Expert Advisor Programming for MetaTrader 4

Creating automated trading systems in the MQL4 language

Andrew R. Young

Expert Advisor Programming
for MetaTrader 4

Creating automated trading systems in the MQL4 language

Andrew R. Young

Edgehill Publishing

Published by Edgehill Publishing, Nashville, TN.

For more information on this book, including updates, news and new editions, please visit our web site at http://www.expertadvisorbook.com/

ISBN: 978-0-9826459-3-2

Table of Contents

Introduction

Since its introduction in 2005, MetaTrader 4 has become the most popular trading platform for Forex and CFDs. It's free, it comes with a full suite of charting tools and indicators, and most of all, it allows the creation of custom trading systems and indicators using the MQL4 programming language. A worldwide community of traders and programmers have created thousands of programs for the MetaTrader 4 platform over the years.

In 2010, MetaQuotes introduced the MetaTrader 5 platform, along with a new programming language, MQL5. The MQL5 language introduced many new features, including object-oriented programming, enumerations, structures, new events and much more. MetaTrader 5 is a radical departure from MetaTrader 4, embracing a position-based trading model, rather than the independent orders that MetaTrader 4 uses.

Despite the many improvements over MetaTrader 4, the Forex trading community was slow to adopt the new platform. Many traders preferred MetaTrader 4, and relied on the large existing code base of expert advisors and indicators written in MQL4. MetaQuotes has confirmed their continued commitment to MetaTrader 4 and added many of the improvements introduced in MetaTrader 5 to the MetaTrader 4 platform.

As of build 600+, the MQL4 language has been updated with many of the features from MQL5, while still maintaining classic MQL4 functionality. Both languages share a common development platform: MetaEditor now compiles both MQL4 and MQL5 programs, and MQL4 programmers can now take advantage of the productivity improvements introduced with MetaEditor 5. With few exceptions, classic MQL4 code will still compile and run on the latest versions of the platform, ensuring backward compatibility with thousands of existing MQL4 programs.

About This Book

This book updates the approach used in the original *Expert Advisor Programming* book, released in 2010. While the code in that book will still work in MetaTrader 4, this book uses the new MQL5 features where appropriate.

The objective of this book is to instruct the reader on the fundamentals of programming expert advisors in MQL4. We will also discuss the creation of indicators, scripts and libraries, although we will not go into depth on these topics. Although prior experience in modern programming languages is helpful, this book assumes no prior programming knowledge. You should have a firm grasp of technical trading systems, the MetaTrader platform, and the Forex market in general.

The list below describes the overall layout of the book. Depending on your experience and your areas of interest, you may choose to skip or simply skim certain chapters:

- **Chapters 2-6** cover the basics of the MQL4 language. If you have experience programming in MQL or similar languages, you can skim these chapters. Otherwise, it is recommended that you read them carefully.
- **Chapters 1, 7 and 8** cover the MetaEditor program, file locations and the basic structure of MQL4 programs.
- **Chapters 9-16** cover order placement, handling, modification and closing.
- **Chapter 18 and 19** cover price data and indicators.
- **Chapters 20-22** cover optional features that can be added to your expert advisors, such as trailing stops, money management and trade timers.
- **Chapter 23** demonstrates various trading systems written in MQL4.
- **Chapter 24** covers miscellaneous MQL4 features that may be useful in your expert advisors.
- **Chapter 25** addresses the creation of custom indicators, scripts and libraries.
- **Chapter 26** demonstrates how to debug your programs, and use the Strategy Tester to test and evaluate performance.

Throughout this book, we will create a framework of classes and functions that will greatly simplify the development of expert advisors, and allow you to concentrate on programming your trading system logic rather than worrying about the details of calculating and executing trades. The source code in this book can be downloaded from the book's website, and used in your own personal projects.

This book is not intended to be a comprehensive overview of all of the new features imported from MQL5, nor does it cover every feature available in the MQL4 language. The *MQL4 Reference* should be your source for learning about all of the features MQL4 has to offer. Many of the new features imported from MQL5 duplicate existing functionality that is already present in MQL4. In these cases, we will generally use the classic MQL4 features instead.

Source Code Download

The source code in this book is freely downloadable from the book's official website at `http://www.expertadvisorbook.com/`. The source code download contains the include files for the expert advisor framework, a printable reference sheet, and several programs that we will create over the course of this book. We will be referring to these files often, so it is recommended that you download and install these files in your MetaTrader 4 data folder.

Conventions Used

`Fixed-width font` refers to program code, file and folder names, URLs or MQL4 language elements.

Italics are for terms that are defined or used for the first time in the text, references to the *MQL4 Reference*, keyboard commands, or interface elements in MetaEditor or MetaTrader.

Chapter 1 - MQL4 Basics

MQL4 Programs

There are three types of programs you can create in MQL4:

- An *expert advisor* is an automated trading program that can open, modify and close orders. You can only attach one expert advisor at a time to a chart. The majority of this book will cover the creation of expert advisors in MQL4.
- An *indicator* displays technical analysis data on a chart or in a separate window using lines, histograms, arrows, bars/candles or chart objects. You can attach multiple indicators to a chart. Chapter 25 will address the creation of indicators in MQL4.
- A *script* is a specialized program that performs a specific task. When a script is attached to a chart, it will execute only once. You can only attach one script at a time to a chart. We will address the creation of scripts in Chapter 25.

File Extensions

An MQ4 file (`.mq4`) contains the source code of an MQL4 program, such as an expert advisor, indicator, script or library. This is the file that is created when we write an MQL4 program in MetaEditor. This file can be opened and edited in MetaEditor or any text editor.

An EX4 file (`.ex4`) is a compiled executable program. When an MQ4 file is compiled in MetaEditor, an EX4 file is produced with the same file name. This is the file that executes when you attach an MQL4 program to a chart. EX4 files are binary files, and cannot be opened or edited.

An MQH file (`.mqh`) is an include file that contains source code for use in an MQL4 program. Like MQ4 files, an MQH file can be opened and edited in MetaEditor.

Other File Types

An *include file* (`.mqh`) is a source code file that contains classes, functions and variables for use in an MQL program. Include files contain useful code that can be reused over and over again. When a program is compiled, the contents of any include files used in the program will be "included" in the compiled program. We will be creating many include files over the course of this book.

A *library* is an executable file that contains reusable functions, similar to an include file. Libraries are in EX4 or Windows DLL format. A library executes in memory as a separate program along with your MQL program.

Libraries are useful if you want to make your functions available to others without making the source code available. We will discuss libraries in more detail in Chapter 25.

An *expert settings* file or preset file (`.set`) contains trade parameters for an expert advisor. Settings files are loaded or saved in the expert advisor *Properties* dialog under the *Inputs* tab. The *Load* button loads parameters from a `.set` file, and the *Save* button saves the current parameters to a `.set` file.

File Locations

As of build 600+, there are two possible locations for the MetaTrader data folder. MetaTrader will place the data folder under the current user's `AppData` folder if any of the following are true:

- You are using Windows Vista or higher, and you have *User Account Control (UAC)* enabled.
- You are logged in via a remote connection.
- The current user account is restricted from writing to the `Program Files` folder.

In this case, the path to the MetaTrader data folder will resemble something like this: `C:\Users\Andrew\AppData\Roaming\MetaQuotes\Terminal\BB190E062770E27C3E79391AB0D1A117\`.

If none of the above are true, then the data folder will be located inside the MetaTrader 4 installation folder. To locate your data folder, open MetaTrader or MetaEditor and select *Open Data Folder* from the *File* menu. A Windows Explorer window will open containing your data folder.

All MQL4 programs are located under the `\MQL4` folder of your MetaTrader 4 data folder. Since all MQL4 programs use the same file extensions, program types are organized in subfolders inside the `\MQL4` folder. Here are the contents of the subfolders in the MQL4 folder:

- **`\Experts`** – This folder contains the MQ4 and EX4 files for expert advisors.
- **`\Indicators`** – This folder contains the MQ4 and EX4 files for indicators.
- **`\Scripts`** – This folder contains the MQ4 and EX4 files for scripts.
- **`\Include`** – This folder contains MQH include files.
- **`\Libraries`** – This folder contains the MQ4, EX4 and DLL files for libraries.
- **`\Images`** – If your program uses bitmap images, they must be stored in this folder in `.bmp` format.
- **`\Files`** – Any external files that you use in your programs, other than include files, libraries, images or other MQL programs, must be stored in this folder.

- **\Presets** – This is the default folder for `.set` files that are loaded or saved from the expert advisor *Properties* dialog or from the *Inputs* tab in the Strategy Tester.
- **\Logs** – The expert advisor logs are saved in this folder. You can view these logs in the *Experts* tab inside the *Toolbox* window in the main MetaTrader interface.

Any references to the above folders in this book assume that they are located under the `\MQL4` folder of the MetaTrader 4 installation folder. So a reference to the `\Experts` folder would refer to the `\MQL4\Experts` subfolder of the MetaTrader 4 installation folder.

MetaEditor

MetaEditor is the IDE (*Integrated Development Environment*) for MQL4 that is included with MetaTrader 4. You can open MetaEditor from the MetaTrader interface by clicking the *MetaEditor* button on the *Standard* toolbar, or by pressing F4 on your keyboard. You can also open MetaEditor from the Windows Start menu.

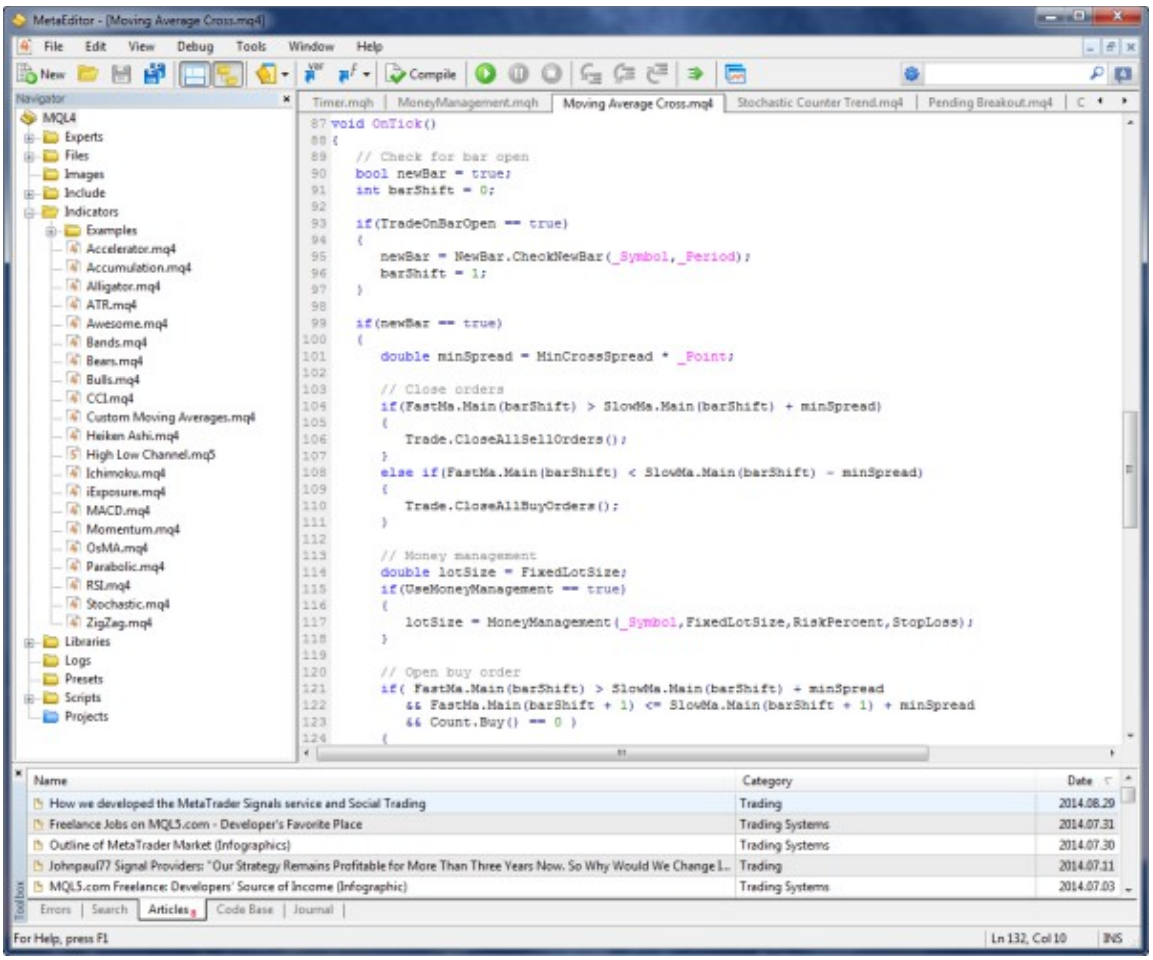


Fig. 1.1 – The MetaEditor interface. Clockwise from top left is the *Navigator* window, the code editor, and the *Toolbox* window.

MetaEditor has many useful features for creating MQL4 programs, including auto completion, parameter info tooltips, search tools, debugging tools and more. Figure 1.2 shows the *List Names* auto completion feature.

Type the first few letters of a variable, type, keyword or function name, and a drop-down list will appear with all matching keywords. Scroll through the list with the up and down arrow keys, and select the keyword to auto-complete by pressing the *Enter* key. You can also select the keyword from the list with the left mouse button. You can recall the List Names drop-down box at any time by pressing *Ctrl+Space* on your keyboard, or by selecting *List Names* from the *Edit* menu.

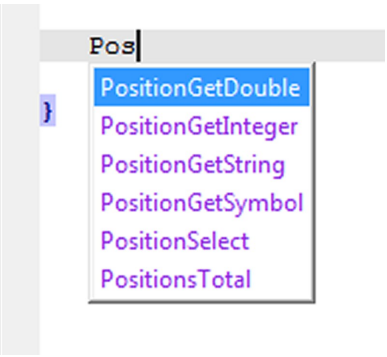


Fig. 1.2 – Auto completion.

Figure 1.3 shows the *Parameter Info* tooltip. When filling out the parameters of a function, the Parameter Info tooltip appears to remind you of the function parameters. The highlighted text in the tooltip is the current parameter.

Some functions have multiple variants – the `SymbolInfoDouble()` function in Figure 1.3 has two variants, as shown by the **[1 of 2]** text that appears in the tooltip. Use the up and down arrows keys to scroll through all variants of the function. You can recall the Parameter Info tooltip at any time by pressing *Ctrl+Shift+Space* on your keyboard, or by selecting *Parameter Info* from the *Edit* menu.

Fig. 1.3 – The *Parameter Info* tooltip.

There are two additional windows inside the MetaEditor interface. The *Navigator* window displays the contents of the MQL4 folder in a folder tree, allowing easy access to your MQL programs. The *Toolbox* window contains several tabs, including the *Errors* tab, which displays compilation errors; the *Search* tab, which displays search results; and the *Articles* and *Code Base* tabs, which display information from the MQL4 website.

MetaEditor has a built-in *MQL4 Reference*, which is useful for looking up MQL4 functions and language elements. Simply position the cursor over an MQL4 keyword and press *F1* on your keyboard. The *MQL4 Reference* will open to the appropriate page. You can also open the *MQL4 Reference* from the *Help* menu.

MQL4 Wizard

The *MQL4 Wizard* is used to create a new MQL4 program. To open the *MQL4 Wizard*, click the *New* button on the toolbar, or select *New* from the *File* menu. A window with the following options will appear:

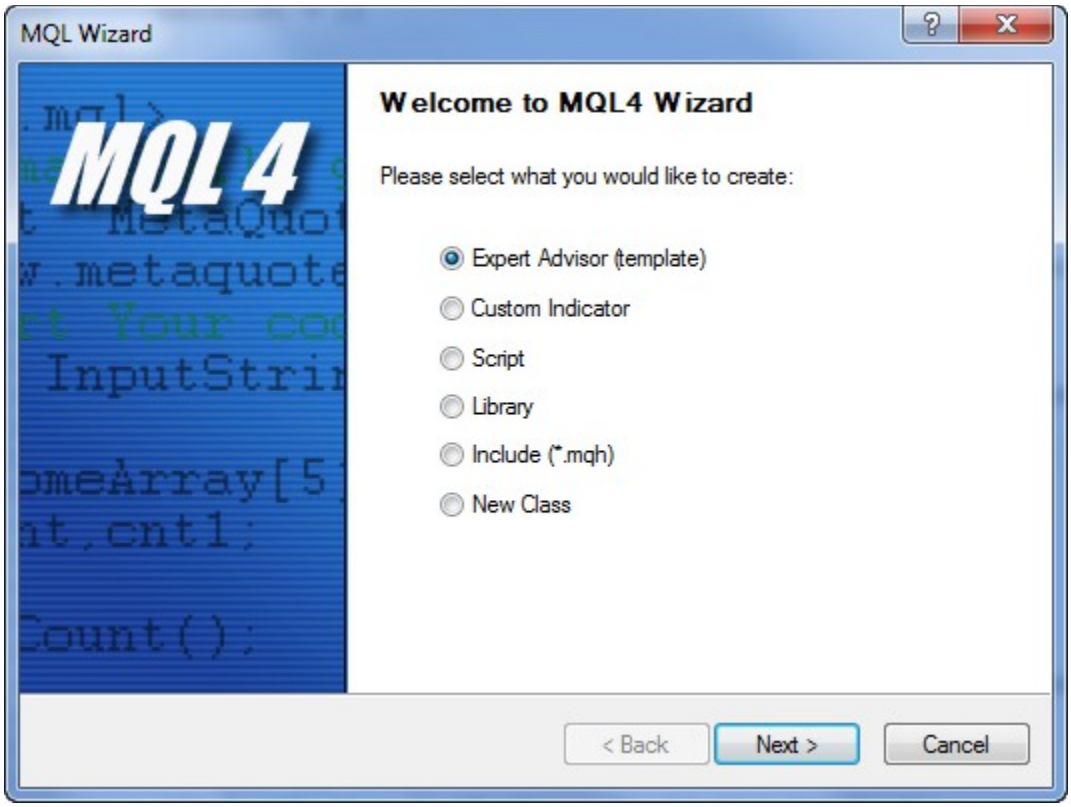


Fig. 1.4 – The *MQL4 Wizard*.

- **Expert Advisor (template)** – This will create a new expert advisor file from a built-in template. The created file is saved in the `\MQL4\Experts` folder or a specified subfolder.
- **Custom Indicator** – This will create a new custom indicator file from a built-in template. The created file is saved in the `\MQL4\Indicators` folder or a specified subfolder.
- **Script** – This will create a blank script file from a built-in template. The created file is saved in the `\MQL4\Scripts` folder or a specified subfolder.
- **Library** – This will create a blank library file from a built-in template. The created file is saved in the `\MQL4\Libraries` folder or a specified subfolder.
- **Include (*.mqh)** – This will create a blank include file from a built-in template with the `.mqh` extension. The created file is saved in the `\MQL4\Include` folder or a specified subfolder.
- **New Class** – This will create an include file with a class definition from a built-in template. The created file is saved in the `\MQL4\Experts` folder, or a specified subfolder.

We will go more in-depth into the creation of programs using the *MQL4 Wizard* throughout the book.

Compilation

To compile an MQL4 program, simply press the *Compile* button on the MetaEditor toolbar. The current MQ4 file and all included files will be checked for errors, and an EX4 file will be produced. Any compilation errors or warnings will appear in the *Errors* tab of the *Toolbox* window.

Errors will need to be fixed before an EX4 file can be generated. Chapter 26 discusses debugging and fixing program errors. Warnings should be examined, but can usually be safely ignored. A program with warnings will still compile successfully.

Syntax

MQL4 is similar to other modern programming languages such as C++, C# or Java. If you've programmed in any modern programming language with C-style syntax, the syntax and structure of MQL will be very familiar to you.

An expression or operator in MQL4 must end in a semicolon (`;`). An expression can span multiple lines, but there must be a semicolon at the end of the expression. Not adding a semicolon at the end of an expression is a common mistake that new programmers make.

```
// A simple expression
x = y + z;

// An expression that spans multiple lines
x = (y + z)
   / (q - r);
```

The one exception to the semicolon rule is the compound operator. A *compound operator* consists of an operator followed by a pair of brackets (`{}`). In the example below, the operator is the `if(x == 0)` expression. There is no semicolon after the closing bracket. Any expressions within the brackets must be terminated with a semicolon.

```
// A simple compound operator
if(x == 0)
{
   Print("x is equal to zero");
   return;
}
```

Identifiers

When naming variables, functions, objects and classes, you need to use a unique and descriptive *identifier*. The identifier must not be identical to another identifier in the program, not should it be the same as an MQL4 language keyword.

You can use letters, numbers and the underscore character (_), although the first character in an identifier should not be a number. The maximum length for an identifier is 64 characters. This give you a lot of room to be creative, so use identifiers that are clear and descriptive.

Identifiers are case sensitive. This means that `MyIdentifier` and `myIdentifier` are not the same! Programmers use capitalization to distinguish between different types of variables, functions and classes. Here is the capitalization scheme we'll use in this book:

- Global objects, classes and function names will capitalize the first letter of each word. For example: `MyFunction()` or `MyObject`.
- Local variables and objects, which are declared inside a function, will use *camel case*. This is where the first letter is lower case, and the first letters of all other words are upper case. For example: `myVariable` or `localObject`.
- Function parameters will be in camel case, starting with a lower-case "p". Global variables will start with a lower-case "g". For example, `pFuncParam` or `gGlobalVar`.
- Private class variables will begin with an underscore character. For example: `_myVariable`.
- Constants are in all upper case. Use underscores to separate words. For example: `MY_CONSTANT`.

Comments

Comments are used to describe what a section of code does in a program. You'll want to use comments throughout your program to keep it organized. You can also use comments to temporarily remove lines of code from your program. Any line that is commented out is ignored by the compiler.

To add a comment, the first two characters should be a double slash (`//`). This will comment a single line of code:

```
// This is a comment

// The line of code below is commented out
// x = y + z;
```

To comment out multiple lines of code, use a slash-asterisk (`/*`) at the beginning of your comment, and an asterisk-slash (`*/`) at the end of your comment.

```
/* This is a multi-line comment
   These lines will be ignored by the compiler
   x = y + z;  */
```

MetaEditor has a set of useful commenting commands. Select the lines you want to comment by highlighting them with your mouse. In the *Edit* menu, under the *Comments* submenu, the *Comment Lines* menu item will comment out the selected lines, while *Uncomment Lines* will remove comments from selected lines. The *Function Header* menu item will insert a commented function header similar to those in the auto-generated MQ4 files:

```
// Function header generated by Edit menu -> Comments -> Function Header

//+------------------------------------------------------------------+
//|                                                                  |
//+------------------------------------------------------------------+
```

Chapter 2 - Variables & Data Types

Variables

A *variable* is the basic unit of storage in any programming language. Variables hold information that is necessary for our program to function, such as prices, indicator values or trade parameters.

Before a variable can be used, it must be *declared*. You declare a variable by specifying the data type and a unique identifier. Optionally, you can *initialize* the variable with a value. You'll generally declare a variable at the beginning of a program or function, or when it is first used. If you declare a variable more than once, or not at all, you'll get a compile error.

Here's an example of a variable declaration:

```
int myNumber = 1;
```

In this example, the data type is `int` (integer), the identifier is `myNumber`, and we initialize it with a value of `1`. Once a variable has been declared, you can change its value by assigning a new value to it:

```
myNumber = 3;
```

The variable `myNumber` now has a value of 3. You can also assign the value of one variable to another variable:

```
int myNumber;
int yourNumber = 2;
myNumber = yourNumber;
```

The variable `myNumber` now has a value of 2.

If you do not initialize a variable with a value, it will be assigned a default empty value. For numerical types, the initial value will be 0, and for string types it will be an empty string (`""`).

Data Types

When declaring a variable, the data type determines what kind of data that variable can hold. Data types in MQL4 can be organized into three types:

- *Integer types* are whole numbers. For example: 0, 1 and 10563.
- *Real types* are fractional numbers with a decimal point. For example: 1.35635.

- *Strings* are text comprised of Unicode characters. For example: "The brown fox jumped over the lazy dog."

Integer Types

Previously, MQL4 had only one integer type, `int`. MQL5 added many more integer types that hold various ranges of whole numbers, and these have since been added to MQL4. Let's start by examining the signed integer types. A *signed type* can hold positive or negative numbers:

- **`char`** – The `char` type uses 1 byte of memory. The range of values is from -128 to 127.
- **`short`** – The `short` type uses 2 bytes of memory. The range of values is -32,768 to 32,767.
- **`int`** – The `int` type uses 4 bytes of memory. The range of values is -2,147,483,648 to 2,147,483,647.
- **`long`** – The `long` type uses 8 bytes of memory. The range of values is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

So which integer type should you use? You will frequently see the `int` type used in MQL4 functions, so that is the integer type you will use the most. Some functions that have been imported from MQL5 will use the `long` type. You can use a `char` or `short` type for your variables if you wish, but you will rarely encounter them in MQL4.

There are also unsigned integer types, which do not allow negative numbers. The *unsigned types* use the same amount of memory as their signed counterparts, but the maximum value is double that of the signed type.

- **`uchar`** – The `uchar` type uses 1 byte of memory. The range of values is 0 to 255.
- **`ushort`** – The `ushort` type uses 2 bytes of memory. The range of values is 0 to 65,535.
- **`uint`** – The `uint` type uses 4 bytes of memory. The range of values is 0 to 4,294,967,295.
- **`ulong`** – The `ulong` type uses 8 bytes of memory. The range of values is 0 to 18,446,744,073,709,551,615.

In practice, you will rarely use unsigned integer types, but they are available for you to use.

Real Types

Real number types are used for storing numerical values with a fractional component, such as prices. There are two real number types in MQL4. The difference between the two types is the level of accuracy when representing fractional values.

- **`float`** – The `float` type uses 4 bytes of memory. It is accurate to 7 significant digits.

- **`double`** – The `double` type uses 8 bytes of memory. It is accurate to 15 significant digits.

You will use the `double` type frequently in MQL4. The `float` type can be used to save memory when dealing with large arrays of real numbers, but it is not used in MQL4 functions.

String Type

The `string` type is used to store text. Strings must be enclosed in double quotes (`"`). Here's an example of a `string` type variable declaration:

```
string myString = "This is a string";
```

If you need to use a single or double quote inside a string, use a backslash character (`\`) before the quote. This is called *escaping* a character.

```
string myQuote = "We are \"escaping\" double quotes";
Print(myQuote);
// Output: We are "escaping" double quotes
```

If you need to use a backslash inside a string, use two backslash characters like this:

```
string mySlash = "This is a backslash: \\";
Print(mySlash);
// Output: This is a backslash: \
```

You can also add a new line character to a string by using the `\n` escape character:

```
string myNewline = "This string has \n a new line character";
Print(myNewline);
// Output: This string has
//         a newline character
```

You can combine strings together using the *concatenation* operator (`+`). This combines several strings into one string:

```
string insert = "concatenated";
string myConcat = "This is an example of a " + insert + " string.";
Print(myConcat);
// Output: This is an example of a concatenated string.
```

The `StringConcatenate()` function can also be used to concatenate strings. It is more memory-efficient than using the concatenation operator. The first parameter of the `StringConcatenate()` function is the `string` variable to copy the concatenated string to, and the remaining parameters are the strings to concatenate:

```
string newString;
string insert = "concatenated";
StringConcatenate(newString,"This is another example of a ", insert, " string");
Print(newString);
// Output: This is another example of a concatenated string.
```

The `newString` variable contains the concatenated string. Note that the strings to concatenate are separated by commas inside the `StringConcatenate()` function.

Finally, if you have a very long string, you can split the string across multiple lines. You do not need to use the concatenation operator. Each line must be enclosed with double quotes, and there must be a semicolon at the end of the expression:

```
string myMultiline = "This is a multi-line string. "
   "These lines will be joined together.";
Print(myMultiline);
// Output: This is a multi-line string. These lines will be joined together.
```

MQL4 has many string functions for comparing, transforming and extracting data from strings. You can view the string functions in the *MQL4 Reference* under the *String Functions* topic.

Boolean Type

The boolean (`bool`) type is used to store true/false values. Technically, the boolean type is an integer type, since it takes a value of 0 (`false`) or 1 (`true`). Here's an example of a boolean type variable declaration:

```
bool myBool = true;
Print(myBool);
// Output: true
```

If a boolean variable is not explicitly initialized with a value of `true`, the default value will be 0, or `false`. Any non-zero value in a boolean variable will evaluate to `true`:

```
bool myBool;
Print(myBool);
// Output: false
```

```
myBool = 5;
if(myBool == true) Print("myBool is true");
// Output: myBool is true
```

In the example above, we initialize the boolean variable `myBool` without a value. Thus, `myBool` is equal to 0 or `false`. When we assign a value of 5 to `myBool`, `myBool` evaluates as `true` in a boolean operation. We'll talk more about boolean operations in Chapter 3.

Color Type

The `color` type is used to store information about colors. Colors can be represented by predefined color constants, RGB values, or hexadecimal values.

You'll use color constants the most. These are the same colors you'll use when choosing a color for an indicator line or a chart object. You can view the full set of color constants in the *MQL4 Reference* under *Standard Constants... > Objects Constants > Web Colors*.

Here's an example of a `color` variable declaration using a color constant:

```
color lineColor = clrRed;
```

The `color` variable `lineColor` is initialized using the color constant for red, `clrRed`. Here's an example using the RGB value for red:

```
color lineColor = C'255,0,0';
```

The RGB value for red is `255,0,0`. An RGB constant begins with a capital C, and the RGB value is enclosed in single quotes. Finally, here's an example using the hexadecimal value for red:

```
color lineColor = 0xFF0000;
```

A hexadecimal value is preceded by '`0x`', and followed by the six-character hexadecimal value, in this case `FF0000`.

RGB and hexadecimal values are used for custom colors – those not defined by color constants. If you are comfortable working with RGB or hexadecimal colors, then you can define your own colors. Otherwise, you'll find the color constants to be easy and useful to work with.

Datetime Type

The `datetime` type is used for storing time and date. The time and date in a `datetime` variable is stored in *Unix time*, which is the number of seconds elapsed since January 1, 1970. For example, January 1, 2012 at midnight in Unix time is 1,325,397,600.

If you need to initialize a `datetime` variable to a specific date and time, use a `datetime` constant. A *datetime constant* begins with a capital D, with the date and time in single quotes. The date and time is represented in the format *yyyy.mm.dd hh:mm:ss*. Here's an example:

```
datetime myDate = D'2012.01.01 00:00:00';
```

The example above initializes the variable `myDate` to January 1, 2012 at midnight. You can omit parts of the `datetime` constant if they are not used. The following examples will demonstrate:

```
datetime myDate = D'2012.01.01 00:00';      // Hour and minute
datetime myDate = D'2012.01.01 00';         // Hour only
datetime myDate = D'2012.01.01';            // Date only
```

All of these examples will initialize the variable `myDate` to the same time – January 1, 2012 at midnight. Since the *hh:mm:ss* part of the `datetime` constant is not used, we can omit it.

So what happens if you leave out the date? The compiler will substitute today's date – the date of compilation. Assuming that today is January 1, 2012, here's what happens when we use a `datetime` constant without a date:

```
datetime myDate = D'';              // 2012.01.01 00:00
datetime myDate = D'02:00';         // 2012.01.01 02:00
```

In the first example, the variable `myDate` is set to today's date at midnight. In the second example, we provide a time in the `datetime` constant. This sets `myDate` to today's date at the specified time.

MQL4 has several predefined constants for the current date and time. The `__DATE__` constant returns the date of compilation. It is the same as using a blank `datetime` constant. The `__DATETIME__` constant returns the current time and date on compilation. Note that there are two underscore characters (_) before and after each constant.

Here's an example using the `__DATE__` constant. We'll assume the current date is January 1, 2012:

```
datetime myDate = __DATE__;         // 2012.01.01 00:00
datetime myDate = D'';              // 2012.01.01 00:00
```

And here's an example using the __DATETIME__ constant. We'll assume the time and date of compilation is January 1, 2012 at 03:15:05:

```
datetime myDate = __DATETIME__;      // 2012.01.01 03:15:05
```

In Chapter 22, we will examine more ways to handle and manipulate `datetime` values.

Constants

A *constant* is an identifier whose value does not change. A constant can be used anywhere that a variable can be used. You cannot assign a new value to a constant like you can for a variable.

There are two ways to define constants in your program. Global constants are defined in your program using the `#define` preprocessor directive. Any `#define` directives are placed at the very beginning of your program. Here's an example of a constant definition:

```
#define COMPANY_NAME "Easy Expert Forex"
```

The `#define` directive tells the compiler that this is a constant declaration. `COMPANY_NAME` is the identifier. The string `"Easy Expert Forex"` is the constant value. The constant value can be of any data type.

A global constant can be used anywhere in your program. Here's an example of how we can use the constant above:

```
Print("Copyright ©2014 ",COMPANY_NAME);
// Output: Copyright ©2014 Easy Expert Forex
```

The constant identifier `COMPANY_NAME` in the `Print()` function is replaced with the constant value `"Easy Expert Forex"`.

Another way of declaring a constant is to use the `const` specifier. By placing a `const` specifier before a variable declaration, you are indicating that the value of the variable cannot be changed:

```
const int cVar = 1;
cVar = 2;            // Compile error
```

The `cVar` variable is set as a constant using the `const` specifier. If we try to assign a new value to this variable, we'll get a compilation error.

Arrays

An *array* is a variable of any type that can hold multiple values. Think of an array as a numerical list. Each number in the list corresponds to a different value. We can iterate through this list numerically, and access each of the values by its numerical index.

Here's an example of an array declaration and assignment:

```
int myArray[3];
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
```

This is called a *static array*. A static array has a fixed size. When a static array is declared, the size of the array is specified inside the square brackets (`[]`). In the example above, the array `myArray` is initialized with 3 elements. The following lines assign an integer value to each of the three elements of the array.

We can also assign the array values when first declaring the array. This line of code accomplishes the same task as the four lines of code in the previous example:

```
int myArray[3] = {1,2,3};
```

The array values are separated by commas inside the brackets (`{}`). They are assigned to the array in the order that they appear, so the first array element is assigned a value of 1, the second array element is assigned a value of 2, and so on. Any array elements that do not have a value assigned to them will default to an empty value – in this case, zero.

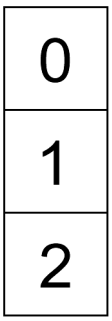


Fig. 2.1 – Array indexing.

Array indexing starts at zero. So if an array has 3 elements, the elements are numbered 0, 1 and 2. See Figure 2.1 to the right. The maximum index is always one less than the size of the array. When accessing the elements of an array, the index is specified in square brackets. In the above example, the first element, `myArray[0]`, is assigned a value of 1, and the third element, `myArray[2]` is assigned a value of 3.

Because this array has only 3 elements, if we try to access an index greater than 2, an error will occur. For example:

```
int myArray[3];
myArray[3] = 4;      // This will cause a compile error
```

Because array indexes start at 0, an index of 3 would refer to the fourth element of an array. This array has only 3 elements, so attempting to access a fourth element will result in an "array out of range" critical error.

Static arrays cannot be resized, but if you need to resize an array, you can declare a dynamic array instead. A *dynamic array* is an array that is declared without a fixed size. Dynamic arrays must be sized before they can be used, and a dynamic array can be resized at any time. The `ArrayResize()` function is used to set the size of a dynamic array.

Here's an example of a dynamic array declaration, sizing and assignment:

```
double myDynamic[];
ArrayResize(myDynamic,3);
myDynamic[0] = 1.50;
```

A dynamic array is declared with empty square brackets. This tells the compiler that this is a dynamic array. The `ArrayResize()` function takes the array name (`myDynamic`) as the first parameter, and the new size (3) as the second parameter. In this case, we're setting the size of `myDynamic` to 3. After the array has been sized, we can assign values to the array.

Dynamic arrays are used in MQL4 for storing indicator values and price data. You will sometimes need to declare a dynamic array for use in an MQL4 function. The functions themselves will take care of properly sizing the array and filling it with data. When using dynamic arrays in your own code, be sure to size them using `ArrayResize()` first.

Multi-Dimensional Arrays

Up to this point, we've been declaring one-dimensional arrays. Let's take a look at a two-dimensional array:

```
double myDimension[3][3];
myDimension[0][1] = 1.35;
```

In the example above, we declare a two-dimensional array named `myDimension`, with both dimensions having three elements. Think of a multi-dimensional array as an "array within an array." It will be easier if we visualize our two-dimensional array as a table. Refer to Fig. 2.2 to the right.

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2
2	2,0	2,1	2,2

Fig. 2.2 – Multi-dimensional array indexing.

The first dimension will be the horizontal rows in our table, while the second dimension will be the vertical columns. So in this example, `myDimension[0][0]` will be the first row, first column; `myDimension[1][2]` would be the second row, third column and so forth. Two-dimensional arrays are very useful if you have a set of data that can be organized in a table format.

Only the first dimension of a multi-dimensional array can be dynamic. All other dimensions must have a static size declared. This is useful if you're not sure how many "rows" your table should have. Let's look at an example:

```
double myDimension[][3];
int rows = 5;
ArrayResize(myDimension,rows);
```

The first dimension is blank, indicating that this is a dynamic array. The second dimension is set to 3 elements. The integer variable `rows` is used to set the size of the first dimension of our array. We'll pass this value to the second parameter of the `ArrayResize()` function. So for example, if `rows = 5`, then the first dimension of our array will be set to 5 elements.

An array can have up to four dimensions, but it is rare that you will ever need more than two or three. In most cases, a *structure* would be an easier method of implementing a complex data structure. We'll cover structures later in the chapter.

Iterating Through Arrays

The primary benefit of arrays is that it allows you to easily iterate through a complete set of data. Here's an example where we print out the value of every element in an array:

```
string myArray[3] = {"cheese","bread","ale"};
for(int index = 0; index < 3; index++)
{
   Print(myArray[index]);
}

// Output:  cheese
            bread
            ale
```

We declare a string array named `myArray`, with a size of 3. We initialize all three elements in the array with the values of "cheese", "bread" and "ale" respectively. This is followed by a `for` loop. The `for` loop initializes a counter variable named `index` to 0. It will increment the value of `index` by 1 on each iteration of the loop. It will keep looping as long as `index` is less than 3. The `index` variable is used as the index of the array. On the first iteration of the loop, `index` will equal 0, so the `Print()` function will print the value of `myArray[0]` to the log – in this case, `"cheese"`. On the next iteration, `index` will equal 1, so the value of `myArray[1]` will be printed to the log, and so on.

As mentioned earlier in the chapter, if you try to access an array element that is larger than the maximum array index, your program will fail. When looping through an array, it is important to know the size of the array. The example above uses a fixed size array, so we know the size of the array beforehand. If you're using a dynamic array, or if you don't know what size an array will be, the `ArraySize()` function can be used to determine the size of an array:

```
int myDynamic[];
ArrayResize(myDynamic,10);

int size = ArraySize(myDynamic);

for(int i = 0; i < size; i++)
{
   myDynamic[i] = i;
   Print(i);                  // Output: 0, 1, 2... 9
}
```

In this example, the dynamic array `myDynamic` is resized to 10 elements. The `ArraySize()` function returns the number of elements in the array and assigns the value to the `size` variable. The `size` variable is then used to set the termination condition for the `for` loop. This loop will assign the value of `i` to each element of the `myDynamic` array, and them print that value to the log.

We'll discuss `for` loops in detail in Chapter 4. Just keep in mind that a loop can be used to iterate through all of the elements of an array, and you will frequently be using arrays for just this purpose.

Enumerations

An *enumeration* is an special integer type that defines a list of constants representing integer values. Only the values defined in an enumeration can be used in variables of that type.

For example, let's say we need to create an integer variable to represent the days of the week. There are only seven days in a week, so we don't need values that are less than 0 or greater than 7. And we'd like to use descriptive constants to specify the days of the week.

Here's an example of an enumeration that allows the user to select the day of the week:

```
enum dayOfWeek
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
```

```
   Friday,
   Saturday,
};
```

We use the type identifier `enum` to define this as an enumeration. The name of our enumeration is `dayOfWeek`. The seven days of the week are listed inside the brackets, separated by commas. The closing bracket terminates with a semicolon.

The members of an enumeration are numbered consecutively, starting at 0. So Sunday = 0, Monday = 1, Saturday = 6 and so on. These integer values happen to correspond with the values that MQL4 uses in the `MqlDateTime` structure for the day of the week. We'll learn more about `MqlDateTime` later in the book. The `ENUM_DAY_OF_WEEK` enumeration built into MQL4 also uses these values.

To use our enumeration, we must define a variable, using the name of our enumeration as the type identifier. This example creates a variable named `day`, and assigns the value for `Monday` to it:

```
dayOfWeek day;
day = Monday;
Print(day);    // Output: 1
```

In the first line, we use the name of our enumeration, `dayOfWeek`, as the type. This is an important concept to note: When we create an enumeration, the name of the enumeration becomes a type, just like the way that `int`, `double` or `string` are types. We then declare a variable of that type by using the enumeration name as the type identifier. This concept also applies to structures and classes, which we'll discuss shortly.

We use the constants defined in our enumeration to assign a value to our `day` variable. In this case, we assign the value of the constant `Monday` to the `day` variable. If we print the value of the `day` variable to the log, the result is 1, because the integer value of the constant `Monday` is 1.

What if you want the enumeration to start at a number other than zero? Perhaps you'd like the members of an enumeration to have non-consecutive values? You can assign a value to each constant in an enumeration by using the assignment operator (=). Here's an example:

```
enum yearIntervals
{
   month = 1,
   twoMonths,     // 2
   quarter,       // 3
   halfYear = 6,
   year = 12,
};
```

The name of our enumeration is `yearIntervals`. The first member name is `month`, which is assigned a value of 1. The next two members, `twoMonths` and `quarter`, are incremented by one and assigned the values of 2 and 3 respectively. The remaining members are assigned their respective values.

One area where enumerations are useful is to provide the user a set of values to choose from. For example, if you're creating a timer feature in your expert advisor, and you want the user to choose the day of the week, you can use the `dayOfWeek` enumeration defined earlier as one of your input parameters. The user will see a drop-down list of constants in the expert advisor *Properties* dialog to choose from.

There are many predefined standard enumerations in MQL4. All of the standard enumerations in MQL4 begin with `ENUM_` and are all uppercase with underscore characters. You can view the standard enumerations in the *MQL4 Reference* under *Standard Constants... > Enumerations and Structures*.

Structures

A *structure* is a set of related variables of different types. The concept is similar to a class, although structures do not have functions like classes do. We will discuss classes in Chapter 6. MQL4 comes with several predefined structures that are used with various functions that have been imported from MQL5. Unless you plan on using the new MQL5 functions, you do not need to worry about them.

Let's look at an example of a structure. This structure could be used to store trade settings:

```
struct tradeSettings
{
   ulong slippage;
   double price;
   double stopLoss;
   double takeProfit;
   string comment;
};
```

The type identifier `struct` defines this as a structure. The name of the structure is `tradeSettings`. There are six member variables defined inside the brackets. Although you can assign a default value to a member variable using the assignment operator (=), typically we assign values after an object has been initialized.

To use a structure, we need to first create an object using the structure name as the type. Then we use the object to access the member variables of the structure. Here's an example of how we would use our structure in code:

```
tradeSettings trade;
trade.slippage = 50;
trade.stopLoss = StopLoss * _Point;
```

Using our structure name `tradeSettings` as the type, we define an object named `trade`. This object allows us to access the member variables of the structure. We'll talk more about objects in Chapter 6. To access the structure members, we use the dot operator (`.`) between the object name and the member name. In this case, we assign the value 50 to the structure member `slippage`, and assign a calculated stop loss value to the structure member `stopLoss`.

Using a structure allows you to group related variables and data into a single object. We will work with structures in Chapter 22, when we create time and date functions.

Typecasting

The process of converting a value from one type to another is called *typecasting*. When you copy the contents of a variable into another variable of a different type, the contents are *casted* into the proper type. This can produce unexpected results if you're not careful.

When copying numerical values from one variable to another, there is the possibility of data loss if copying from one type to another, smaller type. Remember our discussion of integer types on page 14. If you copy an `int` value into a `long` variable, there is no possibility of data loss, since the `long` type can hold a larger range of values.

You can freely copy smaller numerical types into larger ones. A `float` value copied into a `double` variable would suffer no data loss. However, a `double` value copied into a `float` variable would result in the `double` value being truncated. The same can result if you copy an `int` value into a `short` variable. This is especially true if you are casting a floating-number type (such as a `double`) into an integer type. Anything after the decimal point will be lost. This is fine if you don't need the fractional part of the number though.

If you are casting a value of a larger type into a variable of a smaller type, the compiler will warn you with the message "possible loss of data due to type conversion." If you are certain that the value of the larger type won't overflow the range of the smaller type, then you can safely ignore the message. Otherwise, you can *explicitly* typecast the new value to make the warning go away.

For example, if you have a `double` value that you need to pass to a function that requires an integer value, you'll get a "possible loss of data due to type conversion" warning. You can cast the new value as an integer by prefacing the variable with `(int)`. For example:

```
double difference = (high - low) / _Point;
BuyStopLoss(_Symbol,(int) difference);
```

The double variable `difference` is calculated by dividing two floating-point values. The `BuyStopLoss()` function requires an integer value for the second parameter. Passing the `difference` variable will cause a

compiler warning. By prefacing the `difference` variable name with `(int)`, we cast the value of `difference` to an integer, effectively rounding the value down and avoiding the compiler error.

Conversion

Sometimes, you may wish to convert a variable of one type to another. For example, if you have a string that contains a price value, you may wish to convert it to a double. MQL4 has a full set of conversion functions that can be used to convert from one type to another.

This example uses the `StringToDouble()` function to convert a string containing a valid numeric value to a double:

```
string priceStr = “1.35460”;
double price = StringToDouble(priceStr);
```

The price variable now contains a double equal to 1.35460. You can view all of the conversion functions in the *MQL4 Reference* under *Conversion Functions*.

Input Variables

The *input variables* of an MQL4 program are the only variables that can be changed by the user. These variables consist of trade settings, indicator settings, stop loss and take profit values, and so on. They are displayed under the *Inputs* tab of the program's *Properties* dialog.

An input variable is preceded by the `input` keyword. Input variables are placed at the beginning of your program, before any functions or other program code. Input variables can be of any type, including enumerations. Arrays and structures cannot be used as input variables. The identifiers for input variables should be clear and descriptive.

Note that the previous versions of MQL4 used the `extern` keyword for input variables! While this approach will still work for older programs, you should use the `input` keyword when writing new programs or updating old code.

Here's an example of some input variables that you may see in an expert advisor:

```
input int MaPeriod = 10;
input ENUM_MA_METHOD MaMethod = MODE_SMA;
input double StopLoss = 20;
input string Comment = "ea";
```

These input variables set the period and calculation method for a moving average indicator, adds a stop loss value in points, and specifies a comment to be added to the order.

You can set a user-friendly display name in the *Inputs* tab by appending an input variable with a comment. The comment string will appear in the *Variable* column of the *Inputs* tab. Here are the input variables defined above with a descriptive comment:

```
input int MaPeriod = 10;                    // Moving average period
input ENUM_MA_METHOD MaMethod = MODE_SMA;    // Moving average method
input double StopLoss = 20;                 // Stop loss (points)
input string Comment = "ea";                // Trade comment
```

A *static input variable* can be defined by using the `sinput` keyword. The value of a static input variable can be changed, but it cannot be optimized in the Strategy Tester. Static input variables are useful for logical grouping of input parameters. Simply declare an `sinput` variable of the string type and include a comment:

```
sinput string MASettings;     // Moving average settings
```

Figure 2.3 below shows how the input variables above will appear in the *Inputs* tab of the *Properties* window:

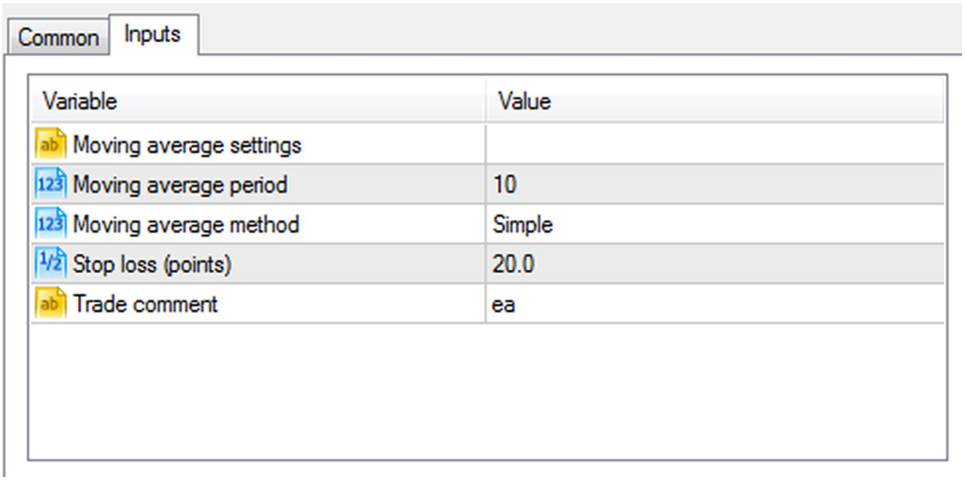


Fig. 2.3 – The *Inputs* tab, showing comments in lieu of variable names.

To use an enumeration that you've created as an input variable type, you'll need to define the enumeration before the input variable itself. We'll use the `dayOfWeek` enumeration that we defined earlier in the book:

```
enum dayOfWeek
{
   Sunday,
   Monday,
```

```
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
};

input dayOfWeek day = Monday;
```

This creates an input variable named `day`, using the `dayOfWeek` enumeration as the type, with a default value of `Monday` or 1. When the user attempts to change the value of the `day` input variable, a drop-down box will appear, containing all of the constants in the enumeration.

Local Variables

A *local variable* is one that is declared inside of a function. Local variables are allocated in memory when the function is first run. Once the function has exited, the variable is cleared from memory.

In this example, we'll create a simple function. We'll discuss functions in more detail in Chapter 5. We will declare a local variable inside our function. When the code in this function is run, the variable is declared and used. When the function exits, the variable is cleared from memory:

```
void myFunction()
{
   int varInt = 5;
   Print(varInt);    // Output: 5
}
```

The name of this function is `myFunction()`. This function would be called from somewhere else in our program. The variable `varInt` is local to this function. This is referred to as the *variable scope*. Local variables cannot be referenced from outside of the function, or anywhere else in the program. They are created when the function is run, and disposed of when the function exits.

Let's take a closer look at *variable scope*. In previous versions of MQL4, a variable only needed to be declared once, anywhere inside a function, at which point it was local to the entire function. With the newer versions of MQL4, the `#property strict` preprocessor directive can be used to modify the variable scope. We will discuss preprocessor directives in more detail in Chapter 7, but for now, let's demonstrate how the `#property strict` directive works.

All preprocessor directives are placed at the top of the source code file:

```
#property strict
```

When the `strict` property is present, a local variable's scope is limited to the block that is is declared in. A *block* is defined as a function, or a compound operator within a function. A block is surrounded by opening and closing brackets (`{}`). Any variables declared inside a block are local only to that block. Let's take a look at a modified example of our function:

```
void myFunction()
{
   bool varBool = true;

   if(varBool == true)
   {
      int varInt = 5;
      Print(varInt);    // Output: 5
   }
}
```

We've added a new local variable – a boolean variable named `varBool`. We've also added an `if` operator block. The `if` operator, its accompanying brackets and the code inside them are a compound operator. Any variables declared inside the brackets are local to the `if` operator block.

The `if` expression can be read as: "If `varBool` is set to `true`, then execute the code inside the brackets." In this case, the expression is true, so the code inside the brackets is run. The `varInt` variable is declared and assigned a value of 5, and that value is printed to the log.

The `varInt` variable is local to the `if` operator block. That means that `varInt` cannot be referenced outside of the block. Once the block is exited, `varInt` is out of scope. If we attempt to reference it from outside the `if` operator block, we'll get a compilation error:

```
if(varBool == true)
{
   int varInt = 5;
   Print(varInt);    // Output: 5
}

varInt = 7;          // Both of these expressions
Print(varInt);       // will produce an error if #property strict is set
```

The expression `varInt = 7` will produce an error on compilation if the `strict` property is set. The variable `varInt` that we declared inside the `if` operator block is out of scope. If we want to correct this code, we can simply declare the `varInt` variable that is outside the `if` operator block:

```
if(varBool == true)
{
   int varInt = 5;
   Print(varInt);    // Output: 5
}

int varInt = 7;      // This is a different variable!
Print(varInt);       // Output: 7
```

Now we have a variable named `varInt` in this scope. Note that this is a different variable than the `varInt` variable declared inside the `if` operator block, even though they both have the same name!

Here's another example: The variable `varInt` is declared at the top of the function, and another variable of the same name and type is declared inside the `if` operator block nested inside the function.

```
void myFunction()
{
   bool varBool = true;
   int varInt = 7;

   if(varBool == true)
   {
      int varInt = 5;
      Print(varInt); // Output: 5
   }

   Print(varInt);                    // Output: 7
}
```

An integer variable named `varInt` is declared at the top of the function, and assigned a value of 7. A second integer variable named `varInt` is declared inside the `if` operator block and initialized with a value of 5. When we print the value of `varInt` inside the `if` operator block, it prints a value of 5.

When the `if` operator block is exited, and we print the value of `varInt` again, this time it prints a value of 7 – the same value that was declared at the top of the function! In other words, the `varInt` variable that was declared inside the `if` operator block overrides the one declared in the scope of the function. By the way, if you try to compile this, you'll get a warning message stating "declaration of 'varInt' hides local declaration..." The example above is not considered good practice, so renaming the second `varInt` variable would fix the warning.

Just to clarify things, let's see what would happen if we declared the `varInt` variable once outside of the `if` operator block, and then referenced it inside the `if` operator block:

```
void myFunction()
{
   bool varBool = true;
   int varInt = 5;

   if(varBool == true)
   {
      Print(varInt);          // Output: 5
   }
}
```

This will work as expected, because the `varInt` variable is declared outside of the `if` operator block, in the function scope. A variable declared in a higher scope is still in scope inside any nested blocks, as long as those nested blocks don't have a variable of the same name and type.

This may seem arbitrary and confusing, but most modern programming languages treat variable scope this way. By requiring that variables be local only inside the block in which they are declared, programming errors are prevented when variables share the same name.

When you create a new expert advisor program using the *MQL4 Wizard*, the wizard will automatically add the `#property strict` preprocessor directive to the top of the source code file. It is suggested that you use the `strict` directive in your programs. We will be using them in the programs throughout this book.

Global Variables

A *global variable* is one that is declared outside of any function. Global variables are defined at the top of your program, generally after the input variables. The scope of a global variable is the entire program.

As demonstrated in the previous section, a local variable declared inside a block will override any variable of the same name and type in a higher scope. If you do this to a global variable, you'll get a compilation warning: "Declaration of variable hides global declaration." There is no practical reason to override a global variable this way, so watch for this.

The value of a global variable can be changed anywhere in the program, and those changes will be available to all functions in the program. Here's an example:

```
// Global variable
int gGlobalVarInt = 5;

void functionA()
{
   gGlobalVarInt = 7;
}
```

```
void functionB()
{
   Print(gGlobalVarInt);            // Output: 7
}
```

The global variable `gGlobalVarInt` is declared at the top of the program, and assigned a value of 5. We'll assume that `functionA()` is executed first. This changes the value of `gGlobalVarInt` to 7. When `functionB()` is run, it will print the changed value of `gGlobalVarInt`, which is now 7.

If you have a variable that needs to be accessed by several functions throughout your program, declare it as a global variable at the top of your program. If you have a local variable whose value needs to be retained between function calls, use a static variable instead.

