Syntax

If you are familiar with programming in any of the many languages derived from C, you will be very comfortable programming in MQL.

But for most non-programmers who want to tackle this language, you have to be aware of a new way of writing.

In MQL, every statement ends in a semicolon and is called an *expression*. An expression can span multiple lines, and there must be a semicolon at the end.

```
extern double StopLoss = 15.0; // single line expression
```

Or this multi-line expression:

```
if (FastMACurrent > SlowMACurrent)
OpenBuy=true; // multi line expression
```

If you are new to programming, you will need to make sure you are placing the semicolon at the end of every statement. To not do so is a common newbie mistake.

The Exception To The Semicolon Rule: The Compound Operator.

A compound operator is lines of code containing multiple expressions within braces {}. Compound operators can include control operators (if, switch), cycle operators (for, while) and function declarations. Below is an example of a control operator:

```
if(Bars<100)
{ Print("Bars less than 100");
return(0); }
```

Notice that you are not placing a semicolon after the initial if operator. You don't have to put a semicolon after the closing brace either. There is a semicolon after the**Print()** function. That is because there can be one or multiple expressions inside the braces, and each expression must end with a semicolon.

Razor TipThere is a quick way to check if your syntax is correct or not. If you press the compile

button from within the editor after completing your expressions or statements, you will get a pile of errors if you do have forgotten to complete an expression with a semicolon, or you have forgotten to complete a left parenthesis or left brace with a matching right parenthesis or right brace. These errors are a warning to go back and check over the semicolons, parenthesis, and braces.

Comments

Similar to comments in other languages, MQL4 comments are sometimes used in code to explain parts of the markup, or to temporarily remove code while testing and debugging. You will find it useful to document your code whenever possible in order to help you make sense of it. Coding language can be rather arcane at times, and it is helpful to spell out in plain English what your code is trying do accomplish.Moreover, you can temporarily remove the line of code, without having to delete it, by using comments.

There are two ways of entering comments: 1) Single Line Comments; and 2) Multi-Line Comments.

Single Line Comment Syntax

```
// We have made a comment
```

Comments follow the two forward slashes (//). Any line that begins with // is free text and ignored by the program.This can be useful for blocking out a line of code for testing purposes.

Multi-Line Comment Syntax

```
/* we are going to block out a bunch of code */
```

Comments are preceded by '/*' and end with '*/'.

This form of commenting is for quickly disabling many lines of code all at once, instead of having to put two forward slashes at the beginning of every line like the first method. When you have a bunch of code to block out, it would be too time consuming to keep putting the double forward slash in front of each line, and far more easy to use multi-line comment technique.

Variables

A variable is a basic storage unit of any programming language, holding data necessary for the program to function. Variables must be declared, and in order to declare a variable, you put three parts together, as in this example:

```
double Lots = 0.1;
```

Here it is broken down into its three parts:

data type (ex: double), followed by an identifier (ex: Lots), followed by the default value (ex: 0.1) after the equal sign.

The data type specifies the type of information the variable holds, such as a number, a text string, a date or a color. Here are some common data types:

Data Type	Description
int	A integer (whole number) such as 0, 3, or -5. Any number assigned to an integer variable is rounded up
double	A fractional number such as 0.01
string	A text string such as "You have shot your musket". Strings must be surrounded by double quotations.
boolean	A true/false value. It can also be represented as 1 (true) or 0 (false).
datetime	A time and date such as 2011.01.01 01:11.

Here my identifier "Lots" could have have been named many things, such as "ilots, "ordersize", "volume".

An *identifier* is the name given to variables and custom functions, and it can be any combination of numbers, letters, and the underscore character (_) up to 31 characters in length. The identifier is ultimately an arbitrary choice of words that should be descriptive of what you intend.

I have seen some coders use letters and numbers, such as "gd_104" as identifiers, instead of using simple words, but that ultimately makes the code unreadable to outsiders (or yourself if you forget what these combined letters and numbers refer to). It is preferable to make them easy to read and remember. Moreover, keep in mind that identifiers are case sensitive (Lots and lots are different identifiers) and also spelling sensitive (Lot and Lots are different identifiers), so make sure all identifiers are correctly cased and spelled (common Newbie mistake).

Once a variable has been declared, one can change its value by assigning a new value to it, in the following example:

```
double Lots = 0.1;
mylotsi = Lots;
// mylotsi is 0.1
```

Note that the assigned variable must be of the same data type, or it will lead to undesirable results.