Static Variables

A *static variable* is a local variable that remains in memory even when the program has exited the variable's scope. We declare a static variable by prefacing it with the `static` modifier. Here's an example of a static variable within a function. On each call of the function, the static variable is incremented by 1. The value of the static variable is retained between function calls:

```
void staticFunction()
{
   static int staticVar = 0;
   staticVar++;              // Increments staticVar by 1
   Print(staticVar);         // Output: 1, 2, 3, etc.
}
```

A static integer variable named `staticVar` is declared with the `static` modifier. On the first call of the function, `staticVar` is initialized to 0. The variable is incremented by 1, and the result is printed to the log. The first entry in the log will read 1. On the next call of the function, `staticVar` will have a value of 1. (Note that it will not be reinitialized to zero!) The variable is then incremented by 1, and a value of 2 is printed to the log, and so on.

Predefined Variables

MQL4 has several predefined variables to access commonly used values. The predefined variables are prefaced by the underscore character (_). All of these variables have equivalent functions as well, but since they are frequently used as function parameters, the predefined variables are easier to read.

Here is a list of commonly-used predefined variables in MQL4. All of these variables refer to the properties of the chart that the program is currently attached to:

- **`_Symbol`** – The symbol of the financial security on the current chart.
- **`_Period`** – The period, in minutes, of the current chart.
- **`_Point`** – The point value of the current symbol. For five-digit Forex currency pairs, the point value is 0.00001, and for three-digit currency pairs (JPY), the point value is 0.001.
- **`_Digits`** – The number of digits after the decimal point for the current symbol. For five-digit Forex currency pairs, the number of digits is 5. JPY pairs have 3 digits.

Chapter 3 - Operations

Operations

We can perform mathematical operations and assign the result to a variable. You can even use other variables in an operation. Let's demonstrate how to perform basic mathematical operations in MQL4.

Addition & Multiplication

```
// Addition
int varAddA = 3 + 5;                // Result: 8
double varAddB = 2.5 + varAddA;     // Result: 10.5

// Multiplication
int varMultA = 5 * 3;               // Result: 15
double varMultB = 2.5 * varMultA;   // Result: 37.5
```

In the first example of each operation, we add or multiply two integers together and assign the result to an `int` variable (`varAddA` and `varMultA`). In the second example, we add or multiply a real number (or floating-point number) by a variable containing an integer value. The result is stored in a `double` variable (`varAddB` or `varMultB`).

If you know that two values will be of integer types, then it's fine to store them in an integer variable. If there's any possibility that one value will be a floating-point number, then use a real number type such as `double` or `float`.

Subtraction & Negation

```
// Subtraction
int varSubA = 5 - 3;          // Result: 2
double varSubB = 0.5 - varSubA;      // Result: -1.5

// Negation
int varNegA = -5;
double varNegB = 3.3 - varNegA;      // Result: 8.3
```

In the subtraction example, we first subtract two integers and assign the value to an `int` variable, `varSubA`. Then we subtract our integer variable from a floating point number, 0.5. The result is a negative floating point number that is assigned to a `double` variable, `varSubB`.

You can specify a numerical constant as a negative number simply by placing a minus sign (-) directly in front of it. The `int` variable `varNegA` has a value of -5 assigned to it. Next, we subtract -5 from 3.3 and assign the result to `varNegB`. Since we are subtracting a negative number from a positive number, the result is an addition operation that results in a value of 8.3.

Division & Modulus

```
// Division
double varDivA = 5 / 2;       // Result: 2.5
int varDivB = 5 / 2;          // Result: 2

// Modulus
int varMod = 5 % 2;           // Result: 1
```

The first division example, `varDivA`, divides two integers and stores the result in a `double` variable. Note that 5 / 2 doesn't divide equally, so there is a fractional remainder. Because of the possibility of fractional remainders, you should always use a real number type (such as `double`) when dividing!

The second example demonstrates: We divide 5 / 2 and store the result in an `int` variable, `varDivB`. Because integer types don't store fractional values, the value is rounded down to the nearest whole number.

The modulus example divides 5 by 2, and stores the integer remainder in `varMod`. You can only use the modulus operator (%) on two integers. Therefore, it is safe to store the remainder in an `int` variable.

Assignment Operations

Sometimes you will need to perform a mathematical operation using a variable, and then assign the result back to the original variable. Here's an example using addition:

```
int varAdd = 5;
varAdd = varAdd + 3;
Print(varAdd);          // Output: 8
```

We declare the variable `varAdd` and initialize it to a value of 5. Then we add 3 to `varAdd`, and assign the result back to `varAdd`. The result is 8. Here's a shorter way to do this:

```
int varAdd = 5;
varAdd += 3;            // varAdd = 8
```

Here we combine a mathematical operator (+) with the assignment operator (=). This can be read as "Add 3 to `varAdd`, and assign the result back to `varAdd`." We can do this with other mathematical operators as well:

```
varSub -= 3;         // Subtraction assignment
varMult *= 3;        // Multiplication assignment
varDiv /= 3;         // Division assignment
varMod %= 3;         // Modulus assignment
```

Increment and Decrement Operations

Frequently, you will need to increment or decrement a variable, especially inside a loop. The increment and decrement operators will add or subtract a value of one to an integer variable. You will see these used a lot inside of `for` loops. We will discuss loops in the next chapter.

To increment an integer variable, use the increment operator (++):

```
int inc = 1;
inc++;         // Add 1 to inc
Print(inc);    // Result: 2
```

The `inc` variable is declared with a value of 1. The increment operator adds 1 to the value of the variable. When we print the value of `inc`, we can see that the value is now 2.

Placing the increment operator after the variable is called a post-increment operation. The value of `inc` isn't increased until after the expression has been evaluated. If you need to immediately increment a variable, place the increment operator before the variable name. This is useful if you need to increment a variable and use it inside an expression on the same line:

```
int inc = 1;
int add = 2;

if(add - ++inc == 0) // The value of inc is now 2
{
   Print(true);         // Result: true
}
```

The `if` expression increments the value of `inc`, and them subtracts it from the value of `add`. The result is zero, so the value `true` is printed to the log.

The decrement operator subtracts one from an integer variable. Its use is identical to the increment operator:

```
int dec = 2;
dec--;
Print(dec);    // Result: 1
```

Relation Operations

You will often have to compare two values in a greater than, less than, equal or non-equal relationship. The operation evaluates to a boolean result, either `true` or `false`. Let's take a look at *greater than* and *less than* operations:

```
a > b       // Greater than
a < b       // Less than
```

In the first example, if `a` is greater than `b`, the result is true, otherwise false. In the second example, if `a` is less than `b`, the result is true. You can also check for equality as well:

```
a >= b      // Greater than or equal to
a <= b      // Less than or equal to
```

In the first example, if `a` is greater than or equal to `b`, the result is true. In the second example, if `a` is less than or equal to `b`, the result is true. Let's look at equal and non-equal operations:

```
a == b      // Equal to
a != b      // Not equal to
```

In the first example, if `a` is equal to `b`, the result is true. In the second example, if a is not equal to b, the result is true. Note that the equality operator (==) is not the same as the assignment operator (=)! This is a common mistake made by new programmers.

When using real numbers, it is important to normalize or round the numbers to a specific number of decimal places. This is done using the `NormalizeDouble()` function. The first argument is the `double` value to normalize. The second argument is the number of digits after the decimal point to round to. Here's an example:

```
double normalA = 1.35874934;
double normalB = 1.35873692;

normalA = NormalizeDouble(normalA,4);       // Result: 1.3587
normalB = NormalizeDouble(normalB,4);       // Result: 1.3587

if(normalA == normalB)
{
   Print("Equal");
}
```

In this example, we have two `double` variables containing fractional values to 8 decimal places. We use the `NormalizeDouble()` function to round these numbers to 4 decimal places and assign the results to their original variables. We can then compare them in an equality statement. In this case, `normalA` and `normalB` are equal, so the string `"Equal"` is printed to the log.

If we tried to perform an equality operation on two prices without normalizing the numbers, it's unlikely we would ever get an equal result. Internally, prices and indicator values are calculated out to a large number of significant digits. By normalizing the numbers, we can check for equality using a smaller number of digits.

Boolean Operations

A boolean operation compares two or more operations (mathematical, boolean or relation) using logical operators, and evaluates whether the expression is true or false. There are three logical operations: AND, OR and NOT.

An AND operation is true if all of the operations in the expression are true. The logical operator for AND is && (two ampersands). Here's an example of an AND operation:

```
int a = 1;
int b = 1;
int c = 2;

if(a == b && a + b == c)
{
   Print(true);         // Result: true
}
```

First, we declare and initialize three integer variables: `a`, `b` and `c`. We use an `if` operator to evaluate our boolean operation. If the operation is true, the code inside the brackets is run.

The value of `a` is 1, and the value of `b` is 1. The expression `a == b` is true, so we go on to the next expression. The addition operation `a + b` equals 2. The value of `c` is also 2, so this expression is true. The boolean AND operation evaluates to true, so a value of `true` is printed to the log.

Let's see what happens if we have a false expression in our AND operation:

```
int a = 1;
int b = 1;
int c = 3;
```

```
if(a == b && a + b == c)
{
   Print(true);
}
else
{
   Print(false);        // Result: false
}
```

In this example, we initialize the value of `c` to 3. Since `a + b` is not equal to `c`, the expression evaluates to false, and thus the boolean AND operation evaluates to false. In this case, the execution skips to the `else` operator, and a value of `false` is printed to the log.

Next, we'll examine the OR boolean operation. An OR operation is true if any of the operations in the expression evaluate to true. The OR operator is `||` (two pipes):

```
int a = 1;
int b = 1;
int c = 3;

if(a == b || a + b == c)
{
   Print(true);         // Result: true
}
```

This code is almost identical to the previous example, except we are using an OR operator. The expression `a == b` is true, but `a + b == c` is not. Since at least one of the expressions is true, the boolean OR operation evaluates to true, and a value of `true` is printed to the log.

Finally, we'll examine the NOT boolean operation. The NOT operator (!) is applied to a single boolean expression. If the expression is true, the NOT operation evaluates to false, and vice versa. Essentially, the NOT operator reverses the true/false value of a boolean expression. Here's an example:

```
bool not = false;

if(!not)
{
   Print(true);         // Result: true
}
```

The variable `not` is initialized to false. The boolean expression `!not` evaluates to true. Thus, a value of `true` is printed to the log. The NOT operator works on more complex expressions as well:

```
int a = 1;
int b = 2;

if(!(a == b))
{
   Print(true);          // Result: true
}
```

The expression `a == b` is false. By enclosing the expression in parentheses and applying the NOT operator, the expression evaluates to true and a value of `true` is printed to the log.

Chapter 4 - Conditional & Loop Operators

Conditional Operators

One of the most basic functions of any program is making decisions. Conditional operators are used to make decisions by evaluating a true/false condition. There are three conditional operators in MQL4: the `if-else` operator, the ternary operator, and the `switch-case` operator.

The if Operator

You've already been introduced to the `if` operator. The `if` operator is the most commonly-used conditional operator in MQL4, and one that you'll use often. It is a compound operator, meaning that there is usually more than one expression contained inside the operation.

The `if` operator evaluates a condition to true or false. The condition can be a relational or boolean operation. If the condition is true, the expression(s) inside the compound operator are executed. If the condition is false, control passes to the next line of code. Let's look at a simple `if` expression:

```
bool condition = true;

if(condition == true)   // If the condition in parentheses is true...
{
   Print(true);         // Execute the code inside the brackets
}
```

We declare a boolean variable named `condition`, and set its value to `true`. Next, we evaluate the boolean condition in the `if` operator. In this case, `condition == true`, so we execute the code inside the brackets, which prints a value of `true` to the log.

When an `if` compound operator has only one expression, you can omit the brackets and place it on the same line as the `if` operator. You can even place the expression on the next line. Note that this only works with a single expression, and that expression must be terminated with a semicolon:

```
// Single line
if(condition == true) Print(true);

// Multi line
if(condition == true)
   Print(true);
```

You can have multiple `if` expressions next to each other. Each `if` expression will be evaluated individually. In the example below, two `if` operators evaluate a relational operation. Both `Print()` functions will be executed, since 2 is greater than 1 and less than 3:

```
int number = 2;

if(number > 1) Print("number is greater than 1");
if(number < 3) Print("number is less than 3");

// Result:  number is greater than 1
//          number is less than 3
```

The else Operator

The *else* operator is the companion to the `if` operator. The `else` operator is placed after an `if` operator. When the `if` operator evaluates to false, the expression(s) inside the `else` operator are executed instead:

```
bool condition = false;

if(condition == true) Print(true);          // If condition is true
else Print(false);                          // If condition is false
                                            // Result: false
```

In this example, the `if` operator evaluates to false, so the expression inside the `else` operator is run and a value of `false` is printed to the log. The `else` operator is useful if you have a default action that you want to be carried out when all other conditions are false.

The `else` operator can be combined with the `if` operator, allowing multiple conditions to be evaluated. When one or more `else if` operators are placed in an `if-else` block, the first true condition will end execution of the `if-else` block. An `else if` operator must be placed after the first `if` operator, and before any `else` operator:

```
int oneOrTwo = 2;

if(oneOrTwo == 1)
   Print("oneOrTwo is 1");

else if(oneOrTwo == 2)              // If previous if operator is false
   Print("oneOrTwo is 2");          // Result: oneOrTwo is 2

else                                // If both if operators are false
   Print("oneOrTwo is not 1 or 2");
```

In this example, we declare an integer variable, `oneOrTwo`, and assign a value of 2. The condition for the `if` operator, `oneOrTwo == 1`, is false, so we move on to the `else if` operator. The `else if` operator condition, `oneOrTwo == 2` is true, so the string `"oneOrTwo is 2"` is printed to the log.

The `if` and `else if` operators are evaluated in order until one of them evaluates to true. Once an `if` or `else if` operator evaluates to true, the expression(s) inside the operator are executed, and the program resumes execution after the `if-else` block. If none of the `if` or `else if` operators evaluates to true, then the expression(s) inside the `else` operator will execute instead.

For example, if `oneOrTwo` is assigned a value of 1, the expression in the first `if` operator, `oneOrTwo == 1`, will be true. The message `"oneOrTwo is 1"` will be printed to the log. The following `else if` and `else` operators will not be evaluated. The program will resume execution after the `else` operator.

If all of the `if` and `else if` operators are false, the expression in the `else` operator is executed. In this case, the message `"oneOrTwo is not 1 or 2"` will be printed to the log:

```
int oneOrTwo = 3;

if(oneOrTwo == 1)
   Print("oneOrTwo is 1");

else if(oneOrTwo == 2)
   Print("oneOrTwo is 2");

else                                  // Result: oneOrTwo is not 1 or 2
   Print("oneOrTwo is not 1 or 2");
```

Note that the `else` operator is not required in any `if-else` block. If it is present, it must come after any `if` or `else if` operators. You can have multiple `else if` operators, or none at all. It all depends on your requirements.

Ternary Operator

The *ternary* operator is a single-line shortcut for the `if-else` operator. The ternary operator is new to MQL4. A ternary operator consists of three parts. The first part is the true/false condition to be evaluated. The second part is the expression to be executed if the condition is true. The third part is the expression to be executed if the condition is false. The result of the expression is assigned to a variable. Here's an example:

```
bool condition = true;
bool result = condition ? true : false;

Print(result);                        // Output: true
```

We declare a boolean variable named `condition`, and set the value to `true`. This variable is used as the condition for the ternary operator. A question mark (`?`) separates the condition from the expressions. The first expression assigns a value of true to the boolean variable `result`. The second expression assigns a value of false to the `result` variable. The expressions are separated by a colon (`:`).

In this case, `condition == true`, so the first expression, `true`, is assigned to the variable `result`. Here's how we would express this using the if-else operator:

```
bool condition = true;
bool result;

if(condition == true) result = true;
else result = false;

Print(result);                        // Output: true
```

We saved two lines of code using the ternary operator. Whichever you prefer to use is up to you.

Switch Operator

The *switch* operator compares an expression to a list of constants, each using the `case` operator. When a constant value is matched, the accompanying expressions are executed. Here's an example:

```
int x = 1;

switch(x)
{
   case 1:
      Print("x is 1");        // Output: x is 1
      break;

   case 2:
      Print("x is 2");
      break;

   default:
      Print("x is not 1 or 2");
}
```

We declare an integer variable, `x`, and assign a value of 1. The `switch` operator contains the expression to evaluate. In this case, the expression is the variable `x`. The `case` operators are labels, each assigned to a different constant value. Since `x` is equal to 1, the string "`x is 1`" will be printed to the log. The `break` operator ends execution of the `switch` operator.

If the expression inside the `switch` operator does not match any of the `case` operators, the optional `default` operator will execute instead. For example, if `x` does not match any of the `case` operators, the expressions after the `default` operator are executed instead. So if `x` were assigned a value of 3, the string `"x is not 1 or 2"` is printed to the log.

Unlike an `if-else` block, execution of the `switch` operator block does not stop when a `case` constant is matched. Unless a `break` operator is encountered, execution will continue until all remaining expressions in the `switch` operator have been executed. Here's an example:

```
int x = 1;

switch(x)
{
   case 1:

   case 2:

   case 3:
      Print("x is 1, 2 or 3");              // Output: x is 1, 2 or 3
      break;

   default:
      Print("x is not 1, 2, or 3");
      break;
}
```

Note that there are no expressions following the `case 1` or `case 2` labels. If either of these labels are matched, the program will begin executing any expressions following the case labels until a `break` operator is encountered, a `default` operator is encountered, or the `switch` operator ends.

In this example, the variable `x` has a value of 1. The expression `x` matches the first `case` label. Execution continues past the `case 3` label, to the `Print()` function. The string `"x is 1, 2 or 3"` is printed to the log, and the `break` operator exits the `switch` block. If `x` did not match any of the `case` labels, then the expressions in the `default` operator would execute instead.

The `switch` operator is useful is a few specific situations. In most cases, an `if-else` block will work just as well, although a `switch-case` block may be more efficient and compact. Here's a useful example of a `switch-case` block. This code will evaluate the chart period in minutes, and return a string if the chart period matches several common chart periods:

```
int period = _Period;
string printPeriod;
```

```
switch(period)
{
   case 60:
      printPeriod = "H1";
      break;

   case 240:
      printPeriod = "H4";
      break;

   case 1440:
      printPeriod = "D1";
      break;

   default:
      printPeriod = "M" + period;
}
```

The integer variable `period` is assigned the period of the current chart, in minutes, using the predefined `_Period` variable. The `switch` operator compares `period` to several common chart periods, including H1, H4 and D1. If `period` matches any of these chart periods, the appropriate string is assigned to the string variable `printPeriod`. In the event that none of these chart periods are matched, a chart period string is constructed using the prefix "M" and the period in minutes.

For example, if `period == 240`, the variable `printPeriod` is assigned the string `"H4"`. If `period == 15`, the expression following the `default` operator will execute, and `printPeriod` is assigned the string `"M15"`. The variable `printPeriod` can be used to print a user-friendly chart period to the log or to the screen.

Loop Operators

Sometimes it is necessary for a program to repeat an action over and over again. For this, we use loop operators. There are three loop operators in MQL4: *while*, *do-while* and *for*.

The while Operator

The `while` loop is the simplest loop in MQL4. The `while` operator checks for a boolean or relational condition. If the condition is true, the code inside the brackets will execute. As long as the condition remains true, the code inside the brackets will continue to execute in a loop. Here is an example:

```
bool loop = true;
int count = 1;
```

```
while(loop == true)
{
   Print(count);                    // Output: 1, 2, 3, 4, 5
   if(count == 5) loop = false;
   count++;
}
```

We start by declaring a boolean variable named `loop` to use as the loop condition. We also declare an integer variable named `count` and initialize it to 1. The `while` operator checks to see if the variable `loop == true`. If so, the code inside the brackets is run.

We print the value of the `count` variable to the log. Next we check to see if `count == 5`. If so, we set `loop = false`. Finally we increment `count` by 1, and check the loop condition again. The result of this loop is that the numbers 1 - 5 are printed to the log.

When `count == 5`, the `loop` variable is set to `false`, and the condition for the `while` operator is no longer true. Thus, the loop stops executing and control passes to the expression following the closing bracket. Let's look at a second example, using a relational condition:

```
int count = 1;

while(count <= 5)
{
   Print(count);             // Output: 1, 2, 3, 4, 5
   count++;
}
```

This code produces the same result as the loop above. In this example, the `count` variable is used as the loop condition. If `count` is less than or equal to 5, the loop will execute. On each execution of the loop, `count` will be incremented by 1 and the result printed to the log. Once `count` is greater than 5, the loop will exit.

The `while` loop condition is checked at the beginning of the loop. If the condition is false, the loop is never run. If you need to check the condition at the end of the loop, or you need the loop to run at least once, then use the `do-while` loop.

The do-while Operators

The `do-while` operators check the loop condition at the end of the loop, instead of the beginning. This means that the loop will always run at least once. The `do` operator is new to MQL4. Here's an example:

```
int count = 1;
```

```
do
{
   Print(count);        // Output: 1, 2, 3, 4, 5
   count++;
}
while(count < 5)
```

Again, this is identical to the previous `while` loop example, except in this case, the value of `count` will be printed to the log at least once. For example, if `count == 1`, the numbers 1-5 will be printed to the log. If `count == 6`, the number 6 will be printed to the log.

In both the `while` and `do-while` loops, the condition to halt loop execution must occur sometime during the loop, or independently of the loop (such as an external event). If the condition to stop the loop does not occur, you'll end up with an infinite loop and your program will freeze. Here's an example:

```
bool loop = true;
int count;

do
{
   Print(count);
   count++;
}
while(loop == true)
```

This example will loop infinitely because the variable `loop` is always equal to `true`.

If you don't know how many times a loop will need to execute, or if you need to use a boolean condition as the loop condition, then use a `while` or `do-while` loop. If you know how many times a loop needs to execute, or if you need more advanced iteration, then use a `for` loop instead.

The for Operator

If you are using an integer variable to iterate through a loop (such as the `count` variable in the previous examples), the `for` loop is a better choice. The `for` operator contains three expressions separated by semicolons:

- The first expression is a variable(s) to initialize at the start of the loop. This variable is generally used to iterate through the loop.
- The second expression is the loop condition. This is generally a relational expression. When this expression is true, the loop executes. When it is false, the loop exits.

- The third expression is executed at the end of each loop. This is generally a mathematical expression to increment the iterator variable.

Here's an example of a `for` loop:

```
for(int count = 1; count <= 5; count++)
{
   Print(count);
}
```

The first expression in the `for` operator, `int count = 1`, declares an integer variable named `count` and initializes it to 1. The second expression, `count <= 5`, is the loop condition. If count is less than or equal to 5, the loop will execute. The third expression, `count++`, is executed at the end of each loop. This expression increments the `count` variable by 1. Note that there are semicolons after the first and second expressions, but not after the third expression.

Like the previous examples, this code prints the numbers 1-5 to the log. If you compare the code above to the `while` loop example on the previous page, the `for` loop requires fewer lines of code. Just about anything you can do with a `while` loop can also be done with a `for` loop.

You can omit any of the three expressions in the `for` loop, but the semicolons separating them must remain. If you omit the second expression, the loop is considered to be constantly true, and thus becomes an infinite loop.

You can declare multiple variables in the first expression of a `for` loop, as well as calculate multiple expressions in the third expression of a `for` loop. The additional expressions must be separated by commas. For example:

```
for(int a = 1, b = 2; a <= 5; a++, b += 2)
{
   Print("a=",a," b=",b);                    // Output: "a=1 b=2", "a=2 b=4", etc...
}
```

We declare two integer variables, `a` and `b`, and initialize them with the values of 1 and 2 respectively. The loop will execute if `a` is less than or equal to 5. On each iteration of the loop, `a` is incremented by 1, while `b` is incremented by 2. The values of `a` and `b` are printed to the log on each iteration.

The break Operator

Sometimes you need to exit a loop before it has finished iterating, or when a certain condition is met. The `break` operator immediately exits the nearest `while`, `do-while` or `for` loop. It also exits the `switch` operator, as explained on page 46.

Generally, the `break` operator is used to exit a loop when a certain condition is met. For example:

```
for(int count = 1; count <= 5; count++)
{
   if(count == 3) break;
   Print(count);              // Output: 1, 2
}
```

In this example, when `count == 3`, the `break` operator is called and the `for` loop is exited.

The continue Operator

The `continue` operator works similar to the `break` operator. Instead of exiting the loop entirely, the continue operator exits the current iteration of the loop and skips to the next iteration.

```
int count = 1;

while(count <= 5)
{
   if(count == 3) continue;
   Print(count);              // Output: 1, 2, 4, 5
   count++
}
```

This example will print the value of `count` to the log for every value except for 3.

Chapter 5 - Functions

Functions

A *function* is a block of code that performs a specific task, such as placing an order or adjusting the stop loss of a position. We will be creating our own functions to carry out many trading-related activities in this book. In addition, MQL4 has dozens of built-in functions that do everything from retrieving order information to performing complex mathematical operations.

Functions are designed to be flexible and reusable. Whenever you need to perform a specific action, such as placing an order, you call a function to perform that action. The function contains all of the code and logic necessary to perform the task. All you need to do is pass the required parameters to the function, if necessary, and handle any values returned by the function.

For example, when we place an order, we will call a function that specifically places orders. We will pass parameters to the function that instruct it to place an order on the specified symbol, at the specified price with the specified number of lots. Once the function has finished executing, it will return a value such as an order confirmation.

A function declaration consists of a return type, an identifier, and an optional list of parameters. Here's an example of a function declaration:

```
double BuyStopLoss(string pSymbol, int pStopPoints, double pOpenPrice = 0)
{
   // Function body
}
```

The name of our function is `BuyStopLoss()`. This function will calculate the stop loss for a buy order. The return type is `double`, which means that this function will calculate a value of type `double`, and return that value to our program.

This function has three parameters: `pSymbol`, `pStopPoints` and `pOpenPrice`. In this book, we will preface all function parameter identifiers with a lower case "p". The parameters are separated by commas. Each parameter must have a type and an identifier. A parameter can also have a default value. We'll discuss default values in more detail shortly.

The first two parameters are required – which means they must be passed to the function when the function is called. The first parameter, `pSymbol`, is a string value representing the symbol of the instrument. The second parameters, `pStopPoints`, is an integer value representing the stop loss value in points. The third parameter,

pOpenPrice, is a double value representing the order opening price. We've specified a default value of zero. We'll discuss default values shortly.

Here is the function in its entirety. This function will be placed somewhere on the global scope of our program – which means that it can't be inside another function. It can be placed in an include file, or even inside another program that is included in our program:

```
double BuyStopLoss(string pSymbol,int pStopPoints, double pOpenPrice = 0)
{
   if(pStopPoints <= 0) return(0);

   double openPrice;
   if(pOpenPrice > 0) openPrice = pOpenPrice;
   else openPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);

   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double stopLoss = openPrice - (pStopPoints * point);

   long digits = SymbolInfoInteger(pSymbol,SYMBOL_DIGITS);
   stopLoss = NormalizeDouble(stopLoss,(int)digits);

   return(stopLoss);
}
```

This function will calculate a stop loss price for a buy market order. Here's an example of how we would call this function in our program:

```
// Input variables
input int StopLoss = 500;

// OnTick() event handler
double useStopLoss = BuyStopLoss(_Symbol,StopLoss);
```

The input variable StopLoss is an integer that contains the stop loss value in points. This will be located at the beginning of our program, and will be set by the user. Later in our program, inside the OnTick() event handler, we call the BuyStopLoss() function, and pass the current chart symbol (_Symbol) and the StopLoss input variable as the function parameters. The BuyStopLoss() function calculates the stop loss for a buy market order and stores the return value in the useStopLoss variable. This variable would then be used to add a stop loss to an order.

Default Values

A function parameter can be assigned a default value when the function is declared. If a parameter has a default value, it must be placed at the end of the parameter list. For example, the `pOpenPrice` parameter in our `BuyStopLoss()` function has a default value assigned:

```
double BuyStopLoss(string pSymbol, int pStopPoints, double pOpenPrice = 0)
{
   //...
}
```

The `pOpenPrice` parameter is assigned a default value of zero. If you are using the default value when calling the function, the parameter can be omitted:

```
BuyStopLoss(_Symbol,StopLoss);
```

In the above example, the default value of 0 will be used for the `pOpenPrice` parameter.

You can have multiple parameters with default values, but they must all be at the end of the parameter list. If a function has multiple parameters with default values, and you are passing a value to a parameter with a default value, then any parameters before it must have values passed as well. Here's an example:

```
int MyFunction(string pSymbol, int pDefault1 = 0, int pDefault2 = 0)
{
   // Function body
}
```

`MyFunction()` has two parameters with default values: `pDefault1` and `pDefault2`. If you need to pass a non-default value to `pDefault2`, but not to `pDefault1`, then a value must be passed to `pDefault1` as well:

```
int nonDefault = 5;
MyFunction(_Symbol,0,nonDefault);
```

The parameter `pDefault1` is passed the default value of 0, while `pDefault2` is passed a value of 5. You cannot skip parameters when calling a function, unless all remaining parameters are using their default values. Of course, if you don't need to pass a value to `pDefault1` and `pDefault2`, or just `pDefault2`, then you can omit it from the function call:

```
int point = 5;
MyFunction(_Symbol,point);
```

In this example, `pDefault1` is passed a value of 5, but `pDefault2` uses its default value of 0. If you are using the default value for `pDefault1` as well, the only parameter that needs to be specified is `pSymbol`.

In summary, any function parameter(s) with a default value must be at the end of the parameter list. You cannot skip parameters when calling the function, so if a parameter with a default value is passed a different value when calling the function, then any parameters before it must have a value passed to it as well.

The return Operator

Any function that returns a value must have at least one `return` operator. The `return` operator contains the variable or expression to return to the program. The type of the expression must match the return type of the function. Generally, the `return` operator is the last line in your function, although you may have several `return` operators in your function, depending on your requirements.

The return type of a function can be of any type, including structures and enumerations. You cannot return an array from a function, although you can return an element from an array. If you need a function to return an array, you can pass an array to a function by reference. We'll discuss passing by reference shortly.

Here is our `BuyStopLoss()` function again. This function returns a value of the `double` type. Notice that there are two `return` operators. At the beginning of the function, if `pStopPoints` is less than or equal to zero, we exit the function early and return a value of zero. At the end of the function, we return the value of `stopLoss` to the program:

```
double BuyStopLoss(string pSymbol,int pStopPoints, double pOpenPrice = 0)
{
   if(pStopPoints <= 0) return(0);

   double openPrice;
   if(pOpenPrice > 0) openPrice = pOpenPrice;
   else openPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);

   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double stopLoss = openPrice - (pStopPoints * point);

   long digits = SymbolInfoInteger(pSymbol,SYMBOL_DIGITS);
   stopLoss = NormalizeDouble(stopLoss,(int)digits);

   return(stopLoss);
}
```

Note that `return(stopLoss)` and `return stopLoss` are both valid syntax for the `return` operator.

The void Type

Not every function needs to return a value. There is a special type called `void`, which specifies a function that does not return a value. A `void` function can accept parameters, but does not need to have a `return` operator. Here is an example of a function of `void` type with no parameters:

```
void TradeEmail()
{
   string subject = "Trade placed";
   string text = "A trade was placed on " + _Symbol;

   SendMail(subject,text);
}
```

This function will send an email using the mail settings specified in the *Tools* menu > *Settings* > *Email* tab. Note that there is no `return` operator at the end of the function because the function is of `void` type. You can use a `return` operator if you need to exit a function early, though.

Passing Parameters by Reference

Normally, when you pass parameters to a function, the parameters are passed by *value*. This means that the value of the parameter is passed to the function, and the original variable remains unchanged.

You can also pass parameters by *reference*. When you pass a parameter by reference, you are passing the actual memory location of the variable to the function. Thus, any changes made to a variable inside a function will be reflected in the original variable. This is useful when you need a function to modify an array or an object. Passing by reference can be used to return multiple values from a function.

Here's an example of passing by reference using an array. In this example, we'll pass a dynamic array to a function by reference. We specify that a parameter is being passed by reference by prefixing the identifier with an ampersand (&):

```
void FillArray(int &array[])
{
   ArrayResize(array,3);
   array[0] = 1;
   array[1] = 2;
   array[2] = 3;
}
```

The sole parameter of the `FillArray()` function, `&array[]`, is passed by reference. In this example, we resize the dynamic array `&array[]` to three elements, and assign the values 1 – 3 to each element of the array. Here is how we would call this function from our program:

```
int fill[];
FillArray(fill);
Print(fill[0]);      // Output: 1
```

We declare an empty dynamic array named `fill[]`, and pass it as the parameter to the `FillArray()` function. The `fill[]` array now contains the values that were modified inside the `FillArray()` function.

Overloading Functions

Sometimes you may need to create multiple functions that perform essentially the same task. Each of these functions will have different input parameters, but the end result is the same. In previous versions of MQL4, it would be necessary to give these functions different identifiers. MQL5 introduced *function overloading*, which allows you to have multiple functions with the same name. This feature has now been added to MQL4.

Each identically-named function must have different parameters, either in number or in type. Let's demonstrate by using two trailing stop functions that we'll create later in this book. Both functions have the same name, and do basically the same thing. The difference is that the first function has an integer parameter named `pTrailPoints`, and the second has a double parameter named `pTrailPrice`:

```
bool TrailingStop(string pSymbol, int pTrailPoints, int pMinProfit = 0, int pStep = 10);
bool TrailingStop(string pSymbol, double pTrailPrice, int pMinProfit = 0, int pStep = 10);
```

The `int` parameter in the first function, `pTrailPoints`, accepts a trailing stop value in points. This is used to calculate a trailing stop price, relative to the current Bid or Ask price. The `double` parameter in the second function, `pTrailPrice`, accepts a price to be used as the trailing stop price.

By having two identically-named functions with different parameters, we have some flexibility as to how to administer the trailing stop, depending on the trading system. Since both functions share the same name, the programmer does not have to remember two (or more) different function names. In the end, this simply makes life easier for the programmer.

The compiler will know which function to use based on its unique parameter signature. The first function has a `string` parameter, followed by three `int` parameters. The second has a `string` parameter, followed by a `double` parameter and two `int` parameters. Here's how we would call the first variant of the function:

```
// Input variables
input int TrailingPoints = 500;

// OnTick() event handler
TrailingStop(_Symbol,TrailingPoints);
```

An `int` input variable named `TrailingPoints` allows the user to set a trailing stop in points. This value is used as the second parameter in the `TrailingStop()` function call. Because the `TrailingPoints` variable is of type `int`, the compiler knows to use the first variant of the function. Since we are using the default values for the `pMinProfit` and `pStep` parameters, we have omitted them from the function call.

And here's how we would call the second variant of the function:

```
// Input variables
input int TrailingPoints = 500;

// OnTick() event handler
double trailingPrice = SymbolInfoDouble(_Symbol,SYMBOL_ASK) - (TrailingPoints * _Point);
TrailingStop(_Symbol,trailingPrice);
```

The local `double` variable `trailingPrice` will contain a price to use as the trailing stop. Since the second parameter of the `TrailingStop()` function call is of type `double`, the compiler knows to use the second variant of the function.

Chapter 6 - Object-oriented Programming

One of the most exciting new features in MQL4 is the addition of object-oriented programming. *Object-oriented programming* (OOP for short) encourages code reuse and hides unnecessary implementation details from the user. This allows a much more flexible and compact style of programming.

The concepts of object-oriented programming are abstract in nature and often confounded by technical jargon. They can be difficult for the new programmer to grasp. But once you learn the fundamentals of OOP, you'll find them to be incredibly useful.

Object-oriented programming is based around the concepts of classes and objects. A *class* is a collection of variables and functions that perform a set of related tasks. The variables and functions contained inside a class are referred to as the *members* of a class.

A class is like a blueprint for an object. Take a car, for example. A car has a steering wheel, a gear shifter, a turn signal, headlights and so on. A class for a car object would contain all of the variables describing the car's state (speed, gear, whether the headlights are on or off), and all of the functions to perform a specific task (accelerating or decelerating, shifting gears, turning the headlights on and off, etc.)

An *object* is created using the class as a template. The class describes the car, while the object is the car itself. Each object has a unique name, similar to how each car has a unique vehicle identification number. You can create as many objects as necessary, just like a manufacturer can build many different cars of the same model. The variables of an object are distinct from the variables of other objects, just like how different cars are going different speeds on the highway.

For example, in an expert advisor program you may have several indicators. For a moving average cross, you will have at least two moving average indicators. Each moving average will have a different period setting, and may have different calculation mode and price settings.

A moving average indicator can be represented by a class. The moving average indicator class contains all of the variables and functions necessary to create the indicator and to retrieve the current indicator value during program execution. Using this class, we create one or more objects, each of which will have their own identifier, variables and return values.

We don't have to worry about how to create the moving average indicator. All of these details are handled in the class implementation. All we need to do is create an object with the appropriate parameters, and use the class functions to return the prices that we need.

Classes

Classes are declared on the global scope, just like functions. A class can be placed in your program file or inside an include file. A class declaration uses the `class` keyword, followed by a unique identifier. The members of the class are placed inside the brackets, sorted by *access keywords*. The closing bracket of a class declaration is terminated with a semicolon.

Here's an example of a class declaration for an indicator object:

```
class CIndicator
{
   protected:
      string _symbol;
      int _timeFrame;
      int _digits;

      void Init(string pSymbol, int pTimeFrame);
};
```

The name of the class is `CIndicator`. It has four protected members, including the `_symbol`, `_timeFrame` and `_digits` variables, as well as the `Init()` function. Notice that every function and variable declaration inside the class declaration is terminated with a semicolon. The closing bracket of the class declaration itself is terminated with a semicolon as well.

We'll discuss the `CIndicator` class in more detail in Chapter 17, so don't worry if you don't understand how it works just yet. In this chapter, we will be using the `CIndicator` class as an example to explain the concepts of object-oriented programming.

Access Modifiers

The labels `public`, `private` and `protected` are *access keywords*. They determine whether a variable or function is available for use outside of a class. Here are the descriptions of the access keywords:

- **Public** members of a class are available for use outside of the class. This is the method by which the program interacts with an object. Public members are generally functions that perform important tasks. Public functions can access and modify the private and protected members of a class.
- **Private** members of a class are only available for use by functions inside the class. A private member cannot be accessed outside the class. Classes that are derived from this class will not inherit these members. (We'll discuss inheritance shortly.) Private members are generally internal functions and variables that are accessed by public members of a class.

- **Protected** members of a class are essentially private members that will be inherited by a derived class. Use the `protected` keyword unless you're certain that you won't be deriving any classes from the current class.

This concept of hiding class members from the rest of the program is an OOP concept referred to as *encapsulation*. By hiding class members, we ensure that they won't be used or modified unnecessarily.

The `CIndicator` class is meant to be used as a parent class for other indicator classes, such as a moving average indicator class. We've created this class to implement features that every indicator will use. For example, every indicator takes a symbol and timeframe parameter, so we've added the `_symbol` and `_timeFrame` member variables. Both of these variables are `protected`, which means they can't be accessed outside of our class, and can only be accessed by public members of the class, or from derived classes.

Derived Classes

One of the most useful features of OOP is the concept of *inheritance*. In OOP, you can create a class using another class as a template. The new class inherits all of the functions and variables of the parent class (except for those that use the `private` access keyword). You can then extend upon that class by adding new functions and variables.

This is exactly what we'll do with our `CIndicator` class. Remember that the `CIndicator` class is meant to be a parent class for other indicator classes. The specifics of implementing a particular indicator are handled in the derived class, while the basic variables and functions are already defined in the parent class.

Here's an example of a derived class for a moving average indicator:

```
class CiMA : CIndicator
{
   private:
      int _maPeriod;
      int _maShift;
      int _maMethod;
      int _appliedPrice;

   public:
      CiMA(string pSymbol, int pTimeFrame, int pMaPeriod, int pMaShift, int pMaMethod,
         int pAppliedPrice);
      double Main(int pShift = 0);
};
```

The name of our derived class is `CiMA`. Notice the colon (`:`), followed by `CIndicator` in the class declaration. This specifies that the `CiMA` class is derived from the `CIndicator` class. All of the public and protected members of the `CIndicator` class are now part of the `CiMA` class. The `CiMA` class contains four private

variables, which will hold the indicator settings. It also has two public functions – a class constructor, as well as a `Main()` function to return the moving average price for a specified bar.

Constructors

When an object is created from a class, a function called the *constructor* is executed automatically. The constructor is used to initialize the variables inside our object. If no constructor is explicitly defined, then the compiler creates a default constructor to initialize the variables. This default constructor is not visible to the programmer.

In our `CiMA` function, we have declared our own constructor, also called `CiMA()`. The name of a constructor must match that of the class identifier. It is not necessary to specify a return type for a default constructor, as the type is always `void`. The access level of a constructor is always `public`.

Here is the `CiMA()` constructor declaration inside the `CiMA` class declaration again:

```
class CiMA : CIndicator
{
   // ...

   public:
      CiMA(string pSymbol, int pTimeFrame, int pMaPeriod, int pMaShift, int pMaMethod,
         int pAppliedPrice);
};
```

A constructor can take input parameters if necessary. Our `CiMA` class constructor has six input parameters, although many class constructors have none. Below is the body of our `CiMA` class constructor. The purpose of the `CiMA` class constructor is to set the parameters for calculating the moving average cross indicator:

```
void CiMA::CiMA(string pSymbol,int pTimeFrame,int pMaPeriod,int pMaShift,int pMaMethod,
   int pAppliedPrice)
{
   Init(pSymbol, pTimeFrame);

   _maPeriod = pMaPeriod;
   _maMethod = pMaMethod;
   _maShift = pMaShift;
   _appliedPrice = pAppliedPrice;
}
```

The `CiMA()` class constructor calls the `Init()` function from the `CIndicator` class. Then it assigns the values from the input parameters of the constructor to the private variables of the `CiMA` class. It is good practice to

keep the variables in a class `private` or `protected`, and to use public functions to change or access those values.

Remember, you do not need to explicitly declare a constructor if you do not need one. If there are actions you want carried out automatically upon the creation of an object, then create a constructor to do this.

There are more advanced things you can do with constructors, such as parametric constructors and initialization lists. There is also the destructor, which is called upon destruction of an object. Since we won't be using those features in this book, it will be up to the reader to learn more. You can learn about constructors and destructors in the *MQL4 Reference* under *Language Basics* > *Data Types* > *Structures and Classes*.

Virtual Functions

Sometimes, you will need to change the way a function operates in a derived class. Or you may want to define a function in a parent class, but take care of the implementation details in the derived class. You can accomplish this by using *virtual functions*.

Let's use the example of a car again: A car class can have a function to change gears. However, the process of changing gears in a car with a manual transmission is different than changing gears with an automatic transmission. Therefore, we would declare a virtual gear changing function in our parent class, and then write the actual function in our derived classes.

Here's what a car class with a gear-shifting function would look like in code:

```
class Car
{
   public:
      virtual int ShiftGears(int gear) { return(gear); }
};
```

The name of our class is `Car`. We've declared a single function – a virtual function named `ShiftGears()`. We've added an empty function body to the `ShiftGears()` declaration, containing only a single `return` operator. The `ShiftGears()` function is declared with the `virtual` keyword. This means that the function will be defined in the derived classes.

Let's create a derived class for a manual transmission car:

```
class ManualCar : Car
{
   public:
      int ShiftGears(int gear);
};
```

This class is named `ManualCar`, and is for a manual transmission vehicle. Here is where we define the `ShiftGears()` function. The function is declared with the same type and parameters as the function in the `Car` class. Notice that we do not use the `virtual` keyword here. The body of the function is defined elsewhere, and contains the logic for shifting gears using a manual transmission. Other classes derived from the `Car` class would define `ShiftGears()` in a similar manner.

If a derived class has a function with the same name as a function in the parent class, the function in the derived class will override the function in the parent class. This process of redefining functions in derived classes is an OOP concept known as *polymorphism*.

The classes in our MQL4 include files do not contain any virtual functions, so we will not address them further. For more information on virtual functions, consult the MQL4 Reference under *Language Basics* > *Object-Oriented Programming* > *Virtual Functions*.

Objects

Now that we've created a class for a moving average indicator, let's create an object. You create a class object the same way you create a variable, enumeration or structure object: The class name is used as the type, and the object is given a unique identifier:

```
CiMA objMa(_Symbol,_Period,MaPeriod,0,MaMethod,MaPrice);
```

This creates an object named `objMa`, based on the class `CiMA`. We have passed the appropriate input values to the class constructor. When an object is created, the constructor for that object is executed automatically. Here is the constructor for the `CiMA` class again for reference:

```
void CiMA::CiMA(string pSymbol,int pTimeFrame,int pMaPeriod,int pMaShift,int pMaMethod,
   int pAppliedPrice)
{
   Init(pSymbol, pTimeFrame);

   _maPeriod = pMaPeriod;
   _maMethod = pMaMethod;
   _maShift = pMaShift;
   _appliedPrice = pAppliedPrice;
}
```

Once the object has been created, we can call any of the public functions in the class. The `CiMA` class has one public function, `Main()`, which returns the moving average price for the specified bar:

```
double ma = objMa.Main(0);
```

We call the `Main()` function of the `objMa` object using dot notation. The example above returns the moving average price for the current bar, and saves the value to the `ma` variable.

You can create as many objects as necessary for your program. If you need a second moving average indicator, then declare it using a different unique identifier, and access the public members of the object as shown above.

By creating classes to perform common tasks, you save time and reduce errors, as well as reducing the amount of code in your program. If you don't understand object-oriented programming just yet, don't worry. The topics discussed above should become clearer as you work your way through the book. Remember that OOP in MQL4 is completely optional, and is not necessary for programming expert advisors or indicators.

We will spend much of this book creating classes and objects to carry out common trading tasks. Even if you choose not to create your own classes, you should understand how to create an object and access it's member functions and variables.

Chapter 7 - The Structure of an MQL4 Program

Before we start developing MQL4 programs, let's take a minute to address the structure of an MQL4 program. All MQL4 programs share the same basic structure. At the top of the file will be the preprocessor directives. Next are the input and global variables. Finally, the functions, classes and event handlers of the program are defined.

Preprocessor Directives

The *preprocessor directives* are used to set program properties, define constants, include files and import functions. Preprocessor directives are typically declared at the very top of the program file. Let's start with the `#property` preprocessor directive, which you'll see in nearly every MQL4 program.

#property Directive

The `#property` directive defines properties for the program, such as descriptive information about the program, and whether the program is an indicator, a script or a library. We'll discuss properties for indicators, scripts and libraries in Chapter 25.

When you create a program using the *MQL4 Wizard*, the `copyright`, `link`, `version` and `strict` properties will be inserted automatically. You can also add the `description` property manually. These will be displayed on the *Common* tab in the expert advisor *Properties* dialog. This is useful if you decide to distribute your program. The `icon` property can be used to replace the file icon for your compiled expert advisor program.

The `#property` directives will be placed at the very top of your program. They must be defined in your main program file, as any property directives in include files will be ignored. Here's an example of the descriptive `#property` directives:

```
#property copyright "Andrew R. Young"
#property link "http://www.expertadvisorbook.com"
#property version "1.02"
#property description "Dual moving average cross with trailing stop"
#property strict
```

And here's how these `#property` directives above will display in the *About* tab of the expert advisor *Properties* window:

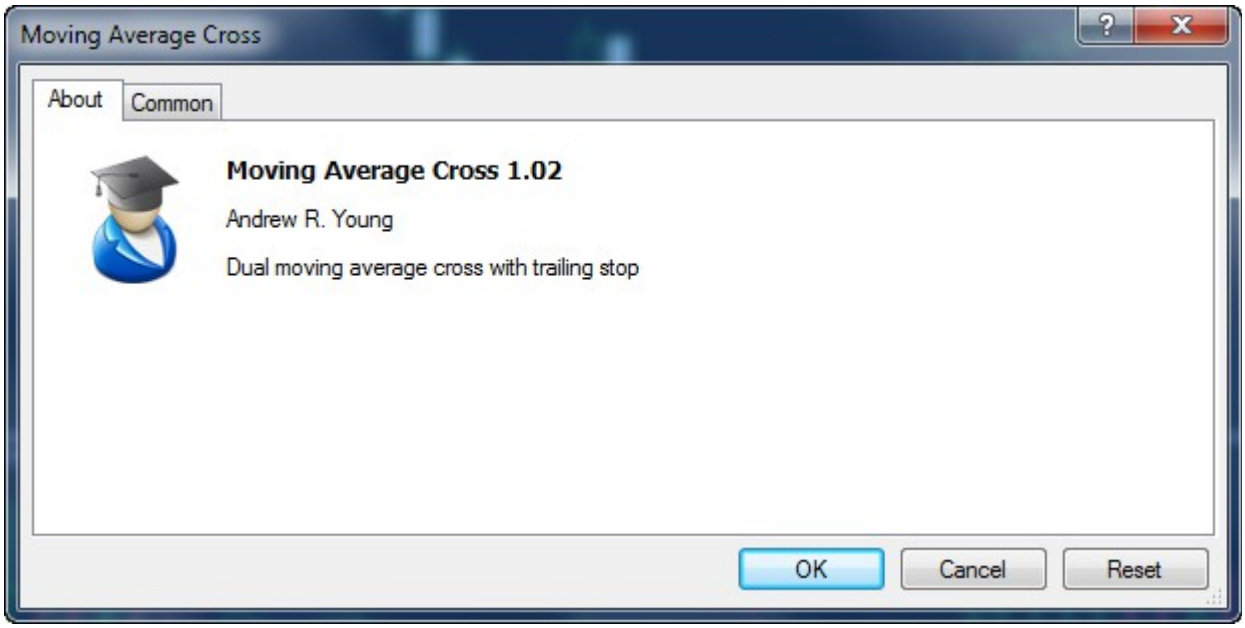


Fig. 7.1 – The *About* tab of the expert advisor *Properties* window, displaying the `#property` directives defined in the source code file.

The `copyright` property above ("Andrew R. Young") doubles as a hyperlink. Placing the mouse over it and clicking will take the user to the website defined in the `link` property.

The #property strict Directive

The newest versions of MQL4 include the `strict` property directive, which controls various elements of program compilation. The most important consideration when using the `strict` property directive is variable scope. When the `strict` property is present, variables are only local to the block in which they are declared. Additionally, any functions have specify a return type other than `void` must have a `return` statement that returns a value of that type.

If you are compiling a program written for a previous version of MQL4, you can omit the `strict` property directive. For new programs, it is advised to leave the `strict` property directive in the program.

#define Directive

The `#define` directive is used to define constants for use throughout the program. We addressed constants earlier on page 19. To summarize, the `#define` directive specifies an identifier with a constant value. The convention is to use all capital letters for the identifier name. Here are some examples of constants using the `#define` directive:

```
#define PI 3.14159265
#define MAX_BARS 100
#define COMPANY_NAME "Easy Expert Forex"
```

To use a constant, you substitute the identifier name for the constant value. For example, if you wanted to use the value of Pi in your program, the identifier `PI` would be interpreted as 3.14159265 by the compiler:

```
double diameter = 5;
double circumference = PI * diameter;
Print(circumference);                        // Output: 15.70796325
```

The example above calculates the circumference of a circle by multiplying the value of Pi by the diameter of a circle.

#include Directive

The `#include` directive specifies an include file to be included in the program. An include file contains variables, function and classes to be used in your program. There are two variations of the `#include` directive:

```
#include <Trade.mqh>
#include "Trade.mqh"
```

The first variant of the `#include` directive encloses the include file name in angle brackets (<>). This indicates that the compiler will look for the include file in the default include directory, which is the `\MQL4\Include` subfolder of your MetaTrader 4 data folder.

If the file is located in a subfolder of `\MQL4\Include`, then you must add the subfolder name. For example, if the `Trade.mqh` file is located in the `\MQL4\Include\Mql4Book` folder, then we will need to include the file like this:

```
#include <Mql4Book\Trade.mqh>
```

The compiler does not care whether you use a forward slash or a back slash in the file path.

The second variant of the `#include` directive encloses the include file name in double quotes ("). This tells the compiler to look for the include file in the same directory as the current file. If for some reason you've stored the include file in the same directory as your program, then use double quotes in your `#include` directive.

There is an additional preprocessor directive, the `#import` directive, which is used to import functions from libraries and DLLs. We'll address the usage of the `#import` directive in Chapter 25.

Input and Global Variables

After the preprocessor directives, the next section of your MQL4 program will be the input and global variable declarations. It doesn't necessarily matter which comes first, but the convention is to put the input variable declarations first. As you may recall, input variables are the user-adjustable settings for your program.

Any global variables that you're using must be declared outside of any functions or event handlers. The convention is to put them at the top of the file, after the input variables. This ensures that they won't be called by any functions before they have been declared.

Classes and Functions

Custom classes and functions can be defined anywhere in your program, especially in include files that are included using the `#include` directive. Generally, any classes or functions that are present in the main program file can go before or after the event handlers, but should be placed below any input or global variables in the file.

Event Handlers

An *event handler* is a function that is executed whenever a certain event occurs. Event handlers are the method by which an MQL4 program runs. For example, when an incoming price quote is received by an expert advisor, the `OnTick()` event handler executes. The `OnTick()` event handler contains code that runs every time a price change occurs.

Each program type has its own event handlers. Expert advisors and indicators use the `OnInit()` event handler, which runs once at the start of the program. Scripts use the `OnStart()` event handler. Indicators use the `OnCalculate()` event handler to execute indicator calculations. We'll go into more detail on event handlers for each program type in the relevant chapters.

An Example Program

Here's a brief example showing all of the elements described above and how they would fit into an MQL4 program. Not every element will be in every program – for example, an `#include` directive is not needed if you're not including functions and variables from an external file. The `#property` directives are usually optional. Most programs will have input variables, but global variables are optional. And you may or may not need to create your own classes or functions.

The event handlers will vary depending on the program type. This example shows an expert advisor program with the `OnInit()` and `OnTick()` event handlers:

```
// Preprocessor directives
#property copyright "Andrew R. Young"
#property link "http://www.expertadvisorbook.com"
#property description "An example of MQL4 program structure"

#define PI 3.14159265

// Input variables
input double Radius = 1.5;

// Global variables
double gRadiusSq;

// Event handlers
int OnInit()
{
   gRadiusSq = MathPow(Radius,2);
   return(0);
}

void OnTick()
{
   double area = CalcArea();
   Print("The area of a circle with a radius of "+Radius+" is "+area);
}

// Functions
double CalcArea()
{
   double result = PI * gRadiusSq;
   return result;
}
```

Above is a simple program that calculates the area of a circle, given the radius. The `#property` directives come first, with some descriptive information about the program. A `#define` directive defines a constant for Pi. An input variable named `Radius` allows the user to enter the radius of a circle. Finally, a global variable named `gRadiusSq` is available to all functions of the program.

This program has two event handlers. The `OnInit()` event handler runs once at the start of the program. The square of the `Radius` variable is calculated and stored in the global variable `gRadiusSq`. After the `OnInit()` event handler has run, the `OnTick()` event handler will run on each incoming price change.

The `OnTick()` event handler calls the function `CalcArea()`, which is defined at the bottom of our program. The function calculates the area of a circle, and returns the result to the `OnTick()` function. The `Print()` function will print the following string to the log, assuming that the default value for `Radius` is used:

```
The area of a circle with a radius of 1.5 is 7.0685834625
```

That's it. We haven't added any trading functions to this program, as we simply want to demonstrate the basic structure of an MQL4 program. In the next chapter, we'll begin creating expert advisors.

Include Files

An include file contains code that can be re-used in multiple programs. Throughout this book, we will create many new functions and classes that will be contained in various include files. An include file can contain classes, functions, variables, preprocessor directives and input variables. It cannot contain event handlers.

If you have downloaded and installed the source code from the book's website, the include files are located in `\MQL4\Include\Mql4Book`. MetaTrader 4 comes with a large number of include files that have been adapted from MetaTrader 5's standard library. Although these are available for you to use, they are currently undocumented and thus we will not be covering them in this book.

Chapter 8 - Expert Advisor Basics

Expert Advisor Event Handlers

In the last chapter, we discussed the structure of an MQL4 program and introduced the reader to event handlers. Let's discuss the most common event handlers that are used in expert advisor programs:

OnInit()

The `OnInit()` event handler runs when the program is initialized. Normally, the `OnInit()` event handler runs once at the start of a program. If there are any changes in the expert advisor properties, or if the current chart symbol or period is changed, the expert advisor will reinitialize and the `OnInit()` function will run again.