Variables that are given the *extern* beforehand are called external variables.

```
extern double Lots = 0.1;
extern double StopLoss = 50;
```

They are useful to have in the first part of the EA in order to make their values accessible, and manipulatable from within the program properties windows.

Constants

If a variable holds a data value that can later be changed or modified, a constant is its opposite, a data value that never changes.

For example, all numbers from 0 to 9 are integer constants, true and false are boolean constants, red is a color constant, and 2011.01.01 00:00 is a datetime constant for January 10, 2010.

There is a wide variety of standard constants for things like price data, chart periods, colors and trade operations. You will see that **PERIOD_M1** is a constant for the M1 chart time frame, and OP_SELL refers to a sell market order.

You can learn more about constants in the Standard Constants of MQL4 Reference.

Functions

Functions are the building blocks of this language. A function is a block of code that is designed to carry out a specific task, such as placing an order or calculating a trailing stop.

Standard Functions

There are more than 220 standard functions in MQL4, and this is apart from the functions of technical indicators. Whenever you run across these functions in the code, they will often be in their own color (such as purple) and refer to an expression within parenthesis, such as the print function made referred to earlier.

```
Print("Bars less than 100");
```

The Print () function is a common function for declaring things to us outside the program, putting it in a similar category to functions like Comment () function, PlaySound() function, and MessageBox() function. You can read all about these standard (or native) functions here.

Functions have the advantage of being reusable, i.e., they can be executed from as many different points in the program as required. Generally, a function groups a number of program statements into a unit and gives it a name. This unit can be then invoked from other parts of a program.

We will learn more about working with standard and custom functions in the articles that follow.

Razor TipThere are so many native functions and constants in MQL4 that is next to impossible to expect anyone to learn and memorize them all. It is far more practical to learn them when you need to, on a case by case basis. Over time and through practice you will learn many of them. You will discover that most of the standard constants and functions have been color coded, and quick and easy way of learning about them is to hover your mouse over the color coded word and press F1. This will bring up a handy MQL4 reference for that word at the bottom of the editor.

Custom Functions

Outside of the standard or native functions, there are the functions that we can create for our own needs, in order to exploit that reusable advantage of functions. One you create a function, such as a close order function, you can reuse it over and over again in different parts of the code, or copy and paste the function for convenient use in other expert advisors you might work on.

Simple Logical (Relational) Operators

This article describes a class of operators known as *logical operators*. We humans use these operators every day without thinking much about them. For instance, we are constantly processing the logical operations of AND and OR. I will not leave the house when it is 10 degrees outside unless I first have my warm winter jacket AND hat AND I have someplace warm to go to. I will fall asleep tonight in my bed if I count endless sheep, OR imagine being part of a 17th Century Dutch landscape painting, OR read a thick Russian novel. We use these logical operations all the time but don't usually write them down (or think of them) as machine instructions.

Our MT4 program needs to make decisions, right or wrong, and these decisions require the use of logical operators.

Simple logical operators evaluate to *true* or *false,* and often propose the relationship between two or more arguments or conditions, which is why the are often called relational operators. Here is a table of logical (relational) operators:

Sign	Meaning	Function	Example
==	Equal To	true, if left hand argument has the same value as the right	If x == y, the condition is true
!=	Not Equal To, Inequality	opposite of equality, if left hand argument does not have the same value as the right	If x != y, the condition is false
>	Greater Than	true, if the left hand argument is greater than the right-hand argument	If x > y, the condition is true
<	Less Than	true, if the left hand argument is less than the	If x < y, the

		right-hand argument	condition is true
>=	Greater Than or Equal To	true, if left hand argument is greater than or equal to right	If x >= y, the condition is true
<=	Less Than or Equal To	true, if left hand argument is less than or equal to right	If x <= y, the condition is true
&&	AND	true, if both left and right-hand arguments are true	If x && y, the condition is true
||	OR	true, if either left or right-hand arguments are true	If x || y, the condition is true
!	NOT	true, if its argument is false; otherwise, false	If !x, the condition is true

NoteThe logical value FALSE is represented with an integer zero value, while the logical value TRUE is represented with any value differing from zero. The value of expressions containing operations of relation or logical operations is 0 (FALSE) or 1 (TRUE).
All the entries except the last are relational or comparison operators. I am going to try to provide more specific examples of these relational operators.