If there are any actions you wish to execute once at the start of the program, place them in the `OnInit()` event handler. The `OnInit()` event handler is not required in your program, but it is recommended. We will use `OnInit()` to initialize certain variables and carry out startup actions.

OnDeinit()

The `OnDeinit()` event handler runs when the program is stopped, or deinitialized. If the expert advisor properties are changed, the current chart symbol or period is changed, or if the program is exited, the `OnDeinit()` event will run.

If there are any actions you wish to execute when the program ends, place them in the `OnDeinit()` event handler. The `OnDeinit()` event handler is not required in your program. One common use of the `OnDeinit()` event handler is to remove objects from the chart when removing an indicator or program that has placed them.

OnTick()

The `OnTick()` event handler runs when a price change is received from the server. Depending on current market activity, price changes can occur several times a minute or even several times a second. Each time a price change occurs, the `OnTick()` event handler will run.

The `OnTick()` event handler is the most important event handler in your program, and is required in your expert advisor. Almost all of your trading system logic will occur in the `OnTick()` event handler, and many of the code examples in this book will go inside the `OnTick()` event handler.

OnTimer()

The `OnTimer()` event handler runs when a timer defined by the `EventSetTimer()` function is activated. This allows you to execute actions at specified intervals. The `OnTimer()` event handler is optional. We'll discuss the usage of the `OnTimer()` event handler in Chapter 18.

Creating An Expert Advisor in MetaEditor

The easiest way to create an expert advisor file in MetaEditor is to use the *MQL4 Wizard*. If you have a template that you want to use to create your expert advisors (like the one included in the source code download), you can open the file and save it under a different name in the `\MQL4\Experts` directory. Unfortunately, the current version of MetaEditor does not allow for the use of predefined template files.

Click the *New* button on the MetaEditor toolbar to open the *MQL4 Wizard*. Ensure that *Expert Advisor (template)* is selected, and click *Next*. The *General properties* dialog allows you to enter a file name and some descriptive information about your program. You can optionally add input variables in the *Parameters* window. The path to the `\MQL4\Experts` folder is already included in the file name. Type the name of your expert advisor, preserving the "`Experts\`" path preceding it:

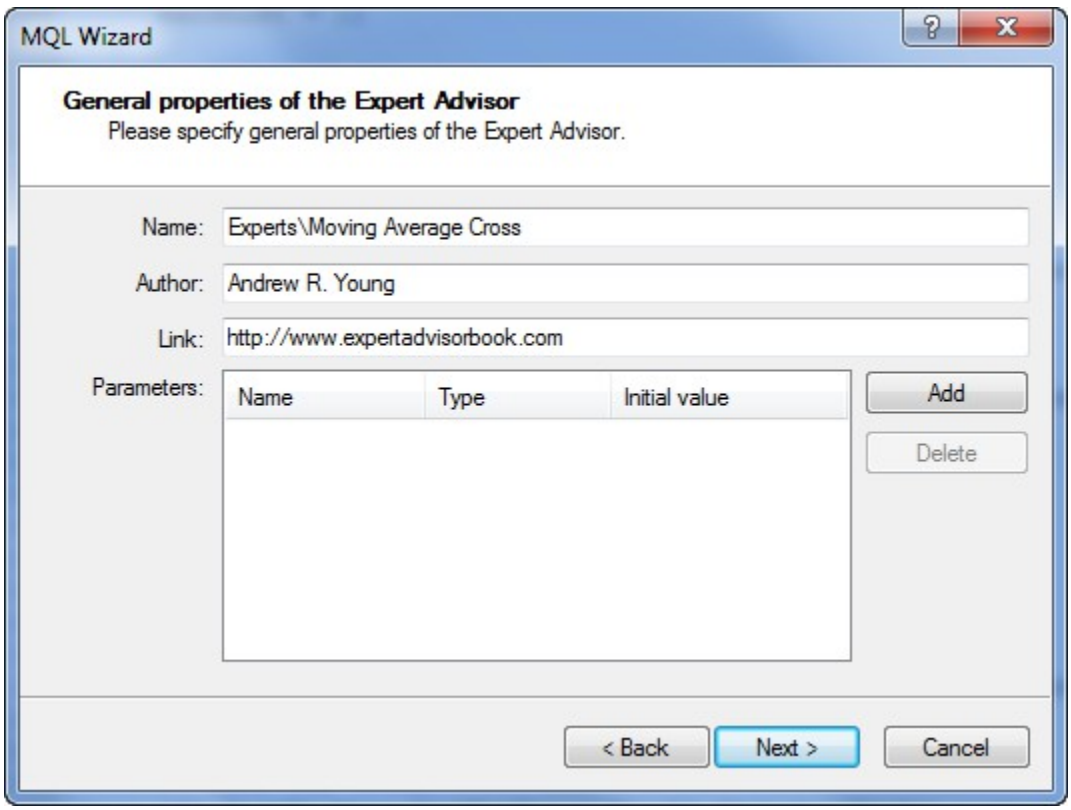


Fig 8.1 – The Expert Advisor properties dialog of the *MQL4 Wizard*.

Click *Next*, and you'll be prompted to insert additional event handlers. We will not be needing additional event handlers at this time. Press *Next* until you get to the last screen. After clicking *Finish*, a new expert advisor MQ4 file is created in the `\MQL4\Experts` folder, and the file is opened in MetaEditor.

Here is what an empty expert advisor template looks like with the basic event handlers added:

```
//+------------------------------------------------------------------+
//|                                        Simple Expert Advisor.MQ4 |
//|                                                     Andrew Young |
//|                                  http://www.easyexpertforex.com |
//+------------------------------------------------------------------+
#property copyright "Andrew Young"
#property link      "http://www.easyexpertforex.com"
#property version   "1.00"
//+------------------------------------------------------------------+
//| Expert initialization function                                   |
//+------------------------------------------------------------------+
int OnInit()
  {
//---

//---
   return(0);
  }
//+------------------------------------------------------------------+
//| Expert deinitialization function                                 |
//+------------------------------------------------------------------+
void OnDeinit(const int reason)
  {
//---

  }
//+------------------------------------------------------------------+
//| Expert tick function                                             |
//+------------------------------------------------------------------+
void OnTick()
  {
//---

  }
//+------------------------------------------------------------------+
```

The expert advisor file generated by the *MQL4 Wizard* includes three descriptive `#property` directives and the `OnInit()`, `OnDeinit()` and `OnTick()` functions by default. If you specified any additional event handlers or input parameters in the *MQL4 Wizard*, they will appear here as well.

Chapter 9 - Order Placement

Bid, Ask & Spread

As a Forex trader, you're already familiar with the Bid and Ask prices. The *Bid* price is what you see on the MetaTrader charts. It is usually what we think of when we think of the "current price." The *Ask* price is generally just a few points above the Bid price. The difference between the Bid and the Ask is the *spread*, which is the broker's commission for placing the order.

The Ask price is where we open buy orders, and close sell orders. The Bid price is where we open sell orders, and close buy orders. You'll need to indicate the correct price when opening or closing a market order.

Order Types

There are two types of orders that can be placed in MetaTrader: market orders and pending orders. Market orders are the most common. A *market order* opens a position immediately at the prevailing Bid or Ask price, while a *pending order* is a request to open an order at a specified price. Pending orders are further classified by *stop* and *limit* order types.

Market Orders

To place a market order, you will need to specify at minimum the order *volume* and an order *opening price*. The opening price is either the current Ask price (for buy orders) or the current Bid price (for sell orders). Some brokers will allow you to specify a stop loss and/or a take profit price, as well as the slippage in points. You can also add a comment to the order.

Execution Type

The process by which an order is executed depends on the trade server's *execution type*. There are three execution types in MetaTrader 4. The execution type is determined by the broker, and is indicated in the *Type* field of the *New Order* dialog box. Most Forex brokers use either market or instant execution.

Instant execution is the classic execution mode familiar to longtime MetaTrader 4 users. The trader specifies the trade type (buy or sell), the symbol to trade, the order volume, a stop loss and take profit price, and the deviation or *slippage* in points. If the difference in points between the current market price and the last quoted price is greater than the slippage, a requote is triggered, and the trader is asked to accept or reject the new price. Otherwise, the trade is placed at the current market price.

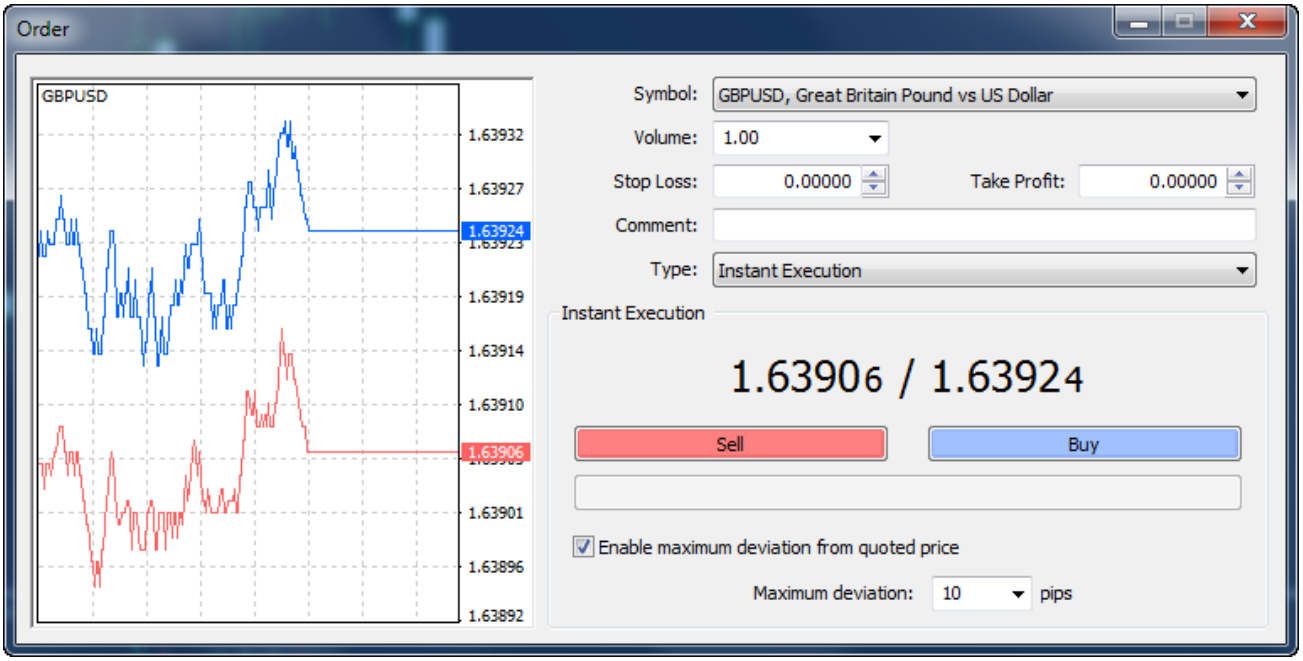


Fig 9.1 – The MetaTrader order window. This broker uses Instant execution.

One advantage of instant execution is that the trader can specify a stop loss and take profit when placing the order, which saves a step when trading manually. In the event of rapidly moving prices and significant price deviation (also referred to as *slippage*), the trader has the option to reject the order and wait for calmer market conditions.

The disadvantage of instant execution is slippage. When the slippage exceeds the specified deviation, the order will not be placed, which creates difficulties when auto trading. Even if the order is placed, the stop loss and take profit price may be a few points off relative to the order execution price.

Most brokers now use market execution, especially the ECN/STP brokers. With the *market execution* type, the trader specifies the order volume only. The trade is executed at the current market price, with no requotes. No stop loss or take profit is placed with the market execution type. The trader will have to modify the position to add a stop loss and take profit after the order is filled.

There is a third execution type called *request executio*n, but it is not common among Forex brokers. To determine your broker's execution type, open the *New Order* dialog in MetaTrader. It will indicate whether the broker uses market or instant execution.

To ensure that our expert advisors are compatible with a wide variety of brokers, we will code our order placement to comply with the market execution type. This means we will place the order at the current market price, and then modify the order to add a stop loss and/or take profit price.

Pending Orders

There are two types of pending orders: stop and limit. A *buy stop* order is placed above the current price, while a *sell stop* order is placed below the current price. The expectation is that the price will eventually rise or fall to that level and continue in that direction, resulting in a profit.

A *limit* order is the opposite of a stop order. A *buy limit* order is placed below the current price, while a *sell limit* order is placed above the current price. The expectation is that the price will rise or fall to that level, triggering the order, and then reversing. Limit orders are not used very often in automated trading.

An *expiration* time can be set for pending orders. If the order is not filled by the expiration time, the order is automatically deleted. Not all brokers support trade expiration.

To place a pending order, you must specify at minimum the order symbol, the trade type (buy stop, sell stop, buy limit or sell limit), the order volume, and an order opening price that is above or below the current market price. You can optionally specify a stop loss and/or take profit price, an expiration time and an order comment.

OrderSend()

The `OrderSend()` function is used to place orders in MQL4. The syntax is as follows:

```
int OrderSend(string Symbol, int Type, double Lots, double Price,
   int Slippage, double StopLoss, double TakeProfit, string Comment = NULL,
   int MagicNumber = 0, datetime Expiration = 0, color Arrow = CLR_NONE);
```

- **Symbol** – A string representing the symbol of the instrument to trade, for example "`GBPUSD`". The predefined `_Symbol` variable is used to represent the symbol of the current chart.
- **Type** – An integer indicating the type of order to place: buy or sell; market, stop or limit. An integer constant can be used for convenience:
  - `OP_BUY` – Buy market order (integer value 0).
  - `OP_SELL` – Sell market order (integer value 1).
  - `OP_BUYSTOP` – Buy stop order (integer value 2).
  - `OP_SELLSTOP` – Sell stop order (integer value 3).
  - `OP_BUYLIMIT` – Buy limit order (integer value 4).
  - `OP_SELLLIMIT` – Sell limit order (integer value 5).
- **Lots** – The order volume. You can specify mini lots (0.1) or micro lots (0.01) if your broker supports it.

- **Price** – The price at which to open the order. For a buy market order, this will be the Ask price. For a sell market order, this will be the Bid price. For pending orders, this will be any valid price that is above or below the current price. Even though market execution brokers do not require an order opening price, we need to specify it anyway.
- **Slippage** – The maximum slippage in points. Use a sufficiently large setting when auto trading. Brokers that do not use slippage will ignore this parameter.
- **StopLoss** – The stop loss price. For a buy order, the stop loss price is below the order opening price, and for a sell order, above. If set to 0, no stop loss will be placed.
- **TakeProfit** – The take profit price. For a buy order, the take profit is above the order opening price, and for a sell order, below. If set to 0, no take profit will be placed.
- **Comment** (optional) – A string that will serve as an order comment. Comments are shown under the *Trade* tab in the *Terminal* window. Order comments can also be used as an identifier for distinguishing different types of orders.
- **MagicNumber** (optional) – An integer that will identify the order as being placed by a specific expert advisor. It is recommended that you use a unique magic number for each expert advisor that you have trading in your terminal, especially those that trade on the same symbol.
- **Expiration** (optional) – The expiration time for pending orders. Not all brokers accept trade expiration times – for these brokers, an error will result if an expiration time is specified.
- **Arrow** (optional) – The color of the arrow that will be drawn on the chart, indicating the opening price and time. If no color is specified, the arrow will not be drawn.

The `OrderSend()` function returns the ticket number of the order that was just placed. If no order was placed, the return value will be `-1`.

We can save the order ticket to a global or static variable for later use. If the order was not placed due to an error condition, we can analyze the error and take appropriate action based on the returned error code.

Placing A Market Order

Here's an example of a buy market order placement. We'll assume that the variables `lotSize`, `Slippage` and `MagicNumber` have already been assigned values that are valid:

```
gBuyTicket = OrderSend(_Symbol,OP_BUY,lotSize,Ask,Slippage,0,0,"Buy order",
   MagicNumber,0,clrGreen);
```

The _Symbol variable returns the current chart symbol. You can also use the Symbol() function, but the variable is easier to read, so we will use it instead. Unless your expert advisor places orders on multiple symbols, we will be using the current chart symbol the majority of the time.

OP_BUY indicates that this is a buy market order. The lotSize variable contains the trade volume. Ask is a predefined variable that stores the most recent Ask quote for the current chart symbol. The Slippage variable is an input variable that holds the slippage value.

No stop loss or take profit price is indicated, as we will modify the order later to place the stops. We've added the generic comment "Buy Order" to this order. MagicNumber is an input variable that is used to indicate this order as being placed by our expert advisor. Since there is no expiration for market orders, the Expiration parameter is 0. Finally, we specify the color constant clrGreen to draw a green arrow on the chart when the order is placed.

The gBuyTicket variable is a global variable that will contain the order ticket number once the order is placed. We will use this value to further modify the order.

Here is an example of a sell market order, using the same parameters as above:

```
gSellTicket = OrderSend(_Symbol,OP_SELL,lotSize,Bid,Slippage,0,0,"Sell order",
   MagicNumber,0,clrRed);
```

We use OP_SELL as the order type, and Bid as the order opening price. "Sell Order" is our order comment, and we use clrRed as the arrow color to draw on the chart when the order is placed. The resulting ticket number will be stored in the gSellTicket variable.

Placing a Pending Stop Order

The difference between placing pending orders and market orders is that the order opening price will be something other than the current market price. Unlike a market order, we will be adding a stop loss and/or take profit price with the order. The stop loss and take profit prices will be calculated relative to the pending order opening price.

In these examples, we will use the variable pendingPrice for our pending order price. It will usually be calculated based on our trading algorithm. For a buy stop order, pendingPrice must be greater than the current Ask price. We'll assume that buyStopLoss and buyTakeProfit have been calculated correctly, relative to pendingPrice. Here's an example of a buy stop order placement:

```
gBuyTicket = OrderSend(_Symbol,OP_BUYSTOP,lotSize,pendingPrice,Slippage,buyStopLoss,
   buyTakeProfit,"Buy Stop Order",MagicNumber,0,clrGreen);
```

Note that we use `OP_BUYSTOP` to indicate a buy stop order, and `pendingPrice` for our order opening price. The `buyStopLoss` and `buyTakeProfit` variables contain the stop loss and take profit prices. No expiration time has been indicated for this order.

For a sell stop order, `pendingPrice` must be less than the current Bid price. In this example, we'll add an order expiration time, using the `expiration` variable. The expiration time must be greater than the current server time. Here's an example of a sell stop order placement:

```
gSellTicket = OrderSend(_Symbol,OP_SELLSTOP,lotSize,pendingPrice,Slippage,sellStopLoss,
   sellTakeProfit,"Sell Stop Order",MagicNumber,expiration,clrRed);
```

Placing a Pending Limit Order

Limit orders are similar to stop orders, except that the pending order price is reversed, relative to the current price and the order type. For buy limit orders, the pending order price must be less than the current Bid price. Here's an example of a buy limit order:

```
gBuyTicket = OrderSend(_Symbol,OP_BUYLIMIT,lotSize,pendingPrice,Slippage,buyStopLoss,
   buyTakeProfit,"Buy Limit Order",MagicNumber,0,clrGreen);
```

Note that we used `OP_BUYLIMIT` to indicate a buy limit order. Otherwise, our parameters are identical to those for stop orders. For a sell limit order, the pending order price must be greater than the current Ask price. Here's an example of a sell limit order:

```
gSellTicket = OrderSend(_Symbol,OP_SELLLIMIT,lotSize,pendingPrice,Slippage,sellStopLoss,
   sellTakeProfit,"Sell Limit Order",MagicNumber,expiration,clrRed);
```

Chapter 10 - Handling, Modifying and Closing Orders

Selecting Orders

Once we've successfully placed an order, we'll need to retrieve some information about the order if we want to modify or close it. We do this using the `OrderSelect()` function. To use `OrderSelect()`, we can either use the ticket number of the order, or we can loop through the pool of open orders and select each one in the order that they were placed.

Once we've selected an order using `OrderSelect()`, we can use a variety of order information functions to return information about the order, including the current stop loss, take profit, order opening price, closing price and more.

OrderSelect()

Here is the syntax for the `OrderSelect()` function:

```
bool OrderSelect(int Index, int Select, int Pool = MODE_TRADES)
```

- **Index** – This is either the ticket number of the order that we want to select, or the position in the order pool. The `Select` parameter will indicate which of these we are using.
- **Select** – A constant indicating whether the `Index` parameter is a ticket number or an order pool position:
  - `SELECT_BY_TICKET` – The value of the `Index` parameter is an order ticket number.
  - `SELECT_BY_POS` – The value of the `Index` parameter is an order pool position.
- **Pool** – An optional constant indicating the order pool: pending/open orders, or closed orders.
  - `MODE_TRADES` – By default, we are examining the pool of currently opened orders.
  - `MODE_HISTORY` – Examines the closed order pool (the order history).

If the `OrderSelect()` function locates the order successfully, the return value will be `true`, otherwise, the return value will be `false`. If you do not check the output of `OrderSelect()` or similar functions, you will get a compiler warning. Therefore, we will always save the output of `OrderSelect()` and other order functions to a variable, even if we do not use that value further.

Here's an example of the `OrderSelect()` function using an order ticket number. The `ticket` variable should contain a valid order ticket:

```
bool selected = OrderSelect(ticket,SELECT_BY_TICKET);
```

If the `OrderSelect()` function has located the order indicated by the ticket number in the `ticket` variable, the `selected` variable will be set to `true`.

After the `OrderSelect()` function has been called, we can use any of the order information functions to retrieve information about that order. A complete listing of functions that can be used with `OrderSelect()` can be found in the *MQL4 Reference* under *Trade Functions*. Here's a list of the most commonly used order information functions:

- `OrderSymbol()` – The symbol that the selected order was placed on.
- `OrderType()` - The order type of the selected order: buy or sell; market, stop or limit. The return value is an integer corresponding to the order type constants on page 81.
- `OrderOpenPrice()` – The opening price of the selected order.
- `OrderLots()` – The lot size of the selected order.
- `OrderStopLoss()` – The stop loss price of the selected order.
- `OrderTakeProfit()` – The take profit price of the selected order.
- `OrderTicket()` – The ticket number of the selected order.
- `OrderMagicNumber()` – The magic number of the selected order.
- `OrderComment()` – The comment that was placed with the order.
- `OrderOpenTime()` – The opening time of the selected order.
- `OrderProfit()` – Returns the profit (in the deposit currency) for the selected order.

We'll need to call `OrderSelect()` and one or more of the above order information functions before closing or modifying an order.

Counting Open Orders

Here's a practical example of how to use the `OrderSelect()` function. It is sometimes necessary to get a count of the orders that are currently opened by the expert advisor. To count the open orders, we will use a `for` loop to loop through the open order pool. For every order that matches the magic number set by the user, we will check the order type and then increment a counter variable:

```
// Current order counts
int buyCount = 0, sellCount = 0;

for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool selected = OrderSelect(order,SELECT_BY_POS);

   if(OrderMagicNumber() == MagicNumber && selected == true)
   {
      if(OrderType() == OP_BUY) buyCount++;
      else if(OrderType() == OP_SELL) sellCount++;
   }
}
```

The `buyCount` and `sellCount` variables will contain the number of open orders of each type. Inside the `for` loop, the `order` variable serves as the loop iterator. The oldest order in the order pool is assigned an index of zero, while the newest order is assigned an index of `OrdersTotal()` - 1. The `OrdersTotal()` function returns a count of all orders currently open in the terminal.

For each open order in the order pool, we call the `OrderSelect()` function. The `SELECT_BY_POS` parameter indicates that the `order` variable will contain the position of the order in the order pool. So if `order` equals 0, `OrderSelect()` will select the oldest order in the pool.

After selecting the order, we call the `OrderMagicNumber()` function to retrieve the magic number assigned to the order. If the magic number is equal to the value of the `MagicNumber` input variable, then we call the `OrderType()` function. If the order type is `OP_BUY` or `OP_SELL`, we increment the relevant variable.

In this book, we use order counts to determine whether there is a position currently open, and we will make trading decisions based on that information.

Order Modification

After placing an order, you can modify the take profit price, stop loss price, pending order price or expiration time using the `OrderModify()` function. To use `OrderModify()`, we'll need the ticket number of the order that we wish to modify. We may also need some information about the order, such as the current stop loss, take profit or order opening price. Here is the syntax for the `OrderModify()` function:

```
bool OrderModify(int Ticket, double Price, double StopLoss, double TakeProfit,
   datetime Expiration, color Arrow = CLR_NONE)
```

- **Ticket** – The ticket number of the order to modify.
- **Price** – The desired pending order price.

- **StopLoss** – The desired stop loss price.
- **TakeProfit** – The desired take profit price.
- **Expiration** – The expiration time for pending orders.
- **Arrow** – A optional color for the arrow to indicate a modified order. If not indicated, no arrow will be displayed.

If the order modification is successful, `OrderModify()` will return a value of `true`. If the order modification failed, the return value will be `false`.

When modifying orders, we must be sure that the values we are passing to the function are valid. For example, the order must still be open – we cannot modify a closed order. When modifying pending orders with the `Price` parameter, the order must not have already been filled – *i.e.* hit its order price. The modified order price also must not be too close to the current Bid or Ask price. We should also check to make sure that the stop loss and take profit prices are valid. We can do this using the stop price verification functions that we will cover later in this book.

If we are not modifying a particular order parameter, we must pass the current value to the `OrderModify()` function. For example, if we are modifying only the stop loss for a pending order, then we must retrieve the current order price and take profit price by calling `OrderSelect()` and then calling the relevant order information functions to pass those values to the `OrderModify()` function.

If you attempt to modify an order without specifying any changed values, you'll get an *error 1: no result*. You should verify why your code is passing unchanged values to the function, but otherwise this error is harmless and can be safely ignored.

In the next chapter, we will examine how to use `OrderModify()` to set a stop loss and take profit price on market orders, and how to modify open pending orders.

Closing Orders

When we close a market order, we are exiting the trade at the current market price. For buy orders, we close at the Bid price, and for sell orders, we close at the Ask. For pending orders, we simply delete the order from the trade pool.

OrderClose()

We close market orders using the `OrderClose()` function. Here is the syntax:

```
bool OrderClose(int Ticket, double Lots, double Price, int Slippage, color Arrow);
```

- **Ticket** – The ticket number of the market order to close.
- **Lots** – The number of lots to close. Most brokers allow partial closes.
- **Price** – The preferred price at which to close the trade. For buy orders, this will be the current Bid price, and for sell orders, the current Ask price.
- **Slippage** – The allowed slippage from the closing price, in points.
- **Color** – A color constant for the closing arrow. If no color is indicated, no arrow will be drawn.

You can close part of a trade by specifying a partial lot size. For example, if you have a trade open with a lot size of 2.00, and you want to close half of the trade, then specify 1 lot for the `Lots` argument. Note that not all brokers support partial closes. If you do partially close an order, then the ticket number of the remaining order will change.

It is recommended that if you need to close a position in several parts, you should place multiple orders and then close each order individually, instead of doing partial closes. Using the example above, you would place two orders of 1.00 lot each, then simply close one of the orders when you want to close out half of the position.

The following example closes a buy market order:

```
bool selected = OrderSelect(closeTicket,SELECT_BY_TICKET);

if(OrderCloseTime() == 0 && OrderType() == OP_BUY && selected == true)
{
   double closeLots = OrderLots();
   double closePrice = Bid;

   bool closed = OrderClose(closeTicket,closeLots,closePrice,Slippage,clrRed);
}
```

The `closeTicket` variable contains the ticket number of the order that we wish to close. The `OrderSelect()` function selects the order, and allows us to retrieve the order information. We use `OrderCloseTime()` to check the order closing time to see if the order has already been closed. If `OrderCloseTime()` returns 0, then we know the order has not been closed yet. We also need to check the order type, since the order type determines the closing price for the order. The `OrderType()` function returns an integer indicating the order type. If it's a buy market order, indicated by the `OP_BUY` constant, we'll continue with closing the order.

Next, we retrieve the order lot size using `OrderLots()`, and store that value in the `closeLots` variable. We assign the current Bid price to the `closePrice` variable. Then we call the `OrderClose()` function to close out the order. If the order has been closed successfully, the value of `closed` will be `true`, otherwise `false`.

To close a sell market order, all you need to do is change the order type to `OP_SELL` and assign the current `Ask` price to `closePrice`:

```
if(OrderCloseTime() == 0 && OrderType() == OP_SELL && selected == true)
{
   double closeLots = OrderLots();
   double closePrice = Ask;

   bool closed = OrderClose(closeTicket,closeLots,closePrice,Slippage,clrRed);
}
```

OrderDelete()

There is a separate function for closing pending orders. `OrderDelete()` has two arguments, the ticket number and the arrow color. No closing price, lot size or slippage is required. Here is the code to close a pending buy stop order:

```
bool selected = OrderSelect(closeTicket,SELECT_BY_TICKET);

if(OrderCloseTime() == 0 && OrderType() == OP_BUYSTOP && selected == true)
{
   bool deleted = OrderDelete(closeTicket,clrRed);
}
```

As we did with the `OrderClose()` function above, we need to check the order type to be sure it is a pending order. The pending order type constants are `OP_BUYSTOP`, `OP_SELLSTOP`, `OP_BUYLIMIT` and `OP_SELLLIMIT`. To close other types of pending orders, simply change the order type.

If the order has been filled, then it is now a market order, and must be closed using `OrderClose()` instead.

Closing Multiple Orders

Most of the time, you will be closing all orders that are currently opened by your expert advisor. This may be a single order, or it may be multiple orders. In either case, we can simply loop through the open order pool and close any orders that match the correct order type and the magic number set by the user.

To loop through the order pool, we use a `for` loop to iterate through the open orders from oldest to newest. Because of the FIFO rules in effect for US brokers, we will close orders in the order that they were placed to ensure compatibility with US-based brokers. The example below closes all sell orders currently opened by an expert advisor:

```
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool selected = OrderSelect(order,SELECT_BY_POS);

   if(OrderType() == OP_SELL && OrderMagicNumber() == MagicNumber && selected == true)
   {
      // Close order
      bool closed = OrderClose(OrderTicket(),OrderLots(),Ask,Slippage,clrRed);
      if(closed == true) order--;
   }
}
```

After selecting the order using `OrderSelect()`, we examine it to see if it fits our closing criteria. If the magic number is equal to the `MagicNumber` input variable, and the order type is equal to `OP_SELL`, then we attempt to close the order.

The `OrderTicket()` function returns the order ticket for the currently selected order, and `OrderLots()` returns the volume of the currently selected order. We pass both of these values to the `OrderClose()` function, along with the current Ask price, the slippage (if necessary), and optionally an arrow color. If the order was closed successfully, the boolean variable `closed` will be set to `true`.

If `closed` is equal to `true`, we will decrement the order variable. Why do we do this? When an order is closed, the newer orders in the order pool shift down by one. If we don't decrement the iterator variable, then orders may be skipped. Of course, if you decide to close orders from newest to oldest, this won't be a concern.

This method of closing orders is easy, and doesn't rely on a previously stored order ticket. We will be using a similar method to close orders in our expert advisors.

Chapter 11 - Stop Loss & Take Profit

Calculating Stop Loss & Take Profit Prices

There are several methods of calculating a stop loss and/or take profit price. The most common method is to specify the number of pips away from the order opening price to place your stop. For example, if we have a stop loss setting of 500 points, that means that the stop loss price will be 500 points away from our order opening price.

We can also use a fixed price, such as a recent swing high or low, an indicator value, an input variable or some other type of calculation. To avoid order modification errors, we will need to verify that the stop loss or take profit price is valid before we attempt to place it.

Calculating in Points

For this, the most common method of calculating stops, we will use an input variable in which the user specifies the number of points for the stop loss and take profit. We then calculate the stops relative to the order opening price.

For market orders, we will first place an order at the current market price. Then we will retrieve the order opening price and calculate the stop loss or take profit price relative to that. For pending orders, we will simply calculate the stop loss or take profit price relative to our anticipated order opening price.

Here are the input variables we'll use for our stop loss and take profit distance:

```
input int StopLoss = 500;
input int TakeProfit = 1000;
```

In this example, we've entered a stop loss of 500 points, and a take profit of 1000 points. To calculate our stop loss price, we'll need to add or subtract 500 points from the order opening price. To do this, we need to convert the integer value of 500 to a fractional value that we'll add or subtract from the opening price.

To convert an integer to the appropriate fractional value, we need to multiply our `StopLoss` input variable by the point value for the trade symbol. `_Point` is a predefined variable in MQL4 that returns the smallest price unit of a currency. For a 5 decimal currency pair, the point is 0.00001, and for a Yen pair, it's 0.001. We will multiply the point value by our input variable, and then add or subtract the fractional value from the order opening price.

Here's an example of a stop loss and take profit price calculation for a buy market order:

```
double stopLoss = 0, takeProfit = 0;
if(StopLoss > 0) stopLoss = OrderOpenPrice() - (StopLoss * _Point);
if(TakeProfit > 0) takeProfit = OrderOpenPrice() + (TakeProfit * _Point);
```

We define two `double` variables, `stopLoss` and `takeProfit`, and initialize them to zero. These will hold the stop loss and take profit prices. If the `StopLoss` and `TakeProfit` input variables are greater than zero, then we will calculate the stop prices by adding or subtracting the `StopLoss` or `TakeProfit` values, multiplied by the `_Point` value, from the order opening price.

Once we've calculate a stop loss and take profit price, we will use `OrderModify()` to add a stop loss and take profit to the order.

Adding Stop Loss and Take Profit to an Order

To add a stop loss and take profit price to a market order, we first need to verify that the order has been placed correctly. We do this by examining the return value of the `OrderSend()` function, which is the ticket number of the order that was just placed. If the order was not placed due to an error condition, the ticket number will be equal to `-1`.

Next, we use the `OrderSelect()` function to retrieve the information for the order that was just placed. We calculate the new stop loss and/or take profit price relative to the order opening price. Finally, we'll use `OrderModify()` to add the stop loss and take profit to the order.

Here's an example where we set the stop loss and take profit for a buy order using the `OrderModify()` function:

```
// Open buy order
gBuyTicket = OrderSend(_Symbol,OP_BUY,lotSize,Ask,Slippage,0,0,"Buy order",
   MagicNumber,0,clrGreen);

// Add stop loss & take profit to order
if(gBuyTicket > 0 && (StopLoss > 0 || TakeProfit > 0))
{
   bool selected = OrderSelect(gBuyTicket,SELECT_BY_TICKET);

   // Calculate stop loss & take profit
   double stopLoss = 0, takeProfit = 0;
   if(StopLoss > 0) stopLoss = OrderOpenPrice() - (StopLoss * _Point);
   if(TakeProfit > 0) takeProfit = OrderOpenPrice() + (TakeProfit * _Point);

   // Modify order
   bool modified = OrderModify(gBuyTicket,0,stopLoss,takeProfit,0);
}
```

After the order is placed with the `OrderSend()` function, we check to see if the conditions warrant placing a stop loss or take profit on the order. If `gBuyTicket` is greater than zero, and either the `StopLoss` or `TakeProfit` input variables are greater than zero, then we will proceed to modify the order.

First, we call the `OrderSelect()` function to select the order that we have just placed. This will allow us to retrieve the order opening price using the `OrderOpenPrice()` function. We calculate the stop loss and take profit prices, relative to `OrderOpenPrice()`. Then we call the `OrderModify()` function. We pass a value of 0 for the `Price` and `Expiration` parameters, since we cannot change the order price or the expiration time for market orders. If `OrderModify()` is successful, the value of the `modified` variable will be `true`.

Modifying a Pending Order

`OrderModify()` can also be used to modify the order price of a pending order. If the pending order price has already been hit and the order has been filled, it is no longer a pending order, and the price cannot be changed.

We'll use the variable `newPendingPrice` to represent our changed order price. We'll assume the price has already been calculated and is valid. In the example below, we will not be changing the stop loss, take profit or expiration time on the order. Here's how we modify a pending order price:

```
bool select = OrderSelect(ticket,SELECT_BY_TICKET);

if(newPendingPrice != OrderOpenPrice() && select == true)
{
   bool modified = OrderModify(ticket,newPendingPrice,OrderStopLoss(),
      OrderTakeProfit(),OrderExpiration());
}
```

As before, we select the order using `OrderSelect()`. This allows us to retrieve the current order information. Before modifying the order, we'll check to make sure that our new pending order price is not the same as the current pending order price, and that the order selection was successful.

For the `OrderModify()` function, we specify our order ticket, the new order opening price stored in `newPendingPrice`, and the unchanged stop loss, take profit and expiration values represented by the `OrderStopLoss()`, `OrderTakeProfit()` and `OrderExpiration()` functions. Remember, when modifying orders, you must pass the current values to the `OrderModify()` function for any parameter that you do not wish to change!

Verifying Stops and Pending Order Prices

Stop loss, take profit and pending order prices must be a minimum distance away from the Bid and Ask prices. If a stop or pending order price is too close to the current price, an error will result, and the order will not be placed. This is one of the most common trading errors, and it can easily be prevented if the trader is careful to set their stops and pending orders a sufficient distance from the current price.

But during periods of rapid price movement, valid stop loss prices can be made invalid by widening spreads. Different brokers have varying stop levels, so a stop loss that is valid on one broker may be too close for another. Some trading systems will set stops and pending order prices based on indicator values, highs or lows, or some other method of calculation where a minimum distance is not guaranteed.

For these reasons, it is recommended that you automatically verify that a stop loss, take profit or pending order price is valid, and not too close to the current market price. We verify this by checking the symbol's *stop level*.

Stop Levels

The stop level is the minimum number of points away from the current Bid or Ask price that all stops and pending orders must be placed. Figure 11.1 illustrates the stop levels in relation to the prices. Think of the price as not being just a single value (such as the Bid), but rather a thick line the width of the spread. On either side of that price line are boundaries, indicated by the stop levels. All stop loss, take profit and pending orders must be placed outside of these boundaries.

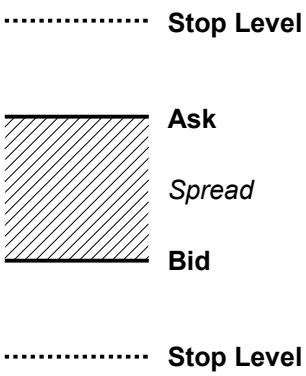


Fig. 11.1 – Stop levels

The `MarketInfo()` function with the `MODE_STOPLEVEL` parameter is used to retrieve the stop level for a trading symbol. The stop level is expressed as an integer, and will need to be converted to a fractional value using `_Point`.

Here's an example of how to retrieve the stop level for the current chart symbol:

```
double stopLevel = MarketInfo(_Symbol,MODE_STOPLEVEL) * _Point;
```

For example, if the stop level for EURUSD is 30 points, then the `stopLevel` variable will contain 0.0003. We will need to add this value to the current Ask price and subtract it from the current Bid price to calculate our stop level prices.

The code below will calculate the upper and lower stop levels based on the current market prices:

```
double stopLevel = MarketInfo(_Symbol,MODE_STOPLEVEL) * _Point;

double upperStopLevel = Ask + stopLevel;
double lowerStopLevel = Bid - stopLevel;
```

The `upperStopLevel` variable contains the stop level price in relation to the Ask price. All buy stop, sell limit, buy take profit and sell stop loss prices must be greater than this price. The `lowerStopLevel` variable contains the stop level in relation to the Bid price. All sell stop, buy limit, sell take profit and buy stop loss prices must be less than this price.

Verifying Stop Loss and Take Profit Prices

Before placing or modifying an order with a stop loss or take profit price, we will compare it to the current stop level price to ensure that it is valid. We can automatically adjust any invalid stop loss or take profit price so that it is just outside of the stop level price. Here's an example of checking the stop loss and take profit price for a buy order, and adjusting it if necessary:

```
if(takeProfit <= upperStopLevel && takeProfit != 0) takeProfit = upperStopLevel + _Point;
if(stopLoss >= lowerStopLevel && stopLoss  != 0) stopLoss = lowerStopLevel - _Point;
```

If the `takeProfit` value is less than or equal to the `upperStopLevel` value, and `takeProfit` is not zero, we will set the take profit price to the upper stop level, plus one point. If the stop loss is greater than or equal to the lower stop level, and the stop loss is not zero, we will set the stop loss to the lower stop level, minus one point.

Instead of automatically adjusting an invalid price, you could also display an error message and halt trade execution. This way the user would be required to readjust their stop loss or take profit setting before continuing. Here's an example of how to do this:

```
if(stopLoss >= lowerStopLevel && stopLoss != 0)
{
   Alert("The stop loss setting is too small!");
}
```

If the stop loss is invalid, the `Alert()` function will display a pop-up message to the user.

In this book, we will be automatically adjusting invalid prices, with the assumption that is is better to place a corrected order than to not place one at all. It may be useful to document when this happens by printing a message to the log:

```
if(stopLoss >= lowerStopLevel && stopLoss  != 0)
{
   stopLoss = lowerStopLevel - _Point;
   Print("Stop loss is invalid and has been automatically adjusted");
}
```

Verifying Pending Order Prices

We can verify the opening price for a pending order using the same logic as above. Here's how we verify the pending order price for a buy stop or sell limit order. The `pendingPrice` variable contains the pending order price:

```
if(pendingPrice <= upperStopLevel) pendingPrice = upperStopLevel + _Point;
```

Notice that the logic here is identical to the code above that verifies our buy take profit price. To verify an opening price for a sell stop or buy limit order:

```
if(pendingPrice >= lowerStopLevel) pendingPrice = lowerStopLevel - _Point;
```

Chapter 12 - A Simple Expert Advisor

Now that you've learned the basics of expert advisors – including how to open, modify and close orders; calculate and verify stop and pending order prices, and get a count of open orders – lets put it all together with a simple expert advisor program.

This expert advisor is a simple trading system using a single moving average indicator. When the close price of the previous bar is above the moving average, we open a buy order, and when the opposite is true, we open a sell order. Only one order will be open at a time, and we will close the current order before opening another order in the opposite direction.

The source code for this file is `Simple Expert Advisor.mq4`, and is located in the `\MQL4\Experts\Mql4Book` folder. Let's go through the file a section at a time:

```
#property copyright           "Andrew R. Young"
#property link                "http://www.expertadvisorbook.com"
#property description         "A simple expert advisor that places market orders"
#property strict

// Input variables
input int MagicNumber = 101;
input int Slippage = 10;

input double LotSize = 0.1;
input int StopLoss = 0;
input int TakeProfit = 0;

input int MaPeriod = 5;
input ENUM_MA_METHOD MaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;

// Global variables
int gBuyTicket, gSellTicket;
```

The beginning of the file contains the preprocessor directives, our input variables (including trade settings and the moving average settings), and two global variables to hold ticket numbers.

```
// Tick event handler
void OnTick()
{
   // Current order counts
   int buyCount = 0, sellCount = 0;
```

```
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool select = OrderSelect(order,SELECT_BY_POS);

   if(OrderMagicNumber() == MagicNumber && select == true)
   {
      if(OrderType() == OP_BUY) buyCount++;
      else if(OrderType() == OP_SELL) sellCount++;
   }
}

// Moving average and close price from last bar
double ma = iMA(_Symbol,_Period,MaPeriod,0,MaMethod,MaPrice,1);
double close = Close[1];
```

This is the beginning of the `OnTick()` event handler. We have a `for` loop that counts the number of buy and sell market orders that are currently open that match our `MagicNumber` input variable. The order counts are stored in the `buyCount` and `sellCount` variables.

Next, we retrieve the moving average and close prices for the previous bar. These values are stored in the `ma` and `close` variables. We will cover indicators and bar prices in later chapters.

```
// Buy order condition
if(close > ma && buyCount == 0 && gBuyTicket == 0)
{
   // Close sell order
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      bool select = OrderSelect(order,SELECT_BY_POS);

      if(OrderType() == OP_SELL && OrderMagicNumber() == MagicNumber && select == true)
      {
         // Close order
         bool closed = OrderClose(OrderTicket(),OrderLots(),Ask,Slippage,clrRed);
         if(closed == true) order--;
      }
   }
```

This is the start of our buy order block. If the close price of the last bar is greater than the moving average price, there are no buy orders currently open, and the `gBuyTicket` variable is zero (we'll address this in a bit), we will close any open sell orders and open a buy order. A `for` loop searches the order pool for all sell market orders that match the magic number and then closes them.

```
      // Open buy order
      gBuyTicket = OrderSend(_Symbol,OP_BUY,LotSize,Ask,Slippage,0,0,"Buy order",
         MagicNumber,0,clrGreen);
      gSellTicket = 0;

      // Add stop loss & take profit to order
      if(gBuyTicket > 0 && (StopLoss > 0 || TakeProfit > 0))
      {
         bool select = OrderSelect(gBuyTicket,SELECT_BY_TICKET);

         // Calculate stop loss & take profit
         double stopLoss = 0, takeProfit = 0;
         if(StopLoss > 0) stopLoss = OrderOpenPrice() - (StopLoss * _Point);
         if(TakeProfit > 0) takeProfit = OrderOpenPrice() + (TakeProfit * _Point);

         // Verify stop loss & take profit
         double stopLevel = MarketInfo(_Symbol,MODE_STOPLEVEL) * _Point;

         RefreshRates();
         double upperStopLevel = Ask + stopLevel;
         double lowerStopLevel = Bid - stopLevel;

         if(takeProfit <= upperStopLevel && takeProfit != 0)
            takeProfit = upperStopLevel + _Point;
         if(stopLoss >= lowerStopLevel && stopLoss  != 0) stopLoss = lowerStopLevel - _Point;

         // Modify order
         bool modify = OrderModify(gBuyTicket,0,stopLoss,takeProfit,0);
      }
   }
```

Next, we open a buy market order and store the order ticket in the `gBuyTicket` variable. The `gSellTicket` variable is then set to zero. We set the `gBuyTicket` and `gSellTicket` variables to prevent a second order from opening if the first order closes before the price crosses the moving average in the opposite direction. We then reset the value of that variable when an order is opened in the opposite direction.

After placing the buy market order, we calculate the stop loss and take profit prices. First we check to see if the `gBuyTicket` variable is greater than zero (indicating that an order was placed), and that either the `StopLoss` or `TakeProfit` input variables are greater than zero. If so, we select the current buy order using `OrderSelect()` and proceed with calculating the stop loss and take profit prices relative to the order opening price. We verify that the prices are valid, relative to the stop level price, and then modify the order to add the stop loss and/or take profit price.

Below is the sell order block. It is similar to the buy order block:

```
// Sell order condition
if(close < ma && sellCount == 0 && gSellTicket == 0)
{
   // Close buy order
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      bool select = OrderSelect(order,SELECT_BY_POS);

      if(OrderType() == OP_BUY && OrderMagicNumber() == MagicNumber && select == true)
      {
         // Close order
         bool closed = OrderClose(OrderTicket(),OrderLots(),Bid,Slippage,clrRed);

         if(closed == true)
         {
            gBuyTicket = 0;
            order--;
         }
      }
   }

   // Open sell order
   gSellTicket = OrderSend(_Symbol,OP_SELL,LotSize,Bid,Slippage,0,0,"Sell order",
      MagicNumber,0,clrRed);
   gBuyTicket = 0;

   // Add stop loss & take profit to order
   if(gSellTicket > 0 && (StopLoss > 0 || TakeProfit > 0))
   {
      bool select = OrderSelect(gSellTicket,SELECT_BY_TICKET);

      // Calculate stop loss & take profit
      double stopLoss = 0, takeProfit = 0;
      if(StopLoss > 0) stopLoss = OrderOpenPrice() + (StopLoss * _Point);
      if(TakeProfit > 0) takeProfit = OrderOpenPrice() - (TakeProfit * _Point);

      // Verify stop loss & take profit
      double stopLevel = MarketInfo(_Symbol,MODE_STOPLEVEL) * _Point;

      RefreshRates();
      double upperStopLevel = Ask + stopLevel;
      double lowerStopLevel = Bid - stopLevel;

      if(takeProfit >= lowerStopLevel && takeProfit != 0)
         takeProfit = lowerStopLevel - _Point;
      if(stopLoss <= upperStopLevel && stopLoss != 0) stopLoss = upperStopLevel + _Point;
```

```
            // Modify order
            bool modify = OrderModify(gSellTicket,0,stopLoss,takeProfit,0);
         }
      }
   }
```

Even though this is a very simple trading system, most expert advisor programs will follow a similar structure. We will declare the preprocessor directives, input variables and global variables at the top of the file. Next, we have our event handling functions such as `OnTick()`, `OnInit()` or `OnDeinit()`. (We did not use the `OnInit()` function here, but we will later in the book.) If there are any other functions specific to the program, they will go at the bottom.

Most of the action happens inside the `OnTick()` event handling function. First, we will perform any calculations and retrieve any prices that are required for our trade conditions before we attempt to open or close orders. We will have separate order blocks for each order type. Before opening the order, we will close existing orders as appropriate. We will then open the order, and add a stop loss and/or take profit price if it is a market order. If we are modifying stops, such as a trailing stop, we will do that after any order opening or closing operations.

The problem with this program is that all of our order placement and handling logic is contained in a single file. Even a simple trading system such as this contains a lot of code. The process of opening, closing or modifying orders is the same regardless of your trading system. In the following chapters, we will create reusable order classes and functions that will make our expert advisor programs much more compact and maintainable.

Chapter 13 - Order Placement Functions

In Chapters 13 through 22, we will focus on creating reusable classes and functions for many common trading tasks. These classes and functions will be located inside the include files that are available in the source code download.

In each chapter, we will first explain the details of implementing our trading classes and functions. If you simply want to move forward with creating your own expert advisors using these classes and functions, you can skip ahead to the example headings where we show you how to use them in an expert advisor.

Our trading-related functions will be inside the `\MQL4\Include\Mql4Book\Trade.mqh` file. All functions that directly open, close or modify orders will be inside the `CTrade` class. We will start by creating functions to open market and pending orders.

Market Order Functions

Let's create a function that will allow us to open market orders. This function will open an order of the specified type with the specified symbol, volume, order comment and arrow color. The function is named `OpenMarketOrder()`, and it is a member of the `CTrade` class. Below is the function declaration from `Trade.mqh`:

```
#include <stdlib.mqh>

#define MAX_RETRIES 3         // Max retries on error
#define RETRY_DELAY 3000      // Retry delay in ms

class CTrade
{
   private:
      static int _magicNumber;
      static int _slippage;

      int OpenMarketOrder(string pSymbol, int pType, double pVolume, string pComment,
         color pArrow);
};

int CTrade::_magicNumber = 0;
int CTrade::_slippage = 10;
```

We have declared `OpenMarketOrder()` as a private function. We will not call this function directly from our programs – instead we will create public “wrapper” functions that will call it for us. We have also specified two

private, static member variables in our CTrade class: _magicNumber and _slippage. These variables are static, which means that they will be shared among all class objects in our program. Regardless of how many CTrade objects we create, they will all use the same magic number and slippage values.

We initialize both of these variables with a default value. The _magicNumber variable has a default value of 0, while the _slippage variable is assigned a default value of 10 points. Later, we will create public functions to change and access these values.

We have included the stdlib.mqh include file that comes with MetaTrader 4, as it contains a function that we will need to use inside our order placement function. We have also defined two constants, MAX_RETRIES and RETRY_DELAY. We will address the usage of these later in the chapter.

Let's walk through the OpenMarketOrder() function:

```
int CTrade::OpenMarketOrder(string pSymbol, int pType, double pVolume, string pComment,
   color pArrow)
{
   int retryCount = 0;
   int ticket = 0;
   int errorCode = 0;

   double orderPrice = 0;

   string orderType;
   string errDesc;
```

The pSymbol parameter is the trade symbol, pType is the order type (OP_BUY or OP_SELL), pVolume is the lot size, pComment is the order comment, and pArrow is the chart arrow color. We start by initializing several variables that we will use inside our function.

```
   // Order retry loop
   while(retryCount <= MAX_RETRIES)
   {
      while(IsTradeContextBusy()) Sleep(10);

      // Get current bid/ask price
      if(pType == OP_BUY) orderPrice = MarketInfo(pSymbol,MODE_ASK);
      else if(pType == OP_SELL) orderPrice = MarketInfo(pSymbol,MODE_BID);

      // Place market order
      ticket = OrderSend(pSymbol,pType,pVolume,orderPrice,_slippage,0,0,pComment,
         _magicNumber,0,pArrow);
```

One of the features of our order placement function is a retry loop. If an error occurs, we can retry the order placement up to a specified number of times, depending on the error code. The `MAX_RETRIES` constant is the maximum number of times to retry the order. It is set to three retries by default, although you can adjust this if you wish.

The first thing we do before attempting to place the order is to check the trade context. MetaTrader 4 only has one thread for communicating with the trade server. That means it can only do one trade operation at a time. The `IsTradeContextBusy()` function checks to see if another trade operation is in progress. If so, we wait 10 milliseconds and check again. This is a simple way of ensuring that two or more expert advisors do not attempt to trade at the same time. It's not foolproof, but in the event that the trade context is busy when we attempt to place our order, we will simply retry the order operation.

Next, we retrieve the current Bid or Ask price for the specified symbol, depending on the order type. The current price is stored in the `orderPrice` variable. Then we place the order using the `OrderSend()` function. The resulting order ticket number is assigned to the `ticket` variable.

The remaining code in this function is used for retry functionality and error handling:

```
// Error handling
if(ticket == -1)
{
   errorCode = GetLastError();
   errDesc = ErrorDescription(errorCode);
   bool checkError = RetryOnError(errorCode);
   orderType = OrderTypeToString(pType);

   // Unrecoverable error
   if(checkError == false)
   {
      Alert("Open ",orderType," order: Error ",errorCode," - ",errDesc);
      Print("Symbol: ",pSymbol,", Volume: ",pVolume,", Price: ",orderPrice);
      break;
   }

   // Retry on error
   else
   {
      Print("Server error detected, retrying...");
      Sleep(RETRY_DELAY);
      retryCount++;
   }
}
```

```
            // Order successful
            else
            {
               orderType = OrderTypeToString(pType);
               Comment(orderType," order #",ticket," opened on ",pSymbol);
               Print(orderType," order #",ticket," opened on ",pSymbol);
               break;
            }
         }
```

If the value of the `ticket` variable is -1, indicating that the order was not placed, we'll retrieve some information about the error. The `GetLastError()` function returns the error code from the last attempted trade operation, and assigns it to the `errorCode` variable. The `ErrorDescription()` function is defined in the `stdlib.mqh` include file that is installed with MetaTrader 4. We've included this file at the top of `Trade.mqh`. It returns a text description of the error code that is saved to the `errDesc` variable.

The `RetryOnError()` function is declared elsewhere in our `Trade.mqh` file. It contains a list of error codes that we will attempt to retry the order operation on. If an error code is not listed in the `RetryOnError()` function, it will return `false`. The result is stored in the `checkError` variable. The `OrderTypeToString()` function is another function declared in our `Trade.mqh` file that returns a readable, user-friendly string for each order type.

Now that we have some information about the error, we will determine whether to retry the order operation or not. If the `checkError` variable is `false`, we will display an alert to the user with the `Alert()` function, and log troubleshooting information to the experts log with the `Print()` function. We will then `break` out of the retry loop.

If the `checkError` variable is true, we will print the message "Server error detected, retrying..." to the log, sleep for the number of milliseconds specified by the `RETRY_DELAY` constant (default is three seconds), iterate the `retryCount` variable, and then return to the top of the `while` loop.

If the value of the `ticket` variable is not equal to -1, then we can assume the order was placed successfully. We will print a comment to the top-left corner of the chart using the `Comment()` function, print a message to the log using the `Print()` function, and `break` out of the while loop.

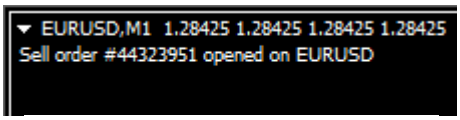


Fig 13.1 – Trade comment

```
   // Failed after retry
   if(retryCount > MAX_RETRIES)
   {
      Alert("Open ",orderType," order: Max retries exceeded. Error ",errorCode,
         " - ",errDesc);
      Print("Symbol: ",pSymbol,", Volume: ",pVolume,", Price: ",orderPrice);
   }
   return(ticket);
}
```

If our attempts to retry the order placement has exceeded the number of retries specified by the `MAX_RETRIES` constant, we will alert the user and print additional troubleshooting information to the log. In any event, the `return` operator will return the `ticket` number to the program.

Public Market Order Functions

The `OpenMarketOrder()` function is a `private` function, which means that we cannot call it directly from outside of the class. We will create two public functions that will call this function for us and pass the appropriate order type:

```
class CTrade
{
   private:
      static int _magicNumber;
      static int _slippage;

      int OpenMarketOrder(string pSymbol, int pType, double pVolume, string pComment,
         color pArrow);

    public:
      int OpenBuyOrder(string pSymbol, double pVolume, string pComment = "Buy order",
         color pArrow = clrGreen);
      int OpenSellOrder(string pSymbol, double pVolume, string pComment = "Sell order",
         color pArrow = clrRed);
};
```

The `OpenBuyOrder()` and `OpenSellOrder()` functions will be used to open market orders. An order symbol and volume will need to be passed to the function. The comment and arrow color are optional. The order type, price and other trade parameters will be handled for you.

Here are the `OpenBuyOrder()` and `OpenSellOrder()` functions in their entirety. As you can see, they simply call the `OpenMarketOrder()` function with the appropriate order type and return the ticket number:

```
int CTrade::OpenBuyOrder(string pSymbol, double pVolume, string pComment = "Buy order",
   color pArrow = 32768)
{
   int ticket = OpenMarketOrder(pSymbol, OP_BUY, pVolume, pComment, pArrow);
   return(ticket);
}

int CTrade::OpenSellOrder(string pSymbol, double pVolume, string pComment = "Sell order",
   color pArrow = 255)
{
   int ticket = OpenMarketOrder(pSymbol, OP_SELL, pVolume, pComment, pArrow);
   return(ticket);
}
```

Market Order Function Examples

We can use our `OpenBuyOrder()` and `OpenSellOrder()` functions anywhere we would use the `OrderSend()` function. Here's an example using the simple expert advisor from the previous chapter. We'll replace the `OrderSend()` function to place a buy order with our `OpenBuyOrder()` function.

First, we will need to include the `Trade.mqh` file and create an object based on our `CTrade` class. This will go at the top of our program file:

```
#include <Mql4Book\Trade.mqh>
CTrade Trade;
```

Our `CTrade` class object is named `Trade`. To open a buy order, we call the public `OpenBuyOrder()` function of our `CTrade` class:

```
// OnTick() event handler
gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
gSellTicket = 0;
```

All we need to specify is the order symbol (in this case, the current chart symbol specified by `_Symbol`), and the order volume. We can optionally set an order comment or change the arrow color if we wish.

By using our order placement functions, we get increased robustness through our order retry functionality, trade context checking, as well as logging and an informational chart comment.

Pending Order Functions

Just like we did with `OpenMarketOrder()`, we're going to create a private function to place all types of pending orders. The `OpenPendingOrder()` function is very similar to our market order placement function. The primary difference is that our pending order function will accept an order opening price, a stop loss and take profit price, and an order expiration time. We will also create public wrapper functions to place different types of pending orders.

Below is our `CTrade` class with the pending order functions added:

```
class CTrade
{
   private:
      static int _magicNumber;
      static int _slippage;

      int OpenMarketOrder(string pSymbol, int pType, double pVolume, string pComment,
         color pArrow);

      int OpenPendingOrder(string pSymbol, int pType, double pVolume, double pPrice,
         double pStop, double pProfit, string pComment, datetime pExpiration, color pArrow);

   public:
      int OpenBuyOrder(string pSymbol, double pVolume, string pComment = "Buy order",
         color pArrow = clrGreen);
      int OpenSellOrder(string pSymbol, double pVolume, string pComment = "Sell order",
         color pArrow = clrRed);

      int OpenBuyStopOrder(string pSymbol, double pVolume, double pPrice, double pStop,
         double pProfit, string pComment = "Buy stop order", datetime pExpiration = 0,
         color pArrow = clrBlue);
      int OpenSellStopOrder(string pSymbol, double pVolume, double pPrice, double pStop,
         double pProfit, string pComment = "Sell stop order", datetime pExpiration = 0,
         color pArrow = clrIndigo);
      int OpenBuyLimitOrder(string pSymbol, double pVolume, double pPrice, double pStop,
         double pProfit, string pComment = "Buy limit order", datetime pExpiration = 0,
         color pArrow = clrCornflowerBlue);
      int OpenSellLimitOrder(string pSymbol, double pVolume, double pPrice, double pStop,
         double pProfit, string pComment = "Sell limit order", datetime pExpiration = 0,
         color pArrow = clrMediumSlateBlue);
};
```

Here is the `OpenPendingOrder()` function. We've highlighted the major differences between this and the `OpenMarketOrder()` function in bold:

```
int CTrade::OpenPendingOrder(string pSymbol,int pType,double pVolume,double pPrice,
   double pStop,double pProfit,string pComment,datetime pExpiration,color pArrow)
{
   int retryCount = 0;
   int ticket = 0;
   int errorCode = 0;

   string orderType;
   string errDesc;

   // Order retry loop
   while(retryCount <= MAX_RETRIES)
   {
      while(IsTradeContextBusy()) Sleep(10);
      ticket = OrderSend(pSymbol, pType, pVolume, pPrice, _slippage, pStop, pProfit,
         pComment, _magicNumber, pExpiration, pArrow);

      // Error handling
      if(ticket == -1)
      {
         errorCode = GetLastError();
         errDesc = ErrorDescription(errorCode);
         bool checkError = RetryOnError(errorCode);
         orderType = OrderTypeToString(pType);

         // Unrecoverable error
         if(checkError == false)
         {
            Alert("Open ",orderType," order: Error ",errorCode," - ",errDesc);
            Print("Symbol: ",pSymbol,", Volume: ",pVolume,", Price: ",pPrice,
               ", SL: ",pStop,", TP: ",pProfit,",Expiration: ",pExpiration);
            break;
         }

         // Retry on error
         else
         {
            Print("Server error detected, retrying...");
            Sleep(RETRY_DELAY);
            retryCount++;
         }
      }
```

```
      // Order successful
      else
      {
         orderType = OrderTypeToString(pType);
         Comment(orderType," order #",ticket," opened on ",pSymbol);
         Print(orderType," order #",ticket," opened on ",pSymbol);
         break;
      }
   }

   // Failed after retry
   if(retryCount > MAX_RETRIES)
   {
      Alert("Open ",orderType," order: Max retries exceeded. Error ",errorCode,
         " - ",errDesc);
      Print("Symbol: ",pSymbol,", Volume: ",pVolume,", Price: ",pPrice,", SL: ",pStop,
         ", TP: ",pProfit,", Expiration: ",pExpiration);
   }

   return(ticket);
}
```

As you can see, the only difference is that we pass an order opening price, stop loss price, take profit price, and expiration time to the function. We do not need these parameters for placing market orders, but we do for pending orders.

And here are the public pending order placement functions. As you can see, they simply call the `OpenPendingOrder()` function and pass the appropriate order type. The comment, expiration time and arrow parameters are optional:

```
int CTrade::OpenBuyStopOrder(string pSymbol, double pVolume, double pPrice, double pStop,
   double pProfit, string pComment = "Buy stop order", datetime pExpiration = 0,
   color pArrow = 16711680)
{
   int ticket = OpenPendingOrder(pSymbol, OP_BUYSTOP, pVolume, pPrice, pStop, pProfit,
      pComment, pExpiration, pArrow);
   return(ticket);
}
```

```
int CTrade::OpenSellStopOrder(string pSymbol, double pVolume, double pPrice, double pStop,
   double pProfit, string pComment = "Sell stop order", datetime pExpiration = 0,
   color pArrow = 8519755)
{
   int ticket = OpenPendingOrder(pSymbol, OP_SELLSTOP, pVolume, pPrice, pStop, pProfit,
      pComment, pExpiration, pArrow);
   return(ticket);
}

int CTrade::OpenBuyLimitOrder(string pSymbol, double pVolume, double pPrice, double pStop,
   double pProfit, string pComment="Buy limit order", datetime pExpiration = 0,
   color pArrow = 15570276)
{
   int ticket = OpenPendingOrder(pSymbol, OP_BUYLIMIT, pVolume, pPrice, pStop, pProfit,
      pComment, pExpiration, pArrow);
   return(ticket);
}

int CTrade::OpenSellLimitOrder(string pSymbol, double pVolume, double pPrice, double pStop,
   double pProfit, string pComment = "Sell limit order", datetime pExpiration = 0,
   color pArrow = 15624315)
{
   int ticket = OpenPendingOrder(pSymbol, OP_SELLLIMIT, pVolume, pPrice, pStop, pProfit,
      pComment, pExpiration, pArrow);
   return(ticket);
}
```

Pending Order Function Examples

Just like we did with our market order placement functions, we can use our pending order functions anywhere we would use the `OrderSend()` function to open a pending order. This example will open a pending buy stop order at the high of the last bar:

```
#include <Mql4Book\Trade.mqh>
CTrade Trade;

// OnTick() event handler
double lastHigh = High[1];
double stopLoss = lastHigh - (StopLoss * _Point);
double takeProfit = lastHigh + (TakeProfit * _Point);

int ticket = Trade.OpenBuyStopOrder(_Symbol,lotSize,lastHigh,stopLoss,takeProfit);
```

As before, we need to include the `Trade.mqh` file and create a `CTrade` class object on the global scope of our program. Inside the `OnTick()` event handler, the `lastHigh` variable will contain the high price of the previous

bar. We calculate the stop loss and take profit prices relative to the high of the previous bar, and save those prices to the `stopLoss` and `takeProfit` variables respectively. If the `OpenBuyStopOrder()` function is successful, we will store the ticket number in the `ticket` variable.

Trade Property Functions

In our `OpenMarketOrder()` and `OpenPendingOrder()` functions, we've referenced several private class variables that hold trade properties, namely the magic number and slippage. The `_magicNumber` and `_slippage` variables have been initialized with default values that may be used if the user does not wish to change them.

Setting the Magic Number

It is highly recommended that you set the magic number for each expert advisor in your trade terminal. This is the primary method by which we differentiate orders between expert advisors. The `SetMagicNumber()` function of our `CTrade` class is used to set the magic number for use by our trading functions.

The `SetMagicNumber()` function simply sets the private `_magicNumber` variable in our `CTrade` class to the value that we pass to the function. The `_magicNumber` variable, as well as the `SetMagicNumber()` function, are `static`, which means that it is shared among all instances of the class:

```
static void CTrade::SetMagicNumber(int pMagic)
{
   if(_magicNumber != 0)
   {
      Alert("Magic number changed! Any orders previously opened by this expert advisor will
         no longer be handled!");
   }

   _magicNumber = pMagic;
}
```

If the value of `_magicNumber` is anything other than zero, an alert will be displayed to the user. If the current instance of the expert advisor in the terminal has orders open, then changing the magic number means that it will no longer be able to handle those orders!

In our expert advisors, we will call the `SetMagicNumber()` function once, inside the `OnInit()` event handler. It takes the `MagicNumber` input variable as a parameter:

```
#include <Mql4Book\Trade.mqh>
CTrade Trade;
```

```
// Input variable
input int MagicNumber = 101;

// OnInit() event handler
int OnInit()
{
   // Set magic number
   Trade.SetMagicNumber(MagicNumber);

   return(INIT_SUCCEEDED);
}
```

If you need to retrieve the value of the `_magicNumber` variable for any reason, we have created a function to return this value:

```
static int CTrade::GetMagicNumber(void)
{
   return(_magicNumber);
}
```

The `GetMagicNumber()` function is also `static`, which means that all instances of the class will return the same magic number. The reason why we don't allow the programmer to reference the `_magicNumber` variable directly is that it's bad practice. By making the variable private and creating functions to get and set the value explicitly, we ensure that it won't accidentally be changed when a programmer attempts to reference the value directly.

Setting the Slippage

We've also created a function to set the slippage value. If your broker is an ECN/STP and does not require a slippage value when placing orders, you can skip this step. Otherwise, it is recommended that you specify a reasonable value for slippage.

The function is called `SetSlippage()`, and can be called from the `OnInit()` event handler:

```
static void CTrade::SetSlippage(int pSlippage)
{
   _slippage = pSlippage;
}
```

As before, the `SetSlippage()` function is `static`, so all instances of the class will share the same slippage value. Calling the `SetSlippage()` function will allow you to set the slippage to something other than its default value of 10 points:

```
#include <Mql4Book\Trade.mqh>
CTrade Trade;

// Input variable
input int MagicNumber = 101;
input int Slippage = 30;

int OnInit()
{
   // Set magic number
   Trade.SetMagicNumber(MagicNumber);

   // Set slippage
   Trade.SetSlippage(Slippage);

   return(INIT_SUCCEEDED);
}
```

Other Internal Functions

The `OpenMarketOrder()` and `OpenPendingOrder()` functions rely on several internal functions for error checking and logging purposes. The most important of these is `RetryOnError()`. This function contains a list of error codes that our order functions will attempt to retry on:

```
bool RetryOnError(int pErrorCode)
{
   // Retry on these error codes
   switch(pErrorCode)
   {
      case ERR_BROKER_BUSY:
      case ERR_COMMON_ERROR:
      case ERR_NO_ERROR:
      case ERR_NO_CONNECTION:
      case ERR_NO_RESULT:
      case ERR_SERVER_BUSY:
      case ERR_NOT_ENOUGH_RIGHTS:
      case ERR_MALFUNCTIONAL_TRADE:
      case ERR_TRADE_CONTEXT_BUSY:
      case ERR_TRADE_TIMEOUT:
      case ERR_REQUOTE:
      case ERR_TOO_MANY_REQUESTS:
      case ERR_OFF_QUOTES:
      case ERR_PRICE_CHANGED:
      case ERR_TOO_FREQUENT_REQUESTS:
```

```
            return(true);
         }
         return(false);
      }
```

The function uses a `switch-case` block to compare the value of the `pErrorCode` input parameter to a list of error code constants returned by the trade server. You can view all of the error codes in the *MQL4 Reference* under *Standard Constants* > *Codes of Errors and Warnings* > *Trade Server Return Codes*.

These errors are ones that we have determined to be recoverable. If we retry the order operation again, it may succeed. If the `pErrorCode` value matches any of the error codes in the list, we'll return a value of `true`. If you come across any error codes not in this list that you wish for your expert advisors to retry on, you can add it to the function in the `Trade.mqh` file.

We also have a second function in our `Trade.mqh` file that converts an order type constant (such as `OP_BUY` or `OP_SELL`) to a user-friendly string. We use this for printing chart comments and for logging:

```
string OrderTypeToString(int pType)
{
   string orderType;
   if(pType == OP_BUY) orderType = "Buy";
   else if(pType == OP_SELL) orderType = "Sell";
   else if(pType == OP_BUYSTOP) orderType = "Buy stop";
   else if(pType == OP_BUYLIMIT) orderType = "Buy limit";
   else if(pType == OP_SELLSTOP) orderType = "Sell stop";
   else if(pType == OP_SELLLIMIT) orderType = "Sell limit";
   else orderType = "Invalid order type";
   return(orderType);
}
```

You can use this function anytime you need to convert an order type constant to a string for display in the log or on the chart.

Chapter 14 - Order Modification Functions

In this chapter, we will create several functions to modify open market orders. We will calculate stop loss and take profit prices, verify them for correctness, and apply them to a recently opened order.

Modify Order Function

Let's start by creating a function to modify orders. This function will modify both market and pending orders, and will serve as a drop-in replacement for the MQL4 `OrderModify()` function. The function is named `ModifyOrder()`, and is located in the `Trade.mqh` file:

```
bool ModifyOrder(int pTicket, double pPrice, double pStop = 0, double pProfit = 0,
   datetime pExpiration = 0, color pArrow = clrOrange)
{
   int retryCount = 0;
   int errorCode = 0;
   bool result = false;
   string errDesc;
```

Note that `ModifyOrder()` is not part of the `CTrade` class, so we can call it directly. The required parameters are `pTicket`, which is the ticket number of the order to modify, and `pPrice`, which is the new order price. The optional `pStop`, `pProfit` and `pExpiration` parameters modify the stop loss price, take profit price and expiration time respectively. You can also modify the arrow color using the `pArrow` parameter.

Remember, if you are not changing a trade parameter, you must pass the current value by selecting the order with `OrderSelect()` first, and then calling the relevant order information function, such as `OrderOpenPrice()`, `OrderStopLoss()`, `OrderTakeProfit()` or `OrderExpiration()`!

```
   // Order retry loop
   while(retryCount <= MAX_RETRIES)
   {
      while(IsTradeContextBusy()) Sleep(10);

      result = OrderModify(pTicket, pPrice, pStop, pProfit, pExpiration, pArrow);
      errorCode = GetLastError();
```

This is the start of our retry loop. We check the trade context, and call the `OrderModify()` function to modify the order. The result is stored in the boolean variable `result`. We also call the `GetLastError()` function to retrieve the error code. We'll see why shortly.

```
            // Error handling - Ignore error code 1
            if(result == false && errorCode != ERR_NO_RESULT)
            {
               errDesc = ErrorDescription(errorCode);
               bool checkError = RetryOnError(errorCode);

               // Unrecoverable error
               if(checkError == false)
               {
                  Alert("Modify order #",pTicket,": Error ",errorCode," - ",errDesc);
                  Print("Price: ",pPrice,", SL: ",pStop,", TP: ",pProfit,
                     ", Expiration: ",pExpiration);
                  break;
               }

               // Retry on error
               else
               {
                  Print("Server error detected, retrying...");
                  Sleep(RETRY_DELAY);
                  retryCount++;
               }
            }

            // Order successful
            else
            {
               Comment("Order #",pTicket," modified");
               Print("Order #",pTicket," modified");
               break;
            }
         }
```

If the value of the `result` variable is `false`, which indicates an error condition, we check the value of `errorCode` before proceeding. If you attempt to modify an order without providing any changed values, an *error 1 – no result* is generated. We can safely ignore this error, but if the error code is equal to anything other than 1, we will proceed to check the error.

As before, if the error is unrecoverable (*i.e.* our `RetryOnError()` function returns `false`), we will alert the user, log some troubleshooting information, and exit the function. Otherwise, we will retry the order modification again. If the order modification was successful, we print a comment to the chart.

Modify Order Function Example

Let's demonstrate how our `ModifyOrder()` function works. As an example, we will borrow the stop loss and take profit modification code from our simple expert advisor, and replace the `OrderModify()` function with our `ModifyOrder()` function:

```
// Add stop loss & take profit to order
if(gBuyTicket > 0 && (StopLoss > 0 || TakeProfit > 0))
{
   bool select = OrderSelect(gBuyTicket,SELECT_BY_TICKET);

   // Calculate stop loss & take profit
   double stopLoss = 0, takeProfit = 0;
   if(StopLoss > 0) stopLoss = OrderOpenPrice() - (StopLoss * _Point);
   if(TakeProfit > 0) takeProfit = OrderOpenPrice() + (TakeProfit * _Point);

   // Verify stop loss & take profit
   double stopLevel = MarketInfo(_Symbol,MODE_STOPLEVEL) * _Point;

   RefreshRates();
   double upperStopLevel = Ask + stopLevel;
   double lowerStopLevel = Bid - stopLevel;

   if(takeProfit <= upperStopLevel && takeProfit != 0) takeProfit = upperStopLevel + _Point;
   if(stopLoss >= lowerStopLevel && stopLoss  != 0) stopLoss = lowerStopLevel - _Point;

   // Modify order
   ModifyOrder(gBuyTicket,0,stopLoss,takeProfit);
}
```

The `ModifyOrder()` function provides the same functionality as the built-in `OrderModify()` function, with added order retry functionality, error handling and logging. We will call our `ModifyOrder()` function from the other functions that we'll create in this chapter anytime that we need to modify an order.

Stop Calculation Functions

Looking at the code above from our simple expert advisor, the process of calculating and verifying a stop loss and take profit price is the largest part of our order placement code. We're going to create several functions to simplify the process of calculating and verifying stop prices. These functions will be located in the `Trade.mqh` include file.

We'll start by creating functions to calculate a stop loss price. The function below calculates the stop loss price for a buy order:

```
double BuyStopLoss(string pSymbol,int pStopPoints, double pOpenPrice = 0)
{
   if(pStopPoints <= 0) return(0);

   double openPrice;
   if(pOpenPrice > 0) openPrice = pOpenPrice;
   else openPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);

   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double stopLoss = openPrice - (pStopPoints * point);

   long digits = SymbolInfoInteger(pSymbol,SYMBOL_DIGITS);
   stopLoss = NormalizeDouble(stopLoss,(int)digits);

   return(stopLoss);
}
```

The `pSymbol` parameter is the trade symbol, `pStopPoints` is the stop loss in points, and `pOpenPrice` is an optional order opening price that is used to calculate a stop loss for a pending buy order.

If the `pStopPoints` parameter is less than or equal to zero, then we will immediately exit the function and return zero. Otherwise, we will retrieve the current Ask price for the specified symbol and store it in the `openPrice` variable, as well as storing the symbol's point value in the `point` variable.

We calculate the stop loss price by subtracting `pStopPoints` (multiplied by `point`) from `openPrice`. This value is stored in the `stopLoss` variable. We use the `NormalizeDouble()` function to round the `stopLoss` variable to the number of digits in the symbol price, and return the stop loss price to our program.

The process of calculating a take profit price for a buy order is very similar – only the arithmetic operations are reversed:

```
double BuyTakeProfit(string pSymbol,int pProfitPoints, double pOpenPrice = 0)
{
   if(pProfitPoints <= 0) return(0);

   double openPrice;
   if(pOpenPrice > 0) openPrice = pOpenPrice;
   else openPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);

   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double takeProfit = openPrice + (pProfitPoints * point);

   long digits = SymbolInfoInteger(pSymbol,SYMBOL_DIGITS);
   takeProfit = NormalizeDouble(takeProfit,(int)digits);
```

```
   return(takeProfit);
}
```

The sell stop loss and take profit calculation functions are very similar. In all, we have four stop calculation functions, `BuyStopLoss()`, `SellStopLoss()`, `BuyTakeProfit()` and `SellTakeProfit()`. You can view them in the `\MQL4\Include\Mql4Book\Trade.mqh` file.

Stop Level Verification Functions

After calculating a stop loss or take profit price, we'll need to verify it against the stop level price. We're going to write two sets of functions to do this. One set of functions will compare a price to either the upper or lower stop level price, and return true if the price is valid. The other set of functions will automatically adjust an invalid price to a valid one.

Let's start with a function to check the upper stop level. This is used to verify buy take profit, sell stop loss, pending buy stop and pending sell limit prices:

```
bool CheckAboveStopLevel(string pSymbol, double pPrice, int pPoints = 10)
{
   double currPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);
   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double stopLevel = SymbolInfoInteger(pSymbol,SYMBOL_TRADE_STOPS_LEVEL) * point;
   double stopPrice = currPrice + stopLevel;
   double addPoints = pPoints * point;

   if(pPrice >= stopPrice + addPoints) return(true);
   else return(false);
}
```

The `pSymbol` parameter is the trade symbol, `pPrice` is the stop or pending order price to verify, and `pPoints` is an additional number of points to add to the stop level. By default, we will add 10 points.

We retrieve the current Ask price, the point value and the stop level for the specified symbol. To calculate the stop level price, we add the stop level (multiplied by the point value) to the current price, and store that in the `stopPrice` variable. Then we multiply the `pPoints` parameter by the point value and store that in the `addPoints` variable. If the `pPrice` parameter is greater than or equal to `stopPrice` plus `addPoints`, we return a value of `true`.

This function and its counterpart, `CheckBelowStopLevel()`, can be used to check if a price is valid relative to the stop level. If it is not, you can notify the user or take other action. The next set of functions will automatically adjust the stop level if it's invalid. The `AdjustAboveStopLevel()` function is similar to the function above. We've highlighted the changes below:

```
double AdjustAboveStopLevel(string pSymbol, double pPrice, int pPoints = 10)
{
   double currPrice = SymbolInfoDouble(pSymbol,SYMBOL_ASK);
   double point = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double stopLevel = SymbolInfoInteger(pSymbol,SYMBOL_TRADE_STOPS_LEVEL) * point;
   double stopPrice = currPrice + stopLevel;
   double addPoints = pPoints * point;

   if(pPrice > stopPrice + addPoints) return(pPrice);
   else
   {
      double newPrice = stopPrice + addPoints;
      Print("Price adjusted above stop level to "+DoubleToString(newPrice));
      return(newPrice);
   }
}
```

If `pPrice` is greater than or equal to `stopPrice` plus `addPoints`, we'll return the value of `pPrice`. Otherwise, we will adjust the price to `stopPrice` plus `addPoints` and return that price. We will use this and the `AdjustBelowStopLevel()` function throughout this book to verify stop and pending order prices. You can view these functions in the `\MQL4\Include\Mql4Book\Trade.mqh` file.

Stop Calculation & Verification Function Examples

Let's use the buy stop loss and take profit modification example from earlier, and show how these functions can simplify our code:

```
// Add stop loss & take profit to order
if(gBuyTicket > 0 && (StopLoss > 0 || TakeProfit > 0))
{
   bool select = OrderSelect(gBuyTicket,SELECT_BY_TICKET);

   // Calculate stop loss & take profit
   double stopLoss = 0, takeProfit = 0;
   stopLoss = BuyStopLoss(_Symbol,StopLoss,OrderOpenPrice());
   takeProfit = BuyTakeProfit(_Symbol,TakeProfit,OrderOpenPrice());

   // Verify stop loss & take profit
   stopLoss = AdjustBelowStopLevel(_Symbol,stopLoss);
   takeProfit = AdjustAboveStopLevel(_Symbol,takeProfit);

   // Modify order
   Trade.ModifyOrder(gBuyTicket,0,stopLoss,takeProfit);
}
```

We've greatly simplified this block of code by using our stop calculation and verification functions. But we can simplify this even more by creating an all-in-one function that will calculate our stop loss and take profit prices, verify them and modify the order for us. We'll do this in the next section.

The stop calculation and verification functions in this chapter can be used for pending orders as well. Simply pass your pending order opening price to the function to calculate your stop loss and take profit prices, and use the stop verification functions to verify your order opening and stop prices:

```
double lastHigh = High[1];
lastHigh = AdjustAboveStopLevel(_Symbol,lastHigh);

double stopLoss = BuyStopLoss(_Symbol,StopLoss,lastHigh);
double takeProfit = BuyTakeProfit(_Symbol,TakeProfit,lastHigh);

stopLoss = AdjustBelowStopLevel(_Symbol,stopLoss);
takeProfit = AdjustAboveStopLevel(_Symbol,takeProfit);

int ticket = trade.OpenBuyStopOrder(_Symbol,lotSize,lastHigh,stopLoss,takeProfit);
```

The `lastHigh` variable contains the pending buy stop order opening price. The `AdjustAboveStopLevel()` function verifies that this price is valid. The `BuyStopLoss()` and `BuyTakeProfit()` functions calculate the stop loss and take profit prices, and the appropriate stop level verification functions verify that these prices are valid.

Stop Modification Functions

The process of calculating a stop loss and take profit price and adding it to a market order is the same for nearly all trading systems. We're going to create two functions to calculate and modify stop prices for an order. One function uses fixed integer values, and the other takes prices directly as input.

The `ModifyStopsByPoints()` function takes a fixed stop loss and take profit value, calculates the stop prices, verifies them, and adds them to a market order:

```
bool ModifyStopsByPoints(int pTicket, int pStopPoints, int pProfitPoints = 0,
   int pMinPoints = 10)
{
   if(pStopPoints == 0 && pProfitPoints == 0) return false;

   bool result = OrderSelect(pTicket,SELECT_BY_TICKET);
```

```
      if(result == false)
      {
         Print("Modify stops: #",pTicket," not found!");
         return false;
      }

      double orderType = OrderType();
      string orderSymbol = OrderSymbol();
      double orderOpenPrice = OrderOpenPrice();

      double stopLoss = 0;
      double takeProfit = 0;

      if(orderType == OP_BUY)
      {
         stopLoss = BuyStopLoss(orderSymbol,pStopPoints,orderOpenPrice);
         if(stopLoss != 0) stopLoss = AdjustBelowStopLevel(orderSymbol,stopLoss,pMinPoints);

         takeProfit = BuyTakeProfit(orderSymbol,pProfitPoints,orderOpenPrice);
         if(takeProfit != 0)
            takeProfit = AdjustAboveStopLevel(orderSymbol,takeProfit,pMinPoints);
      }
      else if(orderType == OP_SELL)
      {
         stopLoss = SellStopLoss(orderSymbol,pStopPoints,orderOpenPrice);
         if(stopLoss != 0) stopLoss = AdjustAboveStopLevel(orderSymbol,stopLoss,pMinPoints);

         takeProfit = SellTakeProfit(orderSymbol,pProfitPoints,orderOpenPrice);
         if(takeProfit != 0)
            takeProfit = AdjustBelowStopLevel(orderSymbol,takeProfit,pMinPoints);
      }

      result = ModifyOrder(pTicket,0,stopLoss,takeProfit);
      return(result);
   }
```

The required parameters are `pTicket`, the order ticket to modify, and `pStopPoints`, the stop loss value in points. The optional parameters are `pProfitPoints`, the take profit value in points, and `pMinPoints`, the minimum distance in points from the stop level. In the event that a stop loss or take profit price is inside the stop level, the `pMinPoints` parameter determines how many points outside of the stop level to place the stop.

If no value has been specified for either `pStopPoints` or `pProfitPoints`, the function will exit and return `false`. Otherwise, we'll select the order specified by `pTicket` and retrieve the order type, opening price and symbol. Depending on the order type, we call the relevant stop calculation and verification functions that we discussed earlier. Then we pass those values to our `ModifyOrder()` function.

A second function, `ModifyStopsByPrice()`, takes a stop loss and take profit price instead of a fixed value. You can use this function if your stop loss or take profit prices are not a fixed distance from the order opening price, or if you are uncertain about the validity of the prices:

```
bool ModifyStopsByPrice(int pTicket, double pStopPrice, double pProfitPrice = 0,
   int pMinPoints = 10)
{
   if(pStopPrice == 0 && pProfitPrice == 0) return false;

   bool result = OrderSelect(pTicket,SELECT_BY_TICKET);

   if(result == false)
   {
      Print("Modify stops: #",pTicket," not found!");
      return false;
   }

   double orderType = OrderType();
   string orderSymbol = OrderSymbol();

   double stopLoss = 0;
   double takeProfit = 0;

   if(orderType == OP_BUY)
   {
      if(stopLoss != 0) stopLoss = AdjustBelowStopLevel(orderSymbol,pStopPrice,pMinPoints);
      if(takeProfit != 0)
         takeProfit = AdjustAboveStopLevel(orderSymbol,pProfitPrice,pMinPoints);
   }

   else if(orderType == OP_SELL)
   {
      if(stopLoss != 0) stopLoss = AdjustAboveStopLevel(orderSymbol,pStopPrice,pMinPoints);
      if(takeProfit != 0)
         takeProfit = AdjustBelowStopLevel(orderSymbol,pProfitPrice,pMinPoints);
   }

   result = ModifyOrder(pTicket,0,stopLoss,takeProfit);
   return(result);
}
```

The `ModifyStopsByPrice()` function simply takes a stop loss (and optional take profit) price, adjusts it if it is not valid, and modifies the order.

Stop Modification Function Example

Let's return to our stop loss and take profit modification code from earlier. We can now replace all of this code with our `ModifyStopsByPoints()` function. Here is the entire buy order block from our simple expert advisor:

```
if(close > ma && buyCount == 0 && gBuyTicket == 0)
{
   // Close sell order(s)
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      OrderSelect(order,SELECT_BY_POS);

      if(OrderType() == OP_SELL && OrderMagicNumber() == MagicNumber)
      {
         // Close order
         bool closed = OrderClose(OrderTicket(),OrderLots(),Ask,Slippage,clrRed);
         if(closed == true) order--;
      }
   }

   // Open buy order
   gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
   gSellTicket = 0;

   // Add stop loss & take profit to order
   ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
}
```

We've replaced over a dozen lines of code with one simple function call. No longer do you need to worry about how to properly calculate stop prices, or whether those prices are valid. The `ModifyStopsByPoints()` and `ModifyStopsByPrice()` functions will take care of it for you.

If for some reason, these functions do not meet the needs of your specific trading system, you can easily write your own stop placement function that uses the `ModifyOrder()` function along with the stop verification functions described in this chapter.

Chapter 15 - Order Closing Functions

In this chapter, we will create functions to close market orders and delete pending orders. We will create functions to close individual orders, as well as all orders of a specified type.

Close Market Order Function

We'll start with a function that will close an individual market order. `CloseMarketOrder()` is a public function in our `CTrade` class, and is located in the `Trade.mqh` include file:

```
bool CTrade::CloseMarketOrder(int pTicket, double pVolume = 0, color pArrow = clrRed)
{
   int retryCount = 0;
   int errorCode = 0;

   double closePrice = 0;
   double closeVolume = 0;

   bool result;
   string errDesc;
```

The required parameter is `pTicket`, which is the ticket number to close. `pVolume` is the trade volume to close. If `pVolume` is zero, then the entire order will be closed.

```
   // Select ticket
   result = OrderSelect(pTicket,SELECT_BY_TICKET);

   // Exit with error if order select fails
   if(result == false)
   {
      errorCode = GetLastError();
      errDesc = ErrorDescription(errorCode);

      Alert("Close order: Error selecting order #",pTicket,". Error ",errorCode,
         " - ",errDesc);
      return(result);
   }

   // Close entire order if pVolume not specified, or if pVolume is greater than order volume
   if(pVolume == 0 || pVolume > OrderLots()) closeVolume = OrderLots();
   else closeVolume = pVolume;
```

We begin by selecting the order ticket. If `OrderSelect()` returns false, we will show an alert to the user. Otherwise, we'll calculate the order volume to close, based on the `pVolume` input parameter:

```
// Order retry loop
while(retryCount <= MAX_RETRIES)
{
   while(IsTradeContextBusy()) Sleep(10);

   // Get current bid/ask price
   if(OrderType() == OP_BUY) closePrice = MarketInfo(OrderSymbol(),MODE_BID);
   else if(OrderType() == OP_SELL) closePrice = MarketInfo(OrderSymbol(),MODE_ASK);

   result = OrderClose(pTicket,closeVolume,closePrice,_slippage,pArrow);
```

This is the beginning of the order operation retry loop. We check the trade context and retrieve the current Bid or Ask price, depending on the order operation. Then we call the `OrderClose()` function to close the order at the specified price, volume and slippage.

```
  if(result == false)
  {
     errorCode = GetLastError();
     errDesc = ErrorDescription(errorCode);
     bool checkError = RetryOnError(errorCode);

     // Unrecoverable error
     if(checkError == false)
     {
        Alert("Close order #",pTicket,": Error ",errorCode," - ",errDesc);
        Print("Price: ",closePrice,", Volume: ",closeVolume);
        break;
     }

     // Retry on error
     else
     {
        Print("Server error detected, retrying...");
        Sleep(RETRY_DELAY);
        retryCount++;
     }
  }
```

```
      // Order successful
      else
      {
         Comment("Order #",pTicket," closed");
         Print("Order #",pTicket," closed");
         break;
      }
   }

   // Failed after retry
   if(retryCount > MAX_RETRIES)
   {
      Alert("Close order #",pTicket,": Max retries exceeded. Error ",errorCode,
         " - ",errDesc);
       Print("Price: ",closePrice,", Volume: ",closeVolume);
    }

    return(result);
}
```

As before, the rest of the function checks for an error condition and retries or logs the result as appropriate. If the order is closed successfully, `CloseMarketOrder()` returns a value of `true`.

The `CloseMarketOrder()` function can be used as drop-in replacement for the MQL4 `OrderClose()` function. We will also use it in our function to close multiple orders later in the chapter.

Delete Pending Order Function

Next, we'll create a function to delete pending orders. The `ClosePendingOrder()` function can be used as a drop-in replacement for MQL4's `OrderDelete()` function. The only required parameter is the ticket number of the order to delete:

```
bool CTrade::DeletePendingOrder(int pTicket, color pArrow = clrRed)
{
   int retryCount = 0;
   int errorCode = 0;

   bool result = false;
   string errDesc;

   // Order retry loop
   while(retryCount <= MAX_RETRIES)
   {
      while(IsTradeContextBusy()) Sleep(10);
      result = OrderDelete(pTicket,pArrow);
```

```
            if(result == false)
            {
               errorCode = GetLastError();
               errDesc = ErrorDescription(errorCode);
               bool checkError = RetryOnError(errorCode);

               // Unrecoverable error
               if(checkError == false)
               {
                  Alert("Delete pending order #",pTicket,": Error ",errorCode," - ",errDesc);
                  break;
               }

               // Retry on error
               else
               {
                  Print("Server error detected, retrying...");
                  Sleep(RETRY_DELAY);
                  retryCount++;
               }
            }

            // Order successful
            else
            {
               Comment("Pending order #",pTicket," deleted");
               Print("Pending order #",pTicket," deleted");
               break;
            }
         }

         // Failed after retry
         if(retryCount > MAX_RETRIES)
         {
            Alert("Delete pending order #",pTicket,": Max retries exceeded. Error ",errorCode,
               " - ",errDesc);
         }

         return(result);
      }
```

Close Multiple Market Orders Functions

Most of the time, we will be closing all orders of a certain type when a trade condition is met. We will be using this approach in the trading systems that we will write in this book. The `CloseMultipleOrders()` function is a

private function in the `CTrade` class that will close market orders by type. We will not be calling this function directly from our expert advisors – rather, we will create public wrapper functions for each order type we want to close.

We have created an enumeration containing each of the order types that we want to close. The options are: close buy orders only, close sell orders only, or close all orders. The `CLOSE_MARKET_TYPE` enumeration is a private member of our `CTrade` class:

```
enum CLOSE_MARKET_TYPE
{
   CLOSE_BUY,
   CLOSE_SELL,
   CLOSE_ALL_MARKET
};
```

The only required parameter for our `CloseMultipleOrders()` function is `pCloseType`, which uses our `CLOSE_MARKET_TYPE` enumeration as the type. We pass one of the values of our enumeration to the function, which determines which orders will be closed:

```
bool CTrade::CloseMultipleOrders(CLOSE_MARKET_TYPE pCloseType)
{
   bool error = false;
   bool closeOrder = false;

   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      int orderType = OrderType();
      int orderMagicNumber = OrderMagicNumber();
      int orderTicket = OrderTicket();
      double orderVolume = OrderLots();

      // Determine if order type matches pCloseType
      if( (pCloseType == CLOSE_ALL_MARKET && (orderType == OP_BUY || orderType == OP_SELL))
         || (pCloseType == CLOSE_BUY && orderType == OP_BUY)
         || (pCloseType == CLOSE_SELL && orderType == OP_SELL) )
      {
         closeOrder = true;
      }
      else closeOrder = false;
```

```
            // Close order if pCloseType and magic number match currently selected order
            if(closeOrder == true && orderMagicNumber == _magicNumber)
            {
               result = CloseMarketOrder(orderTicket,orderVolume);

               if(result == false)
               {
                  Print("Close multiple orders: ",OrderTypeToString(orderType)," #",orderTicket,
                     " not closed");
                  error = true;
               }
               else order--;
            }
         }

         return(error);
      }
```

We loop through the order pool from oldest to newest. We select each order and retrieve information about that order, including the order type, magic number, the order ticket, and the lot size. Depending on the value of `pCloseType` and the order type of the currently selected order, we will either attempt to close the order or go to the next order in the pool.

If the order type matches the type of order that we want to close, we'll check to see if the magic number matches the value in the `_magicNumber` class variable. If so, then we'll call our `CloseMarketOrder()` function to close the order. If the order is closed successfully, we must decrement the `order` variable by one. The function returns a value of `true` if all orders were closed successfully.

Now we will create our public functions to close multiple market orders by type. As you can see, these functions simply call the `CloseMultipleOrders()` function using the appropriate value from our `CLOSE_MARKET_TYPE` enumeration:

```
      bool CTrade::CloseAllBuyOrders(void)
      {
         bool result = CloseMultipleOrders(CLOSE_BUY);
         return(result);
      }

      bool CTrade::CloseAllSellOrders(void)
      {
         bool result = CloseMultipleOrders(CLOSE_SELL);
         return(result);
      }
```

```
bool CTrade::CloseAllMarketOrders(void)
{
   bool result = CloseMultipleOrders(CLOSE_ALL_MARKET);
   return(result);
}
```

The `CloseAllBuyOrders()` functions closes all buy orders that match the magic number that is set in the `CTrade` class. The `CloseAllSellOrders()` function does the same for sell orders. `CloseAllMarketOrders()` will close all market orders, both buy and sell.

Close Multiple Market Orders Example

Here's an example of how we can close all sell orders before opening a buy order. We'll use the same code from the simple expert advisor from earlier:

```
if(close > ma && buyCount == 0 && gBuyTicket == 0)
{
   // Close sell order(s)
   Trade.CloseAllSellOrders();

   // Open buy order
   gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
   gSellTicket = 0;

   // Add stop loss & take profit to order
   ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
}
```

The code above accomplishes the same task that the code in our simple expert advisor does, plus we have added additional functionality including retry on error, logging and alerting capabilities. Our expert advisor programs will contain only the code necessary to execute the trading system logic. All of the order placement and management details will be handled by the code in our include files.

Delete Multiple Pending Order Functions

We also have a private function in the `CTrade` class to delete multiple pending orders. As before, we have created an enumeration to contain the values to pass to the function:

```
enum CLOSE_PENDING_TYPE
{
   CLOSE_BUY_LIMIT,
   CLOSE_SELL_LIMIT,
   CLOSE_BUY_STOP,
```

```
      CLOSE_SELL_STOP,
      CLOSE_ALL_PENDING
   };
```

The options for closing multiple pending orders are: close all buy limit orders, close all sell limit orders, close all buy stop orders, close all sell stop orders, and close all pending orders.

The function is named `DeleteMultipleOrders()`, and it takes a single parameter from our `CLOSE_PENDING_TYPE` enumeration:

```
bool CTrade::DeleteMultipleOrders(CLOSE_PENDING_TYPE pDeleteType)
{
   bool error = false;
   bool deleteOrder = false;

   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      int orderType = OrderType();
      int orderMagicNumber = OrderMagicNumber();
      int orderTicket = OrderTicket();
      double orderVolume = OrderLots();

      // Determine if order type matches pCloseType
      if( (pDeleteType == CLOSE_ALL_PENDING && orderType != OP_BUY && orderType != OP_SELL)
         || (pDeleteType == CLOSE_BUY_LIMIT && orderType == OP_BUYLIMIT)
         || (pDeleteType == CLOSE_SELL_LIMIT && orderType == OP_SELLLIMIT)
         || (pDeleteType == CLOSE_BUY_STOP && orderType == OP_BUYSTOP)
         || (pDeleteType == CLOSE_SELL_STOP && orderType == OP_SELLSTOP) )
      {
         deleteOrder = true;
      }
      else deleteOrder = false;

      // Close order if pCloseType and magic number match currently selected order
      if(deleteOrder == true && orderMagicNumber == _magicNumber)
      {
         result = DeletePendingOrder(orderTicket);

         if(result == false)
         {
            Print("Delete multiple orders: ",OrderTypeToString(orderType)," #",orderTicket,
               " not deleted");
```

```
            error = true;
         }
         else order--;
      }
   }
   return(error);
}
```

This function is very similar to the `CloseMultipleOrders()` function from earlier. If the order type matches the value of the `pDeleteType` variable, and the order magic number matches the `_magicNumber` class variable, then we delete the order.

As before, we'll create wrapper functions to delete all open orders of each type:

```
bool CTrade::DeleteAllBuyLimitOrders(void)
{
   bool result = DeleteMultipleOrders(CLOSE_BUY_LIMIT);
   return(result);
}

bool CTrade::DeleteAllBuyStopOrders(void)
{
   bool result = DeleteMultipleOrders(CLOSE_BUY_STOP);
   return(result);
}

bool CTrade::DeleteAllSellLimitOrders(void)
{
   bool result = DeleteMultipleOrders(CLOSE_SELL_LIMIT);
   return(result);
}

bool CTrade::DeleteAllSellStopOrders(void)
{
   bool result = DeleteMultipleOrders(CLOSE_SELL_STOP);
   return(result);
}

bool CTrade::DeleteAllPendingOrders(void)
{
   bool result = DeleteMultipleOrders(CLOSE_ALL_PENDING);
   return(result);
}
```

Chapter 16 - Order Counting Functions

To finish up our order-related functions, we have created a class that counts all open orders by type. The class is named `CCount`, and it is located in the `Trade.mqh` file:

```
class CCount
{
   private:
      enum COUNT_ORDER_TYPE
      {
         COUNT_BUY,
         COUNT_SELL,
         COUNT_BUY_STOP,
         COUNT_SELL_STOP,
         COUNT_BUY_LIMIT,
         COUNT_SELL_LIMIT,
         COUNT_MARKET,
         COUNT_PENDING,
         COUNT_ALL
      };

      int CountOrders(COUNT_ORDER_TYPE pType);

   public:
      int Buy();
      int Sell();
      int BuyStop();
      int SellStop();
      int BuyLimit();
      int SellLimit();
      int TotalMarket();
      int TotalPending();
      int TotalOrders();
};
```

The class consists of a private function, `CountOrders()`, and an enumeration, `COUNT_ORDER_TYPE`, that is used as the type for the input parameter to the `CountOrders()` function. We also have nine public functions that return the current count for each type of order.

Let's start by examining the private `CountOrders()` function. This is a function that loops through the open orders, counts the open orders by type, and then returns the specified value:

```
int CCount::CountOrders(COUNT_ORDER_TYPE pType)
{
   // Order counts
   int buy = 0, sell = 0, buyStop = 0, sellStop = 0, buyLimit = 0, sellLimit = 0,
      totalOrders = 0;

   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      int orderType = OrderType();
      int orderMagicNumber = OrderMagicNumber();

      // Add to order count if magic number matches
      if(orderMagicNumber == CTrade::GetMagicNumber())
      {
         switch(orderType)
         {
            case OP_BUY:
               buy++;
               break;

            case OP_SELL:
               sell++;
               break;

            case OP_BUYLIMIT:
               buyLimit++;
               break;

            case OP_SELLLIMIT:
               sellLimit++;
               break;

            case OP_BUYSTOP:
               buyStop++;
               break;

            case OP_SELLSTOP:
               sellStop++;
               break;
         }
         totalOrders++;
      }
   }
```

First, we initialize several `int` variables to hold the open order counts. The heart of the function consists of a `for` loop that loops through all open orders in the order pool. We check the magic number on the current order to see if it is one that was placed by our expert advisor. Note the `CTrade::GetMagicNumber()` function call. This is how you call a static function in another class without creating a class object.

If the magic number matches, we check the order type and increment the correct integer variable. The `totalOrders` variable is incremented for every order that we count.

```
// Return order count based on pType
int returnTotal = 0;
switch(pType)
{
   case COUNT_BUY:
      returnTotal = buy;
      break;

   case COUNT_SELL:
      returnTotal = sell;
      break;

   case COUNT_BUY_LIMIT:
      returnTotal = buyLimit;
      break;

   case COUNT_SELL_LIMIT:
      returnTotal = sellLimit;
      break;

   case COUNT_BUY_STOP:
      returnTotal = buyStop;
      break;

   case COUNT_SELL_STOP:
      returnTotal = sellStop;
      break;

   case COUNT_MARKET:
      returnTotal = buy + sell;
      break;

   case COUNT_PENDING:
      returnTotal = buyLimit + sellLimit + buyStop + sellStop;
      break;
```

```
            case COUNT_ALL:
               returnTotal = totalOrders;
               break;
         }

         return(returnTotal);
      }
```

After counting all of the open orders by type, we check the value of the `pType` parameter and calculate and return the correct order count. For the `COUNT_MARKET` and `COUNT_PENDING` `case` labels, we add several variables together and return the result.

As before, we've made several public wrapper functions to call `CountOrders()` with the relevant order type. The return values of each function should be self-explanatory. The `TotalMarket()` function returns a count of all market orders, `TotalPending()` returns a count of all pending orders, and `TotalOrders()` returns a count of all orders, market and pending:

```
int Buy();
int Sell();
int BuyStop();
int SellStop();
int BuyLimit();
int SellLimit();
int TotalMarket();
int TotalPending();
int TotalOrders();
```

We can use the `CCount` class and its functions anywhere we need to get a count of open orders:

```
// Include file and object initialization
#include <Mql4Book\Trade.mqh>
CTrade Trade;
CCount Count;

// Buy order block in OnTick()
if(close > ma && Count.Buy() == 0 && gBuyTicket == 0)
{
   // Close sell order(s)
   Trade.CloseAllSellOrders();

   // Open buy order
   gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
   gSellTicket = 0;
```

```
      // Add stop loss & take profit to order
      Trade.ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
   }
```

An object of the `CCount` class will need to be declared on the global scope. We'll name our object `Count`. By calling the appropriate class function, we can return a count of open orders of that type without any additional code.

Chapter 17 - A Simple Expert Advisor, Revisited

In Chapter 12, we used the techniques that we had learned in earlier chapters to code a simple trading system that opened, closed and modified orders. In the last four chapters, we have demonstrated how to create reusable trading functions that will greatly simplify the creation of expert advisors. We will demonstrate by replacing much of the code in our Simple Expert Advisor with the functions that we have created.

Below is the entire file, located in `\MQL4\Experts\Mql4Book\Simple Expert Advisor with Functions.mq4`. The changes are highlighted in bold:

```
#property strict

// Include and objects
#include <Mql4Book\Trade.mqh>
CTrade Trade;
CCount Count;


// Input variables
input int MagicNumber = 101;
input int Slippage = 10;

input double LotSize = 0.1;
input int StopLoss = 0;
input int TakeProfit = 0;

input int MaPeriod = 5;
input ENUM_MA_METHOD MaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;


// Global variables
int gBuyTicket, gSellTicket;


// OnInit() event handler
int OnInit()
{
   // Set magic number
   Trade.SetMagicNumber(MagicNumber);
   Trade.SetSlippage(Slippage);

   return(INIT_SUCCEEDED);
}
```

```
// OnTick() event handler
void OnTick()
{
   // Moving average and close price from last bar
   double ma = iMA(_Symbol,_Period,MaPeriod,0,MaMethod,MaPrice,1);
   double close = Close[1];

   // Buy order condition
   if(close > ma && Count.Buy() == 0 && gBuyTicket == 0)
   {
      // Close sell order(s)
      Trade.CloseAllSellOrders();

      // Open buy order
      gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
      gSellTicket = 0;

      // Add stop loss & take profit to order
      ModifyStopsByPoints(gBuyTicket,StopLoss,TakeProfit);
   }

   // Sell order condition
   if(close < ma && Count.Sell() == 0 && gSellTicket == 0)
   {
      // Close buy order(s)
      Trade.CloseAllBuyOrders();

      // Open sell order
      gSellTicket = Trade.OpenSellOrder(_Symbol,LotSize);
      gBuyTicket = 0;

      // Add stop loss & take profit to order
      ModifyStopsByPoints(gSellTicket,StopLoss,TakeProfit);
   }
}
```

This works identically to the expert advisor we created in chapter 12. However, the code is much shorter and easier to read, because the implementation details are hidden in the classes and functions contained in the `Trade.mqh` include file. We have also added additional functionality, including retry on error and enhanced logging and informational capabilities.

Let's go through this file a section at a time. You may want to compare this file to the one we created in chapter 12:

```
// Include and objects
#include <Mql4Book\Trade.mqh>
CTrade Trade;
CCount Count;

// Input variables
input int MagicNumber = 101;
input int Slippage = 10;

input double LotSize = 0.1;
input int StopLoss = 0;
input int TakeProfit = 0;

input int MaPeriod = 5;
input ENUM_MA_METHOD MaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;


// Global variables
int gBuyTicket, gSellTicket;
```

We've included the `Trade.mqh` file from the `\MQL4\Include\Mql4Book` folder and created two class objects, `Trade`, based on the `CTrade` class, and `Count`, based on the `CCount` class. The input and global variables are the same from the original program.

```
// OnInit() event handler
int OnInit()
{
   // Set magic number
   Trade.SetMagicNumber(MagicNumber);
   Trade.SetSlippage(Slippage);

   return(INIT_SUCCEEDED);
}
```

We've added the `OnInit()` event handler to this program. The `SetMagicNumber()` and `SetSlippage()` functions set the magic number and slippage parameters for our order operations.

```
// OnTick() event handler
void OnTick()
{
   // Moving average and close price from last bar
   double ma = iMA(_Symbol,_Period,MaPeriod,0,MaMethod,MaPrice,1);
   double close = Close[1];
```

```
      // Buy order condition
      if(close > ma && Count.Buy() == 0 && gBuyTicket == 0)
      {
         // Close sell order(s)
         Trade.CloseAllSellOrders();

         // Open buy order
         gBuyTicket = Trade.OpenBuyOrder(_Symbol,LotSize);
         gSellTicket = 0;

         // Add stop loss & take profit to order
         Trade.ModifyStopsByPoints(gBuyTicket,StopLoss,TakeProfit);
      }

      // Sell order condition
      if(close < ma && Count.Sell() == 0 && gSellTicket == 0)
      {
         // Close buy order(s)
         Trade.CloseAllBuyOrders();

         // Open sell order
         gSellTicket = Trade.OpenSellOrder(_Symbol,LotSize);
         gBuyTicket = 0;

         // Add stop loss & take profit to order
         Trade.ModifyStopsByPoints(gSellTicket,StopLoss,TakeProfit);
      }
   }
```

Above is the contents of the `OnTick()` event handler. We use the `Count.Buy()` and `Count.Sell()` functions to return the current open order counts. The `Trade.CloseAllBuyOrders()` and `Trade.CloseAllSellOrders()` functions close any currently opened trade. The `Trade.OpenBuyOrder()` and `Trade.OpenSellOrder()` functions handle order placement, and `ModifyStopsByPoints()` is used to calculate and add the stop loss and take profit prices to the order.

In Chapter 23, we will create several trading systems using the techniques we have just shown here.

Chapter 18 - Bar and Price Data

In this chapter, we will examine how to use price data in your expert advisors. We will show you how to retrieve the current Bid and Ask prices for a symbol, as well as retrieving the open, high, low and close prices for each bar. We will demonstrate how to copy price data to arrays, determine the highest and lowest price of a price series, and look at price-based trading signals using candlestick patterns.

Retrieving Current Prices

In previous chapters, we introduced the predefined variables `Bid` and `Ask` to retrieve the current Bid and Ask prices for the current chart symbol. The value of these variables is set each time the `OnTick()` event handler is called.

If you are performing multiple order operations, especially inside a loop, it will be necessary to refresh the contents of the predefined variables by calling the `RefreshRates()` function. Here's an example of a loop that modifies a trailing stop on all open orders. At the top of the loop, we call the `RefreshRates()` function to ensure that we are always using the most current prices:

```
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   OrderSelect(order,SELECT_BY_POS);

   RefreshRates();

   if(OrderType() == OP_BUY)
   {
      double trailStopPrice = Bid - (TrailingStop * _Point);
      ModifyOrder(OrderTicket(),0,trailStopPrice,OrderTakeProfit());
   }

   else if(OrderType() == OP_SELL)
   {
      double trailStopPrice = Ask + (TrailingStop * _Point);
      ModifyOrder(OrderTicket(),0,trailStopPrice,OrderTakeProfit());
   }
}
```

On each iteration of the `for` loop, the `RefreshRates()` function is called, ensuring that the `Bid` and `Ask` variables contain the most current prices.

Retrieving Current Prices with MarketInfo()

The predefined `Bid` and `Ask` variables only contain the prices for the current chart symbol. If you need to retrieve a current price on another symbol, use the `MarketInfo()` function. To retrieve the current Bid price for a symbol, call `MarketInfo()` with the `MODE_BID` parameter. To retrieve the current Ask price, call `MarketInfo()` with the `MODE_ASK` parameter:

```
string symbol = “EURUSD”;
double bid = MarketInfo(symbol,MODE_BID);
double ask = MarketInfo(symbol,MODE_ASK);
```

The example above returns the current Bid and Ask prices for EURUSD and stores them in the `bid` and `ask` variables respectively.