Here are some lines of code taken from a custom **OrdersTotalMagicOpen()** function:

```
if (OrderSymbol() != Symbol() || OrderMagicNumber() != MagicNumber) continue;
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
```

Within these two lines are contained four relational operators: **equal to** (==), **not equal to** (!=), **AND** (&&), and **OR** (||). The first line of code is a good representation of the **unequal** (!=) operator and the **OR** (||) operator. It is saying that if the open trade's symbol is NOT the symbol of the EA's underlying chart, OR if the open trade's magic number is not that of the strategy, then the program can **continue**. What does **continue** mean? Well, it is another operator that gives control to the beginning of the nearest outward cycle **while** or **for** operator, that is, it skips the immediate series of computations because they no longer apply. The second and third line are a good example of the **equal to** (==) and **AND** (&&) operators. It is saying that if the open trade's symbol is the symbol of the EA's underlying chart, and the open trade has the EA's magic number, then we can process the next computations.

We usually see these greater to or less than relational operators when comparing price points or indicators with each. For instance, let us look at one way of representing the buy condition of bollinger bands:

```
bool BuyCondition1 == false;

if (iBands(NULL,bandp, bandpx, banddev,0,PRICE_CLOSE,MODE_LOWER,1) < iClose (NULL,0,1)
&& iBands(NULL,bandp, bandpx, banddev,0,PRICE_CLOSE,MODE_LOWER,0) >= iClose (NULL,0,0)
BuyCondition1 == true;
```

Here you can see a buy condition that becomes true only if two arguments joined by AND (&&) are also true: the first argument has the **less than** (<) operator in effect, and the second has the **greater than equal to** (>=) operator in effect. The first argument is saying that the lower band of the previous BB must have been less than the previous close. The second argument is saying that the lower band of the current BB must now be greater than or equal to the current close. The two arguments combined translates into: *buy when the close crosses over the lower band*. Since there is no resident cross over function, the cross over must be constructed in two parts: what occurred in the immediate past (lower band was below close), and what is occurring in the present (lower band is now touching or above close).

Notethe **equal to** (==) operator is not the same as the **assignment** (=) operator.
The **assignment** operator is used when assigning a value to a variable. The **equal to**operator is used to evaluate a true/false condition.
You can compare any two values as long as they are of the same data type. You can compare a boolean value to the constants true or false.

Boolean Operations

We use the boolean operators AND (&&) and OR (||) to combine relation operations. The AND operator evaluates whether all conditions are true. If so, the entire statement is true. If any of the conditions are false, the entire statement is false.

```
if (BooleanVar1 == true && Indicator1 > Indicator2)
{
// Open order
}
```

If BooleanVar1 is equal to true, and Indicator1 is greater than indicator2, the statement evaluates to true, and the code between the braces is run. If either of these conditions are false, the entire statement evaluates to false,a nd the code in the braces is not run. There any number of conditions combined together with the && operator, and they must all evaluate to true.

The OR operator evaluates whether any one of the conditions are true. If at least one condition is true, the entire statement is true. If all the conditions are false, the statement evaluates to false.

```
if (BooleanVar1 = true || Indicator1 > Indicator2)
```

If either Booleanvar1 is equal to true, or Indicator1 is greater than Indicator2, the statement is evaluated as true. if both of these conditions are false, the statement evaluates to false.

You can combine AND and OR operations to create more complex trading conditions. When doing so, use parantheses to establish the order of operations.

```
if (BooleanVar1 == true && Indicator1 > Indicator2 || BooleanVar1 == false)
```

The statement within parenthesis is evaluated first. If both of these conditions are true, the statement evaluates to true, and we are left with an OR operation.

Basic Structure Of An Expert Advisor

There are approximately six structural elements in any given Expert Advisor, with five being optional and only one being essential. That's right, the structure of an EA can be as simple as just knowing the one essential part, though you will eventually want to know them all.

It is easy to see most of the structural pieces when one clicks on the *Expert Advisor Wizard* (select New / File in MetaEditor, or New button on the toolbar, or pressing Ctrl + N on keyboard). From the dialog choose Expert Advisor, press *Next*, type in whatever Name, Author and Link you want. The EA will be saved to the **\experts** folder under the name chosen.