Bar Data

There are several predefined arrays that contain the OHLC data for the current chart. The `Open[]`, `High[]`, `Low[]` and `Close[]` arrays can be used to retrieve the prices for each bar on the chart. An index of 0 indicates the current bar, an index of 1 is the previous bar, and so on.

The examples below demonstrates how to use predefined price arrays to find patterns. This example compares the high and low prices of the last two bars, and prints a message to the log if the most recent bar has a higher high or lower low than the previous bar:

```
if(High[1] > High[2])
{
   Print("New higher high");
}

if(Low[1] < Low[2])
{
   Print("New lower low");
}
```

You can compare the open and close prices of the same bar to see if it is a bullish or bearish candle:

```
if(Close[1] > Open[1])
{
   Print("Bullish candle");
}
```

```
else if(Close[1] < Open[1])
{
   Print("Bearish candle");
}
```

If you need to retrieve bar prices from another symbol or timeframe, you can use the `iClose()`, `iOpen()`, `iHigh()` and `iLow()` functions. The function definition for `iClose()` is below:

```
double iClose(string symbol, int timeframe, int shift);
```

The `symbol` parameter is the symbol to retrieve the close price for. The `timeframe` parameter is the chart period. You can use the number of minutes in the period, or you can use the constants in the `ENUM_TIMEFRAMES` enumeration:

PERIOD_CURRENT	0	Current timeframe
PERIOD_M1	1	1 minute
PERIOD_M5	5	5 minutes
PERIOD_M15	15	15 minutes
PERIOD_M30	30	30 minutes
PERIOD_H1	60	1 hour
PERIOD_H4	240	4 hours
PERIOD_D1	1440	1 day
PERIOD_W1	10080	1 week

The `shift` parameter is the number of bars from the current bar to retrieve the price for – 0 is the current bar, 1 is the previous bar, etc.

Here's an example of how to use the `iClose()` function to get the close price of the previous bar on a specified timeframe:

```
double h4Close = iClose(_Symbol,PERIOD_H4,1);
```

The example above retrieves the close price of the previous bar on the H4 timeframe for the current chart symbol. You can also specify a symbol other than the current chart symbol to retrieve a price from:

```
string symbol = “EURUSD”;
double close = iClose(symbol,PERIOD_H1,1);
```

The example above will retrieve the close price for the previous bar on the H1 timeframe for EURUSD.

Copy...() Functions

If you prefer to copy price data into an array, you can use one of the following functions. These functions are new to MQL4:

- **`double CopyClose()`** - Copies the close prices into a specified array.
- **`double CopyOpen()`** - Copies the open prices into a specified array.
- **`double CopyHigh()`** - Copies the high prices into a specified array.
- **`double CopyLow()`** - Copies the low prices into a specified array.

To use these functions, first declare a dynamic array of type `double`. Next, set it as a series array using `ArraySetAsSeries()`. Then use the appropriate `Copy...()` function to copy data into the array.

We'll demonstrate using the `CopyClose()` function. Here is the function definition for `CopyClose()`:

```
int CopyClose(
   string symbol_name,              // symbol name
   ENUM_TIMEFRAMES timeframe,       // period
   int start_pos,                   // start position
   int count,                       // data count to copy
   double close_array[]             // target array to copy
  );
```

The `symbol_name` parameter is the symbol to copy data for, `timeframe` is the chart period, `start_pos` is the starting position to copy from, `count` is the number of bars to copy to the array, and `close_array` is the array to copy the data to. There are several variants of the function that allow you to specify start and end dates as well. You can view these in the *MQL4 Reference* under *Timeseries and Indicator Access*.

To use the `Copy...()` functions, you'll need to create a series array. A series array is an array where the indexing is reversed, so the most recent value is indexed at zero. To create a series array, first declare a dynamic array of type `double`, then set it as a series array using the `ArraySetAsSeries()` function:

```
double close[];
ArraySetAsSeries(close,true);
```

Remember, to declare a dynamic array, simply leave the brackets empty. The example above declares a dynamic array named `close[]`. The `ArraySetAsSeries()` function sets the `close[]` array as a series array. If the second parameter of the function is set to `true`, the array name in the first parameter is set as a series array.

Here's an example of the usage of the `CopyClose()` function:

```
double close[];
ArraySetAsSeries(close,true);

CopyClose(_Symbol,_Period,0,3,close);

double currentClose = close[0];
```

The `CopyClose()` function copies the close prices of the last three bars from the current chart period and symbol into the `close[]` array. Typically, you will not need more than a few bars worth of data. To retrieve the current bar's close price, we reference the zero index of the `close[]` array. As long as you use a start position of zero when copying price data to your array, the copied price data will function identically to the predefined price arrays from earlier in the chapter.

Candlestick Patterns

The practice of using candlesticks to determine trader psychology was popularized by authors such as Steve Nisson. We can use price arrays to locate candlestick patterns on the chart.

A common reversal pattern in Forex trading is the engulfing pattern. An engulfing candle is one that completely engulfs the body of the previous candle. If it occurs near the top or bottom of a trend, it can be a powerful reversal pattern.

True engulfing candles rarely occur in Forex, as the opening price of a candle is typically the same (or 1-2 points off) as the close price of the previous candle. What we are looking for is a candle that is trending in the opposite direction of the previous candle, and where the close price is greater or less than the open price of the previous candle.

Fig 18.1 – Engulfing pattern

Here is an example of how to locate a bullish engulfing pattern:

```
if(Close[1] > Open[2] && Close[1] > Open[1] && Close[2] < Open[2])
{
   Alert("Bullish engulfing pattern detected!");
}
```

We are looking for the following conditions to be true:

- The close of the most recent bar is greater than the open of the previous bar.
- The close of the most recent bar is greater than its open, indicating that the candle is bullish.
- The close of the previous bar is less than its open, indicating that the candle is bearish.

You can also examine the high and low prices of a bar to find various candlestick patterns. Take the hammer/hanging man pattern, for example. This pattern has a long lower shadow that is at least twice the size of the body, and little or no upper shadow. If a hammer candle occurs near the bottom of a downtrend, it can be a powerful reversal signal.

Here's an example of how to locate a hammer/hanging man candle. We are assuming that the lower shadow is at least twice the size of the body, and the upper shadow is smaller than the body:

```
double body = MathAbs(Close[1] - Open[1]);
double lowerShadow, upperShadow;

if(Close[1] < Open[1])  // Bearish
{
   lowerShadow = Close[1] - Low[1];
   upperShadow = High[1] - Open[1];
}
else                    // Bullish
{
   lowerShadow = Open[1] - Low[1];
   upperShadow = High[1] - Close[1];
}

if(lowerShadow * 2 > body && upperShadow < body)
{
   Alert("Hammer/hanging man candle detected!");
}
```

Fig 18.2 – Hammer candle

We calculate the size of the candle body by subtracting the open from the close, and using the `MathAbs()` function to return the absolute value in points. Next, we need to determine whether the body of the candle is bullish or bearish. This will allow us to accurately calculate the size of the upper and lower shadows.

If the size of the lower shadow, multiplied by 2, is greater than the body, and the size of the upper shadow is less than the body, we will conclude that this is a hammer/hanging man pattern.

By using price arrays, you can detect potential candlestick patterns and trading setups. There is a large degree of discretion when it comes to trading with candlesticks, so you may not want to open trades using candlesticks alone. But with manual confirmation or trend-detection methods, price action can be a good tool for making trading decisions.

Highest and Lowest Prices

If you need to find the highest or lowest price over a specified number of bars, the `iHighest()` and `iLowest()` functions will return the index of the bar with the highest or lowest value. For example, you may

want to use the highest high or lowest low of the last *x* bars to set a stop loss, or to determine a pending order price.

Here is the function definition for `iHighest()`:

```
int  iHighest(
   string symbol,       // symbol
   int timeframe,       // timeframe
   int type,            // timeseries
   int count,           // count of bars to copy
   int start            // start position
  );
```

The `symbol` parameter is the symbol to use, `timeframe` is the chart period, and `type` is the price series to use. Below is a list of price series identifiers to use for the `type` parameter:

ID	Value	Description
MODE_OPEN	0	Open price
MODE_LOW	1	Low price
MODE_HIGH	2	High price
MODE_CLOSE	3	Close price

The `count` parameter is the number of bars to search, and the `start` parameter is the start position to search from. Here's an example of how to use `iHighest()` to find the highest high of the last ten bars:

```
int shift = iHighest(_Symbol,_Period,MODE_HIGH,10,0);
double highestHigh = High[shift];
```

The `MODE_HIGH` parameter indicates that we are searching for the highest high price. You can also use this function to find the highest open, close or low if you wish – simply use the appropriate value for the `type` parameter. We are counting back ten bars, starting from the current bar. The `iHighest()` function returns the index of the bar that contains the highest high – not the price itself! We store this value in the `shift` variable, and then pass that value into the predefined `High[]` price array. The `highestHigh` variable will contain the highest high price of the last ten bars.

The `iLowest()` function works similarly to the `iHighest()` function, except that it locates the lowest price in a series:

```
int shift = iLowest(_Symbol,_Period,MODE_LOW,10,0);
double lowestLow = Low[shift];
```

This example returns the lowest low of the last ten bars for the current chart period and symbol.

The `iHighest()` and `iLowest()` functions can be used to determine recent swing highs and lows. You can use this information to place breakout trades, or to place a stop loss on your orders. In Chapter 23, we will create a trading system that uses swing highs and lows to place trades and stops.

Chapter 19 - Using Indicators in Expert Advisors

Many expert advisors uses indicators to generate trading signals. In this chapter, we will examine how to use MetaTrader's built-in indicators in your expert advisors, as well as custom indicators that you can find online or in MetaTrader 4's `\MQL4\Indicators\Examples` folder. We will create classes to simplify the use of indicators, and examine some common indicator-based trading signals.

Single Buffer Indicators

Earlier in this book, we used a moving average indicator in our simple expert advisor. Here is the code to add the moving average indicator to an expert advisor:

```
// Input variables
input int MaPeriod = 10;
input int MaShift = 0;
input ENUM_MA_METHOD MaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;

// OnTick() event handler
double ma = iMA(_Symbol,_Period,MaPeriod,MaShift,MaMethod,MaPrice,1);
```

The input variables include four settings to adjust the moving average parameters: `MaPeriod` adjusts the period of the indicator. `MaShift` shifts the indicator forward or backward on the chart. `MaMethod` changes the calculation method of the indicator (simple, exponential, etc.), and `MaPrice` selects the price series used to calculate the moving average (close, open, etc.).

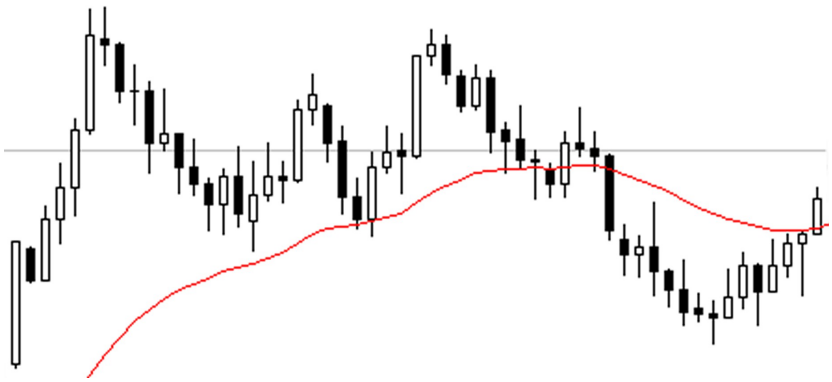


Fig 19.1 – A 30 period exponential moving average on a daily chart.

Note that you can name these input variables whatever you wish. We've simply chosen some descriptive and consistent names for our input variables.

In the `OnTick()` event handler, we call the `iMA()` function to retrieve the moving average price from the previous bar for the current chart symbol and period. Every built-in MetaTrader indicator has a similar function. You can view them in the *MQL4 Reference* under *Technical Indicators*.

Here is the function definition for the `iMA()` indicator:

```
double iMA(
   string symbol,       // symbol name
   int timeframe,       // period
   int ma_period,       // averaging period
   int ma_shift,        // horizontal shift
   int ma_method,       // smoothing type
   int applied_price    // price series
   int shift
 );
```

Every technical indicator function begins with the `symbol` and `timeframe` parameters. The `symbol` parameter is the name of the symbol to calculate the indicator for. You can use the predefined `_Symbol` variable, or another variable that contains a valid symbol name. The `timeframe` parameter is the chart period to calculate for. You can use the predefined `_Period` variable, or a value from the `ENUM_TIMEFRAMES` enumeration.

The remaining parameters depend on the indicator. The moving average indicator has four parameters that can be adjusted – `ma_period`, `ma_shift`, `ma_method` and `applied_price`. Some indicators have just one or two parameters, and a few have none at all. The last parameter, `shift`, is the bar to calculate the indicator value for. 0 is the current bar, 1 is the previous bar, and so on.

Here are the input variables for the moving average indicator again:

```
// Input variables
input int MaPeriod = 10;
input int MaShift = 0;
input ENUM_MA_METHOD MaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;
```

Note how we use the `ENUM_MA_METHOD` and `ENUM_APPLIED_PRICE` types for the `MaMethod` and `MaPrice` parameters. These are enumerations that hold the correct values for the `ma_method` and `applied_price` parameters of the `iMA()` function. Any indicator function that has the `ma_method` or `applied_price` parameter can use `ENUM_MA_METHOD` or `ENUM_APPLIED_PRICE` values as input.

The *MQL4 Reference* will indicate if any parameters in a function will accept an enumeration as input. Using these enumeration types for our input variables has the added benefit of giving us a user-friendly drop-down box in the *Expert Properties* dialog:

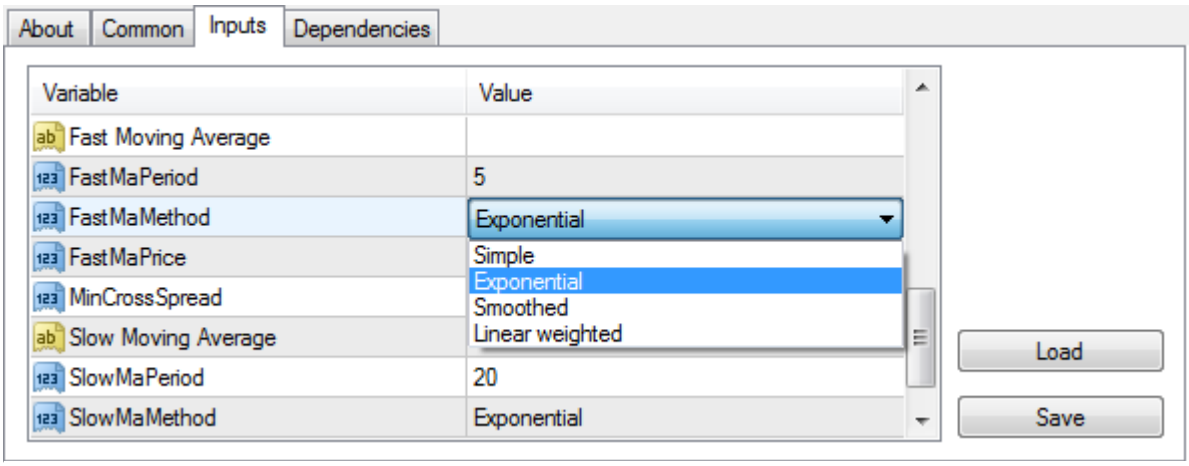


Fig 19.2 – The `ENUM_MA_METHOD` enumeration values as they appear in the *Inputs* tab

Let's take a look at a second indicator that only uses a single buffer. The RSI indicator is very popular, and is often used to locate price extremes. Here is the function definition for the RSI indicator:

```
double iRSI(
   string symbol,       // symbol
   int timeframe,       // timeframe
   int period,          // period
   int applied_price,   // applied price
   int shift            // shift
 );
```

Just like the `iMA()` function, the first two parameters are `symbol` and `period`. The RSI takes two input parameters, one for the averaging period (`period`) and another for the price series (`applied_price`).

Fig 19.3 – The RSI indicator.

Here's how we would add the RSI indicator to an expert advisor:

```
// Input parameters
input int RSIPeriod = 10;
input ENUM_APPLIED_PRICE RSIPrice = PRICE_CLOSE;
```

```
// OnTick() event handler
double rsi = iRSI(_Symbol,_Period,RSIPeriod,RSIPrice,1);
```

We declare the input variables `RSIPeriod` and `RSIPrice` with the appropriate types at the beginning of our program. Inside the `OnTick()` event handler, we call the `iRSI()` function and calculate the RSI value for the previous bar for the current chart symbol and period.

Multi-Buffer Indicators

When using indicators with multiple lines (and multiple buffers), we'll need to call the relevant indicator function multiple times, once for each line of the indicator. Let's start with the stochastic indicator. The stochastic is an oscillator similar to the RSI. Along with the main stochastic line, a second signal line is also present.

Here is the function definition for `iStochastic()`:

```
double iStochastic(
   string symbol,        // symbol
   int timeframe,        // timeframe
   int Kperiod,          // K line period
   int Dperiod,          // D line period
   int slowing,          // slowing
   int method,           // averaging method
   int price_field,      // price (Low/High or Close/Close)
   int mode,             // line index
   int shift             // shift
  );
```

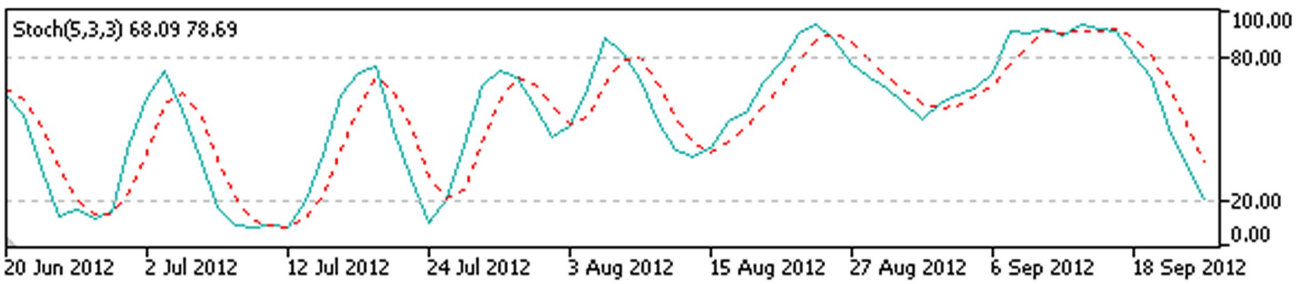


Fig. 19.4 – The stochastic indicator. The red dashed line is the signal line.

The stochastic indicator has five input parameters: `Kperiod`, `Dperiod`, `slowing`, `method` and `price_field`. The `mode` parameter indicates which line to calculate. There are two valid values for the `mode` parameter on the stochastic indicator: `MODE_MAIN`, which is the main stochastic line, and `MODE_SIGNAL`, which is the signal line. A

listing of all indicator line constants can be found in the *MQL4 Reference* under *Standard Constants... > Indicator Constants > Indicator Lines*.

Here is how we would add the stochastic indicator to an expert advisor:

```
// Input parameters
input int KPeriod = 10;
input int DPeriod = 3;
input int Slowing = 3;
input ENUM_MA_METHOD StochMethod = MODE_SMA;
input ENUM_STO_PRICE StochPrice = STO_LOWHIGH;

// OnTick() event handler
double stochastic = iStochastic(_Symbol,_Period,KPeriod,DPeriod,Slowing,StochMethod,
   StochPrice,MODE_MAIN,1);
double signal = iStochastic(_Symbol,_Period,KPeriod,DPeriod,Slowing,StochMethod,
   StochPrice,MODE_SIGNAL,1);
```

Note the `ENUM_MA_METHOD` and `ENUM_STO_PRICE` types for the `StochMethod` and `StochPrice` variables. We've discussed `ENUM_MA_METHOD` in the previous section. The `ENUM_STO_PRICE` enumeration contains the valid price fields for the stochastic indicator.

We call the `iStochastic()` function twice – first using `MODE_MAIN` as the `mode` parameter, and then using `MODE_SIGNAL` as the `mode` parameter. This will calculate both the main and the signal lines for the stochastic indicator.

All multi-buffer indicators work the same way. You will need to call the indicator function for each line that is drawn by the indicator by changing the `mode` parameter to the appropriate line.

The CIndicator Class

We're going to create classes to simplify the process of adding indicators to a program. We'll start by creating a base class that contains the variables shared by every indicator. Then we will create new classes derived from this base class that address the specific needs of each indicator.

All of the built-in indicators have parameters to set the symbol and chart period. Our base class will contain protected variables to hold these values. As you may recall, *protected* members of a class are similar to private members, with the exception that derived classes will inherit them. We're going to name our base class `CIndicator`. It will be placed in the `\MQL4\Include\Mql4Book\Indicators.mqh` include file. This file will be used to hold all of our indicator related classes and functions.

Here is the class declaration for the `CIndicator` class:

```
class CIndicator
{
   protected:
      string _symbol;
      int _timeFrame;
      int _digits;

   void Init(string pSymbol, int pTimeFrame);
};
```

The `CIndicator` class has three protected variables and one protected function. The `_symbol` variable will hold the symbol name for our indicator, `_timeFrame` will contain the chart period, and `_digits` contains the number of digits in the symbol price. All three of these variables will be used in our indicator classes.

The `Init()` function is a protected function that will be used to set the values of the protected variables in the `CIndicator` class. We will call this function from the constructor of our derived indicator classes. Here is the function body for `CIndicator::Init()`:

```
void CIndicator::Init(string pSymbol,int pTimeFrame)
{
   _symbol = pSymbol;
   _timeFrame = pTimeFrame;
   _digits = (int)MarketInfo(pSymbol,MODE_DIGITS);
}
```

The `Init()` function has two parameters to set the symbol and chart timeframe. We set the `_symbol` and `_timeFrame` variables to these parameters, and call the `MarketInfo()` function with the `MODE_DIGITS` parameter to return the number of decimal places in the symbol quote. These variables will be used later in our derived classes.

Derived Classes

The `CIndicator` class will be the basis for derived classes that will be used to initialize and retrieve data from MetaTrader's built-in indicators. Since the implementation of each indicator will be different, we will need to create a separate class for each indicator that we want to use.

Let's start with the moving average indicator. The moving average has just one buffer, so we can create a derived class with a minimum of effort. Here is the class declaration for the `CiMA` class:

```
class CiMA : CIndicator
{
   private:
      int _maPeriod;
      int _maShift;
      int _maMethod;
      int _appliedPrice;

   public:
      CiMA(string pSymbol, int pTimeFrame, int pMaPeriod, int pMaShift, int pMaMethod,
         int pAppliedPrice);
      double Main(int pShift = 0);
};
```

Note the : followed by `CIndicator` after the class name. This indicates that the `CiMA` class is derived from `CIndicator`. The `CiMA` class inherits all of the public and protected functions and variables from the `CIndicator` class.

The `CiMA` class has four `private` variables to hold the moving average indicator settings. It also has two `public` functions. The `CiMA()` function is the class constructor, and it has six parameters. When creating an object based on the `CiMA` class, we must specify these parameters after the object name. The `Main()` function retrieves the moving average value for a specified bar.

Here is the code for the `CiMA()` class constructor:

```
CiMA::CiMA(string pSymbol,int pTimeFrame,int pMaPeriod,int pMaShift,int pMaMethod,
   int pAppliedPrice)
{
   Init(pSymbol, pTimeFrame);

   _maPeriod = pMaPeriod;
   _maMethod = pMaMethod;
   _maShift = pMaShift;
   _appliedPrice = pAppliedPrice;
}
```

When creating an object based on the `CiMA` class, this function will be called automatically. The constructor has six parameters that must be specified when creating the object. The first thing we do in every constructor that we create for our indicator classes is to call the `Init()` function from the `CIndicator` class. This sets the `_symbol`, `_timeFrame` and `_digits` variables that we inherited from the base class. Next, we assign the input parameters to the private class variables. These will be used when calculating the indicator value.

The `Main()` function of the `CiMA` class returns the value of the indicator for the specified bar. Here is the code for the `Main()` function:

```
double CiMA::Main(int pShift = 0)
{
   double value = iMA(_symbol,_timeFrame,_maPeriod,_maShift,_maMethod,_appliedPrice,pShift);
   return NormalizeDouble(value,_digits);
}
```

The `pShift` parameter is the bar that we want to calculate the indicator value for. The default value of `pShift` is zero, so calling `Main()` with no parameters will return the indicator value for the current bar.

The `Main()` function simply calls the `iMA()` function, using the private class variables that we set earlier in our class constructor as the parameters for the function. The resulting indicator price is normalized to the number of digits in the quote using the `NormalizeDouble()` function, and returned to the program.

Moving Average Class Example

Here is how we would use the `CiMA` class (and its parent class, `CIndicator`) in an expert advisor:

```
#include <Mql4Book\Indicators.mqh>

// Input variables
input int MaPeriod = 10;
input ENUM_MA_METHOD MAMethod = MODE_EMA;
input int MaShift = 0;
input ENUM_APPLIED_PRICE MaPrice = PRICE_CLOSE;

// Create object based on the CiMA class
CiMA ma(_Symbol,_Period,MaPeriod,MaShift,MaMethod,MaPrice);

// OnTick() event handler
double currentMa = ma.Main();
double previousMa = ma.Main(1);
```

First, we include the `Indicators.mqh` file at the top of the program. Next are the input variables for a moving average indicator. Below that, on the global scope of the program, we create our moving average indicator object based on the `CiMA` class. The name of the object is `ma`, and we have passed in the parameters for the class constructor.

Inside the `OnTick()` event handler (or anywhere else in our program where we might need to access the moving average indicator), we simply call the `Main()` function of our `ma` object to retrieve the moving average value for that bar. Calling `Main()` without any parameters retrieves the moving average value for the current bar. We save this value to the `currentMa` variable. The `previousMa` variable contains the moving average value for the previous bar.

Using our indicator classes allows us to pass the input parameters to the indicator function only once – when the object is created. We can then access indicator values using short and easy-to-remember functions. If you are using a simple built-in indicator with only one buffer, you may prefer to use the classic method detailed earlier in the chapter. But if you're using an indicator with multiple buffers, or if you need to access indicator values for multiple bars, then using these indicator classes is a much easier method.

Stochastic Indicator Class

Let's create a second indicator class, using an indicator with multiple buffers. We discussed the stochastic indicator earlier in the chapter, which has two buffers. Here is the class declaration for the `CiStochastic` class:

```
class CiStochastic : CIndicator
{
   private:
      int _kPeriod;
      int _dPeriod;
      int _slowing;
      int _method;
      int _appliedPrice;

   public:
      CiStochastic(string pSymbol, int pTimeFrame, int pKPeriod, int pDPeriod, int pSlowing,
         int pMethod, int pAppliedPrice);
      double Main(int pShift = 0);
      double Signal(int pShift = 0);
};
```

The `CiStochastic` class has five private variables, all of which hold the stochastic indicator settings. There are three public functions: the class constructor, the `Main()` function and the `Signal()` function.

Here is the class constructor for the `CiStochastic` class:

```
void CiStochastic::CiStochastic(string pSymbol,int pTimeFrame,int pKPeriod,int pDPeriod,
   int pSlowing,int pMethod,int pAppliedPrice)
{
      Init(pSymbol,pTimeFrame);

      _kPeriod = pKPeriod;
      _dPeriod = pDPeriod;
      _slowing = pSlowing;
      _method = pMethod;
      _appliedPrice = pAppliedPrice;
}
```

Just like our `CiMA` class, the `CiStochastic` class constructor calls the `Init()` function of the `CIndicator` class to set the `_symbol`, `_period` and `_digits` variables, and the remainder of the function assigns the input parameters to our private class variables.

And here are the `Main()` and `Signal()` functions:

```
double CiStochastic::Main(int pShift=0)
{
   double value = iStochastic(_symbol,_timeFrame,_kPeriod,_dPeriod,_slowing,_method,
      _appliedPrice,MODE_MAIN,pShift);
   return NormalizeDouble(value,_digits);
}

double CiStochastic::Signal(int pShift=0)
{
   double value = iStochastic(_symbol,_timeFrame,_kPeriod,_dPeriod,_slowing,_method,
      _appliedPrice,MODE_SIGNAL,pShift);
   return NormalizeDouble(value,_digits);
}
```

Both functions call the MQL4 `iStochastic()` function, using our private class variables as the parameters. The only difference between the two functions is the indicator line that we are calculating. `Main()` calls `iStochastic()` with the `MODE_MAIN` parameter, while `Signal()` calls `iStochastic()` with the `MODE_SIGNAL` parameter.

Stochastic Indicator Class Example

We add the stochastic indicator to our expert advisor the same way we added the moving average – create the stochastic indicator object by calling the class constructor with the appropriate parameters, and then use the `Main()` and `Signal()` functions to calculate the indicator data:

```
#include <Mql4Book\Indicators.mqh>

// Input variables
input int KPeriod = 10;
input int DPeriod = 3;
input int Slowing = 3;
input ENUM_MA_METHOD StochMethod = MODE_SMA;
input ENUM_STO_PRICE StochPrice = STO_LOWHIGH;

// Create a CiStochastic class object
CiStochastic stoch(_Symbol,_Period,KPeriod,DPeriod,Slowing,StochMethod,StochPrice);
```

```
// OnTick() event handler
double currentStoch = stoch.Main();
double currentSignal = stoch.Signal();
```

We create an object based on the `CiStochastic` class named `stoch`. To retrieve the indicator values, we call the `Main()` and `Signal()` functions of the `stoch` object. The `currentStoch` and `currentSignal` variables contain the *%K* and *%D* line values for the current bar.

The `\MQL4\Include\Mql4Book\Indicators.mqh` file contains classes for the most popular built-in indicators for MetaTrader 4. If you need to create a class for an indicator that is not listed, you can do so using the techniques listed in this chapter.

Custom Indicators

You can also use custom indicators in your expert advisors. To use a custom indicator, you will need to determine the number of buffers in the indicator, as well as their usage. You will also need to determine the names and types of the indicator parameters.

You can download custom indicators online through the *MQL4 Codebase*, the *MQL4 Market*, or from MetaTrader-related forums and websites. It will be much easier to work with a custom indicator if you have the `.mq4` file for it. Even if you only have the `.ex4`, it is still possible to use the indicator in your project, though it will take a bit more detective work.

Let's use one of the custom indicators that come standard with MetaTrader 4. The *Custom Indicators* tree in the *Navigator* window contains custom indicator examples of many built-in MetaTrader indicators, as well as a few extras. We'll use the *Bands* custom indicator, which plots Bollinger Bands on the chart.

When examining a custom indicator, open the *Data window* (*Ctrl+D*) to see how many lines are plotted on the chart, as well as the labels that are assigned to them, if any. The Bands indicator has three lines, which are labeled *Bands SMA*, *Bands Upper* and *Bands Lower*.

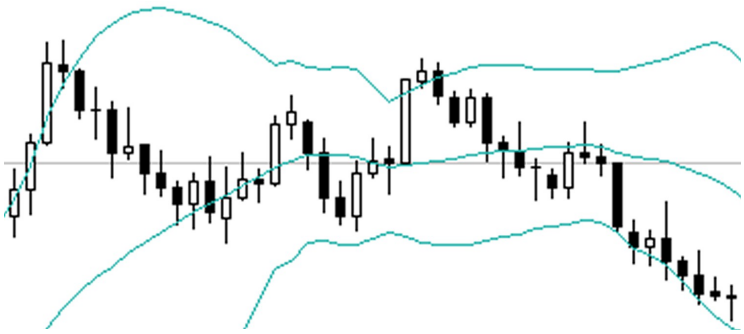


Fig. 19.5 – The Bands custom indicator.

Open the `MQL4\Indicators\Bands.mq4` file in MetaEditor. We need to find out how many buffers or plots the indicator uses, as well as the relevant buffer numbers. Look in the `OnInit()` event handler for the `SetIndexBuffer()` functions. This assigns the arrays defined in the program to specific indicator buffer numbers:

```
//--- 1 additional buffer used for counting.
   IndicatorBuffers(4);
   IndicatorDigits(Digits);
//--- middle line
   SetIndexStyle(0,DRAW_LINE);
   SetIndexBuffer(0,ExtMovingBuffer);
   SetIndexShift(0,InpBandsShift);
   SetIndexLabel(0,"Bands SMA");
//--- upper band
   SetIndexStyle(1,DRAW_LINE);
   SetIndexBuffer(1,ExtUpperBuffer);
   SetIndexShift(1,InpBandsShift);
   SetIndexLabel(1,"Bands Upper");
//--- lower band
   SetIndexStyle(2,DRAW_LINE);
   SetIndexBuffer(2,ExtLowerBuffer);
   SetIndexShift(2,InpBandsShift);
   SetIndexLabel(2,"Bands Lower");
//--- work buffer
   SetIndexBuffer(3,ExtStdDevBuffer);
```

The first parameter of the `SetIndexBuffer()` function is the buffer number, and the second parameter is the array that is used for that buffer. The indicator has three lines, but the code shows four buffers! Looking at the code above, we can conclude that buffer 0 is the middle line, buffer 1 is the upper band, and buffer 2 is the lower band. Buffer 3 is used for internal calculations. Most indicators won't be labeled as clearly as this one! If you're not sure, attach the indicator to a chart and view the data window. Most of the time, it will be obvious which buffer goes to which line.

Now that we know our buffer numbers, let's figure out the input parameters for the Bands indicator. We can find the input variables right near the top of the file:

```
input int InpBandsPeriod=20;                // Period
input int InpBandsShift=0;                  // Shift
input double InpBandsDeviations=2.0;        // Deviation
```

We have three input parameters – two `int` variables for the period and shift, and a `double` variable for the deviation. Feel free to copy and paste the input variables from the custom indicator file to your expert advisor file. You can change the names of the input variables in your program if you wish.

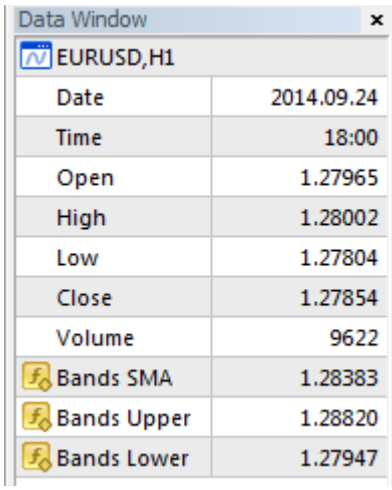


Fig. 19.6 – The Data Window.

The iCustom() Function

The `iCustom()` function works similarly to the built-in indicator functions examined previously in this chapter. Here is the function definition from the *MQL4 Reference*:

```
double iCustom(
   string symbol,            // symbol
   int timeframe,            // timeframe
   string name,              // path/name of the custom indicator compiled program
   ...                       // custom indicator input parameters (if necessary)
   int mode,                 // line index
   int shift                 // shift
  );
```

Just like the other technical indicator functions, the `iCustom()` parameters begin with `symbol` and `timeframe`. The `name` parameter is the indicator name, minus the file extension. The indicator file must be located in the `\MQL4\Indicators` folder. If the file is located in a subfolder, then the subfolder name must precede the indicator name, separated by a double slash (`\\`).

The . . . in the `iCustom()` function definition is where the input parameters for the custom indicator go. The Bands custom indicator has three input parameters. We need to pass those parameters to the `iCustom()` function in the order that they appear in the indicator file:

```
input int     InpBandsPeriod=20;       // Period
input int     InpBandsShift=0;         // Shift
input double  InpBandsDeviations=2.0;  // Deviation
```

We would need to pass three parameters, of type `int`, `int` and `double` to the `iCustom()` function. The `mode` parameter is the buffer number to calculate. As indicated above, we have three buffers in this indicator, numbered 0 to 3. Finally, the `shift` parameter is the bar to calculate the indicator value for.

Here's an example of how we would use the `iCustom()` function to add the Bands indicator to an expert advisor:

```
// Input variables
input int BandsPeriod = 20;       // Period
input int BandsShift = 0;         // Shift
input double BandsDeviation = 2;  // Deviation

// OnTick() event handler
double midleBand = iCustom(_Symbol,_Period,"Bands",BandsPeriod,BandsShift,
   BandsDeviation,0,1);
```

```
double upperBand = iCustom(_Symbol,_Period,"Bands",BandsPeriod,BandsShift,
   BandsDeviation,1,1);
double lowerBand = iCustom(_Symbol,_Period,"Bands",BandsPeriod,BandsShift,
   BandsDeviation,2,1);
```

The input variables are presented in the order that they appear in the Bands indicator file. We've renamed them to `BandsPeriod`, `BandsShift` and `BandsDeviation`. Inside the `OnTick()` event handler, we call the `iCustom()` function and pass it the symbol, period, indicator name ("`Bands`"), the three input variables, the buffer number (0 = middle band, 1 = upper band, 2 = lower band) and finally the shift value. The code above will calculate all three lines of the Bands indicator for the most recently closed bar.

You can add any custom indicator to your expert advisor using the procedure above. First, note the buffer numbers for the corresponding lines/plots in the custom indicator that you wish to use. Next, note the input parameter names and their types. Add the input parameters to your expert advisor, renaming or omitting them as you wish. Pass the input parameters (or appropriate default values) to the `iCustom()` function along with the indicator name.

A Custom Indicator Class

The implementation of custom indicators in MQL4 makes it difficult to create a class specifically for custom indicators. You can, however, extend the `CIndicator` class for a specific custom indicator. Here's an example of a class using the Bands indicator above:

```
class Bands() : CIndicator
{
   private:
      int _bandsPeriod;
      int _bandsShift;
      double _bandsDeviation;

   public:
      void Bands(string pSymbol, int pTimeFrame, int pPeriod, int pShift, double pDeviation);
      double Mid(int pShift = 0);
      double Upper(int pShift = 0);
      double Lower(int pShift = 0);
};

void Bands::Bands(string pSymbol, int pTimeFrame, int pPeriod, int pShift,
   double pDeviation);
{
   Init(pSymbol, pTimeFrame);

   _bandsPeriod = pPeriod;
   _bandsShift = pShift;
```

```
   _bandsDeviation = pDeviation;
}

double Bands::Mid(int pShift=0)
{
   double band = iCustom(_Symbol,_Period,"Bands",_bandsPeriod,_bandsShift,
      _bandsDeviation,0,pShift);
   return band;
}

double Bands::Upper(int pShift=0)
{
   double band = iCustom(_Symbol,_Period,"Bands",_bandsPeriod,_bandsShift,
      _bandsDeviation,1,pShift);
   return band;
}

double Bands::Lower(int pShift=0)
{
   double band = iCustom(_Symbol,_Period,"Bands",_bandsPeriod,_bandsShift,
      _bandsDeviation,2,pShift);
   return band;
}
```

Just like we did for the built-in MetaTrader indicators, we declare several private variables for the indicator settings, a class constructor that calls the `Init()` function of the `CIndicator` class and assigns the input parameters to the private class variables, and three public functions that return a value for the specified shift for each line in the indicator. Based on the material covered earlier in this chapter, the code above should be self-explanatory.

Indicator-based Trading Signals

Let's examine a few of the ways that indicators can be used to create trading signals. Most indicator-based trading signals fall into one of the following categories:

- The relationship between an indicator line and the price.
- The relationship between two or more indicators lines.
- The change or slope of an indicator line between two bars.
- The value of an indicator relative to a fixed value.

A trading signal can occur when the price is above or below an indicator line. In this chapter, we discussed the Bands custom indicator, which is an implementation of the popular Bollinger Bands indicator. We can use this indicator as an example of a price/indicator relationship.

The Bollinger Bands can be used to open trades at price extremes – for example, when the price moves outside the bands. We can express these trading conditions like so:

```
if(Close[1] > upperBand && Count.Buy() == 0)
{
   // Open buy position
}
else if(Close[1] < lowerBand && Count.Sell() == 0)
{
   // Open sell position
}
```

If the current close price is above the upper band and no buy position is currently open, then we would open a buy position. The same is true for a sell if the close is below the lower band.

You can use two or more indicator lines to create a trading signal. For example, the moving average cross uses two moving averages, one fast and one slow. When the fast moving average is above the slow moving average a buy signal is implied, and vice versa for sell.

For example, lets use a 10 period exponential moving average for the fast MA, and a 30 period simple moving average for the slow MA:

```
if(fastMA > slowMA && Count.Buy() == 0 && glBuyPlaced == false)
{
   // Open buy position
}

if(fastMA < slowMA && Count.Sell() && glSellPlaced == false)
{
   // Open sell position
}
```

When the 10 EMA (the `fastMA` variable) is greater than the 30 SMA (the `slowMA` variable), a buy signal occurs. When the opposite is true, a sell signal occurs.

You can also use indicators of different types, provided they are all drawn in the same window. For example, you may wish to use a moving average with the Bollinger bands in the earlier example instead of the price:

```
if(ma[1] > bbUpper && Count.Buy() == 0)
{
   // Open buy position
}
```

In this example, we check if the moving average value of the previous bar (`ma[1]`) is greater than the upper bollinger band (`bbUpper`). If so, a buy signal occurs.

You can also compare two lines of the same indicator, such as the main and signal lines of the stochastic indicator:

```
if(stoch[1] > signal[1] && Count.Buy() == 0)
{
   // Open buy position
}
```

The `stoch[]` array represents the main or *%K* line of the indicator, while the `signal[]` array is the *%D* line. If the *%K* line is greater than the *%D* line, this indicates a bullish condition.

For trending indicators, you may wish to base trading decisions upon the direction in which the indicator is trending. You can do this by checking the indicator value of a recent bar against the value of a previous bar. For example, if we want a buy signal to occur when a moving average is sloping upward, we can compare the value of the most recently closed bar to the value of the previous bar:

```
if(ma[1] > ma[2] && Count.Buy() == 0)
{
   // Open buy position
}
```

This example checks to see if the moving average value of the most recently closed bar (`ma[1]`) is greater than the value of the previous bar (`ma[2]`). If so, the moving average is trending upward, and a buy signal occurs. You can do this with any indicator that trends up or down with the price, including oscillators such as RSI and MACD.

Many indicators, namely oscillators, appear in a separate chart window. They are not plotted according to price, but rather respond to changes in price. The RSI, for example, has a minimum value of 0 and a maximum value of 100. When the RSI value is below 30, that indicates an oversold condition, and when it is above 70, that indicates an overbought condition.

We can check for these overbought and oversold conditions simply by comparing the current RSI value to a fixed value. The example below checks for an oversold condition as part of a buy signal:

```
if(rsi[1] <= 30 && Count.Buy() == 0)
{
   // Open buy position
}
```

If the RSI value of the most recently closed bar is less than or equal to the oversold level of 30, that indicates a buy signal.

Histogram indicators such as the MACD oscillate around a zero line. Ascending values above zero indicate a bullish trend, while descending values below zero indicate bearish price movement. You may wish to base a trading signal around the indicator value relative to zero:

```
if(macd[1] > 0 && Count.Buy() == 0)
{
   // Open buy position
}
```

If the MACD value of the most recently closed bar is greater than zero, then a buy condition is indicated.

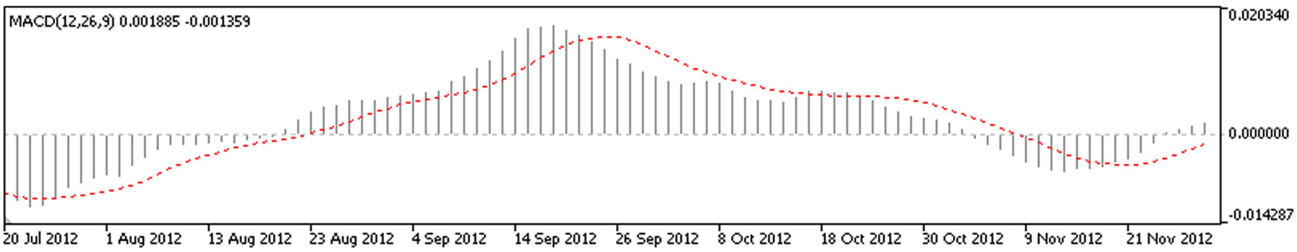


Fig. 19.6 – The MACD indicator.

You may come up with more elaborate indicator conditions than these. But nearly all indicator-based trading conditions are a combination of relationships between indicator readings and/or prices. To combine multiple price and indicator conditions for a trading signal, simply use boolean AND or OR operations between the expressions:

```
if(Close[1] < bbLower && (rsi[1] < 30 || stoch[1] < signal[1]) && Count.Buy() == 0)
{
   // Open buy position
}
```

In the expression above, if the close price is less than the lower Bollinger band and either the RSI is less than 30, or the stochastic is less than its signal line, we open a buy position. When combining AND and OR operations, use parentheses to establish which expressions will be evaluated first.

Turning Indicators On and Off

In an expert advisor with multiple indicators, you may wish to turn individual indicators on and off. To do so, we need to add a boolean input variable that will allow the user to deactivate an indicator if necessary. It will also be necessary to modify the trading condition to allow for both on and off states.

For example, let's create an expert advisor that uses the Bollinger Bands and an RSI. If the price is below the lower band and the RSI is in oversold territory (below 30), then we will open a buy position. We will add an input variable to turn the RSI condition on and off:

```
// Input variables
input bool UseRSI = true;

// OnTick() event handler
if(close[1] < lowerBand && ((rsi <= 30 && UseRSI == true) || UseRSI == false)
   && Count.Buy() == 0)
{
   // Open buy position
}
```

In the example above, we add an input variable named `UseRSI`. This turn the RSI trade condition on and off. Inside the `if` operator, we check for both true and false states. If `UseRSI == true`, we check to see if the `rsi` value is less than 30. If so, and if the close price is below the lower band and there is no buy order currently open, we open a new buy order. If `UseRSI == false`, and the other conditions are true, we also open a buy position.

The parentheses establish the order in which the conditions are evaluated. The inner set of parentheses contain the operation `(rsi <= 30 && UseRSI == true)`. This is evaluated first. If this condition is false, we then evaluate the condition inside the outer set of parentheses, `UseRSI == false`. Note the OR operator (`||`) separating both operations. If one of these conditions evaluates as true, then the entire expression inside the parentheses is true.

Remember that the indicator "on" state is inside the innermost set of parentheses, and is evaluated first. The indicator "off" state is inside the outermost set of parentheses, and is separated from the "on" condition by an OR operator (`||`). Combining AND and OR operations using parentheses to establish order of evaluation allows you to create complex trading conditions. When doing so, be sure to watch your opening and closing parentheses to make sure that you do not leave one out, add one too many, or nest them incorrectly.

Chapter 20 - Trailing Stops

In the next few chapters, we will examine popular trading features that can be used to enhance your expert advisors. We have created several include files to hold the classes and functions for these features. Use of these features in your expert advisors is optional.

What is a Trailing Stop?

A trailing stop is a stop loss that moves as a position increases in profit. For a buy order, the trailing stop moves up in price as the position gains in profit, and for a sell order, the trailing stop moves down in price as the position gains in profit. A trailing stop never moves in reverse.

The trailing stop typically follows the current price by a specified number of points. For example, if a trailing stop is set to 500 points, then the stop loss begins moving once the current price is at least 500 points away from the stop loss price. We can delay a trailing stop by requiring that a minimum level of profit be reached first. And while a trailing stop typically follows the price point by point, we can trail the stop in larger increments.

Let's examine how to implement a simple trailing stop. To calculate the trailing stop price, we add or subtract the trailing stop in points from the current Bid or Ask price. If the distance between the order's current stop loss and the current price is greater than the trailing stop in points, we modify the position's stop loss to match the trailing stop price.

The code below will add a simple trailing stop to an expert advisor. This code would be placed below the order placement code, near the end of the `OnTick()` event handler. You can view this code, including the further modifications that we'll make in this chapter, in the `Experts\Simple Expert Advisor.mq4` file:

```
// Input variables
input int TrailingStop = 500;

// OnTick() event handler
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool select = OrderSelect(order,SELECT_BY_POS);

   if(TrailingStop > 0 && OrderMagicNumber() == MagicNumber && select == true)
   {
      // Check buy order trailing stops
      if(OrderType() == OP_BUY)
      {
         RefreshRates();
```

```
                  double trailStopPrice = Bid - (TrailingStop * _Point);
                  trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

                  double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);

                  if(trailStopPrice > currentStopLoss)
                  {
                     bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
                        OrderTakeProfit(),0);
                  }
               }

               // Check sell order trailing stops
               else if(OrderType() == OP_SELL && TrailingStop > 0)
               {
                  RefreshRates();
                  double trailStopPrice = Ask + (TrailingStop * _Point);
                  trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

                  double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);

                  if( (trailStopPrice < currentStopLoss || currentStopLoss == 0) )
                  {
                     bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
                        OrderTakeProfit(),0);
                  }
               }
            }
         }
```

An input variable named `TrailingStop` is used to set the trailing stop in points. The trailing stop code consists of a `for` loop that iterates through the open order pool and selects each order. If the `TrailingStop` input variable is greater than zero, and the order magic number matches the `MagicNumber` input variable, we proceed to calculate the trailing stop.

We call the `RefreshRates()` function to refresh the values in the `Bid` and `Ask` variables. Since we are performing order operations in a loop, the prices will be out of date if we don't refresh them. For a buy order, we calculate the trailing stop price by subtracting the fractional value of `TrailingStop` from the current `Bid` price. Next, we use the `NormalizeDouble()` function to normalize the trailing stop price to the number of digits in the symbol (using the predefined `_Digits` variable). We store the result value in the `trailStopPrice` variable.

Next, we get the current stop loss for the selected order and normalize that price, saving it to the `currentStopLoss` variable. If the trailing stop price is greater than the current stop loss price, we modify the order to move the stop to the trailing stop price.

For a sell order, we simply reverse the operations. When comparing the trailing stop price to the current stop loss, we need to account for the event that there is no stop loss currently on the order. If the stop loss is zero, or if the trailing stop price is less than the current stop loss, we move the stop to the trailing stop price.

Minimum Profit

Let's add some modifications to our simple trailing stop. For example, maybe you want the trailing stop to kick in only after a minimum amount of profit has been achieved. To do this, we'll add a minimum profit setting to our expert advisor. First we determine the profit of the current position in points. Then we compare this to the minimum profit setting. If the position's current profit in points is greater than the minimum profit, then the trailing stop will activate. The changes are highlighted in bold:

```
// Input variables
input int TrailingStop = 500;
input int MinimumProfit = 200;

// OnTick() event handler
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool select = OrderSelect(order,SELECT_BY_POS);

   if(TrailingStop > 0 && OrderMagicNumber() == MagicNumber && select == true)
   {
      RefreshRates();

      // Check buy order trailing stops
      if(OrderType() == OP_BUY)
      {
         double trailStopPrice = Bid - (TrailingStop * _Point);
         trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

         double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);
         double currentProfit = Bid - OrderOpenPrice();
         double minProfit = MinimumProfit * _Point;

         if(trailStopPrice > currentStopLoss && currentProfit >= minProfit)
         {
            bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
               OrderTakeProfit(),0);
         }
```

```
      }

      // Check sell order trailing stops
      else if(OrderType() == OP_SELL && TrailingStop > 0)
      {
         double trailStopPrice = Ask + (TrailingStop * _Point);
         trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

         double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);
         double currentProfit = OrderOpenPrice() - Ask;
         double minProfit = MinimumProfit * _Point;

         if( (trailStopPrice < currentStopLoss || currentStopLoss == 0)
            && currentProfit >= minProfit )
         {
            bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
               OrderTakeProfit(),0);
         }
      }
   }
```

We've added an input variable named `MinimumProfit`. This is the minimum number of points in profit before the trailing stop kicks in. To calculate the current profit, we subtract the order opening price from the current Bid price (for a buy order), or subtract the current Ask price from the order opening price (for a sell order). We store this value in the `currentProfit` variable.

The `MinimumProfit` value is converted to a fractional value by multiplying it by the symbol's point value. We store this result in the `minProfit` variable. If the value of `currentProfit` is greater than the value of `minProfit`, and the other trailing stop conditions are true, then we move the stop loss to the trailing stop price.

Stepping A Trailing Stop

One final modification to the trailing stop is to add a step value. The code above will modify the trailing stop on every minor price change in the direction of profit. This can be a little overwhelming for the trade server, so we will enforce a minimum step value of 10 points and allow the user to specify a larger step value:

```
// Input variables
input int TrailingStop = 500;
input int MinimumProfit = 200;
input int Step = 10;
```

```
for(int order = 0; order <= OrdersTotal() - 1; order++)
{
   bool select = OrderSelect(order,SELECT_BY_POS);

   if(TrailingStop > 0 && OrderMagicNumber() == MagicNumber && select == true)
   {
      RefreshRates();

      // Check buy order trailing stops
      if(OrderType() == OP_BUY)
      {
         double trailStopPrice = Bid - (TrailingStop * _Point);
         trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

         double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);
         double currentProfit = Bid - OrderOpenPrice();
         double minProfit = MinimumProfit * _Point;

         int getStep = 0;
         if(Step < 10) getStep = 10;
         double step = getStep * _Point;

         if(trailStopPrice > currentStopLoss + step && currentProfit >= minProfit)
         {
            bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
               OrderTakeProfit(),0);
         }
      }

      // Check sell order trailing stops
      else if(OrderType() == OP_SELL && TrailingStop > 0)
      {
         double trailStopPrice = Ask + (TrailingStop * _Point);
         trailStopPrice = NormalizeDouble(trailStopPrice,_Digits);

         double currentStopLoss = NormalizeDouble(OrderStopLoss(),_Digits);
         double currentProfit = OrderOpenPrice() - Ask;
         double minProfit = MinimumProfit * _Point;

         int getStep = 0;
         if(Step < 10) getStep = 10;
         double step = getStep * _Point;
```

```
            if((trailStopPrice < currentStopLoss - step || currentStopLoss == 0)
               && currentProfit >= minProfit)
            {
               bool modify = OrderModify(OrderTicket(),OrderOpenPrice(),trailStopPrice,
                  OrderTakeProfit(),0);
            }
         }
      }
   }
```

We've added an input variable named `Step`, with a default value of 10 points. If `Step` is set to anything less than 10, we will automatically adjust the value to 10. We convert `Step` to a point value by multiplying it by the symbol's `_Point` value, and storing the result in the variable `step`. When checking the trailing stop condition, we add or subtract the `step` value from the `currentStopLoss` variable. This ensures that the trailing stop moves in 10 point increments.

Trailing Stop Include Functions

We're going to create a set of trailing stop functions that are robust and flexible enough for any situation that a trader may require. We will create a new include file to hold our trailing stop functions. The file is named `TrailingStop.mqh`, and will be located in the `\MQL4\Include\Mql4Book` folder.

Let's start with a function that will calculate and adjust a fixed trailing stop for a buy or sell order:

```
#include "Trade.mqh"

bool TrailingStop(int pTicket, int pTrailPoints, int pMinProfit = 0, int pStep = 10)
{
   if(pTrailPoints <= 0) return(false);
   bool result = OrderSelect(pTicket,SELECT_BY_TICKET);

   if(result == false)
   {
      Print("Trailing stop: #",pTicket," not found!");
      return false;
   }
```

We are including the `Trade.mqh` file from the current folder. Our `TrailingStop()` function has four parameters: `pTicket` is the ticket number of the order to modify, `pTrailPoints` is the trailing stop in points, `pMinProfit` is the minimum profit in points, and `pStep` is the step size in points.

If `pTrailPoints` is not set to a positive number, we exit the function. Otherwise, we select the order specified by `pTicket`. If an order selection error occurs, we exit the function.

```
bool setTrailingStop = false;

// Get order and symbol information
double orderType = OrderType();
string orderSymbol = OrderSymbol();
double orderStopLoss = OrderStopLoss();
double orderTakeProfit = OrderTakeProfit();
double orderOpenPrice = OrderOpenPrice();

double point = MarketInfo(orderSymbol,MODE_POINT);
int digits = (int)MarketInfo(orderSymbol,MODE_DIGITS);

// Convert inputs to prices
double trailPoints = pTrailPoints * point;
double minProfit = pMinProfit * point;
double step = pStep * point;

double trailStopPrice = 0;
double currentProfit = 0;
```

We retrieve the necessary order information and set up any variables needed for calculations.

```
// Calculate trailing stop
if(orderType == OP_BUY)
{
   double bid = MarketInfo(orderSymbol,MODE_BID);

   trailStopPrice = bid - trailPoints;
   trailStopPrice = NormalizeDouble(trailStopPrice,digits);

   currentProfit = bid - orderOpenPrice;

   if(trailStopPrice > orderStopLoss + step && currentProfit >= minProfit)
   {
      setTrailingStop = true;
   }
}
else if(orderType == OP_SELL)
{
   double ask = MarketInfo(orderSymbol,MODE_ASK);

   trailStopPrice = ask + trailPoints;
   trailStopPrice = NormalizeDouble(trailStopPrice,digits);

   currentProfit = orderOpenPrice - ask;
```

```
            if((trailStopPrice < orderStopLoss - step || orderStopLoss == 0)
               && currentProfit >= minProfit)
            {
               setTrailingStop = true;
            }
         }
```

Depending on the order type, we retrieve the current Bid or Ask price, and calculate the trailing stop and current profit for the currently selected order. If the trailing stop conditions are met, the `setTrailingStop` variable is set to true.

```
         // Set trailing stop
         if(setTrailingStop == true)
         {
            result = tsTrade.ModifyOrder(pTicket,0,trailStopPrice,orderTakeProfit);

            if(result == false)
            {
               Print("Trailing stop for #",pTicket," not set! Trail Stop: ",trailStopPrice,",
                  Current Stop: ",orderStopLoss," Current Profit: ",currentProfit);
            }
            else
            {
               Comment("Trailing stop for #",pTicket," modified");
               Print("Trailing stop for #",pTicket," modified");
               return true;
            }
         }

         return false;
      }
```

If `setTrailingStop` is true, we call the `ModifyOrder()` function from the `Trade.mqh` file to modify the stop loss on the order. We then log the result and return a `true` or `false` value.

Trailing Stop with Price Function

The previous examples used a fixed trailing stop. What if you wanted to trail an indicator, such as a moving average or Parabolic SAR? Or perhaps you want to trail the high or low price of a previous bar?

We're going to create another function that will trail the stop loss based on a specified price. If the price is greater (or less) than the current stop loss, the stop loss will be trailed to that price. We will use the same

function name as the `TrailingStop()` function that we defined in the previous section. The only difference will be the function parameters. This is known as *overloading* a function.

Our second `TrailingStop()` function has a parameter of type `double`, named `pTrailPrice`. This is the price that we will use as the trailing stop. The remainder of the function is similar to our first trailing stop function:

```
bool TrailingStop(int pTicket, double pTrailPrice, int pMinProfit = 0, int pStep = 10)
{
   if(pTrailPrice <= 0) return false;

   bool result = OrderSelect(pTicket,SELECT_BY_TICKET);

   if(result == false)
   {
      Print("Trailing stop: #",pTicket," not found!");
      return false;
   }

   bool setTrailingStop = false;

   // Get order and symbol information
   double orderType = OrderType();
   string orderSymbol = OrderSymbol();
   double orderStopLoss = OrderStopLoss();
   double orderTakeProfit = OrderTakeProfit();
   double orderOpenPrice = OrderOpenPrice();

   double point = MarketInfo(orderSymbol,MODE_POINT);
   int digits = (int)MarketInfo(orderSymbol,MODE_DIGITS);
```

As before, we select the order and retrieve the necessary order information.

```
   // Convert inputs to prices
   double minProfit = pMinProfit * point;
   double step = pStep * point;

   double currentProfit = 0;

   // Calculate trailing stop
   if(orderType == OP_BUY)
   {
      pTrailPrice = AdjustBelowStopLevel(orderSymbol,pTrailPrice);

      double bid = MarketInfo(orderSymbol,MODE_BID);
      currentProfit = bid - orderOpenPrice;
```

```
            if(pTrailPrice > orderStopLoss + step && currentProfit >= minProfit)
            {
               setTrailingStop = true;
            }
         }
         else if(orderType == OP_SELL)
         {
            pTrailPrice = AdjustAboveStopLevel(orderSymbol,pTrailPrice);

            double ask = MarketInfo(orderSymbol,MODE_ASK);
            currentProfit = orderOpenPrice - ask;

            if((pTrailPrice < orderStopLoss - step || orderStopLoss == 0)
               && currentProfit >= minProfit)
            {
               setTrailingStop = true;
            }
         }
```

Depending on the order type, we call the `AdjustAboveStopLevel()` or `AdjustBelowStopLevel()` function from our `Trade.mqh` file, and adjust the trailing stop price relative to the stop level. This is necessary because the trailing stop price may be too close to the current price, which would cause a trade error.

```
         // Set trailing stop
         if(setTrailingStop == true)
         {
            result = tsTrade.ModifyOrder(pTicket,0,pTrailPrice,orderTakeProfit);

            if(result == false)
            {
               Print("Trailing stop for #",pTicket," not set! Trail Stop: ",pTrailPrice,
                  ", Current Stop: ",orderStopLoss," Current Profit: ",currentProfit);
            }
            else
            {
               Comment("Trailing stop for #",pTicket," modified");
               Print("Trailing stop for #",pTicket," modified");
               return true;
            }
         }

         return false;
      }
```

If `setTrailingStop` is set to true, we modify the order to the value set in `pTrailPrice`, log the result, and return the appropriate value.

These functions will modify the trailing stop for a single order. Most of the time though, you will be modifying the trailing stop on all open orders. Next, we will create functions to loop through the order pool and apply a trailing stop to all orders opened by the expert advisor.

Multiple Order Trailing Stop Functions

We're going to create a pair of functions that will modify the trailing stop for all open orders that match the magic number set in the input parameters. The function name is `TrailingStopAll()`, and like our `TrailingStop()` function, it is overloaded. The first function uses a fixed trailing stop:

```
void TrailingStopAll(int pTrailPoints, int pMinProfit = 0, int pStep = 10)
{
   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      // Skip if magic number doesn't match or order is pending
      int magicNumber = OrderMagicNumber();
      double orderType = OrderType();

      if(magicNumber == CTrade::GetMagicNumber()
         && (orderType == OP_BUY || orderType == OP_SELL))
      {
         TrailingStop(OrderTicket(),pTrailPoints,pMinProfit,pStep);
      }
   }
}
```

As always, we use a `for` loop to iterate through the order pool and select each order. If the magic number on the order matches the magic number that is set in the expert advisor, and the order type is a market order, we will call the `TrailingStop()` function to check and modify the trailing stop. (Note the `GetMagicNumber()` function call: This is a static function that returns the magic number that is shared by all instances of the `CTrade` class.) The `pTrailPoints` parameter is of type `int`, so when we call the `TrailingStop()` function, it will call the first variant of that function.

The second variant of the `TrailingStopAll()` function modifies orders by price. The only difference is the `pTrailPrice` input parameter, which sets the trailing stop to a specified price:

```
void TrailingStopAll(double pTrailPrice, int pMinProfit = 0, int pStep = 10)
{
   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      // Skip if magic number doesn't match or order is pending
      int magicNumber = OrderMagicNumber();
      double orderType = OrderType();

      if(magicNumber == CTrade::GetMagicNumber()
         && (orderType == OP_BUY || orderType == OP_SELL))
      {
         TrailingStop(OrderTicket(),pTrailPrice,pMinProfit,pStep);
      }
   }
}
```

As long are you are applying the same trailing stop to all open orders, the `TrailingStopAll()` functions can be used to handle your trailing stop needs.

Trailing Stop Function Examples

Let's demonstrate how to use our multiple order trailing stop functions in an expert advisor. Any trailing stop functionality will be placed near the end of the `OnTick()` event handler, after any order opening and closing logic.

```
// Input variables
input int TrailingStop = 0;
input int MinimumProfit = 200;
input int Step = 10;

// OnTick() event handler
if(TrailingStop > 0)
{
   TrailingStopAll(TrailingStop,MinProfit,Step);
}
```

We define three input variables for the trailing stop distance, minimum profit and step interval. The trailing stop feature can be disabled by setting the `TrailingStop` variable to zero. Otherwise, we will call the `TrailingStopAll()` function, which will check all orders currently opened by this expert advisor and modify the trailing stop if appropriate.

If you wish to use a price as your trailing stop (for example, a recent high/low price or an indicator value), simply pass a value of type `double` as the first parameter of the `TrailingStopAll()` function. The compiler will select the correct variant of the function.

Break Even Stop

Sometimes a trader prefers to move the stop loss to the order opening price when a specified amount of profit is reached. The stop loss is moved only once – unlike a trailing stop, which keeps moving the stop as long as the trade increases in profit. This is referred to as a *break even stop*.

The principle behind a break even stop is similar to a trailing stop. First, check to see if the trade has reached a specified minimum profit. If so, check to see if the stop loss is less (or greater) than the position opening price. If so, the stop loss is moved to the order opening price. You can use a break even stop alongside a trailing stop. Just be sure to adjust the minimum profit setting for both stop types so that the trailing stop does not activate before the break even stop.

We've added a break even stop function to our `TrailingStop.mqh` file. It functions similarly to the fixed trailing stop functions from earlier in the chapter. The `BreakEvenStop()` function has three parameters: `pTicket` is the ticket number to modify, `pMinProfit` is the minimum profit in points, and `pLockProfit` adds a specified number of points of profit to the break even stop:

```
bool BreakEvenStop(int pTicket, int pMinProfit, int pLockProfit = 0)
{
   if(pMinProfit <= 0) return false;

   bool result = OrderSelect(pTicket,SELECT_BY_TICKET);

   if(result == false)
   {
      Print("Break even stop: #",pTicket," not found!");
      return false;
   }

   bool setBreakEvenStop = false;

   // Get order and symbol information
   double orderType = OrderType();
   string orderSymbol = OrderSymbol();
   double orderStopLoss = OrderStopLoss();
   double orderTakeProfit = OrderTakeProfit();
   double orderOpenPrice = OrderOpenPrice();

   double point = MarketInfo(orderSymbol,MODE_POINT);
```

```
int digits = (int)MarketInfo(orderSymbol,MODE_DIGITS);

// Convert inputs to prices
double minProfit = pMinProfit * point;
double lockProfit = 0;
if(pLockProfit > 0) lockProfit = pLockProfit * point;

double breakEvenStop = 0;
double currentProfit = 0;
```

As before, we select the order and retrieve the relevant order information, and initialize variables.

```
if(orderType == OP_BUY)
{
   double bid = MarketInfo(orderSymbol,MODE_BID);

   breakEvenStop = orderOpenPrice + lockProfit;
   breakEvenStop = NormalizeDouble(breakEvenStop,digits);

   currentProfit = bid - orderOpenPrice;
   currentProfit = NormalizeDouble(currentProfit,digits);
   orderStopLoss = NormalizeDouble(orderStopLoss,digits);

   if(breakEvenStop > orderStopLoss && currentProfit >= minProfit)
   {
      setBreakEvenStop = true;
   }
}
else if(orderType == OP_SELL)
{
   double ask = MarketInfo(orderSymbol,MODE_ASK);

   breakEvenStop = orderOpenPrice - lockProfit;
   breakEvenStop = NormalizeDouble(breakEvenStop,digits);

   currentProfit = orderOpenPrice - ask;
   currentProfit = NormalizeDouble(currentProfit,digits);
   orderStopLoss = NormalizeDouble(orderStopLoss,digits);

   if((breakEvenStop < orderStopLoss || orderStopLoss == 0) && currentProfit >= minProfit)
   {
      setBreakEvenStop = true;
   }
}
```

We determine the break even stop price by adding the number of points in the `pLockProfit` variable (if specified) to the order opening price. This value is stored in the `breakEvenStop` variable. Next, we calculate the current profit in points. For a buy order, if the break even stop is greater than the current stop loss, and the current profit is greater than the minimum profit, we will modify the stop loss.

```
   if(setBreakEvenStop == true)
   {
      result = ModifyOrder(pTicket,0,breakEvenStop,orderTakeProfit);

      if(result == false)
      {
         Print("Break even stop for #",pTicket," not set! Break Even Stop: ",breakEvenStop,
            ", Current Stop: ",orderStopLoss," Current Profit: ",currentProfit);
      }

      else
      {
         Comment("Break even stop for #",pTicket," modified");
         Print("Break even stop for #",pTicket," modified");
         return true;
      }
   }

   return false;
}
```

If `setBreakEvenStop` is true, we will pass the `breakEvenStop` variable as the first parameter to the `ModifyOrder()` function of our `CTrade` class, and set the stop loss to the break even price.

This function will modify the break even stop for a single order. If you wish to apply a break even stop to all open orders, use the `BreakEvenStopAll()` function:

```
void BreakEvenStopAll(int pMinProfit, int pLockProfit = 0)
{
   // Loop through open order pool from oldest to newest
   for(int order = 0; order <= OrdersTotal() - 1; order++)
   {
      // Select order
      bool result = OrderSelect(order,SELECT_BY_POS);

      // Skip if magic number doesn't match or order is pending
      int magicNumber = OrderMagicNumber();
      double orderType = OrderType();
```

```
            if(magicNumber == CTrade::GetMagicNumber()
               && (orderType == OP_BUY || orderType == OP_SELL))
            {
               BreakEvenStop(OrderTicket(),pMinProfit,pLockProfit);
            }
         }
      }
```

Break Even Stop Function Example

Here's an example of how to include a break even stop in your expert advisor. Just like a trailing stop, you want the break even stop code to go near the bottom of your `OnTick()` event handler, after any order opening and closing logic:

```
// Input variables
input int BreakEven = 0;
input int LockProfit = 0;

// OnTick() event handler
if(BreakEven > 0)
{
   BreakEvenAll(BreakEven,LockProfit);
}
```

The `BreakEven` input variable is the number of points in profit that is required before the break even stop activates. By default, the stop loss is moved to the order opening price. If you want to place the stop loss above or below the order opening price, simply add or subtract the number of points using the `LockProfit` input variable.

You can use a break even stop along with any of the trailing stop functions. If you want the break even stop to trigger first, be sure that the minimum profit setting for your trailing stop is greater than the break even stop setting.

Chapter 21 - Money Management & Trade Sizing

Up to this point, we've been using an input variable to specify order volume. Programming an expert advisor offers the opportunity to use automated money management techniques that can manage your risk and grow or shrink your trade size as your account balance changes. We'll examine techniques for automatically managing trade sizes, as well as verifying trade volumes for correctness.

Money Management

Money management is a method of optimally adjusting position size according to risk. Most traders use the same fixed trade volume for every trade. This can result in trades that are too large or too small for the amount of money that is being risked.

To calculate an optimal trade size, we will use the distance of the stop loss price from the trade entry price as well as a percentage of the current balance to determine the maximum risk per trade. A good guideline is to limit your risk per trade to 2-3% of your current balance. If for some reason we cannot calculate a trade volume (*i.e.* a stop loss or percentage has not been specified), we will fall back to a specified fixed trade volume.

Here's an example of how we can add simple money management to an expert advisor. The money management code goes before any order opening code, and after the stop loss in points has been determined:

```
// Input variables
input bool UseMoneyManagement = true;
input double RiskPercent = 2;
input double FixedLotSize = 0.1;
input int StopLoss = 0;

// OnTick() event hander
double lotSize = FixedLotSize;

if(UseMoneyManagement == true)
{
   double margin = AccountBalance() * (RiskPercent / 100);
   double tickSize = MarketInfo(_Symbol,MODE_TICKVALUE);

   lotSize = (margin / StopLoss) / tickSize;

   if(lotSize < MarketInfo(_Symbol,MODE_MINLOT)) lotSize = MarketInfo(_Symbol,MODE_MINLOT);
   else if(lotSize > MarketInfo(_Symbol,MODE_MAXLOT))
      lotSize = MarketInfo(_Symbol,MODE_MAXLOT);
```

```
      else if(MarketInfo(_Symbol,MODE_LOTSTEP) == 0.1) lotSize = NormalizeDouble(lotSize,1);
      else NormalizeDouble(lotSize,2);
   }
```

The `UseMoneyManagement` input variable determines whether we use money management or not. If it is set to `false`, we will use the lot size specified in `FixedLotSize`. Otherwise, the `RiskPercent` input variable is the percentage of our account balance that we are willing to risk per trade. The `StopLoss` input variable must be entered to determine our risk.

Inside the `OnTick()` event handler, the `lotSize` variable will hold our calculated lot size, and is set to the value of `FixedLotSize` by default. If `UseMoneyManagement` is set to true, we will calculate a lot size using the `StopLoss` and `RiskPercent` variables. Otherwise, we will use the value in `FixedLotSize` as our trade volume.

To calculate the lot size, first we determine the percentage of our account balance that we are willing to risk. We multiply the percentage specified in `RiskPercent` by our account balance, and store the result in the `margin` variable. Next, we retrieve the *tick value* for the current symbol using the `MarketInfo()` function. The tick value is the amount of profit or loss represented by a single point move.

To calculate the lot size, the formula is *(max margin / stop loss) / tick size*. The maximum margin is the maximum amount of equity that we will risk on this trade. The stop loss is the stop loss in points, and the tick size is the amount of profit or loss represented by a single point move. By dividing these three together, we end up with a lot size that will limit our risk to a specified percentage of our account, using the specified stop loss.

Let's clarify this with an example: We want to place an order risking no more than 2% of our account balance of $5000. The initial stop loss will be placed 500 points away from the order opening price. The symbol is EURUSD and we're using mini lots, so the tick size will be $1 per point. 2% of $5000 is $100, so this value will be saved in the `margin` variable. The value of the `tickSize` variable will be $1.

$100 divided by 500 points is 0.2. Every point of movement will equal approximately $0.20 of profit or loss. 0.2 divided by $1 equals 0.2, so our trade volume will be 0.2 lots. If this trade of 0.2 lots hits its initial stop loss 500 points away, the loss will be approximately $100. If we set the stop loss to 200 points, the trade volume will be 0.5 lots, but the maximum loss is still $100.

Verifying Trade Volume

Before we pass a lot size to the `OrderSend()` function, we should ensure that the trade volume is valid. We do this by comparing the trade volume to the minimum and maximum lot sizes, as well as the broker's minimum step size.

This excerpt from our money management code above takes the lot size that we calculate and verifies that it is correct:

```
if(lotSize < MarketInfo(_Symbol,MODE_MINLOT)) lotSize = MarketInfo(_Symbol,MODE_MINLOT);
else if(lotSize > MarketInfo(_Symbol,MODE_MAXLOT))
   lotSize = MarketInfo(_Symbol,MODE_MAXLOT);
else if(MarketInfo(_Symbol,MODE_LOTSTEP) == 0.1) lotSize = NormalizeDouble(lotSize,1);
else NormalizeDouble(lotSize,2);
```

We use the `MarketInfo()` function with the `MODE_MINLOT` and `MODE_MAXLOT` parameters to retrieve the minimum and maximum lot sizes. If the value of the `lotSize` variable is less than the minimum lot size or greater than the maximum lot size, we resize the trade volume to be equivalent to either the minimum or maximum lot size.

If the `MarketInfo()` function with the `MODE_LOTSTEP` parameter returns 0.1, then the broker does not accept micro lots, and we must normalize the lot size to a single decimal place. Otherwise, we normalize it to two decimal places.

Money Management Include Functions

We've created a new include file for money management functions. The file is named `MoneyManagement.mqh`, and is located in the `\MQL4\Include\Mql4Book` folder. This file was borrowed from the MQL5 code in the *Expert Advisor Programming for MetaTrader 5* book. Since it does not use any MQL5-specific features, it will work just fine in our MQL4 programs.

We've created a money management function named `MoneyManagement()`:

```
#define MAX_PERCENT 10

double MoneyManagement(string pSymbol, double pFixedVol, double pPercent, int pStopPoints)
{
   double tradeSize;

   if(pPercent > 0 && pStopPoints > 0)
   {
      if(pPercent > MAX_PERCENT) pPercent = MAX_PERCENT;

      double margin = AccountInfoDouble(ACCOUNT_BALANCE) * (pPercent / 100);
      double tickSize = SymbolInfoDouble(pSymbol,SYMBOL_TRADE_TICK_VALUE);

      tradeSize = (margin / pStopPoints) / tickSize;
      tradeSize = VerifyVolume(pSymbol,tradeSize);

      return(tradeSize);
   }
```

```
      else
      {
         tradeSize = pFixedVol;
         tradeSize = VerifyVolume(pSymbol,tradeSize);

         return(tradeSize);
      }
   }
```

The `pSymbol` parameter is the trade symbol, `pFixedVol` is the default trade volume, `pPercent` is the percentage of the current balance to use, and `pStopPoints` is the stop loss distance in points.

The `tradeSize` variable will hold our calculated trade volume. First, we check to see if `pPercent` and `pStopPoints` are both greater than zero. If not, then we cannot calculate the trade volume, and will fall back on the default trade volume specified by `pFixedVol`.

If `pPercent` and `pStopPoints` are valid, then we will proceed with calculating the trade volume. First, we compare `pPercent` to the maximum volume. As mentioned before, it is recommended to limit your risk to no more than 2-3% of your balance. The `MAX_PERCENT` constant, defined at the top of our `MoneyManagement.mqh` file, specifies a maximum risk of 10 percent. If `pPercent` exceeds this, then it will be adjusted to no more than 10 percent.

Next, we calculate the amount of margin to risk. We retrieve the account balance using the `AccountInfoDouble()` function with the `ACCOUNT_BALANCE` parameter. We multiply this by `pPercent` (divided by 100 to obtain a fractional value), and store the result in the `margin` variable. Then we retrieve the tick value of the symbol from the trade server using the `SymbolInfoDouble()` function with the `SYMBOL_TRADE_TICK_VALUE` parameter, and store the result in the `tickSize` variable.

To calculate our trade volume, we divide the margin to risk (`margin`) by the stop loss in points (`pStopLoss`), and divide that result by `tickSize`. Then we pass our calculated trade volume to the `VerifyVolume()` function. We will cover this function shortly. The verified result is returned to the program.

Here's an example of how we can use our money management function in an expert advisor:

```
// Include directive
#include <Mql4Book/MoneyManagement.mqh>

// Input variables
input double RiskPercent = 2;
input double FixedVolume = 0.1;
input int StopLoss = 500;
```

```
// OnTick() event handler
double lotSize = MoneyManagement(_Symbol,FixedVolume,RiskPercent,StopLoss);
```

To use our money management function, we need to include the `MoneyManagement.mqh` file in our expert advisor. The `RiskPercent` input variable is our trade risk as a percentage of the current trade balance. If you do not want to use money management, set this to zero. `FixedVolume` is the fixed trade volume to use if `RiskPercent` or `StopLoss` is zero.

The `lotSize` variable will hold our calculated trade volume. The `MoneyManagement()` function will take the specified input variables and return the calculated and verified trade volume. We can then pass the `lotSize` variable to one of our order placement functions as defined in the previous chapters.

The examples above assume a fixed stop loss that is specified by an input variable. What if you want to use a dynamic stop loss price, such as an indicator value or a support/resistance price? We will need to calculate the distance between the desired order opening price (either a pending order price, or the current Bid or Ask price) and the desired stop loss price.

For example, we want to open a buy position. The current Ask price is 1.39426, and the stop loss price we want to use is 1.38600. The difference between the two is 826 points. We'll create a function that will determine the difference in points between the desired opening price and the stop loss price:

```
double StopPriceToPoints(string pSymbol, double pStopPrice, double pOrderPrice)
{
   double stopDiff = MathAbs(pStopPrice - pOrderPrice);
   double getPoint = SymbolInfoDouble(pSymbol,SYMBOL_POINT);
   double priceToPoint = stopDiff / getPoint;
   return(priceToPoint);
}
```

The `pStopPrice` parameter is our desired stop loss price, and `pOrderPrice` is our desired order opening price. First, the function calculates the difference between `pStopPrice` and `pOrderPrice`, and returns the absolute value using the `MathAbs()` function. We store the result in the `stopDiff` variable. Next we retrieve the symbol's point value and store it in the variable `getPoint`. Finally, we divide `stopDiff` by `getPoint` to find the stop loss distance in points.

Here's an example of how we can do this in code. We'll assume that the current Ask price is 1.39426:

```
double stopLossPrice = 1.38600;
double currentPrice = SymbolInfoDouble(_Symbol,SYMBOL_ASK);
double stopLossDistance = StopPriceToPoints(_Symbol,stopLossPrice,currentPrice);
double tradeSize = MoneyManagement(_Symbol,FixedVolume,RiskPercent,stopLossDistance);
```

The `stopLossPrice` variable contains our desired stop loss price of 1.38600, while the `currentPrice` variable holds the current Ask price of 1.39426. The `stopLossDistance` variable will contain the return value of the `StopPriceToPoints()` function, which is 826. We then use `stopLossDistance` as the last parameter of the `MoneyManagement()` function. Assuming a `RiskPercent` value of 2% and a balance of $5000 using standard lots, the trade volume will be 0.08 lots.

As long as you have specified an initial stop loss, the `MoneyManagement()` function can be used to limit your trade risk to a specified percentage of your account balance. You can even use a dynamic stop loss by using the `StopPriceToPoints()` function to calculate the stop loss in points. As your account balance grows or shrinks, the trade size will increase or decrease accordingly.

Verifying Trade Volume

Our `MoneyManagement.mqh` file contains a function to verify trade volume. It is used by the `MoneyManagement()` function to verify the lot size, and can be used independently as well:

```
double VerifyVolume(string pSymbol,double pVolume)
{
   double minVolume = SymbolInfoDouble(pSymbol,SYMBOL_VOLUME_MIN);
   double maxVolume = SymbolInfoDouble(pSymbol,SYMBOL_VOLUME_MAX);
   double stepVolume = SymbolInfoDouble(pSymbol,SYMBOL_VOLUME_STEP);

   double tradeSize;
   if(pVolume < minVolume) tradeSize = minVolume;
   else if(pVolume > maxVolume) tradeSize = maxVolume;
   else tradeSize = MathRound(pVolume / stepVolume) * stepVolume;

   if(stepVolume >= 0.1) tradeSize = NormalizeDouble(tradeSize,1);
   else tradeSize = NormalizeDouble(tradeSize,2);

   return(tradeSize);
}
```

The `pSymbol` parameter is the trade symbol, and `pVolume` is the trade volume to verify. This function will return the adjusted trade volume to the program.

Chapter 22 - Working with Time and Date

In this chapter, we'll learn how to work with time and date in MQL4. We'll create a class that will allow us to place orders only on the open of a new bar. We'll create classes that will allow us to implement a fully-featured trade timer in our expert advisors. We'll also learn about the *Timer* event, which allows us to carry out actions at a predefined interval.

Trading On New Bar Open

By default, an expert advisor executes on every new incoming *tick*, or change in price. This means that orders will be opened intrabar, unless you program your expert advisor to trade otherwise. We can program our expert advisor to trade only at the open of each new bar by saving the timestamp of the current bar to a static, global or class variable. When the timestamp of the current bar changes, we'll know that a new bar has opened. At this time, we can check for order opening and closing conditions.

When trading on the open of a new bar, we will use the price and indicator values of the previous bar when making trade decisions. If you're looking at a chart of historical trades, keep in mind that the trade criteria is based on the previous bar, and not the bar that the trade opened on!

Note that the Strategy Tester has an *Open prices only* execution mode that mimics this behavior. If your strategy only opens and closes orders on the open of a new bar, then you can use *Open prices only* to perform quick and accurate testing of your expert advisor.

The CNewBar Class

Let's create a class that will keep track of the timestamp of the current bar and allow us to determine when a new bar has opened. We'll create this class in a new include file named `\MQL4\Include\Mql4Book\Timer.mqh`. All of the classes and functions in this chapter will go in this file. Note that this file was borrowed from the MQL5 code in *Expert Advisor Programming for MetaTrader 5*. Since there is no MQL5-specific code in this file, it will work fine in our MQL4 programs.

Here is the class declaration for our `CNewBar` class:

```
class CNewBar
{
   private:
      datetime _time[], _lastTime;

   public:
      void CNewBar();
```

```
        bool CheckNewBar(string pSymbol, ENUM_TIMEFRAMES pTimeframe);
};
```

We have two private variables in our class. The `_time[]` array contains the timestamps of the most recent bars, and the `_lastTime` variable contains the timestamp for the most recently checked bar. Note that we declared both variables on the same line, since they share the same type. We will compare these two values to determine if a new bar has opened. Our class also contains a constructor and the `CheckNewBar()` function. The class constructor simply sets the `_time[]` array as a series array:

```
void CNewBar::CNewBar(void)
{
   ArraySetAsSeries(_time,true);
}
```

The `CheckNewBar()` function is used to check for the open of a new bar. Here is the function declaration:

```
bool CNewBar::CheckNewBar(string pSymbol, ENUM_TIMEFRAMES pTimeframe)
{
   bool firstRun = false, newBar = false;
   CopyTime(pSymbol,pTimeframe,0,2,_time);

   if(_lastTime == 0) firstRun = true;

   if(_time[0] > _lastTime)
   {
      if(firstRun == false) newBar = true;
      _lastTime = _time[0];
   }

   return(newBar);
}
```

This function takes two parameters: the symbol and the period of the chart that we are using. We initialize two boolean variables, `firstRun` and `newBar`, and explicitly set them to `false`. The `CopyTime()` function updates the `_time[]` array with the timestamp of the current bar.

We check the `_lastTime` class variable to see if it has a value assigned to it. The first time this function runs, `_lastTime` will be equal to zero, so we will set the `firstRun` variable to `true`. This indicates that we have just attached the expert advisor to the chart. Since we do not want the expert advisor to trade intrabar, this is a necessary check to ensure that we don't attempt to open an order right away.

Next, we check to see if `_time[0]` is greater than `_lastTime`. If this is the first time we've run this function, then this expression will be true. This expression will also evaluate to `true` every time a new bar opens. If the

timestamp of the current bar has changed, we check to see if `firstRun` is set to `false`. If so, we set `newBar` to `true`. We update the `_lastTime` variable with the current bar's timestamp, and return the value of `newBar`.

The execution of `CheckNewBar()` works like this: When the expert advisor is first attached to a chart, the function will return a value of `false` because `firstRun` will be set to `true`. On each subsequent check of the function, `firstRun` will always be `false`. When the current bar's timestamp (`_time[0]`) is greater than `_lastTime`, we set `newBar` to `true`, update the value of `_lastTime`, and return a value of `true`. This indicates that a new bar has opened, and we can check our trading conditions to open and close orders.

Here is how we would use the `CNewBar` class in an expert advisor:

```
// Include file and object declaration
#include <Mql4Book\Timer.mqh>
CNewBar NewBar;

// Input variables
input bool TradeOnNewBar = true;

// OnTick() event handler
bool newBar = true;
int barShift = 0;

if(TradeOnNewBar == true)
{
   newBar = NewBar.CheckNewBar(_Symbol,_Period);
   barShift = 1;
}

if(newBar == true)
{
   // Order placement code
}
```

We declare an object based on the `CNewBar` class named `NewBar`. The `TradeOnNewBar` input variable allows us to select between trading on a new bar, or trading on every tick. In the `OnTick()` event handler, we declare a local `bool` variable named `newBar` and initialize it to `true`, as well as an `int` variable named `barShift`.

If `TradeOnNewBar` is set to `true`, we call our `CheckNewBar()` function and save the return value to the `newBar` variable. Most of the time, this value will be `false`. We will also set the value of `barShift` to 1. If `TradeOnNewBar` is set to `false`, then `newBar` will be `true`, and `barShift` will be 0.

If `newBar` is `true`, we'll go ahead and check our order placement conditions. Any code that you want to run only at the open of a new bar will go inside these brackets. This includes all order opening conditions and possibly your closing conditions as well.

When using the CNewBar class to check for the open of a new bar, the barShift variable will be used to set the bar index of any price or indicator values. For example, we'll use the moving average and close price functions that we've defined in previous chapters:

```
double close = Close[barShift];
double ma = fastMa.Main(barShift);
```

The close and ma variables will be assigned the value for the current bar (barShift = 0) if TradeOnNewBar is set to false, or the value of the previous bar (barShift = 1) if TradeOnNewBar is set to true.

The datetime Type

We addressed the datetime type previously on page 18. The datetime type is used to hold date and time values. In a datetime variable, the date and time is represented as the number of seconds elapsed since January 1, 1970 at midnight. This makes it easy to compare two datetime values, and manipulate them mathematically.

For example, let's say we have two datetime variables, date1 and date2. We know that date1 is equal to July 12, 2014 at 3:00, and date2 is equal to July 14, 2014 at 22:00. If we wanted to know which date is earlier, we can compare the two :

```
datetime date1 = D'2014.07.12 03:00';
datetime date2 = D'2014.07.14 22:00';

if(date1 < date2)
{
   Print("date1 is sooner");                // True
}

else if(date1 > date2)
{
   Print("date2 is sooner");
}
```

The correct result is "date1 is sooner". Another advantage of working with datetime values is that if you want to add or subtract hours, days or any period of time, you can simply add or subtract the appropriate number of seconds to or from the datetime value. For example, we'll add 24 hours to the date1 variable above:

```
datetime addDay = date1 + 86400;            // Result:  2014.07.13 03:00
```

There are 86,400 seconds in a 24 hour period. By adding 86400 to a `datetime` value, we have advanced the date by exactly 24 hours. The `date1` variable now has a value equal to July 13, 2014 at 3:00.

A `datetime` value is not human-readable, although the MetaTrader 4 terminal will automatically convert `datetime` values to a readable string constant when printing them to the log. If you need to convert a `datetime` value to a string for other purposes, the `TimeToString()` function will convert a `datetime` variable to a string in the format *yyyy.mm.dd hh:mm*. In this example, we will convert `date2` using the `TimeToString()` function:

```
string convert = TimeToString(date2);
Print(convert);                              // Result: 2014.07.14 22:00
```

The `TimeToString()` function converts the `date2` variable to a string, and saves the result to the `convert` variable. If we print out the value of `convert`, it will be `2014.07.14 22:00`.

We can create a `datetime` value by constructing a string in the format *yyyy.mm.dd hh:mm:ss*, and converting it to a `datetime` variable using the `StringToTime()` function. For example if we want to convert the string `2014.07.14 22:00` to a `datetime` value, we pass the string to the `StringToTime()` function:

```
string dtConst = "2014.07.14 22:00";
datetime dtDate = StringToTime(dtConst);
```

The `dtDate` variable is assigned a `datetime` value equivalent to `2014.07.14 22:00`. We can also create a `datetime` constant by enclosing the string in single quotes and prefacing it with a capital D. This `datetime` constant can be assigned directly to a `datetime` variable:

```
datetime dtDate = D'2014.07.14 22:00';
```

This method allows for more flexibility when constructing the `datetime` constant string. For example, you could use *dd.mm.yyyy* as your date format and leave off the time. The `datetime` constant topic in the *MQL4 Reference* under *Language Basics* > *Data Types* > *Integer Types* > *Datetime Type* has more information on formatting a `datetime` constant.

We can also use the `MqlDateTime` structure. This structure allows us to retrieve specific information from a `datetime` variable, such as the hour or the day of the week.

The MqlDateTime Structure

The `MqlDateTime` structure contains variables that hold information about a `datetime` value. This feature was introduced in MQL5. Here is the structure definition from the *MQL4 Reference*:

```
struct MqlDateTime
{
   int year;           // Year
   int mon;            // Month
   int day;            // Day
   int hour;           // Hour
   int min;            // Minutes
   int sec;            // Seconds
   int day_of_week;    // Day of week (0 = Sunday, 1 = Monday, ... 6 = Saturday)
   int day_of_year;    // Day number of the year (January 1st = 0)
};
```

Using the `MqlDateTime` structure, we can retrieve any date or time element from a `datetime` value. We can retrieve the hour, minute or day and assign that value to an integer variable. We can even retrieve the day of the week or the day of the year. We can also assign values to these variables to construct a `datetime` value.

The `TimeToStruct()` function is used to convert a `datetime` value to an object of the `MqlDateTime` type. The first parameter of the function is the `datetime` value to convert. The second parameter is the `MqlDateTime` object to load the values into. Here's an example:

```
datetime dtTime = D'2014.10.15 16:36:23';

MqlDateTime timeStruct;
TimeToStruct(dtTime,timeStruct);

int day = timeStruct.day;
int hour = timeStruct.hour;
int dayOfWeek = timeStruct.day_of_week;
```

First, we construct a `datetime` variable, `dtTime`, using a `datetime` constant. Next, we declare an object of the `MqlDateTime` type named `timeStruct`. We use the `TimeToStruct()` function to convert the `datetime` value in `dtTime` to the `MqlDateTime` structure `timeStruct`. Finally, we show how to retrieve the day, hour and day of week from the `timeStruct` object. The `day` is 15, the `hour` is 16, and the `day_of_week` is 3 for Wednesday.

We can also create an `MqlDateTime` object, assign values to its member variables, and convert it to a `datetime` value using the `StructToTime()` function. The `day_of_week` and `day_of_year` variables are not used when using `StructToTime()`. Here's an example of how to create a `datetime` value using the `StructToTime()` function:

```
MqlDateTime timeStruct;

timeStruct.year = 2014;
```

```
timeStruct.mon = 10;
timeStruct.day = 20;
timeStruct.hour = 2;
timeStruct.min = 30;
timeStruct.sec = 0;

datetime dtTime = StructToTime(timeStruct);
```

Be sure to explicitly assign values to all of the variables of the `MqlDateTime` object, or else you may not get the results you expect! The `StructToTime()` function converts the `timeStruct` object to a `datetime` value, and stores the result in the `dtTime` variable. If we print out the value of the `dtTime` variable, we can see the values that we assigned to the `timeStruct` object:

```
Print(dtTime);               // Result: 2014.10.20 02:30:00
```

Creating A Trade Timer Class

Now that we've learned how to create and convert `datetime` values and `MqlDateTime` objects, we can use this information to create a trade timer class for our expert advisor. We'll need to create `datetime` values for our start and end times. Since we'll usually be working with times that fall within the current day or week, we'll create a function that will calculate a `datetime` value for a specified time for the current date. We can then manipulate that value to get a specific date if necessary.

The CreateDateTime() Function

Let's create a function that will create a `datetime` value for the current date. We'll specify the hour and minute, and the function will return a value for that time for the current date. An `MqlDateTime` object will be used to create our `datetime` value.

All of the code in this chapter is in the `\MQL4\Include\Mql4Book\Timer.mqh` include file. Here is the code for the `CreateDateTime()` function:

```
datetime CreateDateTime(int pHour = 0, int pMinute = 0)
{
   MqlDateTime timeStruct;
   TimeToStruct(TimeCurrent(),timeStruct);

   timeStruct.hour = pHour;
   timeStruct.min = pMinute;

   datetime useTime = StructToTime(timeStruct);
```

```
   return(useTime);
}
```

First, we create an `MqlDateTime` object named `timeStruct`. We use the `TimeToStruct()` function to fill the `timeStruct` object with the current date and time using the `TimeCurrent()` function. Next, we assign the `pHour` and `pMinute` parameter values to the `hour` and `min` variables of the `timeStruct` object. Finally, we use `StructToTime()` to convert the `timeStruct` object to a `datetime` value, which we assign to the `useTime` variable. This function will return a `datetime` value for the specified hour and minute of the current date.

Once we have a `datetime` value for a specified time, we can easily manipulate this value if necessary. For example, we want our trade timer to start at 20:00 each day. We will stop trading for the day at 8:00 the following day. First, we will use our `CreateDateTime()` function to create `datetime` values for the start and end time:

```
datetime startTime = CreateDateTime(22,0);
datetime endTime = CreateDateTime(8,0);

Print("Start: ",startTime,", End: ",endTime);
// Result: Start: 2014.10.22 22:00:00, End: 2014.10.22 08:00:00
```

The problem here is that our end time is earlier than our start time. Assuming that today is October 22, and we want our timer to start at 20:00, we'll need to advance the end time by 24 hours to get the correct time. You'll recall from earlier in the chapter that 24 hours is equal to 86400 seconds. We can simply add this value to `endTime` to get the correct time.

Instead of having to remember that 86400 seconds equals a day, let's define some constants that we can easily refer to if we need to add or subtract time from a `datetime` variable. These will go at the top of our `Timer.mqh` include file:

```
#define TIME_ADD_MINUTE 60
#define TIME_ADD_HOUR 3600
#define TIME_ADD_DAY  86400
#define TIME_ADD_WEEK 604800
```

Using these constants, we can easily add or subtract the specified time period. Let's add 24 hours to our `endTime` variable:

```
endTime += TIME_ADD_DAY;
Print("Start: ",startTime," End: ",endTime);
// Result: Start: 2014.10.22 22:00:00, End: 2014.10.23 08:00:00
```

Now our start and end times are correct relative to each other.

The CTimer Class

Let's examine the simplest type of trade timer. This timer will take two `datetime` values and determine whether the current time falls between those values. If so, our trade timer is active and trading can commence. If the current time is outside of those values, then our trade timer is inactive, and trades will not be placed.

The `CTimer` class will hold our timer-related functions. We will start with the `CheckTimer()` function, which will check two `datetime` variables to see if trading is allowed:

```
class CTimer
{
   public:
      bool CheckTimer(datetime pStartTime, datetime pEndTime, bool pLocalTime = false);
};
```

Here is the body of the `CheckTimer()` function:

```
bool CTimer::CheckTimer(datetime pStartTime, datetime pEndTime, bool pLocalTime = false)
{
   if(pStartTime >= pEndTime)
   {
      Alert("Error: Invalid start or end time");
      return(false);
   }

   datetime currentTime;
   if(pLocalTime == true) currentTime = TimeLocal();
   else currentTime = TimeCurrent();

   bool tradeEnabled = false;
   if(currentTime >= pStartTime && currentTime < pEndTime)
   {
      tradeEnabled = true;
   }

   return(tradeEnabled);
}
```

The `pStartTime` parameter holds the trading start time, while the `pEndTime` parameter holds the trading end time. The `pLocalTime` parameter is a boolean value that determines whether we use local time or trade server time.

First, we check to see if `pStartTime` is greater than or equal to `pEndTime`. If so, we know that our start and end times are invalid, so we show an error message and exit with a value of `false`. Otherwise, we continue checking our timer condition.

We declare a `datetime` variable named `currentTime` that will hold the current time. If `pLocalTime` is `true`, we will use the local computer time. If it is `false`, we use the trade server time. The `TimeLocal()` and `TimeCurrent()` functions retrieve the current time from the local computer or the server, respectively. Normally, we will use the server time, but it is a simple modification to allow the user to use local time, so we have added that option.

Once the `currentTime` variable has been assigned the current time, we compare it to our trading start and end times. If `currentTime >= pStartTime` and `currentTime < pEndTime`, then the trade timer is active. We will set the `tradeEnabled` variable to `true` and return the value of `tradeEnabled` to the program.

Let's demonstrate how to create a simple trade timer using `CheckTimer()`. We'll be using two `datetime` input variables. The MetaTrader 4 interface makes it easy to input a valid `datetime` value in the *Expert Properties* dialog using a date/time picker. We'll set a start and end time, and use the `CheckTimer()` function to check for a valid trade condition:

```
#include <Mql4Book/Timer.mqh>
CTimer Timer;

// Input variables
input datetime StartTime = D'2014.10.15 08:00';
input datetime EndTime = D'2014.10.19 20:00';

// OnTick() event handler
bool tradeEnabled = Timer.CheckTimer(StartTime,EndTime);

if(tradeEnabled == true)
{
   // Order placement code
}
```

First, we include the `Timer.mqh` file and declare the `Timer` object, based on our `CTimer` class. The input variables `StartTime` and `EndTime` have a valid start and end time entered. In the `OnTick()` event handler, we call the `CheckTimer()` function and pass it our `StartTime` and `EndTime` input variables. If the current time is between the start and end times, the `tradeEnabled` variable is set to `true`. When we check our order opening conditions, we check if `tradeEnabled` is `true`. If so, we proceed with checking the order conditions and performing trade operations as appropriate.

The problem with using `datetime` input variables to set our start and end times is that they have to be constantly updated. What if you just wanted to trade the same hours each day, for example?

The DailyTimer() Function

Most savvy Forex traders know that certain hours of the day are more conducive to trading. The open of the London session through the second half of the New York session is the most active trading period of the day. We can improve the performance of our expert advisors by limiting our trading to these hours.

We'll create a timer function that will allow us to trade the same hours each day. We set a start and end hour and minute, and unless we decide to change our trading hours we need not edit it again. Our daily timer operates by the same principle as the simple timer in the previous section. We create two `datetime` values using our `CreateDateTime()` function. We compare our start and end times, and increment or decrement them by one day as necessary. If the current time falls between the start and end times, trading is enabled.

Let's add our daily timer function to the `CTimer` class. We're also going to add two private `datetime` variables – `_startTime` and `_endTime` – to the `CTimer` class. These will be used to save our current start and end times, just in case we need to retrieve them elsewhere in our program:

```
class CTimer
{
   private:
      datetime _startTime, _endTime;

   public:
      bool CheckTimer(datetime pStartTime, datetime pEndTime, bool pLocalTime = false);
      bool DailyTimer(int pStartHour, int pStartMinute, int pEndHour, int pEndMinute,
         bool pLocalTime = false);
};
```

Here is the body of our `DailyTimer()` function:

```
bool CTimer::DailyTimer(int pStartHour, int pStartMinute, int pEndHour, int pEndMinute,
   bool pLocalTime = false)
{
   datetime currentTime;
   if(pLocalTime == true) currentTime = TimeLocal();
   else currentTime = TimeCurrent();

   _startTime = CreateDateTime(pStartHour,pStartMinute);
   _endTime = CreateDateTime(pEndHour,pEndMinute);
```

```
      if(_endTime <= _startTime)
      {
         _startTime -= TIME_ADD_DAY;

         if(currentTime > _endTime)
         {
            _startTime += TIME_ADD_DAY;
            _endTime += TIME_ADD_DAY;
         }
      }

      bool tradeEnabled = CheckTimer(_startTime,_endTime,pLocalTime);
      return(tradeEnabled);
   }
```

For this function, we input the start and end hour and minute using the `int` variables `pStartHour`, `pStartMinute`, `pEndHour` and `pEndMinute`. We also include the `pLocalTime` parameter for selecting between server and local time. After assigning the current time to the `currentTime` variable, we call the `CreateDateTime()` function using the `pStartHour`, `pStartMinute`, `pEndHour` and `pEndMinute` variables. The resulting `datetime` values are stored in `_startTime` and `_endTime` respectively.

Next, we determine whether we need to increment or decrement the start and/or end time values. First, we check to see if the `_endTime` value is less than the `_startTime` value. If so, we subtract one day from the start time (using the `TIME_ADD_DAY` constant) so that the value of `_startTime` is sooner than the value of `_endTime`. If the value of `currentTime` is greater than `_endTime`, the timer has already expired and we need to set it for the next day. We increment both `_startTime` and `_endTime` by one day, so that the timer is set for the following day.

Finally, we pass the `_startTime` and `_endTime` values to the `CheckTimer()` function, along with our `pLocalTime` parameter. This determines whether the timer is currently active or not. We save the return value of `CheckTimer()` to the `tradeEnabled` variable, and return that value to the program.

Let's demonstrate how we can use the daily timer in our expert advisor programs:

```
#include <Mql4Book/Timer.mqh>
CTimer Timer;

// Input variables
input bool UseTimer = false;
input int StartHour = 0;
input int StartMinute = 0;
input int EndHour = 0;
input int EndMinute = 0;
input bool UseLocalTime = false;
```

```
// OnTick() event handler
bool tradeEnabled = true;

if(UseTimer == true)
{
   tradeEnabled = Timer.DailyTimer(StartHour,StartMinute,EndHour,EndMinute,UseLocalTime);
}

if(tradeEnabled == true)
{
   // Order placement code
}
```

We've added an input variable named `UseTimer` to turn the timer on and off. We have inputs for the start and end hour and minute, as well as for local/server time. In the `OnTick()` event handler, we declare a boolean variable named `tradeEnabled` (not to be confused with the local variable of the same name in our `DailyTimer()` function!) The value of this variable determines whether we can trade or not. We will initialize it to `true`.

If the `UseTimer` input variable is set to `true`, we call the `DailyTimer()` function. The result of the function is saved to the `tradeEnabled` variable. If `tradeEnabled` is `true`, trading will commence, and if it is `false`, we will wait until the start of the next trading day. Note that `tradeEnabled` will always be `true` if `UseTimer` is set to `false`.

The PrintTimerMessage() Function

One issue from a user perspective is knowing whether or not trading is enabled by the timer. Let's create a short function that writes a comment to the chart and to the log, to inform the user when the timer has started and stopped. The function is named `PrintTimerMessage()`, and we will make it a private member of the `CTimer` class. We will also declare a private variable named `_timerStarted` that will hold our on/off state:

```
class CTimer
{
   private:
      datetime _startTime, _endTime;
      bool _timerStarted;
      void PrintTimerMessage(bool pTimerOn);

   public:
      bool CheckTimer(datetime pStartTime, datetime pEndTime, bool pLocalTime = false);
      bool DailyTimer(int pStartHour, int pStartMinute, int pEndHour, int pEndMinute,
         bool pLocalTime = false);
};
```

Here is the code for the `PrintTimerMessage()` function:

```
void CTimer::PrintTimerMessage(bool pTimerOn)
{
   if(pTimerOn == true && _timerStarted == false)
   {
      string message = "Timer started";
      Print(message);
      Comment(message);
      _timerStarted = true;
   }
   else if(pTimerOn == false && _timerStarted == true)
   {
      string message = "Timer stopped";
      Print(message);
      Comment(message);
      _timerStarted = false;
   }
}
```

The function type is `void`, since we do not need this function to return a value. The `pTimerOn` parameter takes a boolean value from the `DailyTimer()` function or another timer function from this chapter. If `pTimerOn` is `true`, we check to see if the timer is active and print a message to the log and to the screen.

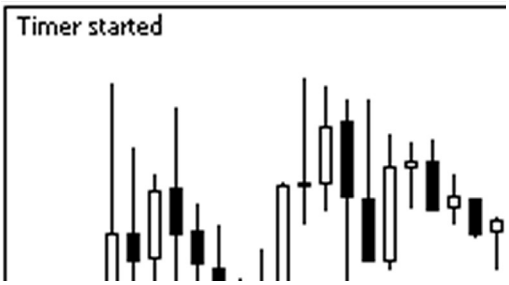


Fig. 22.1 – A chart comment generated by the `PrintTimerMessage()` function.

The `_timerStarted` variable contains the timer activation state. When the timer is first activated and `pTimerOn` is `true`, `_timerStarted` will be `false`. We'll print the message "Timer started" to the chart and to the log, and set `_timerStarted` to `true`. If `pTimerOn` is `false` and `_timerStarted` is `true`, we print the message "Timer stopped" to the chart and log, and set `_timerStarted` to `false`.

Now all we need to do is add this to the end of our `DailyTimer()` function:

```
bool tradeEnabled = CheckTimer(_startTime,_endTime,pLocalTime);
PrintTimerMessage(tradeEnabled);

return(tradeEnabled);
```

The result is that we will have one message printed to the log when the timer activates, and another when the timer deactivates. The message will also be printed to the comment area of the chart, and will remain there until it is overwritten by another comment.

The BlockTimer() Function

The trade timers we've created so far take a single start and end time. The daily timer is sufficient for most traders, but if you need more flexibility in setting your trade timer, we've created the `BlockTimer()` function. We call it the "block timer" because the user will set several blocks of time that the expert advisor will be allowed to trade each week. This allows the user to avoid volatile market events such as the non-farm payroll report.

We will specify the days to start and stop trading by using the `ENUM_DAY_OF_WEEK` enumeration. The user will select the day of the week and specify a start and end hour and minute in the expert advisor inputs. For each block of time, we need a start day, start hour, start minute, end day, end hour and end minute. We'll also need a boolean variable to indicate whether to use that timer block.

A fixed number of timer blocks will need to be specified in the inputs, according to the trader's needs. Five blocks should be enough for most traders. Due to the number of variables required for each timer block, we will create a structure that will hold them. We will then create an array based on this structure that will hold all of the variables for each timer block. This array will be passed to the `BlockTimer()` function, which will determine whether trading should be enabled.

Here is the structure that we will define in the `Timer.mqh` include file. This will be declared on the global scope, at the top of the file:

```
struct TimerBlock
{
   bool enabled;        // Enable or disable timer block
   int start_day;       // Start day (1 = Monday... 5 = Friday)
   int start_hour;      // Start hour
   int start_min;       // Start minute
   int end_day;         // End day
   int end_hour;        // End hour
   int end_min;         // End minute
};
```

Let's demonstrate how we will use this structure to fill an array object with timer information. First, we will declare the input variables in our expert advisor. We will add two timer blocks, each with the necessary inputs:

```
sinput string Block1;   // Timer Block 1
input bool UseTimer = true;
input ENUM_DAY_OF_WEEK StartDay = 5;
input int StartHour = 8;
input int StartMinute = 0;
input ENUM_DAY_OF_WEEK EndDay = 5;
input int EndHour = 12;
input int EndMinute = 25;

sinput string Block2;   // Timer Block 2
input bool UseTimer2 = true;
input ENUM_DAY_OF_WEEK StartDay2 = 5;
input int StartHour2 = 13;
input int StartMinute2 = 0;
input ENUM_DAY_OF_WEEK EndDay2 = 5;
input int EndHour2 = 18;
input int EndMinute2 = 0;

input bool UseLocalTime = false;
```

Each timer block has a `UseTimer` boolean variable to turn the timer block on and off, a `StartDay` and `EndDay` variable of type `ENUM_DAY_OF_WEEK`, and the `StartHour`, `StartMinute`, `EndHour` and `EndMinute` integer variables. Note the `Block1` and `Block2` variables with the `sinput` modifier. These are static input variables that are used for informational and formatting purposes. The comment following the variable identifier is displayed in the input window in MetaTrader:

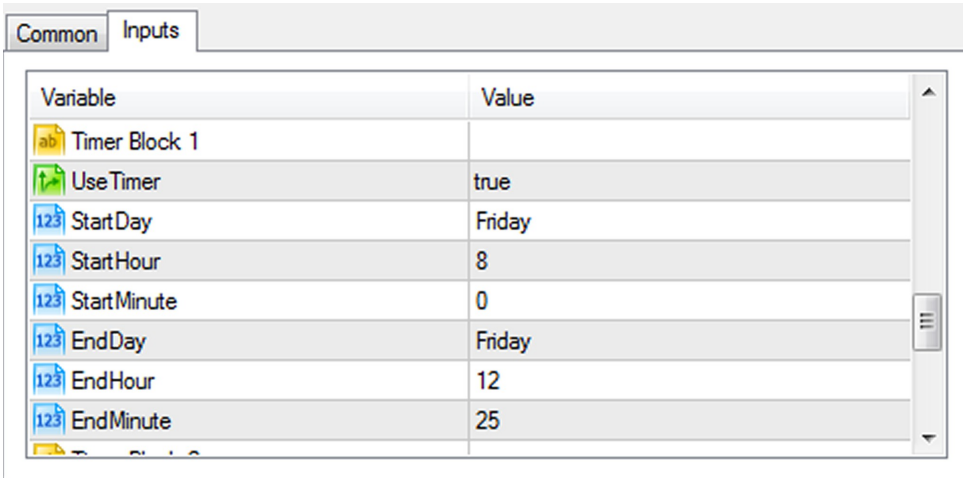


Fig. 22.2 – The block timer inputs as they appear in the expert advisor *Inputs* tab.