Next, you can add a few external variables. Click the *Add* button (this will add a new record to your external variables list), and every record has three fields: *Name* (double click to set the name of the variable); *Type* (double click to set the data type); and *Initial Value* (double click to give your variable initialization value). We just added three variables to our sample BasicTemplate EA:

Name	Type	Initial Value
Lots	double	0.1
TakeProfit	int	250

StopLoss	int	150

It is a waste of time to put too many variables here because most should be put in manually. Press the Finish button and an EA template will open a document that looks like what I have below (note: I have altered it somewhat to make it even more useful to see at a glance what the structure is).

Please note: I remember when I was first learning to program in mql4, I was expecting the Expert Advisor Wizard to help me translate and code my trading ideas. Actually it doesn't do anything but present a hollow shell of a template. It is a very poor wizard indeed! In subsequent articles I'll present you with more complete templates for you to use and learn from.

```
//+--------------------------------------------------------+
//| BasicTemplate.mq4 |
//| Copyright © 2011, Forex Razor |
//| http://www.forexrazor.com |
//+--------------------------------------------------------+
// Structure #1 (Optional): Directives
#property copyright "Copyright © 2011, ForexRazor.COM"
#property link "http://www.forexrazor.com"

// Structure #2 (Optional): Input parameters

extern double Lots=0.1;
extern int TakeProfit = 250;
extern int StopLoss = 150;

// Structural #3 (Optional): expert initialization function

int init() {
//---- start up code
return(0); }

// Structure #4 (Optional): expert deinitialization function

int deinit() {
//----  shutdown code
return(0); }

// Structure #5 (Essential): expert start function

int start() {
//---- your trade conditions and orders
return(0); }

// Structure #6 (Optional): Custom Functions
```

```
int customfunction1(){
// your custom option conditions
return (0); }

void customfunction2(){
// your custom function conditions
}
```

The above is like the bare bones structure of any given EA without the code that gives it life. Each structural element deserves a bit of explanation.

Structural Element #1 (Optional): Preprocessor Directives

The first items to appear in the MQL template are preprocessor directives, prefaced by a #, and you can see that this template has two: **#property copyright**, which is the Author Name, and **#property link**, which is the link you may or may not have entered into the wizard. These are the only two property directives related to Expert Advisors, and they are completely optional to include or not.

Another type of preprocessor directive that you might see in some EAs is the **#include** directive. An include file consists of functions and source code that will be included in your project when it is compiled, the syntax of which is:

```
#include < filename.mqh >
```

A standard include file which you might see is **#include** stdlib.mqh, which comes with MetaEditor and includes several functions that programmers may find useful.

If you want to use a function that is already compiled in another file, such as another EA, library file or windows dll file, you can import these functions directly into your project using **#import** directives. For detailed examples of the import directive, you can check out http://docs.mql4.com/basis/preprosessor/import.

Most library files are called with the **#import** directive. Using preprocessor directives like #include and #import are usually reserved for advanced programmers. I thought I would stick them in here so that you know what they are when when you look under the hood of expert advisors that use them.

Structural Element #2 (Optional): External And Global Variables

The next section in our code are the external variables, and these can be very useful.

A variable is like a small box into which you can store things for later use, particularly numbers. The concept is borrowed from mathematics (remember the statement x = 10, where the value of 1 is stored in the variable x). From that point forward, until the value of x is changed, the value of x is 10. All references to x are replaced by the value of 10.

Of course we can declare any variable internally, from different parts of the program, but putting the variable here in this section, and preceding it with word **extern** allows you to be able to see the variable from the outside, that is, from the *Expert Properties* dialog of the expert advisor.

The external variables is where you can put the adjustable parameters of your program, such as your trade settings (lot size, takeprofit, and stoploss) and indicator settings. When you open the *Expert Properties* dialog for an expert advisor, you are viewing the external variables of that program.

Notice that we specify an external variable by adding **extern** in front of the variable. Extern makes sure that the variable will be external, that is, it will appear in the Expert Properties dialog, viewable and adjustable by the user.

```
extern int StopLoss = 150;
```

As I have hinted at, a *variable* is the basic storage unit of any programming language, holding data necessary for the program to function. Variables must be declared, and in order to declare a variable, you put three parts together:

data type (example: int), space, *identifier* (example: StopLoss), equal sign, *default value* (example: 150).

The data type specifies the type of information the variable holds, such as a number, a text string, a date or a color. For more on variables, see my description of Variables in Basic Concepts.

If you don't put extern in front of the variable, the variable still works, it is just internal, that is, not visible in the Expert Properties dialog. The advantage of declaring internal and external variables in this section is that they are global, meaning that is available to any function within the program. As long as the program is running, the global variables and it's values stays in memory and can be referenced by any function in the program.