Now that we have our input variables declared, we will declare our `TimerBlock` object on the global scope:

```
TimerBlock block[2];
```

This creates a `TimerBlock` array object named `block`, with two elements. Next, we need to load our input values into this array. This will take quite a bit of code, but there's no easy way around this. We will copy the input variables into the array inside the `OnInit()` event handler:

```
// OnInit() event handler
block[0].enabled = UseTimer;
block[0].start_day = StartDay;
block[0].start_hour = StartHour;
block[0].start_min = StartMinute;
block[0].end_day = EndDay;
block[0].end_hour = EndHour;
block[0].end_min = EndMinute;

block[1].enabled = UseTimer2;
block[1].start_day = StartDay2;
block[1].start_hour = StartHour2;
block[1].start_min = StartMinute2;
block[1].end_day = EndDay2;
block[1].end_hour = EndHour2;
block[1].end_min = EndMinute2;
```

All of the input variables from the first timer block are copied into the member variables of the `block[]` array at element 0. The same is done for the second timer block at element 1. Now that the input values are copied into our `block[]` array, let's pass them to our timer function. This is the easy part, and it works just like the other timer functions in this chapter:

```
// OnTick() event handler
bool tradeEnabled = true;

tradeEnabled = Timer.BlockTimer(block,UseLocalTime);
```

Inside the `OnTick()` event handler, we pass the `block[]` array to the `BlockTimer()` function. Note that we do not include the brackets (`[]`) when passing the array name as a function parameter. If the current time falls inside one of our timer blocks, the value of `tradeEnabled` will be set to `true`. We can then check the order opening conditions and perform any actions that occur when our timer is active.

Let's examine our `BlockTimer()` function:

```
bool CTimer::BlockTimer(TimerBlock &pBlock[], bool pLocalTime = false)
{
   MqlDateTime today;
   bool tradeEnabled = false;
   int timerCount = ArraySize(pBlock);

   for(int i = 0; i < timerCount; i++)
   {
      if(pBlock[i].enabled = false) continue;

      _startTime = CreateDateTime(pBlock[i].start_hour, pBlock[i].start_min);
      _endTime = CreateDateTime(pBlock[i].end_hour, pBlock[i].end_min);

      TimeToStruct(_startTime,today);
      int dayShift = pBlock[i].start_day - today.day_of_week;
      if(dayShift != 0) _startTime += TIME_ADD_DAY * dayShift;

      TimeToStruct(_endTime,today);
      dayShift = pBlock[i].end_day - today.day_of_week;
      if(dayShift != 0) _endTime += TIME_ADD_DAY * dayShift;

      tradeEnabled = CheckTimer(_startTime,_endTime,pLocalTime);
      if(tradeEnabled == true) break;
   }

   PrintTimerMessage(tradeEnabled);
   return(tradeEnabled);
}
```

The `BlockTimer()` function has two parameters: The `pBlock[]` parameter takes an array object of the `TimerBlock` structure type passed by reference, and the `pLocalTime` parameter is a boolean variable that determines whether to use server or local time.

First, we declare the variables that will be used in our function. An `MqlDateTime` object named `today` will be used to retrieve the day of the week for the current date. The `timerCount` variable will hold the size of the `pBlock[]` array, which we retrieve by using the `ArraySize()` function.

A `for` loop will be used to process the timer blocks. The number of timer blocks is determined by the `timerCount` variable. We will calculate the start and end times for each timer block, and determine whether the current time falls within those times. If so, we break out of the loop and return a value of `true` to the program.

Inside the `for` loop, the first thing we check is the `enabled` variable of the current timer block. If it is `true`, we continue calculating the start and end time. We use the `CreateDateTime()` function to calculate the `_startTime` and `_endTime` variables using the `start_hour`, `start_min`, `end_hour` and `end_min` variables of the `pBlock[]` object.

Next, we calculate the correct date based on the day of the week indicated for the start and end time. We use `TimeToStruct()` to convert the `_startTime` and `_endTime` values to an `MqlDateTime` structure. The result is stored in the `today` object. The `today.day_of_week` variable will hold the current day of the week. If the current day of the week differs from the `start_day` or `end_day` value of the `pBlock[]` object, we calculate the difference and store it to the `dayShift` variable. The `dayShift` variable is multiplied by the `TIME_ADD_DAY` constant, and added or subtracted from the `_startTime` or `_endTime` variables. The result is that `_startTime` and `_endTime` are set to the correct time and date relative to the current week.

After the start and end times for the current timer block have been determined, we use the `CheckTimer()` function to see if the current time falls within the start time and end time of the current timer block. If the function returns `true`, we `break` out of the `for` loop. The `PrintTimerMessage()` function will print an informative message to the chart and the log, and the value of `tradeEnabled` is returned to the program.

Programming a flexible trade timer is the most complex thing we've had to do so far. With the timer functions we've defined in this chapter, you should be able to implement a trade timer that is flexible enough for your needs.

Retrieving _startTime and _endTime

A final bit of code to add: If for some reason you need to check the current start or end time that the trade timer is using, use the `GetStartTime()` and `GetEndTime()` functions. These functions retrieve the appropriate `datetime` variable and return it to the program. Since these are very short functions, we can declare the entire function in the class declaration:

```
class CTimer
{
   private:
      datetime _startTime, _endTime;

   public:
      datetime GetStartTime() {return(_startTime);};
      datetime GetEndTime() {return(_endTime);};
};
```

For clarity, we've removed the other functions and variables from the `CTimer` class for this example. Note the `return` statement inside the brackets after the function name. This is the function body. The sole purpose of

`GetStartTime()` and `GetEndTime()` is to return the value of the `_startTime` or `_endTime` variable to the program. If you have a short function whose only purpose is to return a value, then you can place the body right in the class declaration.

The OnTimer() Event Handler

So far we've been only been using the `OnInit()` and `OnTick()` event handlers in our expert advisors. There are several optional event handlers that can be used for specific purposes. One of those is the `OnTimer()` event handler, which executes at a predefined time interval.

The `OnTick()` event handler executes only when a change in price is received from the terminal. The `OnTimer()` event, on the other hand, can execute every *X* number of seconds. If you want your trading strategy to perform an action at regular intervals, place that code in the `OnTimer()` event handler.

To use the `OnTimer()` event handler, we need to set the timer interval using the `EventSetTimer()` function. This function is typically declared inside the `OnInit()` event handler or in a class constructor. For example, if we want the expert advisor to perform an action once every 60 seconds, here is how we set the event timer:

```
int OnInit()
{
   EventSetTimer(60);
}
```

This will execute the `OnTimer()` event handler every 60 seconds, if one exists in your program. The *MQL4 Wizard* can be used to add the `OnTimer()` event to your source code file when creating it, but you can always add it later. The `OnTimer()` event handler is of type `void` with no parameters:

```
void OnTimer()
{
   // Perform action every x seconds
}
```

If for some reason you need to stop your event timer, the `EventKillTimer()` will remove the event timer, and the `OnTimer()` event handler will no longer run. This is typically declared in the `OnDeinit()` event handler or a class destructor, although it should work anywhere in the program:

```
EventKillTimer();
```

The `EventKillTimer()` function takes no parameters, and does not return a value.

Chapter 23 - Trading Systems

We've spent the better part of this book creating functions and classes that place, close, modify and manage orders. We've added useful features such as trailing stops, money management and a trade timer. We've also created classes that allow us to work with indicators. Now we're going to show you how to put it all together in creating a fully-featured and robust expert advisor.

Creating A Template

The newest version of MetaEditor doesn't allow the use of custom templates anymore. So to save time, we will need to create our own template, which we will use when creating our expert advisors. When we create a new expert advisor, we'll simply open this file and save it under a new name.

The template that we'll be using is named `Expert Advisor Template.mq4`. To prevent this file from being overwritten, navigate to the `\MQL4\Experts\Mql4Book` folder in *Windows Explorer* and locate the `Expert Advisor Template.mq4` file. Right-click on the file and select *Properties*. Put a check mark in the *Read-only* attribute. When attempting to save a read-only file, MetaEditor will alert you and prompt you to save it to a new file.

Here is our template file. We'll go through it a section at a time. First is our `#include` directives and object declarations:

```
#include <Mql4Book\Trade.mqh>
CTrade Trade;
CCount Count;

#include <Mql4Book\Timer.mqh>
CTimer Timer;
CNewBar NewBar;

#include <Mql4Book\TrailingStop.mqh>
#include <Mql4Book\MoneyManagement.mqh>
#include <Mql4Book\Indicators.mqh>
```

We include all five include files from the `\MQL4\Include\Mql4Book` directory. We've also declared several objects for the classes in those files. The `Trade` object contains our order placement functions, `Count` is for the order counting functions, `Timer` is for the trade timer functions, and `NewBar` contains the new bar functions. If you need to add indicators to your expert advisor, you will do that after declaring the input variables.

Next, we have our input variables:

```
sinput string TradeSettings;         // Trade Settings
input int MagicNumber = 101;
input int Slippage = 10;
input bool TradeOnBarOpen = true;

sinput string MoneyManagement;       // Money Management
input bool UseMoneyManagement = true;
input double RiskPercent = 2;
input double FixedLotSize = 0.1;

sinput string Stops;                 // Stop Loss & Take Profit
input int StopLoss = 0;
input int TakeProfit = 0;

sinput string TrailingStopSettings   // Trailing Stop
input bool UseTrailingStop = true;
input int TrailingStop = 0;
input int MinProfit = 0;
input int Step = 10;

sinput string BreakEvenSettings      // Break Even Stop
input bool UseBreakEvenStop = false;
input int MinimumProfit = 0;
input int LockProfit = 0;

sinput string TimerSettings          // Timer
input bool UseTimer = false;
input int StartHour = 0;
input int StartMinute = 0;
input int EndHour = 0;
input int EndMinute = 0;
input bool UseLocalTime = false;
```

The input variables includes settings for many of the features described in this book, including money management, trailing stop, break even stop, new bar detection and a trade timer. Note the `sinput` variables – these divide our input variables into clearly defined sections in the *Inputs* tab. Any additional input variables required by your trading system can be added anywhere in this section. If you are not using a particular feature, you can remove the input variables for it..

Next, we have our global variables and indicators and the `OnInit()` event handler:

```
int gBuyTicket, gSellTicket;
```

```
int OnInit()
{
   // Set magic number
   Trade.SetMagicNumber(MagicNumber);
   Trade.SetSlippage(Slippage);
   return(INIT_SUCCEEDED);
}
```

We've added the `gBuyTicket` and `gSellTicket` variables, in case you need to keep track of recently opened orders. The `OnInit()` event handler contains code that will run once when the program is first started. We've added the `SetMagicNumber()` and `SetSlippage()` functions to set our magic number and slippage values for the program.

Next, we're going to examine the `OnTick()` event handler:

```
void OnTick()
{
   // Check timer
   bool tradeEnabled = true;
   if(UseTimer = true)
   {
      bool tradeEnabled = Timer.DailyTimer(StartHour,StartMinute,EndHour,EndMinute,
         UseLocalTime);
   }

   // Check for bar open
   bool newBar = true;
   int barShift = 0;

   if(TradeOnBarOpen == true)
   {
      newBar = NewBar.CheckNewBar(_Symbol,_Period);
      barShift = 1;
   }

   // Trading
   if(newBar == true && tradeEnabled == true)
   {
     // Money management
      double lotSize = FixedLotSize;
      if(UseMoneyManagement == true)
      {
         lotSize = MoneyManagement(_Symbol,FixedLotSize,RiskPercent,StopLoss);
      }
```

```
            // Open buy order
            if(  )
            {
               gBuyTicket = Trade.OpenBuyOrder(_Symbol,lotSize);
               ModifyStopsByPoints(gBuyTicket,StopLoss,TakeProfit);
            }

            // Open sell order
            else if(  )
            {
               int gSellTicket = Trade.OpenSellOrder(_Symbol,lotSize);
               ModifyStopsByPoints(gSellTicket,StopLoss,TakeProfit);
            }
         }

         // Break even stop
         if(UseBreakEvenStop == true)
         {
            BreakEvenStopAll(MinimumProfit,LockProfit);
         }

         // Trailing stop
         if(UseTrailingStop == true)
         {
            TrailingStopAll(TrailingStop,MinProfit,Step);
         }
      }
```

We start with our daily timer code. If the `UseTimer` input variable is set to `true`, we check the output of the `DailyTimer()` function and set `tradeEnabled` to `true` or `false`. Next is our new bar detection code. If the `TradeOnBarOpen` input variable is set to true, we check the output of the `CheckNewBar()` function to set the `newBar` variable. If both `tradeEnabled` and `newBar` are true, we will proceed with checking our trade conditions.

The money management code will calculate a lot size if `UseMoneyManagement` is set to `true` – otherwise we use the value in `FixedLotSize`. We have some skeleton code to open buy and sell market orders, and add a fixed stop loss and/or take profit to the order. The order tickets are saved to the `gBuyTicket` and `gSellTicket` variables, respectively. Finally, we check for a break even stop or trailing stop.

This template is just a starting point. The code here will work well for most indicator-based trading systems that open market orders, and includes many of the features we have discussed in this book. You can modify it and remove features as necessary. Note that we haven't included any order closing code, since that depends on your trading system. You'll need to add indicators and other features as necessary.

We're going to create two trading systems based on this template. The first is a moving average cross, which is a common trend trading system. The second is a counter-trend system that uses the stochastic indicator.

Moving Average Cross

The moving average cross uses two moving averages, a fast MA and a slow MA. When the fast MA is greater than the slow MA, we open a buy position. When the fast MA is less than the slow MA, we open a sell position. Only one position is opened in each direction. When the moving averages cross in the opposite direction, we'll close the open order before opening a new one.

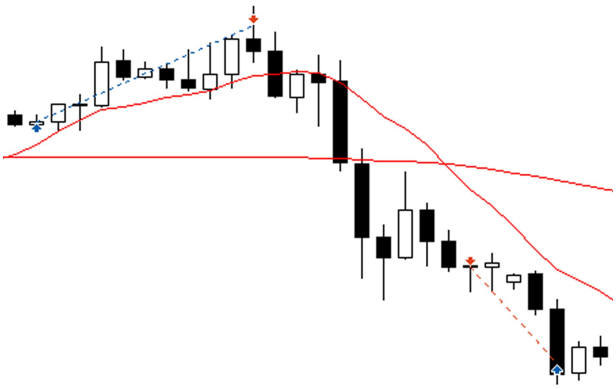


Fig. 23.1 – The moving average cross system.

The code highlighted in bold is the new code that is added to our template above. The first thing we'll need to add is the moving average indicators. First, we add the input variables to adjust the moving average settings, followed by the `CiMA` class objects:

```
sinput string FastMaSettings;   // Fast Moving Average
input int FastMaPeriod = 5;
input ENUM_MA_METHOD FastMaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE FastMaPrice = PRICE_CLOSE;
input int MinCrossSpread = 10;

sinput string SlowMaSettings;   // Slow Moving Average
input int SlowMaPeriod = 20;
input ENUM_MA_METHOD SlowMaMethod = MODE_EMA;
input ENUM_APPLIED_PRICE SlowMaPrice = PRICE_CLOSE;

//+------------------------------------------------------------------+
//| Global variable and indicators                                   |
//+------------------------------------------------------------------+

CiMA FastMa(_Symbol,_Period,FastMaPeriod,0,FastMaMethod,FastMaPrice);
CiMA SlowMa(_Symbol,_Period,SlowMaPeriod,0,SlowMaMethod,SlowMaPrice);
```

Note the `MinCrossSpread` input variable. This is the minimum distance between the fast and slow moving average that is required before we open an order. Our moving average indicator objects are named `FastMa` and `SlowMa`.

In the `OnTick()` event handler, our trading actions are carried out inside the `if(newBar == true)` block. If `newBar` is `true` (meaning that a new bar has opened, or `TradeOnBarOpen` is set to `false`), we will check our order opening and closing conditions:

```
if(newBar == true)
{
   double minSpread = MinCrossSpread * _Point;

   // Close orders
   if(FastMa.Main(barShift) > SlowMa.Main(barShift) + minSpread)
   {
      Trade.CloseAllSellOrders();
   }
   else if(FastMa.Main(barShift) < SlowMa.Main(barShift) - minSpread)
   {
      Trade.CloseAllBuyOrders();
   }
```

First, we set the value of the `minSpread` variable by multiplying `MinCrossSpread` by the point value. We will use this value shortly. Next are the order closing conditions. If the fast moving average price is greater than the slow moving average price, plus the value of `minSpread`, we will close any open sell orders. If the opposite is true, we close any open buy orders.

```
   // Money management
   double lotSize = FixedLotSize;
   if(UseMoneyManagement == true)
   {
      lotSize = MoneyManagement(_Symbol,FixedLotSize,RiskPercent,StopLoss);
   }

   // Open buy order
   if( FastMa.Main(barShift) > SlowMa.Main(barShift) + minSpread
      && FastMa.Main(barShift + 1) <= SlowMa.Main(barShift + 1) + minSpread
      && Count.Buy() == 0 )
   {
      int ticket = Trade.OpenBuyOrder(_Symbol,lotSize);
      Trade.ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
   }

   // Open sell order
   else if( FastMa.Main(barShift) < SlowMa.Main(barShift) - minSpread
      && FastMa.Main(barShift + 1) >= SlowMa.Main(barShift + 1) - minSpread
      && Count.Sell() == 0 )
   {
```

```
      int ticket = Trade.OpenSellOrder(_Symbol,lotSize);
      Trade.ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
   }
}

// Trailing stop
if(UseTrailingStop == true)
{
   TrailingStopAll(TrailingStop,MinProfit,Step);
}
```

Before we open any orders, we determine our lot size by calling the `MoneyManagement()` function and saving the result to the `lotSize` variable. Then we check our order opening conditions. To open a buy order, we are looking for a recent cross of the moving average indicators. If the fast moving average is greater than the slow moving average, but the reverse was true on the previous bar, then we know that a cross has occurred. We add the `minSpread` value to the slow moving average price to ensure that the moving averages are a minimum distance apart.

If our buy order conditions are true, and there is currently not a buy order open, we will open a buy order and set the stop loss and take profit. The conditions for opening a sell order are the inverse of the buy order conditions. Outside of the order placement block, we check for trailing stops on any open orders.

That's it! We now have a functioning moving average cross with money management and trailing stop. You can view the code in the `Moving Average Cross.mq4` file in the `\MQL4\Experts\Mql4Book` folder.

Stochastic Counter Trend

Our second market order trading system uses the stochastic indicator to place counter-trend orders. A counter-trend system attempts to pick the tops and bottoms of trends, and places orders at price extremes. For our stochastic trading system, we will place a buy order when the stochastic reading is oversold, and place a sell order when the stochastic reading is overbought.

Let's start with the input variables and the indicator object declaration:

```
sinput string ST;    // Stochastics Settings
input int KPeriod = 14;
input int DPeriod = 3;
input int Slowing = 5;
input ENUM_MA_METHOD MaMethod = MODE_SMA;
input int PriceField = 0;
```

```
//+------------------------------------------------------------------+
//| Global variables and indicators                                  |
//+------------------------------------------------------------------+

CiStochastic Stoch(_Symbol,_Period,KPeriod,DPeriod,Slowing,MaMethod,PriceField);
```

The `Stoch` object, based on our `CiStochastic` class is for our stochastic indicator. We've added the relevant settings for the stochastic indicator and passed them to the `Stoch` object constructor.

```
void OnTick()
{
   // Money management
   double lotSize = FixedLotSize;
   if(UseMoneyManagement == true)
   {
      lotSize = MoneyManagement(_Symbol,FixedLotSize,RiskPercent,StopLoss);
   }

   // Close buy order
   if(Stoch.Main(1) < Stoch.Signal(1) && Count.Buy() > 0)
   {
      Trade.CloseAllBuyOrders();
   }

   // Close sell order
   if(Stoch.Main(1) > Stoch.Signal(1) && Count.Sell() > 0)
   {
      Trade.CloseAllSellOrders();
   }

   // Open buy order
   if( Stoch.Main(2) < 20 && Stoch.Main(2) < Stoch.Signal(2)
      && Stoch.Main(1) > Stoch.Signal(1) && Count.Buy() == 0 )
   {
      int ticket = Trade.OpenBuyOrder(_Symbol,lotSize);
      Trade.ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
   }

   // Open sell order
   else if( Stoch.Main(2) > 80 && Stoch.Main(2) > Stoch.Signal(2)
      && Stoch.Main(1) < Stoch.Signal(1) && Count.Sell() == 0 )
   {
      int ticket = Trade.OpenSellOrder(_Symbol,lotSize);
      Trade.ModifyStopsByPoints(ticket,StopLoss,TakeProfit);
   }
}
```

This is the entirety of the `OnTick()` function. We will check the order closing conditions first. If the main stochastic line moves below the signal line, we will close any open buy orders. If the reverse is true, we will close any open sell orders.

Next is our order opening conditions. For a buy order, we are looking for the stochastic to dip below 20, followed by a cross of the stochastic line above its signal line. If this occurs, and there are no buy orders currently open, we will open a buy order and set a stop loss and take profit. The sell order conditions are the inverse of the buy order conditions, with the exception that the stochastic value goes above 80.

Fig. 23.2 – The stochastic counter-trend system

Pending Breakout System

The template that we created earlier in this chapter is for market orders, but it can easily be modified to place pending orders as well. We will create a pending order-based trading system that finds the highest and lowest price of the last *X* bars. When the trade timer starts, a pending buy stop order will be placed at the highest high and a pending sell stop order will be placed at the lowest low. The stop loss for both orders will be placed at the opposite price. Only one trade in each direction will be placed, and all trades will be closed at the timer end time.

Here are the input and global variables for this trading system:

```
sinput string MM;    // Money Management
input bool UseMoneyManagement = true;
input double RiskPercent = 2;
input double FixedLotSize = 0.1;

sinput string TS;    // Trade Settings
input int MagicNumber = 101;
input int HighLowBars = 8;
input int TakeProfit = 0;

sinput string BE;    // Break Even Stop
input bool UseBreakEvenStop = false;
input int MinimumProfit = 0;
input int LockProfit = 0;
```

```
sinput string TI;    // Timer
input int StartHour = 0;
input int StartMinute = 0;
input int EndHour = 0;
input int EndMinute = 0;
input bool UseLocalTime = false;

int gBuyTicket, gSellTicket;
```

We've added an input variable named `HighLowBars`, which is the number of recent bars that we will search for the highest high and the lowest low prices. The `gBuyTicket` and `gSellTicket` variables hold our ticket numbers. We set them when we open our pending orders at the daily start time, and reinitialize them to zero at the daily end time.

```
void OnTick()
{
   // Check timer
   bool tradeEnabled = Timer.DailyTimer(StartHour,StartMinute,EndHour,EndMinute,
      UseLocalTime);

   // Close all orders
   if(tradeEnabled == false)
   {
      Trade.CloseAllMarketOrders();
      Trade.DeleteAllPendingOrders();

      gBuyTicket = 0;
      gSellTicket = 0;
   }
```

At the start of the `OnTick()` event handler, we check the trade timer to see if the current time is between our daily start and end times. If so, the `tradeEnabled` variable is set to `true`. If `tradeEnabled` is set to `false`, we close all market and pending orders, and set `gBuyTicket` and `gSellTicket` to zero.

```
   // Open orders
   else
   {
      // Calculate highest high & lowest low
      int hShift = iHighest(_Symbol,_Period,MODE_HIGH,HighLowBars);
      int lShift = iLowest(_Symbol,_Period,MODE_LOW,HighLowBars);

      double hHigh = High[hShift];
      double lLow = Low[lShift];
      double difference = (hHigh - lLow) / _Point;
```

If `tradeEnabled` is set to true, the code above is executed. First, we retrieve the highest high and lowest low of the last *X* bars. The `HighLowBars` input variable determines the number of bars to calculate for. The `iHighest()` and `iLowest()` functions retrieve the shift values of the highest high and lowest low bars. We pass these values into the `High[]` and `Low[]` predefined arrays, and the highest high and lowest low prices are stored in `hHigh` and `lLow` respectively. Lastly, we calculate the difference between the highest high and lowest low, and store that value in the `difference` variable. We'll use this value when calculating the lot size.

```
   // Money management
   double lotSize = FixedLotSize;
   if(UseMoneyManagement == true)
   {
      lotSize = MoneyManagement(_Symbol,FixedLotSize,RiskPercent,(int)difference);
   }

   // Open buy stop order
   if( Count.Buy() == 0 && Count.BuyStop() == 0 && gBuyTicket == 0 )
   {
      double orderPrice = hHigh;
      orderPrice = AdjustAboveStopLevel(_Symbol,orderPrice);

      double buyStop = lLow;
      double buyProfit = BuyTakeProfit(_Symbol,TakeProfit,orderPrice);

      gBuyTicket = Trade.OpenBuyStopOrder(_Symbol,lotSize,orderPrice,buyStop,buyProfit);
   }

   // Open sell stop order
   if( Count.Sell() == 0 && Count.SellStop() == 0 && gSellTicket == 0)
   {
      double orderPrice = lLow;
      orderPrice = AdjustBelowStopLevel(_Symbol,orderPrice);

      double sellStop = hHigh;
      double sellProfit = SellTakeProfit(_Symbol,TakeProfit,orderPrice);

      gSellTicket =
         Trade.OpenSellStopOrder(_Symbol,lotSize,orderPrice,sellStop,sellProfit);
   }

   // Break even stop
   if(UseBreakEvenStop == true)
   {
      BreakEvenStopAll(MinimumProfit,LockProfit);
   }
}
```

To calculate our lot size using the `MoneyManagement()` function, we calculated the difference between the highest high and lowest low, and stored that value in the `difference` variable. We pass that value to the `MoneyManagement()` function as our stop loss distance. Note that we explicitly cast the value of the `difference` variable to an integer `(int)`, since the `difference` variable was calculated as a double.

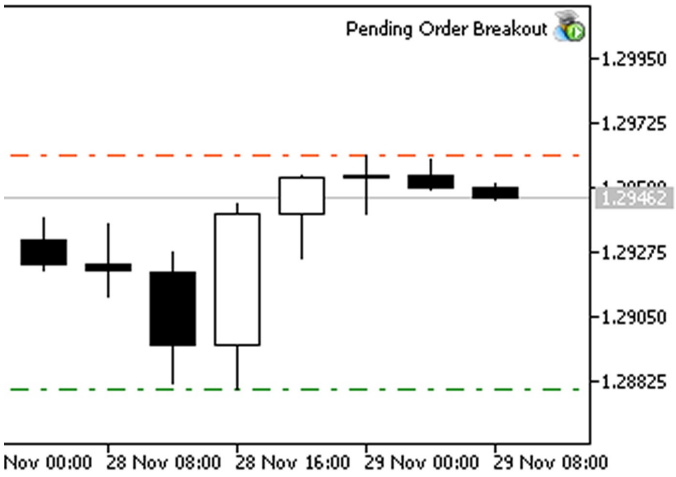


Fig 23.3 – The pending order breakout system.

Next, we open our pending buy and sell orders. Every time the trade timer starts, a buy and sell stop order will be placed. For the buy stop order, the highest high is our order opening price, and the lowest low is the stop loss. The reverse is true for the sell stop order. We verify the order opening prices, calculate a take profit price if one has been specified, and then we place the order, setting the value of `gBuyTicket` and `gSellTicket` with our order ticket numbers. The orders will only be placed once each day.

Once a pending order has opened, the break even stop will come into effect, and move the stop loss to break even once a certain amount of profit has been achieved. If an order hits its stop loss or take profit price, it will be closed for the day. Otherwise, the order will be closed at the timer end time. You can view the code for this expert advisor in the `\Experts\Mql4Book\Pending Order Breakout.mq4` file.

We've just demonstrated how to create several basic trading strategies using the techniques we've discussed in this book. Although a more advanced trading strategy may require significant modification to the template file, the basics are there so you can easily implement your strategy without having to code a new one from scratch.

Chapter 24 - Tips & Tricks

This chapter covers additional MQL4 features that may be useful in your expert advisor projects. We'll discuss methods to inform users of errors and trade events, work with chart objects in MetaTrader, learn how to read and write to CSV files, work with terminal variables, and learn how to stop execution of an expert advisor and close the trade terminal.

User Information and Interaction

It is sometimes necessary to inform the user of trade actions, errors or other events. This section examines dialog boxes, email and mobile notifications, sound alerts and chart comments.

Alert() Function

If you need to alert the user to an error or other adverse condition, the built-in *Alert* dialog is ideal. The Alert dialog window shows a running log of alert messages, indicating the time and symbol for each. If sound events are enabled in the *Options* tab of the *Tools* menu, then an alert sound will play when the dialog appears.

The `Alert()` function is used to show the alert dialog. The `Alert()` function takes any number of arguments, of any type. We've used the `Alert()` function all throughout our include files. For example, here is an `Alert()` function call when an error occurs in our `CTrade::OpenMarketOrder()` function in `Trade.mqh`:

```
if(checkError == false)
{
   Alert("Open ",orderType," order: Error ",errorCode," - ",errDesc);
}
```

This function has several arguments, including strings and an integer variable, all separated by commas.

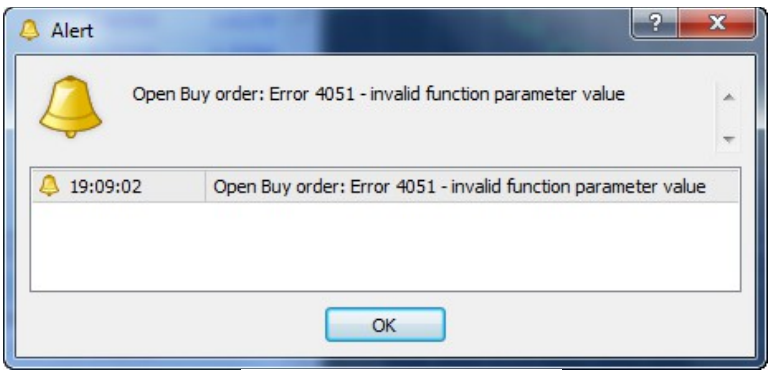


Fig. 24.1 – The *Alert* dialog.

MessageBox() Function

If you prefer something a little more elaborate, or if you need to accept user input, the `MessageBox()` function allows

you to display a standard Windows dialog box with user-defined text, caption, icons and buttons. Here is the definition of the `MessageBox()` function:

```
int  MessageBox(
   string  text,              // Message text
   string  caption=NULL,      // Title bar caption
   int     flags=0            // Button and icon constants
  );
```

The `text` parameter is the text to show in the dialog window. The `caption` parameter is the text to show in the title bar of the dialog window. The `flags` parameter defines which buttons to show, as well as icons and default buttons. The `MessageBox()` flags can be viewed in the *MQL4 Reference* under *Standard Constants*... > *Input/Output Constants* > *MessageBox*. Here are a few of the most commonly-used button flags:

- **MB_OK** – *OK* button
- **MB_OKCANCEL** – *OK* and *Cancel* buttons
- **MB_YESNO** – *Yes* and *No* buttons
- **MB_RETRYCANCEL** – *Retry* and *Cancel* buttons

Let's create a simple message box with text, a caption and yes/no buttons. For example, if you want to prompt to user to place an order when a trading signal is received:

```
int place = MessageBox("Place the order?","Confirmation",MB_YESNO);
```

This will create the message box dialog shown to the right.

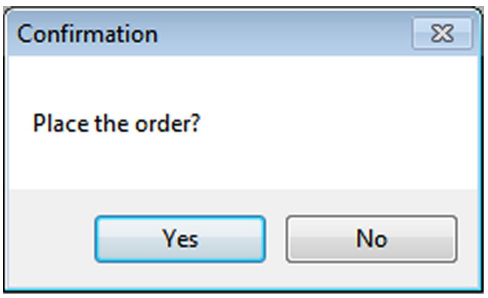


Fig. 24.2 – The *Message Box* dialog.

We can add context to our message box by adding an icon. Since we are prompting the user with a question, we'll use the question mark icon. Here are the flags for the message box icons:

- **MB_ICONERROR** – A red error icon.
- **MB_ICONQUESTION** – A question mark icon.
- **MB_ICONWARNING** – An exclamation/warning icon.
- **MB_ICONINFORMATION** – An information icon.

Here is how we would add the question mark icon to our message box. Flags must be separated with a pipe character (|):

```
int place = MessageBox("Place the order?","Confirmation",MB_YESNO|MB_ICONQUESTION);
```

Fig. 24.3 shows the message box dialog with the question mark icon.

When presented with the message box, the user will click a button and the box will close. If the message box is a simple information or error message with a single OK button, then we do not need to do anything else. But if the message box has several buttons, we will need to retrieve the value of the button that the user pressed and take appropriate action.

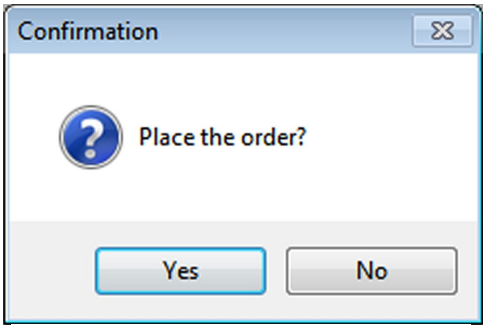


Fig. 24.3 – The *Message Box* dialog with icon.

In the example below, the `place` integer variable will hold the return value of the `MessageBox()` function. The return value for the *Yes* button is `IDYES`, while the return value for the *No* button is `IDNO`. If the user clicked Yes, then we need to proceed to place the order:

```
int place = MessageBox("Place the order?","Confirmation",MB_YESNO|MB_ICONQUESTION);

if(place == IDYES)
{
   // Open position
}
```

You can view additional return value constants for `MessageBox()` buttons in the *MQL4 Reference* under *Standard Constants... > Input/Output Constants > MessageBox*.

SendMail() Function

If you'd like your expert advisor to send you an email whenever a trade is placed, simply add the `SendMail()` function to your expert advisor after the order placement has been confirmed. You will need to enable email notifications in MetaTrader under the *Tools* menu > *Options* > *Email* tab. Under the *Email* tab, you will enter your email server information and your email address.

The `SendMail()` function takes two parameters. The first parameter is the `subject` of the message, and the second is the `message` text. For example, if you want to send an email just after a buy position has been opened:

```
if(ticket > 0)
{
   OrderSelect(ticket);
   SendMail("Buy Order Opened", "Buy order opened on " + OrderSymbol() + " at "
      + OrderOpenPrice() );
}
```

If `ticket` is greater than zero, indicating that an order has been placed, we first use the `OrderSelect()` function to select that order ticket. The `OrderSymbol()` and `OrderOpenPrice()` functions insert the order symbol and price into the `message` parameter. An email will be sent to the email address defined in the *Email* tab of the *Tools* menu > *Options* dialog.

Sending Mobile Notifications

A mobile edition of MetaTrader is available for iPhone and Android devices. Your expert advisors can send trade notifications to the mobile MetaTrader terminal on your smartphone by using the `SendNotification()` function. You will need to configure MetaTrader to send notifications under the *Tools* menu > *Options* > *Notifications* tab. In the *Notifications* tab, you will enter your MetaQuotes ID, which you can retrieve from your mobile version of MetaTrader.

The `SendNotification()` function takes one parameter, a string indicating the text to send. You can use the `SendNotification()` function anywhere you would use the `SendMail()` function. Here's an example:

```
if(ticket > 0)
{
   OrderSelect(ticket);
   SendNotification("Buy order opened on " + OrderSymbol() + " at " + OrderOpenPrice());
}
```

Playing Sound

The `PlaySound()` function will play a WAV sound file located in the `\Sounds` directory inside the MetaTrader 4 installation folder. This can be used to play an audible alert when a trade is placed, or whenever you want the user's attention. MetaTrader comes with several sound files, but you can find more online.

If you want the user to be able to choose the sound file, use an input string variable to enter the file name:

```
input string SoundFile = "alert.wav";
PlaySound(SoundFile);
```

Comment() Function

The `Comment()` function will display text in the top left corner of the chart. This is useful for informing users of actions taken by the expert advisor, such as modifying or closing orders. We've been using the `Comment()` function throughout our expert advisors to write informative comments to the chart. Here's an example from the `CTrade::OpenMarketOrder()` function in `Trade.mqh`:

```
Comment(orderType," order #",ticket," opened on ",pSymbol);
```

The problem with using chart comments is that all programs have access to this function. So if you have an indicator and an expert advisor that both write comments to the chart using the `Comment()` function, they will overwrite each other. If you need to display information on the chart that is always present, then consider using chart objects.

Chart Objects

Chart objects consist of lines, technical analysis tools, shapes, arrows, labels and other graphical objects. You can insert objects on a chart using the *Insert* menu in MetaTrader. MQL4 has a variety of functions for creating, manipulating and retrieving information from chart objects.

Creating and Modifying Objects

The `ObjectCreate()` function is used for creating new chart objects. Here is the function definition for `ObjectCreate()`:

```
bool ObjectCreate(
   string object_name,        // object name
   ENUM_OBJECT object_type,   // object type
   int sub_window,            // window index
   datetime time1,            // time of the first anchor point
   double price1,             // price of the first anchor point
   datetime time2 = 0,        // time of the second anchor point
   double price2 = 0,         // price of the second anchor point
   datetime time3 = 0,        // time of the third anchor point
   double price3 = 0          // price of the third anchor point
  );
```

The `object_name` parameter is the name of the object to create. We will need to reference this name anytime we need to modify or use the object. The `object_type` parameter is the type of object to create. It takes a value of the `ENUM_OBJECT` enumeration type. The object types can be viewed in the *MQL4 Reference* under *Standard Constants...* > *Objects Constants* > *Object Types*.

The `sub_window` parameter is the index of the chart subwindow to draw the object in. The main chart window has an index of 0. Any subwindows created by indicators (such as the RSI or MACD) are numbered starting at 1. The `time1` and `price1` parameters are the time and price for the first anchor point of the object. Most objects have one or more anchor points, although some do not use them, or use only one or the other. The first set of anchor points must always be specified, even if they are not used.

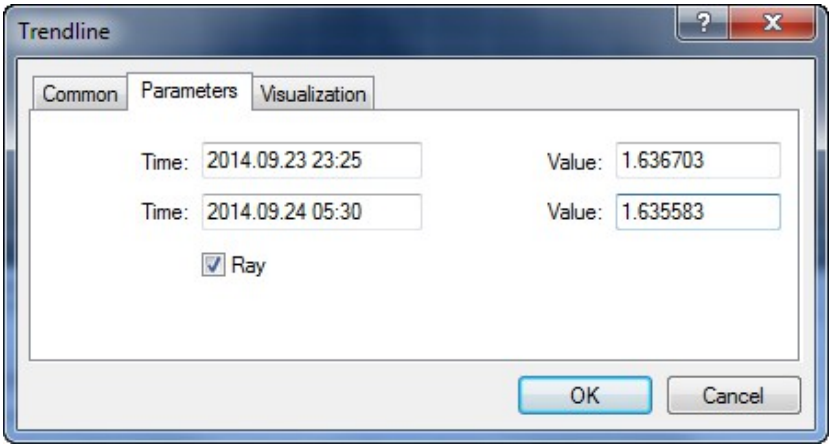


Fig. 24.4 – The *Parameters* tab of the *Object properties* dialog. This Trend line object has two pairs of time and price anchor points.

If the object uses more than one set of anchor points, they must also be specified. For example, most trend line objects use two anchor points. So in the `ObjectCreate()` function, a second set of anchor points would be passed to the function. If in doubt about the number of anchor points required for an object, simply attach the object onto a chart, right-click it and open the object properties dialog. Under the *Parameters* tab, you'll see the number of time and/or price parameters required.

Object Properties

The `ObjectCreate()` function will create the object in the location specified. However, we will still need to set the properties of the object. The `ObjectSet...()` functions are used to set the object's properties. There are three `ObjectSet...()` functions: `ObjectSetInteger()`, `ObjectSetDouble()` and `ObjectSetString()`. These functions were added in MQL5.

The classic MQL4 `ObjectSet()` function can also be used to set common object properties, but the newer `ObjectSet...()` functions allow for a wider range of properties to be set. Let's look at the function definition of `ObjectSetInteger()`:

```
bool ObjectSetInteger(
   long chart_id,         // chart identifier
   string name,           // object name
   int prop_id,           // property
   long prop_value        // value
  );
```

Since we will be modifying objects on the current chart, `chart_id` will be 0. The `name` parameter is the name of the object to modify, as set by the `ObjectCreate()` function. The `prop_id` parameter takes a value from the `ENUM_OBJECT_PROPERTY_INTEGER` enumeration, which indicates the property to modify. The

ENUM_OBJECT_PROPERTY_INTEGER constants can be viewed in the *MQL4 Reference* under *Standard Constants... > Objects Constants > Object Properties*. Finally, the prop_value parameter is the value to set.

As an example, let's create a trend line object and set a few of its properties. The object type for a trend line is OBJ_TREND. Since a trend line has two anchor points, we will need to pass two sets of time and price values. We'll assume that the time1, time2, price1 and price2 variables are filled with the appropriate values:

```
datetime time1, time2;
double price1, price2;

ObjectCreate("Trend",OBJ_TREND,0,time1,price1,time2,price2);
```

This creates a trend line object named "Trend" on the current chart. Next, let's modify a few of the properties of our trend line. We're going to modify the color, the style and the ray right properties:

```
ObjectSetInteger(0,"Trend",OBJPROP_COLOR,clrGreen);
ObjectSetInteger(0,"Trend",OBJPROP_STYLE,STYLE_DASH);
ObjectSetInteger(0,"Trend",OBJPROP_RAY_RIGHT,true);
```

The first ObjectSetInteger() function call adjusts the color property, using the OBJPROP_COLOR constant. The value used is the color constant for green, clrGreen. The second call of the function adjusts the line style, using the OBJPROP_STYLE constant. The value is STYLE_DASH, a constant of the ENUM_LINE_STYLE type. Finally, we use the OBJPROP_RAY_RIGHT constant to set the ray right property to true. The ray right property extends the trend line to the right beyond the second anchor point.

Now we have a green dashed trend line that extends to the right. All of the examples above use integer types to adjust the properties. The object property constants are listed in the *MQL4 Reference* under *Standard Constants... > Objects Constants > Object Properties*.

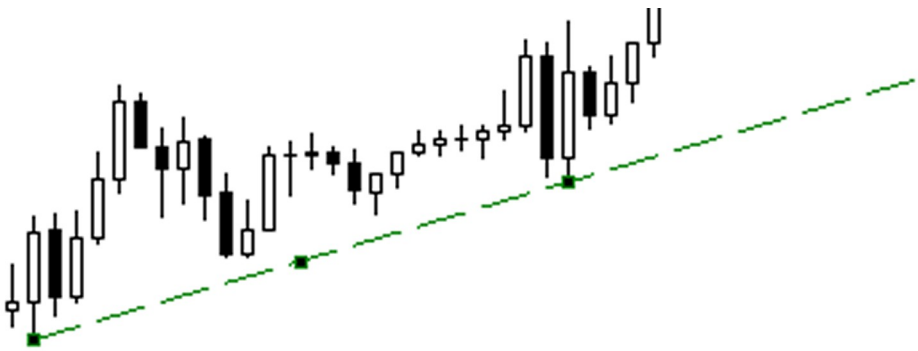


Fig. 24.5 – A trend line object with the color, style and ray right properties set.

If you need to move an object, the `ObjectMove()` function allows you to adjust one of the anchor points of an object. Here is the function definition for `ObjectMove()`:

```
bool  ObjectMove(
   long      chart_id,        // chart identifier
   string    name,            // object name
   int       point_index,     // anchor point number
   datetime  time,            // time
   double    price            // price
 );
```

The `name` parameter is the name of the object to modify. The `point_index` parameter is the anchor point to modify. Anchor point numbers start at 0, so the first anchor point would be 0, the second would be 1, and so on. The `time` and `price` parameters modify the time and price for the specified anchor point.

Retrieving Time and Price From Line Objects

When using line objects such as trend lines or channels, you may need to retrieve the price value of a line at a particular bar. Assuming that we know the timestamp of the bar, we can easily retrieve the price. The `ObjectGetValueByTime()` function will retrieve the price for a specified time:

```
double  ObjectGetValueByTime(
   string    name,         // object name
   datetime  time,         // time
   int       line_id       // line number
 );
```

The `name` parameter is the name of a line or channel object on our chart. The `time` parameter is the timestamp of a bar that intersects the line, and the `line_id` parameter is for channel objects that have several lines. For a trend line, `line_id` would be 0.

Using the `"Trend"` line object we created above, here is how we would retrieve the price for the current bar. We will use `Time[0]` to get the timestamp for the current bar:

```
double trendPrice = ObjectGetValueByTime(0,"Trend",Time[0],0);
```

The `ObjectGetValueByTime()` function will return the price of the trend line for the current bar and save the result to the `trendPrice` variable.

Now, what if you have a price, and you want to know what bar comes closest to that price? The `ObjectGetTimeByValue()` function will return the timestamp of the bar where a line object intersects a given price:

```
datetime  ObjectGetTimeByValue(
   long    chart_id,     // chart identifier
   string  name,         // object name
   double  value,        // price
   int     line_id       // line number
 );
```

The `value` parameter is the price to search for. The function will return the timestamp of the closest bar where the line intersects the price:

```
datetime trendTime = ObjectGetTimeByValue(0,"Trend",1.265,0);
```

Label and Arrow Objects

Earlier in the chapter, we talked about using the `Comment()` function to write information to the current chart. The *label* object can also be used for this purpose. Unlike chart comments, label objects can be placed anywhere on the chart, and they can use any color or font that you like.

The object type constant for the label object is `OBJ_LABEL`. Label objects do not use anchor points, but rather use the *corner*, *x-distance* and *y-distance* properties for positioning. Here's an example of the creation and positioning of a label object:

```
ObjectCreate(0,"Label",OBJ_LABEL,0,0,0);

ObjectSetInteger(0,"Label",OBJPROP_CORNER,1);
ObjectSetInteger(0,"Label",OBJPROP_XDISTANCE,20);
ObjectSetInteger(0,"Label",OBJPROP_YDISTANCE,40);
```

We've created a label object named `"Label"` and set the position to the bottom left corner using the `ObjectSetInteger()` function with the `OBJPROP_CORNER` property. The label is 20 pixels from the left border (`OBJPROP_XDISTANCE`) and 40 pixels above the bottom border (`OBJPROP_YDISTANCE`).

Note that the `time1` and `price1` parameters for the `ObjectCreate()` function are set to 0, since they are not used when creating label objects. The corners are labeled from 0 to 3, starting counter-clockwise from the top-left corner. A value of 1 is the lower-left corner, while 3 would be the upper-right corner. The x-distance and y-distance are set in pixels from the specified corner.

Next, we'll need to set the color, font and text of the label object. We'll use the `ObjectSetInteger()` and `ObjectSetString()` functions to set these properties:

```
double price = Bid;

ObjectSetInteger(0,"Label",OBJPROP_COLOR,clrWhite);
ObjectSetString(0,"Label",OBJPROP_FONT,"Arial");
ObjectSetInteger(0,"Label",OBJPROP_FONTSIZE,10);
ObjectSetString(0,"Label",OBJPROP_TEXT,"Bid: "+(string)price);
```

We've set the color of the label object to white, using 10 point Arial font. The text of the label contains the current Bid price. Every time this code is run, the label object will be updated with the current Bid price.

Fig. 24.6 – The label object.

In summary, the label object is ideal for printing useful information to the chart. The object is anchored to the chart window itself, and will not move if the chart is scrolled. The font, color, position and text are fully adjustable.

The last object type we'll examine is the arrow object. You may wish to draw an arrow on the chart when an order is placed. MQL4 defines two arrow object types that are useful for marking buy and sell signals – `OBJ_ARROW_BUY` and `OBJ_ARROW_SELL`. Arrow objects use one anchor point – the price and time where the arrow should be placed:

```
string time = Time[0];
double price = Ask;

ObjectCreate(0,"BuyArrow"+time,OBJ_ARROW_BUY,0,time,price);
```

The example above will set a buy arrow object at the current Ask price on the current bar. We append the time of the current bar to the object name so that each arrow we draw will have a unique name. This code would be called after a trade has been placed.

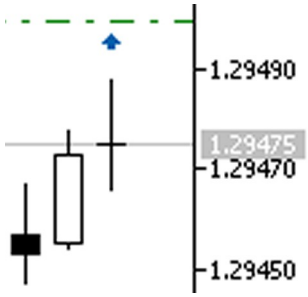


Fig. 24.7 – The buy arrow object.

Deleting Objects

To delete an object, simply call the `ObjectDelete()` function and pass the `chart_id` (usually 0) and the object `name`:

```
ObjectDelete(0,"Label");
```

To delete all objects from a chart, use the `ObjectsDeleteAll()` function. You can choose to delete only objects of a certain type or from certain subwindows. To delete all objects from the current chart, use this call:

```
ObjectsDeleteAll(0);
```

It is a good idea to call this function from your program's `OnDeinit()` function to ensure that all objects are deleted when a program is removed from the chart.

File Functions

MQL4 has a set of functions for reading and writing to files. You can, for example, create a log of your expert advisor trades, or import/export trading signals to a file. All files must be in the `\MQL4\Experts\Files` folder of your MetaTrader 4 installation.

We will examine how to read and write to a CSV file. A CSV (*comma separated value*) file contains data much like a spreadsheet. You can create and view CSV files in programs such as Microsoft Excel or OpenOffice Calc, as well as any text editor. In this example, our CSV file will record information about trades. We will write the symbol, open price, stop loss, take profit and open time of a trade to each line of the file. We will then read that information back into our program.

The `FileOpen()` function opens files for reading and writing. If the file does not exist, it will be created. Here is the definition of the `FileOpen()` function:

```
int  FileOpen(
   string  file_name,           // File name
   int     open_flags,          // Combination of flags
   short   delimiter='\t'       // Delimiter
   uint    codepage=CP_ACP      // Code page
  );
```

The `file_name` parameter is the name of the file to open in the `\MQL4\Experts\Files` directory. The `open_flags` parameter contains a combination of flags describing the file operations. The file opening flags can be viewed in the *MQL4 Reference* under *Standard Constants... > Input/Output Constants > File Opening Flags*. The `delimiter` parameter is the field delimiter for a CSV file. The default is the tab character, but we will use a comma. The `codepage` parameter will be left at its default.

The `FileOpen()` function returns an integer that will serve as the file handle. Every time we perform an operation on the file, we will need to reference the file handle. For the file opening flags, we will be using a combination of `FILE_READ`, `FILE_WRITE` and `FILE_CSV`.

Writing to a CSV File

The `FileWrite()` function is used to write a line of data to a CSV file. The example below shows how to open a file, write a line of data to it, and then close the file:

```
string symbol;
double openPrice, sl, tp;
datetime openTime;

int fileHandle = FileOpen("Trades.csv",FILE_READ|FILE_WRITE|FILE_CSV,",");
FileSeek(fileHandle,0,SEEK_END);
FileWrite(fileHandle,symbol,openPrice,sl,tp,openTime);
FileClose(fileHandle);
```

The `symbol`, `openPrice`, `sl`, `tp` and `openTime` variables will contain the information to write to the file. We'll assume that these variables are filled with the appropriate values. The `FileOpen()` function creates a file named `"Trades.csv"`. The flags specify read and write privileges to a CSV file. The delimiter will be a comma. The file handle is saved to the `fileHandle` variable.

You will need to add the `FILE_READ` flag when using the `FILE_WRITE` flag, even if you are not reading information from the file, because the `FileSeek()` function will not work without it. The `FileSeek()` function moves the file pointer to a specified point in the file. In this example, the pointer is moved to the end of the file. The first parameter of the `FileSeek()` function is the file handle, the second is the shift in bytes, and the third parameter is the start location. The `SEEK_END` constant indicates that we will move the file pointer to the end of the file. When opening a file that already has data in it, failing to move the file pointer to the end of the file will result in data being overwritten. Therefore, we always use `FileSeek()` to locate the end of the file before writing data.

The `FileWrite()` function writes a line of data to the CSV file. The first parameter is the file handle. The remaining parameters are the data to write to the file, in the order that they appear. Up to 63 additional parameters can be specified, and they can be of any type. The delimiter specified in the `FileOpen()` function will be placed between each data field in the CSV file, and a new line character (`\r\n`) will be written at the end of the line.

Finally, the `FileClose()` function will close the file. Be sure to close a file when you are done using it, or else you may not be able to open it in another program. If you plan on keeping a file open for an extended period of time, or are doing subsequent read/write operations, use the `FileFlush()` function to write the data to the file without closing it.

Here is what our `Trades.csv` file will look with several lines of data written to it:

```
EURUSD,1.2345,1.2325,1.2375,2012.11.15 04:17:41
EURUSD,1.2357,1.2337,1.2397,2012.11.15 04:20:04
EURUSD,1.2412,1.2398,1.2432,2012.11.15 04:21:35
```

From left to right, each line contains the trade symbol, opening price, stop loss, take profit and open time, each separated by a comma. After each line is written to the file, a new line is started.

If you need to write several lines of data to a file at once, place the `FileWrite()` function inside a loop, and loop it as many times as you need to. The example below assumes that we have several arrays, each with the same number of elements, that are properly sized and filled with data. (You could also use a structure array for this.) We use a `for` loop to write each line of data to the file:

```
string symbol[];
double openPrice[], sl[], tp[];
datetime openTime[];

int fileHandle = FileOpen("Trades.csv",FILE_READ|FILE_WRITE|FILE_CSV,",");
FileSeek(fileHandle,0,SEEK_END);

for(int i = 0; i < ArraySize(symbol); i++)
{
   FileWrite(fileHandle,symbol[i],openPrice[i],sl[i],tp[i],openTime[i]);
}

FileClose(fileHandle);
```

We use the `ArraySize()` function on the `symbol[]` array to determine the number of times to run the loop. The `i` increment variable is the array index. If each array has five elements, for example, we write five lines of data to the file.

Reading From a CSV File

Next, we'll examine how to read data from a CSV file. The `FileRead...()` functions are used to read data from a field and convert it to an appropriate type. There are four functions used to read data from a CSV file:

- **`FileReadString()`** - Reads a `string` from a CSV file.
- **`FileReadBool()`** - Reads a string from a CSV file and converts it to `bool` type.
- **`FileReadDatetime()`** - Reads a string from a CSV file in the format *yyyy.mm.dd hh:mm:ss* and converts it to `datetime` type.
- **`FileReadNumber()`** - Reads a string from a CSV file and converts it to `double` type.

If you are reading an integer from a CSV file using the `FileReadNumber()` function, you will need to convert it to the appropriate type if you need to use it as an integer type in your program.

If we examine the fields in our `Trades.csv` file, we have a `string`, three `double` values and a `datetime` value on each line. We will need to read each field of data from the file using the appropriate function so that it is converted to the correct type. We're going to read the entire contents of the file, line by line, and save the result to a structure array:

```
struct Trades
{
   string symbol;
   double openPrice;
   double sl;
   double tp;
   datetime openTime;
};

Trades trade[];
int i;

int fileHandle = FileOpen("Trades.csv",FILE_READ|FILE_CSV,",");

while(FileIsEnding(fileHandle) == false)
{
   ArrayResize(trade,ArraySize(trade) + 1);

   trade[i].symbol = FileReadString(fileHandle);
   trade[i].openPrice = FileReadNumber(fileHandle);
   trade[i].sl = FileReadNumber(fileHandle);
   trade[i].tp = FileReadNumber(fileHandle);
   trade[i].openTime = FileReadDatetime(fileHandle);

   i++;
}

FileClose(fileHandle);
```

First, we create a structure named `Trades` to hold the data read from the CSV file. We create an array object named `trade[]`, and initialize the incrementor variable `i`. We open the `Trades.csv` file using the `FILE_READ` and `FILE_CSV` flags. The `while` loop will read each line of data from the file, one field at a time.

The `FileIsEnding()` function returns a value of `true` if the end of the file has been reached, and `false` otherwise. As long as the end of the file has not been reached, we continue reading the next line. The

`ArrayResize()` function resizes our `trade[]` array, one element at a time. We call the `ArraySize()` function to get the current size of the `trade[]` array and add 1 to it to increase the size.

The `FileRead...()` functions reads each field of data from the file and converts it to the appropriate type. The result is saved to the appropriate member variable of our `trades[]` array. After the current line has been read, we increment the `i` variable and check the `FileIsEnding()` condition again. After the loop exits, we close the file. We can now access the data read from our CSV file using the `trade[]` array object.

Global Variables

MetaTrader has the ability to save variables to the terminal, which remain even if the terminal is shut down. These are referred to as *Global Variables*. Global variables that are saved to the terminal are deleted after one month. You can view the global variables saved to your terminal by clicking the *Tools* menu > *Global Variables,* or by pressing the F3 key.

Do not confuse the global variables of the terminal with variables that we define on the global scope of a program! Global variables in a program are available only to that program, while the global variables of the terminal are available to all programs. You can use MetaTrader's Global Variables to save information about your program's state in the event that execution is interrupted.

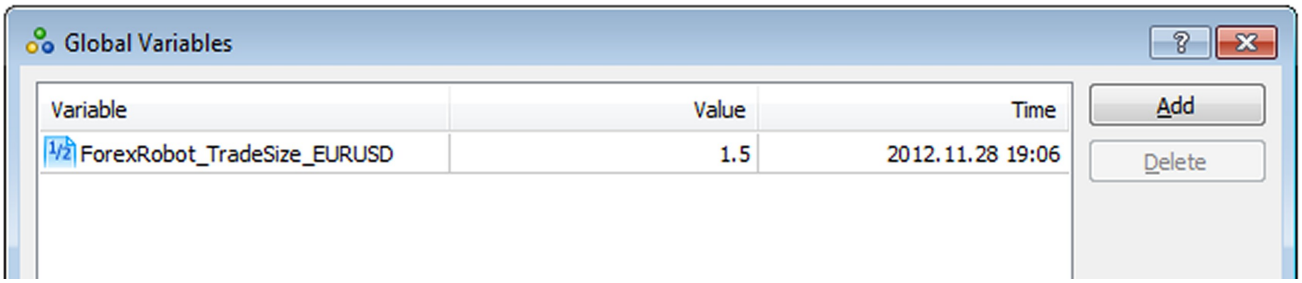


Fig. 24.8 – The *Global Variables* window.

The `GlobalVariableSet()` function is used to save a global variable to the terminal. It has two parameters – the `name` of the variable, and the `value` to assign to it. Make sure that you use unique names for your global variables. For example, you could use the name of your trading system, followed by the name of the variable and the symbol that it is placed on:

```
string varName = "ForexRobot_TradeSize_"+_Symbol;
// Example: ForexRobot_TradeSize_EURUSD

GlobalVariableSet(varName,1.5);
```

In this example, our global variable name is `ForexRobot_TradeSize_EURUSD`. The current symbol is `EURUSD`, so we will be able to identify our global variable based on the symbol we are currently trading. We would save this global variable to the terminal any time its value is set or changed.

If our terminal shut down unexpectedly (a computer crash or power failure), we can read the contents of the global variable using `GlobalVariableGet()`, and continue where we left off. We would usually do this in our `OnInit()` event handler:

```
// OnInit() event handler
string varName = "ForexRobot_TradeSize_"+_Symbol;
double tradeSize = GlobalVariableGet(varName);
```

To prevent our program from using outdated global variables, we will need to delete them if necessary. If we manually remove our expert advisor from the chart, then we need to delete the global variable(s) that are currently saved. We do this in the `OnDeinit()` event handler using the `GlobalVariableDel()` function:

```
void OnDeinit(const int reason)
{
   string varName = "ForexRobot_TradeSize_"+_Symbol;
   GlobalVariableDel(varName);
}
```

The `OnDeinit()` event handler is called for many reasons. Obviously, it is called if the program is removed from the chart, the chart is closed, or the terminal is shut down. But it is also called if the input parameters are changed, the period of the chart is changed, or a template is applied. We don't want to delete any global variables when this occurs. So it is necessary to check the reason for deinitialization before deleting any global variables.

The `reason` parameter of the `OnDeinit()` function contains the reason for deinitialization. You can view the deinitialization codes in the *MQL4 Reference* under *Standard Constants... > Named Constants > Uninitalization Reason Codes*. The codes we are concerned with are `REASON_CHARTCHANGE`, `REASON_PARAMETERS`, and `REASON_TEMPLATE`. If the `reason` parameter contains any of these codes, we will not delete the global variable:

```
void OnDeinit(const int reason)
{
   if(reason != REASON_CHARTCHANGE && reason != REASON_PARAMETERS
      && reason != REASON_TEMPLATE)
   {
      string varName = "ForexRobot_TradeSize_"+_Symbol;
      GlobalVariableDel(varName);
   }
}
```

Stopping Execution

If you wish to stop the execution of an expert advisor programmatically, use the `ExpertRemove()` function. Once the current event is finished executing, the expert advisor will stop its operation and remove itself from the chart.

If you wish to close the terminal, the `TerminalClose()` function will close MetaTrader. The `TerminalClose()` function takes one parameter, a deinitialization code that will be passed to the `OnDeinit()` function. When calling the `TerminalClose()` function, it must be followed by the return operator:

```
TerminalClose(REASON_CLOSE);
return;
```

Chapter 25 - Indicators, Scripts & Libraries

Indicators

In Chapter 19, we examined how to add indicators to our expert advisor programs. MQL4 allows you to create your own custom indicators as well. In this section, we will create a custom indicator that will plot a price channel using the highest high and lowest low of the last *x* bars. This is referred to as a *Donchian channel*.

The OnCalculate() Event Handler

All indicators require the `OnCalculate()` event handler. It is the equivalent of the `OnTick()` event handler for expert advisors, and runs on every incoming tick. Here is the function declaration for `OnCalculate()`:

```
int OnCalculate (const int rates_total,      // size of input time series
                 const int prev_calculated,  // bars handled in previous call
                 const datetime& time[],     // Time
                 const double& open[],       // Open
                 const double& high[],       // High
                 const double& low[],        // Low
                 const double& close[],      // Close
                 const long& tick_volume[],  // Tick Volume
                 const long& volume[],       // Real Volume
                 const int& spread[]         // Spread
   );
```

The `rates_total` parameter contains the total number of bars on the chart, while the `prev_calculated` parameter contains the number of bars that has been previously calculated by the indicator. These are used to determine how many bars to calculate on each call of the function. We'll address this later in the chapter.

The remaining parameters are arrays that contain time, price, spread and volume data for the current chart symbol. We will use the data in these arrays for our indicator calculations. Note that we will need to use the `ArraySetAsSeries()` function to set these arrays as series arrays before using them in our indicator!

MQL4 Wizard

The *MQL4 Wizard* can be used to create a starting template for your custom indicator. It is much more convenient to use the wizard to add our event handlers, buffers and lines than it is to add them manually. Open the *MQL4 Wizard* by clicking the *New* button on the MetaEditor toolbar. Select *Custom Indicator* and click *Next* to continue.

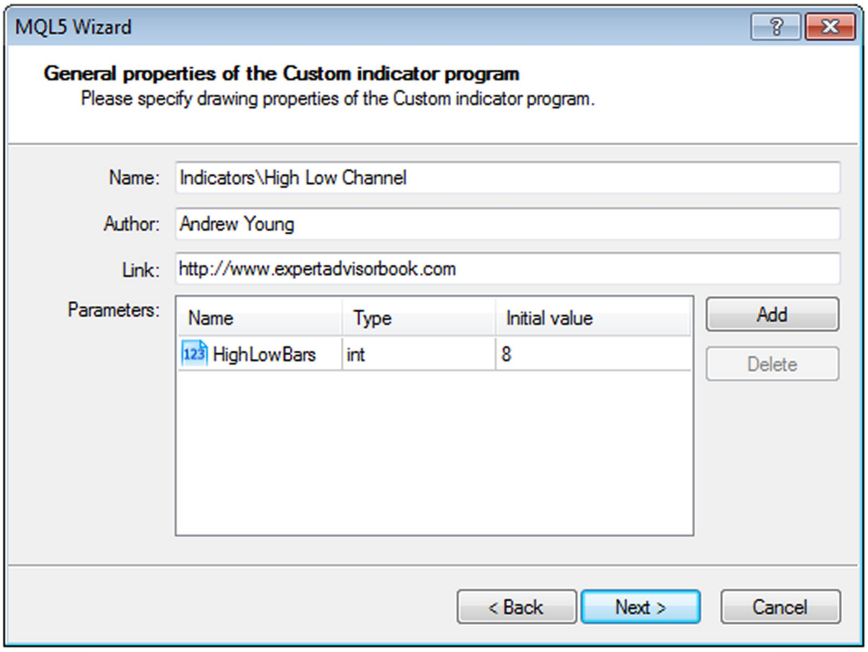


Fig. 25.1 – The custom indicator properties dialog of the *MQL4 Wizard*.

The custom indicators are saved to the `\MQL4\Indicators` folder by default. You can add input variables on this screen if you wish. In Fig. 25.1, we have added an `int` input variable named `HighLowBars`, with a default value of 8.

The next screen allows you to select the event handlers to add to your indicator. The `OnCalculate()` event handler is selected by default. You can add additional event handlers if necessary. The final screen is where you set the drawing properties of the indicator:

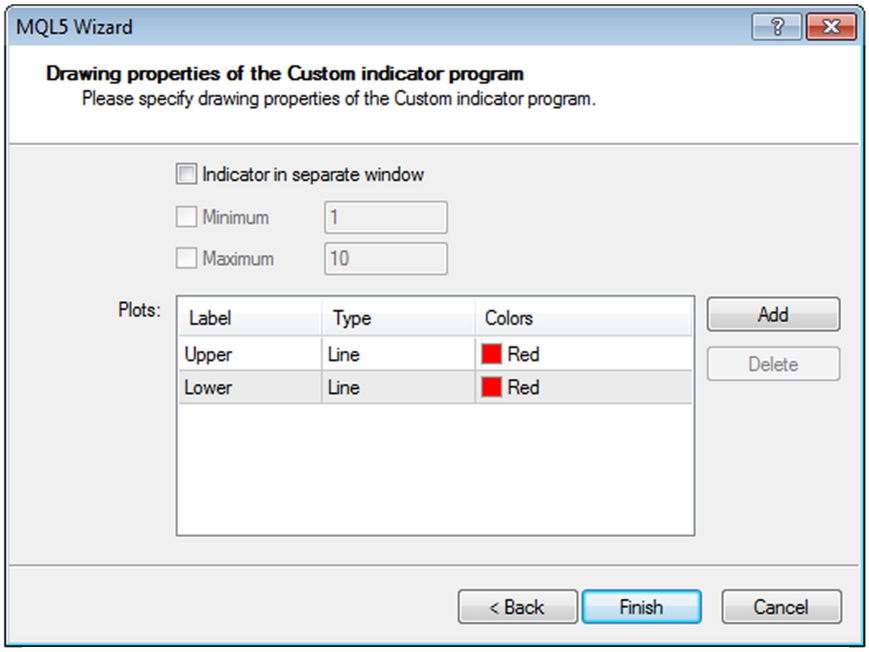


Fig. 25.2 – The drawing properties dialog of the *MQL4 Wizard* for custom indicators.

If this indicator will be displayed in a separate window (such as an oscillator), check *Indicator in separate window* to add the relevant `#property` directive to the indicator file. The *Plots* grid allows you to add and set the properties for the indicator lines.

The *Label* column is the name of the indicator line as it will appear in the *Data Window*. It is also used to create the array buffer name. We have added two indicator lines named `Upper` and `Lower`. Double-click on the *Type* column to reveal a drop-down box to select the line's drawing type. We have left our lines set to the *Line* drawing type. Finally, the *Color* column is where you set the color of the indicator line. We have set both lines to Red.

Click *Finish* to close the Wizard and open the indicator template in MetaEditor.

Indicator Properties

The *MQL4 Wizard* uses `#property` directives to configure the indicator lines. Let's examine the properties in our indicator file:

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

//--- plot HighestHigh
#property indicator_label1  "Upper"
```

```
#property indicator_type1   DRAW_LINE
#property indicator_color1  clrLimeGreen
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1

//--- plot LowestLow
#property indicator_label2  "Lower"
#property indicator_type2   DRAW_LINE
#property indicator_color2  clrCrimson
#property indicator_style2  STYLE_SOLID
#property indicator_width2  1
```

The `indicator_chart_window` property draws the indicator in the main chart window. If we wanted to draw our indicator in a subwindow on the chart, we'd use the `indicator_separate_window` property instead. The `indicator_buffers` and `indicator_plots` properties are the number of buffer arrays and lines that this indicator requires.

For each line in the indicator, we have several `#property` directives to set the line properties. The `N` at the end of each property name refers to the line number:

- `indicator_labelN` - Sets the name of the line that appears in the *Data Window*.
- `indicator_typeN` – The plotting style. Here are the most common plotting styles:
  - `DRAW_LINE` – This is the most common plotting style, and consists of a single line of the specified color. The Moving Average, RSI and many other indicators use this drawing style.
  - `DRAW_HISTOGRAM` – The histogram drawing style is typically used by oscillators such as the MACD, the OsMA and the Bill Williams oscillators. It consists of vertical lines oscillating above and below a zero axis.
  - `DRAW_ARROW` – The arrow plotting style will draw arrow objects on the chart. The Fractals indicator uses arrow objects to indicate swing highs and lows.
  - `DRAW_NONE` – Used for indicator buffers that will not be drawn on the chart.
- `indicator_colorN` – The color of the indicator line.
- `indicator_styleN` – The drawing style. Typically, this is `DRAW_SOLID` for a solid line, but you can specify dashed or dotted lines using `STYLE_DASH`, `STYLE_DOT` or one of the other drawing styles.
- `indicator_widthN` – The width of the indicator line.

You can view all of the indicator `#property` directives in the *MQL4 Reference* under *Language Basics* > *Preprocessor* > *Program Properties*. The indicator plotting and drawing styles can be viewed under *Standard Constants...* > *Indicator Constants* > *Drawing Styles*.

There are other ways to set the indicator properties as well. The `SetIndexStyle()` function can be used to set the drawing properties for each line, and the `IndicatorSet...()` functions are used to set various indicator properties. You can view all of the custom indicator functions in the *MQL4 Reference* under *Custom Indicators*.

Calculating the Indicator

Let's examine the rest of our indicator file. The `HighLowBars` input variable that we added in the *MQL4 Wizard* has been inserted. The wizard also adds two arrays for our indicator buffers, `UpperBuffer[]` and `LowerBuffer[]`. Inside the `OnInit()` event handler, the `SetIndexBuffer()` functions assign the arrays to the appropriate line indexes:

```
//--- input parameters
input int      HighLowBars=8;

//--- indicator buffers
double         UpperBuffer[];
double         LowerBuffer[];

int OnInit()
{
//--- indicator buffers mapping
   SetIndexBuffer(0,UpperBuffer);
   SetIndexBuffer(1,LowerBuffer);

   return(INIT_SUCCEEDED);
}
```

The first parameter of the `SetIndexBuffer()` function is the line index. Line indexes are zero-based, so the index for the first indicator line is 0, the second indicator line is 1, and so on. The second parameter is the name of the buffer array to use. In our indicator above, the `UpperBuffer` array is assigned to line 0, and `LowerBuffer` is asigned to line 1.

Our indicator calculations are carried out in the `OnCalculate()` event handler. The complete `OnCalculate()` event handler for our indicator is shown below:

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
```

```
                  const long &volume[],
                  const int &spread[])
{

   ArraySetAsSeries(UpperBuffer,true);
   ArraySetAsSeries(LowerBuffer,true);
   ArraySetAsSeries(high,true);
   ArraySetAsSeries(low,true);

   int bars = rates_total - 1;
   if(prev_calculated > 0) bars = rates_total - prev_calculated;

   for(int i = bars; i >= 0; i--)
   {
      UpperBuffer[i] = high[ArrayMaximum(high,i,HighLowBars)];
      LowerBuffer[i] = low[ArrayMinimum(low,i,HighLowBars)];
   }

//--- return value of prev_calculated for next call
   return(rates_total);
}
```

The buffer arrays, as well as the price arrays passed in by the `OnCalculate()` event handler, are not set as series arrays by default. We will need to set them as series arrays using the `ArraySetAsSeries()` function. Both of our indicator buffer arrays, as well as the `high[]` and `low[]` arrays passed in by the `OnCalculate()` function will be set as series:

```
ArraySetAsSeries(UpperBuffer,true);
ArraySetAsSeries(LowerBuffer,true);
ArraySetAsSeries(high,true);
ArraySetAsSeries(low,true);
```

Many indicators use a `for` loop to calculate the indicator value for each bar on the chart. We determine the number of bars to process by using the `prev_calculated` and `rates_total` parameters of the `OnCalculate()` event handler. As mentioned earlier, the `rates_total` variable contains the total number of bars on the chart, while `prev_calculated` contains the number of bars calculated by the previous run of the `OnCalculate()` event handler.

For series arrays, the maximum array index is `rates_total - 1`. This refers to the oldest bar on the chart. The most recent bar has an index of zero. When `OnCalculate()` is first run, the value of `prev_calculated` will be zero. If `prev_calculated` is zero, we set the maximum array index to `rates_total - 1`. On subsequent runs, we calculate the maximum array index by subtracting `prev_calculated` from `rates_total`. This ensures that only the most recent bar(s) will be calculated:

```
int bars = rates_total - 1;
if(prev_calculated > 0) bars = rates_total - prev_calculated;
```

The `bars` variable will hold the maximum array index. In our `for` loop below, we assign the value of `bars` to our index variable `i`. We will decrement the value of `i` until `i = 0`:

```
for(int i = bars; i >= 0; i--)
{
   // Indicator calculations
}
```

As long as your price and buffer arrays are set as series, the `for` loop above will work for calculating most indicators. The code to calculate the indicator and fill the buffer arrays goes inside the loop.

To calculate the buffer arrays for our channel indicator, we simply use the `ArrayMaximum()` and `ArrayMinimum()` functions to find the index of the bars that contain the highest high and lowest low prices, relative to the bar that we are currently calculating. The number of bars to search is specified by the `HighLowBars` input variable. The resulting array index is used in the `high[]` and `low[]` arrays to return the highest high and lowest low prices, which are saved to the current index of the `UpperBuffer[]` and `LowerBuffer[]` arrays respectively:

```
for(int i = bars; i >= 0; i--)
{
   int highShift = ArrayMaximum(high,HighLowBars,i);
   int lowShift = ArrayMinimum(low,HighLowBars,i);

   double highestHigh = high[highShift];
   double lowestLow = low[lowShift];

   UpperBuffer[i] = highestHigh;
   LowerBuffer[i] = lowestLow;
}
```

The last step is to return the value of `rates_total` and exit the `OnCalculate()` event handler. This is inserted automatically by the MQL4 Wizard:

```
//--- return value of prev_calculated for next call
   return(rates_total);
}
```

We've just created a simple indicator that takes price data from the `OnCalculate()` event handler and calculates two indicator lines. This indicator can be used to create trading signals, or as a stop loss for trending positions. We've only touched upon you can do with custom indicators in MQL4. To learn more about custom indicator functions, consult the *MQL4 Reference* under *Custom Indicators*.

You can view the source code for this file in `\MQL4\Indicators\High Low Channel.mq4`.

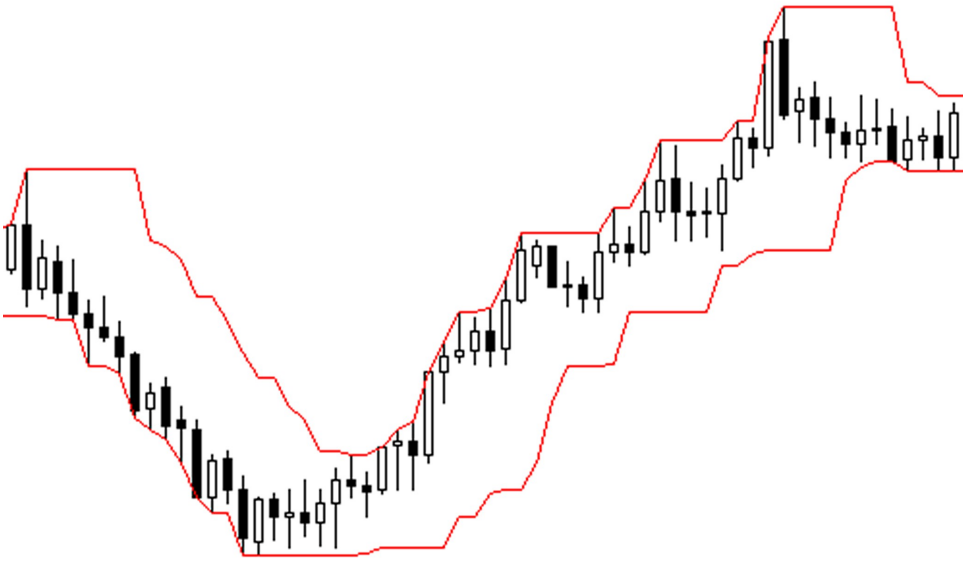


Fig. 25.3 – The High Low Channel custom indicator.

Scripts

A *script* is a simple MQL4 program that executes once when it is attached to a chart. It consists of a single event handler: `OnStart()`. When a script is attached to a chart, the `OnStart()` event handler executes. Unlike an expert advisor or indicator, a script does not repeat its execution after the `OnStart()` event handler has finished. The script is automatically detached from a chart after execution.

To create a script, use the *MQL4 Wizard*. All scripts are saved to the `\MQL4\Scripts` directory. Your new script file will contain an empty `OnStart()` event handler. There are a couple of `#property` directives that control the behavior of your script. If your script has input variables, use the `script_show_inputs` property to show the *Inputs* tab before script execution. If your script does not have input variables, the `script_show_confirm` property displays a confirmation box asking to user whether to execute the script. You'll want to add one of these `#property` directives to your script.

We're going to create a small, but useful script that can be used to close all open orders in the terminal. When attached to a chart, the script will first prompt the user to execute the script. If so, the script will close all open orders on the account:

```
#property script_show_confirm

#include <Mql4Book\Trade.mqh>
CTrade Trade;

void OnStart()
{
   for(int i = 0; i <= OrdersTotal() - 1; i++)
   {
      // Select order
      bool result = OrderSelect(i,SELECT_BY_POS);

      if(result == true)
      {
         bool closed = false;

         if(OrderType() == OP_BUY || OrderType() == OP_SELL)
         {
            closed = Trade.CloseMarketOrder(OrderTicket(),0,clrNONE);
         }
         else
         {
            closed = Trade.DeletePendingOrder(OrderTicket(),clrNONE);
         }

         if(closed == true) i--;
      }
   }
}
```

The `script_show_confirm` property prompts the user with a confirmation dialog before executing the script. We've included our `Trade.mqh` file from the `\MQL4\Include\Mql4Book` folder and created an object based on the `CTrade` class.

When the script is executed, the `OnStart()` event handler runs. We loop through all open orders on the chart from oldest to newest. We select each order using the `OrderSelect()` function, and call the appropriate function to close the order, depending on the order type. If the order closed successfully, we decrement the `i` variable and go to the next open order.

The above script will close all orders on the account. We can modify the program to close orders for a specified symbol or order type. We'll add an input variable named `CloseSymbol`. If a value is specified for `CloseSymbol`, the script will only close orders on the specified symbol. We'll also add the `CloseType` input variable. The type will be an enumeration that allows the user to select the types of orders to close:

```
#property script_show_inputs

#include <Mql4Book\Trade.mqh>
CTrade Trade;

enum CloseTypes
{
   All,        // All Orders
   Market,     // Market Orders
   Pending,    // Pending Orders
   Buy,        // Buy Market
   Sell,       // Sell Market
   BuyStop,    // Buy Stop
   SellStop,   // Sell Stop
   BuyLimit,   // Buy Limit
   SellLimit,  // Sell Limit
};

input CloseTypes CloseType = All;
input string CloseSymbol = "";
```

We use the `script_show_inputs` property directive to show the input window to the user before script execution. The `CloseTypes` enumeration contains the different types of orders to close. You can select whether to close all market orders, all pending orders, all orders regardless of type, or you can choose a specific order type. The comments after each member of the enumeration will be displayed in the *Inputs* tab.

```
void OnStart()
{
   for(int i = 0; i <= OrdersTotal() - 1; i++)
   {
      // Select order
      bool result = OrderSelect(i,SELECT_BY_POS);

      if(result == true)
      {
         string orderSymbol = OrderSymbol();
         string closeSymbol = StringTrimRight(CloseSymbol);

         if(closeSymbol != "" && closeSymbol != orderSymbol) continue;

         int orderType = OrderType();
         int orderTicket = OrderTicket();
```

We retrieve the order symbol and store it in the `orderSymbol` variable, and trim any excess whitespace from the end of the `CloseSymbol` input variable, and store the result in `closeSymbol`. If `CloseSymbol` has a value specified, and that value is not equal to the symbol of the current trade, we skip to the next order.

```
bool closeOrder = false;

switch(CloseType)
{
   case All:
      closeOrder = true;
      break;

   case Market:
      if(orderType == OP_BUY || orderType == OP_SELL) closeOrder = true;
      break;

   case Pending:
      if(orderType >= 2) closeOrder = true;
      break;

   case Buy:
      if(orderType == OP_BUY) closeOrder = true;
      break;

   case Sell:
      if(orderType == OP_SELL) closeOrder = true;
      break;

   case BuyStop:
      if(orderType == OP_BUYSTOP) closeOrder = true;
      break;

   case SellStop:
      if(orderType == OP_SELLSTOP) closeOrder = true;
      break;

   case BuyLimit:
      if(orderType == OP_BUYLIMIT) closeOrder = true;
      break;

   case SellLimit:
      if(orderType == OP_SELLLIMIT) closeOrder = true;
      break;
}
```

This lengthy `switch` block determines whether to close the current order based on it's order type and the value of the `CloseType` setting. If the value of `CloseType` matches the appropriate order type constant, the `closeOrder` boolean variable will be set to `true`.

```
            if(closeOrder == true)
            {
               bool closed = false;

               if(orderType == OP_BUY || orderType == OP_SELL)
               {
                  closed = Trade.CloseMarketOrder(orderTicket,0,clrNONE);
               }
               else
               {
                  closed = Trade.DeletePendingOrder(orderTicket,clrNONE);
               }

               if(closed == true) i--;
            }
         }
      }
   }
```

If the `closeOrder` variable is set to `true`, we close the order using the appropriate function from the `CTrade` class. If the order is closed successfully, the `i` index variable is decremented.

This script can be used to close orders on the chart in a testing scenario, in manual live trading, or in an emergency situation. You can view the source code of the `Close Orders.mq4` script in the `\MQL4\Scripts\Mql4Book` folder.

Libraries

A *library* is an executable file that contains reusable functions for use by other programs. It is similar to an include file, but with several important differences. Unlike an include file, a library does not have classes or variables that can be used by other programs. You can define classes, structures, enumerations and the like in your library, but they will not be usable outside of the library.

You can use native Windows DLLs or other DLLs created in C++ in your MQL4 programs. The process of importing DLLs functions is similar to importing functions from an MQL4 library. Functions contained within libraries have limitations as to the types of parameters that can be passed to them. Pointers and objects that contain dynamic arrays cannot be passed to a library function. If you are importing functions from a DLL, you cannot pass string or dynamic arrays to those functions.

The advantage of a library is that you can distribute it without making the source code available. If you use a library in numerous expert advisors, you can make minor changes to the library without having to recompile every program that depends on it (as long as you don't change the function parameters, that is).

You can create a blank library file using the *MQL4 Wizard*. Libraries are saved in the `\MQL4\Libraries` folder. A library must have the `library` property directive at the top of the file. If it is not present, the file will not compile. Let's create a sample library with two exportable functions. The functions that we wish to export will have the `export` modifier after the function parameters:

```
#property library

#include <Mql4Book\Indicators.mqh>
CiRSI RSI;

bool BuySignal(string pSymbol, ENUM_TIMEFRAMES pTimeframe, int pPeriod,
   ENUM_APPLIED_PRICE pPrice) export
{
   RSI(pSymbol,pTimeframe,pPeriod,pPrice);

   if(RSI.Main() < 30) return(true);
   else return(false);
}

bool SellSignal(string pSymbol, ENUM_TIMEFRAMES pTimeframe, int pPeriod,
   ENUM_APPLIED_PRICE pPrice) export
{
   RSI(pSymbol,pTimeframe,pPeriod,pPrice);

   if(RSI.Main() > 70) return(true);
   else return(false);
}
```

This file is named `SignalLibrary.mq4`, and is located in the `\MQL4\Libraries\Mql4Book` folder. The `#property library` directive indicates that this is a library file. This library contains two functions that will be used to return trade signals to the calling program. These functions use simple RSI overbought and oversold trade signals, but you could create a library with more elaborate trade signals that you can keep hidden from the expert advisors and programmers that use them.

We include the `\MQL4\Include\Mql4Book\Indicators.mqh` file and declare an object based on the `CiRSI` class. Note that this object is not visible outside of the library. The `BuySignal()` and `SellSignal()` functions take parameters that set the parameters for the RSI indicator. Note the `export` modifier after the closing parenthesis. Any function that will be imported into another program must have the `export` modifier!

This file will be compiled just like any other MQL4 program. To use our library functions in another program, we need to import them into that program. We do this using the `#import` directive. Here is how we would import these functions into an expert advisor program:

```
#import "SignalLibrary.ex4"
   bool BuySignal(string pSymbol, ENUM_TIMEFRAMES pTimeframe, int pPeriod,
      ENUM_APPLIED_PRICE pPrice);
   bool SellSignal(string pSymbol, ENUM_TIMEFRAMES pTimeframe, int pPeriod,
      ENUM_APPLIED_PRICE pPrice);
#import
```

The library name is contained in double quotes after the opening `#import` directive. Note the `.ex4` in the library name indicating that this is a compiled executable file. You cannot import a source code file. All libraries must be located in the `\MQL4\Libraries` folder. Following the opening `#import` directive, the functions that we are importing from our library are defined. A closing `#import` directive must be present after the last imported function.

Our imported functions are used just like any other functions. Here's an example of how we could use the `BuySignal()` function in an expert advisor:

```
if(BuySignal(_Symbol,_Period,14,PRICE_CLOSE) == true)
{
   // Open buy position
}
```

The example above calls the `BuySignal()` function, and calculates the RSI value for a 14 period RSI using the close price for the current chart symbol and period. If the RSI is currently oversold, the function returns `true`.

If an imported function from a library has the same name as a function in your program or a predefined MQL4 function, the scope resolution operator (`::`) must be used to identify the correct function. For example, let's assume that our program already has a function named `BuySignal()`. If we import the `BuySignal()` function from our library file, we'll need to preface it with the library name when we call it:

```
SignalLibrary::BuySignal(_Symbol,_Period,10,PRICE_CLOSE);
```

Chapter 26 - Debugging and Testing

Every programmer makes mistakes, and almost every program contains errors. Whether they are compilation errors caused by mistyping a function, or an error in your program logic, you will need to learn how to test and debug your programs. This chapter will discuss errors and procedures for debugging and testing your programs. You'll also learn how to use the Strategy Tester to evaluate your program's performance.

Errors

There are three types of errors related to MQL4 programs: *Compilation errors* occur in MetaEditor when invalid source code is compiled. *Runtime errors* are logic or software errors that occur when a program is executed in MetaTrader. *Trade server errors* occur when a trade request is unsuccessful.

Compilation Errors

It is common to mistype a function call, omit a semicolon or closing bracket, or make a syntax error when coding. When you compile your program, a list of errors will appear in the *Errors* tab in MetaEditor. The first time this happens, it may appear daunting. But don't worry – we're going to address some of the most common syntax errors that occur.

The first thing to remember when confronted with a list of compilation errors is to always start with the first error in the list. More often than not, it is a single syntax error that results in a whole list of errors. Correct the first error, and the remaining errors will disappear.

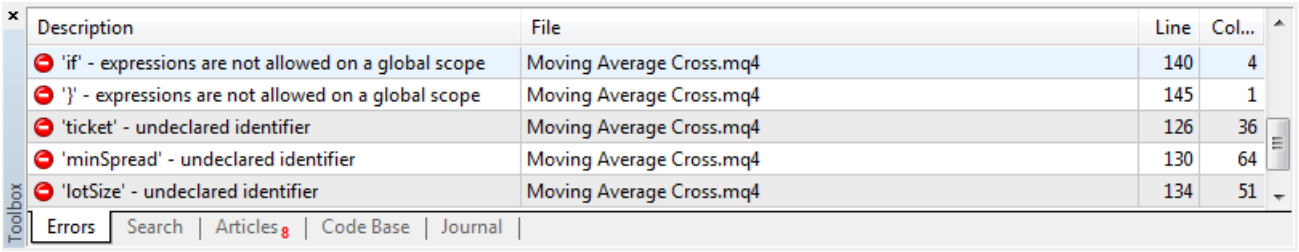


Fig. 26.1 – The *Errors* tab under the *Toolbox* window in MetaEditor. Note that all of these errors are due to a single missing left bracket.

Double-clicking the error in the *Errors* tab will take you to the spot in your program where the error was triggered. More than likely, the error will be right under your cursor, or on the previous line. A missing semicolon, parentheses or bracket may result in misleading compilation errors, so be sure to check the previous lines(s) for these when faced with an error on a line that otherwise looks correct.

Here's a list of the most common syntax errors in MetaEditor, and the most common reasons for these errors:

- **Semicolon expected** – A semicolon is missing in the previous line.
- **Unexpected token** – A semicolon is missing in the previous line, or an invalid character is present in the current line.
- **Unbalanced left/right parenthesis** – An extra parenthesis is present or a parenthesis is missing in the current line.
- **Expressions are not allowed on a global scope** – A left bracket in a compound operator may be missing.
- **Some operator expected** – An operator is missing in the specified location.
- **Wrong parameters count** – Too many or not enough parameters in a function call.
- **Undeclared identifier** – A variable or object name is used without being declared first. Check the spelling and case of the variable name, and declare it if necessary.
- **Unexpected end of program** – A closing bracket is missing in your program. This is a tricky one, since the error refers to the end of the program. Examine the code that you edited recently and look for a missing closing bracket.

Runtime Errors

A runtime error is an error that occurs during the execution of a program. Runtime errors are logic errors – in other words, the program will compile, but is not operating as expected. An error message will print to the log when a runtime error occurs. The error message will indicate the cause of the error, as well as the line on which it occurred in your source code.

You can use the `GetLastError()` function to retrieve the error code of the last runtime error, in case you want to add error handling for runtime errors to your program. After accessing the last error code, use the `ResetLastError()` function to reset the error code to zero.

Programs will continue to run after a runtime error occurs, but there are a few critical errors that will end execution of a program immediately. A *divide by zero* error is where a division operation uses a divisor of zero. A *array out of range* error occurs when the program attempts to access an array element that doesn't exist. This usually occurs by trying to access an array element larger than the size of the array. Attempting to access an invalid pointer will also cause a critical error.

A complete list of runtime errors can be found in the *MQL4 Reference* under *Standard Constants... > Codes of Errors and Warnings > Runtime Errors*.

Trade Server Errors

A trade server error occurs when a trade operation fails, either due to invalid trade parameters or a server issue. When a trade operation fails, the `OrderSend()` function will return false. To retrieve the error code, call the `GetLastError()` function. The `ErrorDescription()` function in MetaTrader's `stdlib.mqh` include file will return a description of the error code.

All of the trading functions that we created in this book check for trade server errors and log them when they occur. If you need to write a class or function that uses the `OrderSend()` function, be sure to add code that will handle trade server errors.

Debugging

New in MetaEditor is a debugger that can be used to execute your programs interactively. By clicking the *Start debugging* button on the MetaEditor toolbar, your program will be opened on a chart in MetaTrader and tested in real time using live data. You can stop or pause the debugging process by clicking the *Stop* or *Pause* buttons:

Fig. 26.2 – Debugging buttons. From left to right are the *Start*, *Pause* and *Stop* debugging buttons.

The debugging process is entered when a *breakpoint* is reached. A breakpoint can be defined in MetaEditor by pressing the F9 key to toggle a breakpoint on the current line. You can also use the `DebugBreak()` function to define a breakpoint. When a breakpoint is reached during program execution, the program pauses and control is turned over to the programmer. Using the *Step Into*, *Step Over* and *Step Out* buttons on the toolbar, the programmer can observe the program execution line by line.

```
// Open sell order
if(PositionType(_Symbol) == -1 && glSellPlaced == false
{
   glSellPlaced = Trade.Sell(_Symbol,tradeSize);

   if(glSellPlaced == true && PositionVerify() == true)
   {
      double openPrice = PositionOpenPrice(_Symbol);
```

Fig. 26.3 – A breakpoint is set in MetaEditor on the current line.

The *Step Into* button moves execution to the next line of the program. *Step Over* will skip over any functions that are encountered during the execution, and *Step Out* exits the current function and returns control to the function that called it (or exits the current event).

Fig. 26.4 – Step buttons. From left to right are the *Step Into*, *Step Over* and *Step Out* buttons.

You can monitor the value of a variable during debugging by using the *Watch* window inside the *Debug* tab in the MetaEditor *Toolbox* window. The *Watch* window displays the current value of watched variables and expressions. To add a variable or expression to the watch window, place your mouse cursor over the variable name, or click-and-drag to select the expression you wish to add to the watch window. Right-click and select *Add Watch* from the popup menu, or press *Shift+F9* on your keyboard.

Expression	Value	Type
positionType	0	long
sar[1]	1.27947	double

Fig. 26.5 – The *Watch* window.

Debugging in MetaEditor is done in real time on live data, so you may need to modify your program to produce the result you are looking for immediately, especially if you are debugging a trading signal or trade operation. If you need to debug a program quickly, try testing your program in the Strategy Tester.

Logging

Sometimes, errors may occur during live or demo trading when a debugger is not available. Or you may need to test multiple trading scenarios at once in the Strategy Tester. You can use the `Print()` function to log information about your program's state to MetaTrader's log files. The `Alert()` function automatically prints the alert string to the log, so any alerts will be displayed in the log as well.

We have already added `Print()` functions throughout our trading classes and functions. These will print the results of trade operations to the log, as well as the values of relevant variables that are in use when an error condition occurs. This example is from the `CTrade::OpenPendingOrder()` function in `Trade.mqh`:

```
Print("Symbol: ",pSymbol,", Volume: ",pVolume,", Price: ",pPrice,", SL: ",pStop,", TP: ",
   pProfit,", Expiration: ",pExpiration);
```

Here is an example of how the result may appear in the strategy tester log:

```
Symbol: EURUSD, Volume: 0.0, Price: 1.35087, SL: 1.3478, TP: 0.0, Expiration: 1970.01.01
```

In this case, we specified an invalid trade volume of zero. You should always build this kind of logging functionality into your programs, especially when performing trade operations or testing new code.

Debugging with `Print()` functions, while less interactive than using the debugger, allows the programmer to examine the output of many trades at once. You can view the Strategy Tester log under the *Journal* tab in the *Strategy Tester* window. Right-click in the *Journal* window and select *Open* from the pop-up menu to open the logs folder and view the files in a text editor.

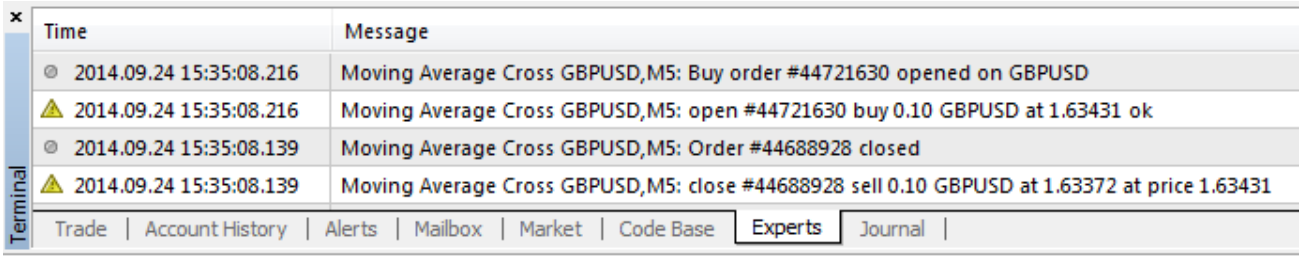


Fig. 26.6– The *Experts* tab in the toolbox window. All `Print()` and `Alert()` output is viewable here.

When trading with an expert advisor on a live chart, MetaTrader uses two different log folders for output. The terminal logs, located in the `\Logs` folder, displays basic trade and terminal information. You can view this log under the *Journal* tab in the *Terminal* window. The experts log, located in `\Experts\Logs`, contains the output of `Print()` and `Alert()` functions, as well as detailed trade information. You can view the experts log under the *Experts* tab in the *Terminal* window. You can open the log folders by right-clicking inside the *Journal* or *Experts* tab and selecting *Open* from the popup menu.

Using the Strategy Tester

The Strategy Tester is the most important tool you have to test and evaluate your trading systems. To access the Strategy Tester, open the *Strategy Tester* window in MetaTrader from the *Standard* toolbar, or press *Ctrl+R* on your keyboard.

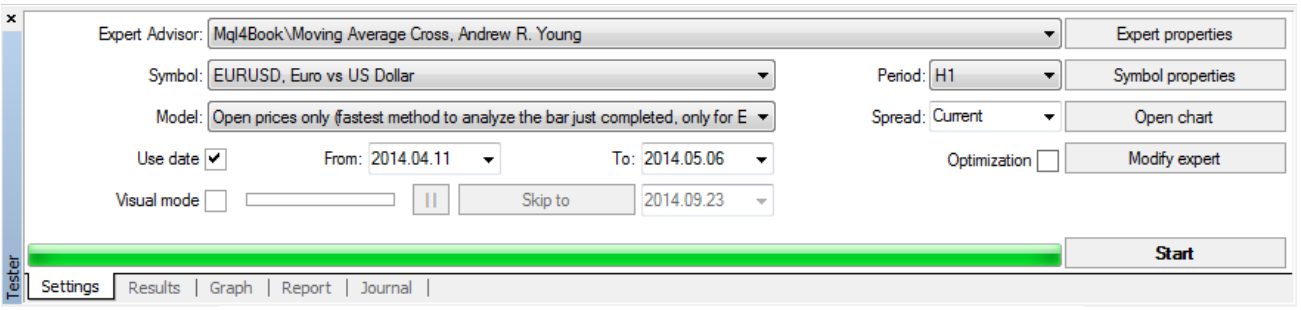


Fig. 26.7 – The *Settings* tab of the Strategy Tester.

The *Settings* tab in the *Strategy Tester* window is where you'll enter the settings for testing. Select the expert advisor to test from the *Expert Advisor* drop-down box, the symbol to test on from the *Symbol* drop-down, and the timeframe from the *Period* drop-down.

The *Model* drop-down determines the testing mode: *Every tick* attempts to model every incoming tick from the server. It is the slowest mode, but the most accurate. *Open prices only* uses only the OHLC of each bar of the selected chart period. This is the quickest mode, but the least accurate. This is useful if you wish to quickly test an expert advisor that only opens orders at the open of a new bar. *Control Points* is an intermediate method of modeling prices that is not recommended for use.

Check the *Use date* check box and select the start and end date for your test. Be sure that you have sufficient history in MetaTrader's *History Center* to carry out the test. Press F3 on your keyboard to open the *History Center*. You can download data from the trade server, or import data from another source.

The *Spread* drop-down box allows you to adjust the spread for the test. This is useful if you are testing on weekends or some other time when the spread is non-optimal. The default is to use the current spread.

If you check the *Visual Mode* checkbox, a chart window will appear when you start the testing. The chart will show the testing tick-by-tick. The speed of the visualization is adjustable, and you can pause it mid-test. The *Optimization* check box runs an optimization instead of a backtest. We'll discuss optimization shortly.

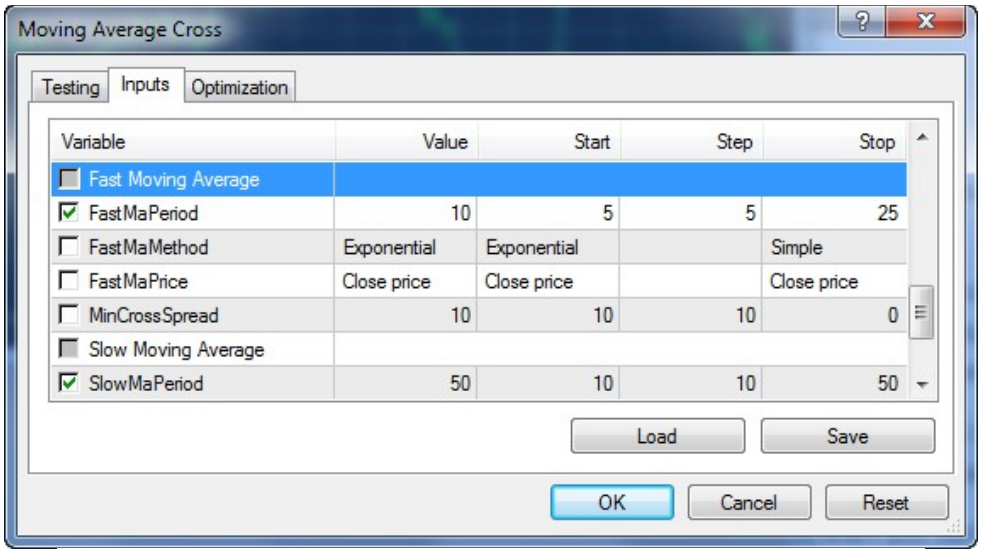


Fig 26.8 – The *Inputs* tab of the Expert Properties dialog.

Press the *Expert properties* button to adjust the expert advisor settings. The *Testing* tab is for adjusting test settings, such as the starting balance. The *Inputs* tab is used to adjust the input parameters of the expert advisor. You can see an example of the *Inputs* tab for an expert advisor in Fig. 26.8 above.

The *Value* column is used to set the values for the current test. The *Start*, *Step* and *Stop* columns are for optimization, and will be addressed shortly. You can save the current settings to a `.set` file, or load settings from an existing `.set` file by using the *Load* and *Save* buttons.

Once your test settings and input parameters are configured, press the *Start* button in the *Settings* tab. After the testing has completed, a chart will open and the results will appear under the *Report* tab. The *Report* tab displays a testing report showing the net profit, drawdown, profit factor and other statistical measures of performance. To view the trades, click on the *Results* tab. This will show where trades opened and closed, and at what prices. The *Graph* tab shows a graph of profitability over time. Finally, the *Journal* tab shows the testing log, which is located in the `\Tester\Logs` folder. The results of your testing, including all log entries and errors, will appear in this file.

Optimization

Running an *optimization* on an expert advisor involves selecting the input parameters to optimize and testing multiple combinations of parameter sets to determine which parameters are most profitable. This is usually followed by a *forward test* that tests the optimization results on out-of-sample data. Optimization and forward testing is the process by which you will evaluate the profitability of your expert advisors.

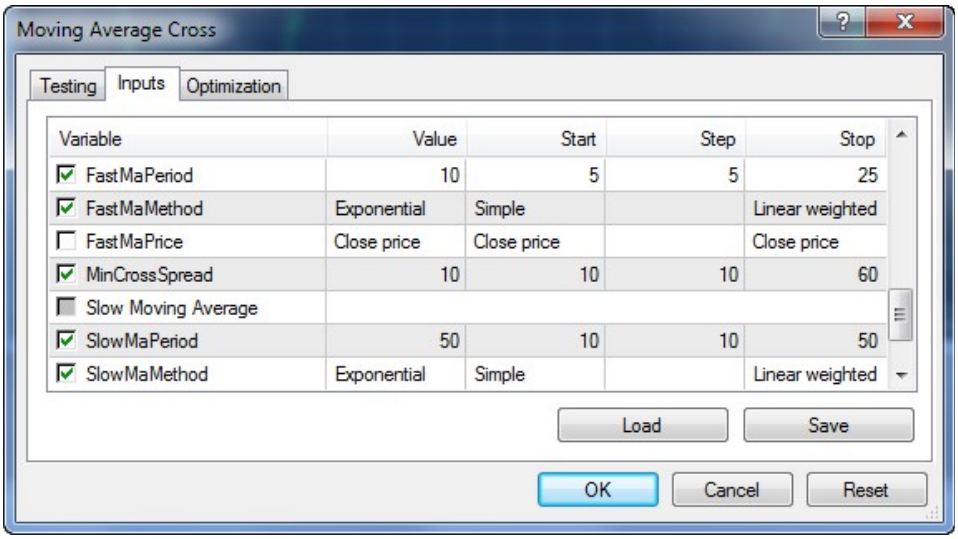


Fig 26.9 – The *Inputs* tab in the Strategy Tester, using the *Start*, *Step* and *Stop* columns for optimization.

Let's start under the *Inputs* tab. The *Start*, *Step* and *Stop* columns are used to set the optimization parameters. To optimize a parameter, select the checkbox to the left of the parameter name in the *Variable* column. The *Start* value is the starting value for the parameter. The *Step* value increments the parameter by the specified amount, and the *Stop* value is the ending value for the parameter. For example, if we have a *Start* value of 10, a *Step* value of 5, and a *Stop* value of 50, then the parameter will be optimized starting at 10, 15, 20... all the way up to 50 for a total of 10 steps.

You can optimize any numerical parameter, including enumeration values such as the moving average methods shown in Fig 22.13. You may want to limit the number of parameters to test, as well as the step value for each parameter. The more parameters/steps there are to test, the longer the optimization will take. Excessive optimization also leads to "curve-fitting", where the parameters are well-optimized for the test data but unprofitable when tested on out-of-sample data.

Pass	Profit	Total trades	Profit factor	Expected Payoff	Drawdown $	Drawdown %
1	-556.50	14	0.68	-39.75	1173.42	11.62
2	-396.00	2	0.00	-198.00	396.00	3.96
3	-1318.00	7	0.00	-188.29	1332.00	13.30
4	161.66	13	1.13	12.44	717.81	6.99
5	-1851.32	11	0.00	-168.30	1851.32	18.51
6	-1112.64	7	0.03	-158.95	1394.64	13.56
7	0.00	0	0.00	0.00	0.00	0.00
8	-1828.00	10	0.00	-182.80	2182.00	21.07
9	-588.00	3	0.00	-196.00	1144.00	10.84
10	209.22	18	1.12	11.62	793.41	7.54
11	-808.20	5	0.00	-161.64	928.72	9.29
12	-340.27	13	0.77	-26.17	1094.22	10.20

Settings | Optimization Results | Optimization Graph | Results | Graph | Report | Journal

Fig. 26.10 – The *Optimization Results* tab in the Strategy Tester

In the *Settings* tab, check the *Optimization* checkbox and press the *Start* button. Depending on the testing model you've selected and the number of parameters to optimize, it may take a long time. Once the optimization is complete, you can view the results of the optimization in the *Optimization Results* tab. The results can be sorted by total profit, profit factor, expected payoff or drop-down.

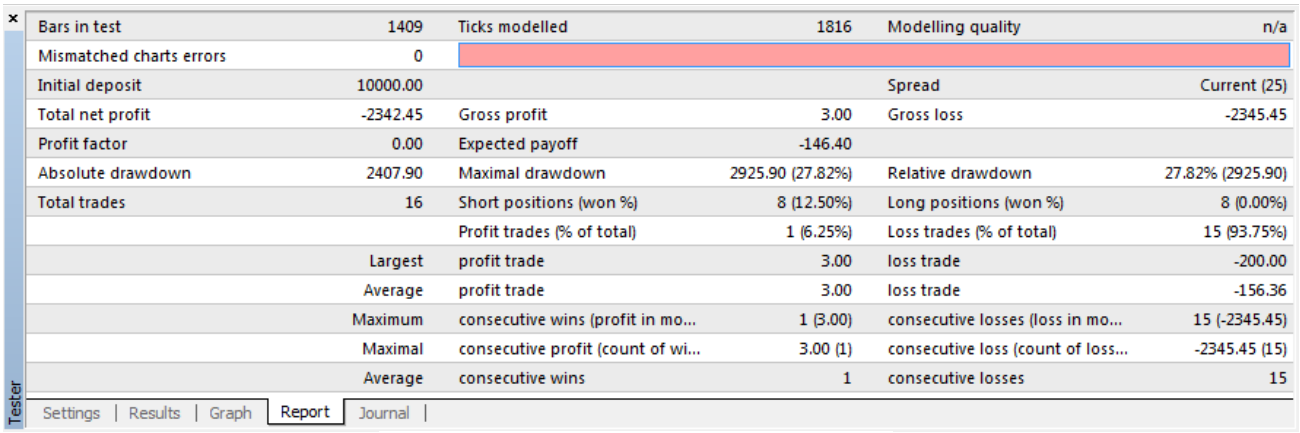
Double-click on a result to load it into the Strategy Tester. To do a quick out-of-sample test, set the starting date of the test to be the same as the end date of the optimization. Set the end date somewhere in the future. The length of the testing period in days should be approximately 25% of the length of the optimization period. Run the test and see how the results compare.

Evaluating Testing Results

The *Results* and *Optimization Results* tabs present a variety of statistical measures to evaluate the profitability and stability of your trading system. In this section, we will examine the most important statistics that appear in the trading and optimization reports:

- **Net Profit** – The *net profit* is calculated as the gross profit minus the gross loss. This is probably the most important statistic, and should always be considered in relation to other statistics. Obviously, a higher net profit is better.

- **Drawdown** – The *drawdown* is the maximum peak to valley loss during the testing period. The absolute drawdown is the maximum drawdown of balance or equity below the original starting balance. The maximal and relative drawdown is the maximum drawdown of equity of balance from the maximum profit to the maximum loss. Relative drawdown is the most important value. Lower values are better.
- **Profit Factor** – The *profit factor* is a simple ratio of gross profit to gross loss. A system that makes zero profit has a profit factor of 1. If profit factor is less than 1, then the system has lost money. A higher profit factor is better.
- **Expected Payoff** – The *expected payoff* is the average/profit or loss of a single trade. Higher values are better.



Bars in test	1409	Ticks modelled	1816	Modelling quality	n/a
Mismatched charts errors	0				
Initial deposit	10000.00			Spread	Current (25)
Total net profit	-2342.45	Gross profit	3.00	Gross loss	-2345.45
Profit factor	0.00	Expected payoff	-146.40		
Absolute drawdown	2407.90	Maximal drawdown	2925.90 (27.82%)	Relative drawdown	27.82% (2925.90)
Total trades	16	Short positions (won %)	8 (12.50%)	Long positions (won %)	8 (0.00%)
		Profit trades (% of total)	1 (6.25%)	Loss trades (% of total)	15 (93.75%)
	Largest	profit trade	3.00	loss trade	-200.00
	Average	profit trade	3.00	loss trade	-156.36
	Maximum	consecutive wins (profit in mo...	1 (3.00)	consecutive losses (loss in mo...	15 (-2345.45)
	Maximal	consecutive profit (count of wi...	3.00 (1)	consecutive loss (count of loss...	-2345.45 (15)
	Average	consecutive wins	1	consecutive losses	15

Tester | Settings | Results | Graph | Report | Journal |

Fig 26.11 – The *Report* tab in the Strategy Tester.

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

_

#

Finance / Investment **$29.95 USD**

Brand new and fully updated for the latest versions of MetaTrader 4, *Expert Advisor Programming for MetaTrader 4* is a practical guide to programming expert advisors in the MQL4 language. Leverage the latest features imported from the MQL5 language, including object-oriented programming, enumerations, structures and more.

This book will teach you the following concepts:

- The basics of the MQL4 language, including variables and data types, operations, conditional and loop operators, functions, classes and objects, event handlers and more.
- Place, modify and close market and pending orders.
- Add a stop loss and/or take profit price to an individual order, or to multiple orders.
- Close orders individually or by order type.
- Get a total of all currently opened orders.
- Work with OHLC bar data, and locate basic candlestick patterns.
- Find the highest high and lowest low of recent bars.
- Work with MetaTrader's built-in indicators, as well as custom indicators.
- Add a trailing stop or break even stop feature to an expert advisor.
- Use money management and lot size verification techniques.
- Add a flexible trading timer to an expert advisor.
- Construct several types of trading systems, including trend, counter-trend and breakout systems.
- Add alerts, emails, sounds and other notifications.
- Add and manipulate chart objects.
- Read and write to CSV files.
- Construct basic indicators, scripts and libraries.
- Learn how to effectively debug your programs, and use the Strategy Tester to test your strategies.

All of the source code in this book is available for download, including an expert advisor framework that allows you to build robust and fully-featured expert advisors with minimal effort.

Whether you're a new trader with limited programming experience, or an experienced programmer who has worked in other languages, *Expert Advisor Programming for MetaTrader 4* is the easiest way to get up and running in MQL4.