Note: Though it is uniform and convenient to declare external and internal variables in this section of the EA, it is not absolutely necessary. You can declare any or all of your internal (though not external) variables from within any function. The small disadvantage is that the variable will only work for that function, not globally *across all functions.*

Structural Element #3 (Optional): Expert Initialization Function

The **init()** function is optional and can be left out if you are not using it.

This is the function that runs before any other functions.

The **init()** function will start to run:

1. after program has been attached to chart;
2. after client terminal has been started;
3. after symbol and/or chart period have been changed;
4. after program has been recompiled in metatrader;
5. after inputs have been changed from the window of expert;
6. after account has been changed.

The **start()** functions runs at first tick, but **init()** runs immediately after attachment (regardless of incoming ticks). It won't run if there is no connection to the server. It will be called the one time during the cycle of the program and will not be called again.

Very little code is usually placed within the **init()** function when it is used. The most common bit of code I see placed within this function are account protection schemes (bits of code to lock the EA to the user's account number), and locking the program to work with specific currency pairs and trade times.

I usually put my code that auto-defines Point and Slippage values in this area, as you can see here.

Structural Element #4 (Optional): Expert Deinitialization Function

The **deinit ()** function is also optional, and it can be left out if you are not using it. It is probably unlikely that you will need it in an expert advisor.

This is the last function the program will call after it is shutdown, and you can put here any removals you want.

The function is called:

1. when we complete the work with MetaTrader 4, or when we close the price chart;
2. when we switch between trading accounts;
3. when we change the time period of the schedule;
4. when we remove a judge from the price chart;
5. when we change the parameters of the expert;
6. when we recompile the program in MetaEditor

Most of the time I have seen **deinit()** never being used.

Structural Element #5 (Essential): Expert Start Function

This function is the most important function of the expert advisor, performing the most important work at any time. It is generally where most of your code will go, unless you are putting most of your code in custom functions, and thus it deserves special attention.

It always starts with:
int **start()** {
and always ends with
return(0); }

In between the start and end lines can be placed all the conditions to open and close your trades, as well as any number of trade procedures and conditions.

The **start()** function is run on every tick, and it will continue to be called for each and every tick of data that is received. Think of it as a loop and every time price moves the start function is triggered, and therefore everything we put in the start function will similarly be triggered. The start function is like the Movie Director of the EA. It calls the action and tells the actors and staff what is it they should be doing. It sets up the code and calls to all other functions.

I have seen many EA's have all the code placed within this function. If you wanted to examine a fully functional EA that contains this one structural element, along with a few external variables, you can open up the *MACD Sample.mq4* from within MetaEditor. You can see that the EA writes all its internal variables, indicators and conditions for entry and exit inside the start function. There are no other structural parts besides a few obvious external variables. It is probably one of the most simply written EAs that you will ever see, and that is probably why it was included with every MT4 download. It may not be that profitable, but it is simply and clearly written, helpful for the notice programmer to learn from.

In the next few articles we will focus mostly on the code placed within the **start()** function, along with some custom functions.

Structural Element #6 (Optional): Custom Functions

Any other functions that your EA may use should be declared after the start() function.

These custom functions can be called from the **start()**, **init()**, or **deinit()** functions, or from other functions that are called from the main program. The idea behind creating custom functions is to create blocks of code that carry out specific tasks, and then once created, you can call on these specific tasks from within any part of the program. Custom functions thus make things very tidy and convenient.

Some examples of custom functions are:

- Lot Sizing Function
- Order Counting Funtion
- Order Placement Function
- Order Closing Function
- Trailing Stop Function
- Much, much more

I have seen many programmers use custom functions to power their EA, so that very little code is needed for the **start()** function. They just plug the custom functions into the start function, and the EA is off and running.

When you find interesting custom functions from other EAs, you can copy and paste them into your own Ea and call on them from different parts of the program. It is not necessary that you fully understand how all the code within the custom function works, just that it works. You can figure out the nuts and bolts later.

In future articles I will be detailing a few useful custom functions that one can implement quickly into any EA.

Note: All custom functions will go below the start function. It must go below and NOT in the start function. If it goes in the start function it will confuse the EA, though it will probably not compile correctly anyway.

Basic Expert Advisor: MA Cross

The best way to illustrate how to code up an expert advisor is by example.

The few manuals and guides that discuss the building of an expert advisor tend to use the moving average cross (MACross) as an example. The reason for this is that it is the most popular indicator based strategy out there, and it so much easier to teach new coding concepts using a trading concept most people are already familiar with.

Following this familiar path, I submit a basic expert advisor based on a simple moving cross (20-200):

```
// Section 1:
// Preprocessor Directives, External & Internal Variables

#property copyright "Copyright © 2008-2010, ForexRazor.Com"
#property link "http://www.forexrazor.com/"

extern string EAName = "MACross";
extern double MagicNumber = 59483;

extern double Lots =0.1;
extern double LotDigits =2;
extern int Slippage = 5;

extern double StopLoss = 80;
extern double TakeProfit =0;

extern bool OppositeClose = true;
extern bool EnterOpenBar = true;

extern int FastMATime = 0;
extern int FastMAPeriod = 2;
extern int FastMAType = 0; //0:SMA 1:EMA 2:SMMA 3:LWMA
extern int FastMAPrice = 0;
extern int FastMAShift = 0;
extern int SlowMATime = 0;
extern int SlowMAPeriod = 30;
extern int SlowMAType = 1; //0:SMA 1:EMA 2:SMMA 3:LWMA
extern int SlowMAPrice = 0;
extern int SlowMAShift = 0;

// Global Variables

int Counter, vSlippage;
```

```
double ticket, number, vPoint;

double
FastMACurrent,
FastMAPrevious,
SlowMACurrent,
SlowMAPrevious;

// Section 2: Initialization

int init(){

if(Digits==3 || Digits==5)
{ vPoint=Point*10; vSlippage=Slippage*10; }
else{ vPoint=Point; vSlippage=Slippage; }

return(0); }

//-------------------------------------------------------
// Section 3: Start

int start()
{

if(Bars<100) { Print("Bars less than 100");
return(0); }

//--------------------------------------------------------
// Section 3A: Define ShortCuts to Common Functions

int Total, OType=-1, Ticket;
double Price, SL, TP, Lot;
bool CloseBuy=false, CloseSell=false, OpenBuy=false, OpenSell=false;

for(int Counter=1; Counter<=OrdersTotal(); Counter++)
{
if (OrderSelect(Counter-1,SELECT_BY_POS)==true)
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)

{
Ticket=OrderTicket();
OType =OrderType();
Price =OrderOpenPrice();
SL =OrderStopLoss();
TP =OrderTakeProfit();
Lot =OrderLots();
}
}
```

```
//------------------------------------------------------
// Section 3B: Indicator Calling

int Current = 0;

FastMACurrent = iMA(NULL, FastMATime, FastMAPeriod, FastMAShift, FastMAType,
FastMAPrice, Current + 0);

FastMAPrevious = iMA(NULL, FastMATime, FastMAPeriod, FastMAShift, FastMAType,
FastMAPrice, Current + 1);

SlowMACurrent = iMA(NULL, SlowMATime, SlowMAPeriod, SlowMAShift, SlowMAType,
SlowMAPrice, Current + 0);

 SlowMAPrevious = iMA(NULL, SlowMATime, SlowMAPeriod, SlowMAShift, SlowMAType,
SlowMAPrice, Current + 1);

//--------------------------------------------------
// Section 3C: Entry Conditions

bool OpenBar=true;
if(EnterOpenBar) if(iVolume(NULL,0,0)>1) OpenBar=false;


if (FastMACurrent > SlowMACurrent&& FastMAPrevious < SlowMAPrevious
&& OpenBar){
OpenBuy=true;
if (OppositeClose) CloseSell=true;
}

if (FastMACurrent<slowmacurrent&& style="outline: none;"> < SlowMACurrent&&
FastMAPrevious > SlowMAPrevious
&& OpenBar){
OpenSell=true;
if (OppositeClose) CloseBuy=true;
}
//---------------------------------------------------
// Section 3D: Close Conditions

while(true)
{
if (OType==0 && CloseBuy==true)
{
close (OP_BUY); // Close Buy
return;
}

if (OType==1 && CloseSell==true)
```

```
{
close (OP_SELL); // Close Sell
return;
}
break;
}
//-------------------------------------------------
// Section 3E: Order Placement

while(true)

{
if (OrdersTotalMagicOpen()==0 && OpenBuy==true)
{

if(StopLoss>0){SL=Bid - StopLoss*vPoint;}else{SL=0;}
if(TakeProfit>0){TP=Bid+TakeProfit*vPoint;}else{TP=0;}
ticket=0;number=0;
while(ticket<=0 && number<100){
RefreshRates();
ticket =
OrderSend(Symbol(),OP_BUY,NormalizeDouble(Lots,LotDigits), Ask,vSlippage,SL,TP,EAName,
MagicNumber, 0, Green);
return (ticket);
}}

if (OrdersTotalMagicOpen()==0 && OpenSell==true)
{
if(StopLoss>0){SL=Ask + StopLoss*vPoint;}else{SL=0;} if(TakeProfit>0){TP=Ask-
TakeProfit*vPoint;}else{TP=0;}
ticket=0;number=0;
while(ticket<=0 && number<100){
RefreshRates();
ticket= OrderSend(Symbol(),OP_SELL,
NormalizeDouble(Lots,LotDigits), Bid,vSlippage,SL,TP, EAName, MagicNumber, 0, Red);
return (ticket);
}}
break;
}
//--------------------------------------------------------
return; // End of start()
}


// Section 4A: Close Function

void close(int type){
if(OrdersTotal()>0){
```

```
for(Counter=OrdersTotal()-1;Counter>=0;Counter--){
OrderSelect(Counter,SELECT_BY_POS,MODE_TRADES);

if(type==OP_BUY && OrderType()==OP_BUY){
if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber) {
RefreshRates();
OrderClose(OrderTicket(),OrderLots(),NormalizeDouble(Bid,Digits), vSlippage);
} }

if(type==OP_SELL && OrderType()==OP_SELL){
if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber) {
RefreshRates();
OrderClose(OrderTicket(),OrderLots(),NormalizeDouble(Ask,Digits),vSlippage);
}}
}}}


// Section 4B: OrdersTotalMagicOpen Function

int OrdersTotalMagicOpen() {
int l_count_0 = 0;
for (int l_pos_4 = OrdersTotal() - 1; l_pos_4 >= 0; l_pos_4--) {
OrderSelect(l_pos_4, SELECT_BY_POS, MODE_TRADES);
if (OrderSymbol() != Symbol() || OrderMagicNumber() != MagicNumber) continue;
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
if (OrderType() == OP_SELL || OrderType() == OP_BUY) l_count_0++;
}
return (l_count_0);
}
</slowmacurrent&&>
```

It you have no prior programming experience, the above code might look a bit arcane and intimidating. One way to get over the intimidation factor is to worry less about the little details and focus on the big picture.

Try not to figure out exactly how each and every bit of the language works and wondering what is happening behind the scenes, at the processor level, and just accept that it is working. You don't need to get caught up in the details of the language and it's format. You don't need to be concerned with the nuts and bolts in order to understand and construct an EA. At this point in time you just need to know how the pieces of the puzzle fit together, and what are the most important pieces that can be manipulated in order to develop new strategies.

I will assist you in putting the puzzle together, and point you in the direction of the most important pieces. I have numbered and labeled each section in the EA to enable easier cross-referencing as I guide you through the understanding of each of the parts.

Section1: Preprocessor Directives, External And Internal Variables

First, a word on **Comments**. You might notice some comments that I have included after the //.

Any line that begins with // is free text and ignored by the program.

We include comments even though the computer ignores them in order to help explain in plain English the meaning of our programming statements. Yes, programming language can be hard to understand at first glance, and adding comments can be useful as you write your code to make your life easier.

Next, a word on **Preprocessor Directives**. Each directive begins with a pound sign (#). There are many advanced forms of directives, such as the #import and #include, but we are just using the simplest of them all, the **#property copyright** preprocessor directive that identifies the code as ours. Other than that, it is not that important and does nothing fancy.

The external variables are next, and they are important. In the previous article, I have explained how a variable is like a small box where can store things for later use. The external variable (which has the word *extern* preceding it) is important because it displays its parameters outside the program in the Expert Dialog box for the user to easily manipulate.

Many of the external variables you see in the basic EA above are self-explanatory. We will discuss EAName, MagicNumber, Lotsize, TakeProfit and Stoploss when you later take up the syntax of the OrderSend function, found in Section 3E, OrderPlacement. These variables refer mostly to that function.

The interesting external variables in this section are: moving average parameter variables (particularly MAPeriod), **OppositeClose**, and **EnterOpenBar**.

Moving Average Parameter Variables.

Notice that I have placed all the moving average parameter values as external variables. I did not have to do this. I could have just just made the most important variable, the MAPeriod, an external variable, leaving the rest of the parameters in their defaulted values within the indicators when I call them in Section 3B, Indicator Calling. I have declared almost all the parameters as external variables just in case I want to optimize any at a later point. For now I'll probably end up just optimizing the MAPeriod, but it could be useful to optimize some of the others in the future. We will discuss more about these parameters when we tackle Section 3B, Indicator Calling.

My True/False Bool (Boolean) Variables: OppositeClose, And EnterOpenBar

When you see a bool in a variable, it is a type used for values of truth. The type bool comes from Boole, the last name of the inventor of the logical calculus. Let's examine the bool OppositeClose.

```
extern bool OpositeClose=true
```

The external bool for this variable allows me to switch on and off the oppositeclose condition. Whenever oppositeclose is referenced in the code, it will be defaulted as true, meaning I want it to be switched on. If set to false, it will be swtiched off. Optionally, instead of using true or false, you can use 0 for false and 1 for true.

The **OppositeClose** bool refers to the idea of being able to close an order on an opposite signal. What does this mean? If set to true, and I am currently in a long position, and a short entry order is triggered, the short entry order closes out my current long before putting on a short trade. The short entry signal is the opposite signal that closes the current long (and vice versa). I have defaulted oppositeclose as true because I definately want it activated. If I had chosen false, that is, decativated the oppositeclose, the short entry signal would not close out my prior long trade, and my long trade would remain open until it was closed by hitting the stoploss or takeprofit. In general it is a good idea to have the oppositeclose set to true and activated. We will discuss the coding of oppositeclose in Section 3D, Close Conditions, as well as its related function found in Section 4A, Close function.

The **EnterOpenBar** bool refers to the idea of entering only at the open of each new bar, instead of interbar or close. When making new strategies based on indicators I prefer to default the EnterOpenBar as true in order to quickly see how the strategy backtests. The strategytester has three types of backtest modes in its drop down menu:*everytick*, *control points*, and *open prices only*. Every tick is more accurate but slower than the others. Open prices are less accurate but faster than the others. Control points sits in the middle of the two in both accuracy and speed. However, if **EnterOpenBar** is set to true, then you can safely backtest on the open prices only mode, vastly increasing your speed of backtesting, while at the same time having the very similar accuracy and results to the everytick mode. Besides the speed of backtesting, I have also noticed that when enteronopenbar is set to true, it improves the overall performance and reliability of the system, particularly if it is based on common indicators. I encourage you to experiment with turning the enteronopenbar to true and false in order to see the differences in results. The coding behind EnterOpenBar can be found in Section 3C, Entry Logic.

Lastingly, in this section I have declared a few internal variables (sometimes called Global Variables), such as

```
double ticket, number, vPoint;
```

Notice how I do not declare a particular value for each identifier. Without a declared value, each indentifier is defaulted at 0, waiting to be later determined. When it is finished with its determination, it reverts back to 0. Also notice that I am listing the identifiers after double, one after another, seperated by commas, till I end the statement with the semicolon. This can be done because none have a globally distinct value. I could have declared these from within the start() function, instead of here, but having them here in this section allows me to reference them globally, from within any function of the code. That is very handy and saves needless repetition.

Razor Tip!Remember, any time you come across specific indentifiers and it is hard to see what part of the code they refer to, there is a fast way to find their matching counterparts. Just copy and paste the indentifier (ex: ExpertName) into the find field (Cnt+F) in order to quickly jump down to the matching identifier residing in other parts of the code. Unless you like playing ISPY with words, you probably will find yourself doing this often to match up the different parts of the code.

Section 2: Initialization

As you can see, there is not much to this section.

What I have included in this section is the code for setting the point value relative to your broker's currency digits (brokers are set up with either a 4 digit or 5 digit quoting system):

```
if(Digits==3 || Digits==5)
{ vPoint=Point*10; vSlippage=Slippage*10; }
else{ vPoint=Point; vSlippage=Slippage; }
```

Plain English Translation: if your currency pair is quoted in digits of 3 or 5, then point value will equal Point*10 , and if not (such as 2 or 4), point value will remain as point value without a multiple.

Inserting code to automatically detect and adjust for fractional 3 or 5 digit brokers is a useful item, and I explain it more in its own article, Auto-Detect/Define Slippage and Point Values for 5 Digit Brokers.

Learn Syntax, Language and Structure. Notice how the if condition is put in parenthesis () and the statements are put in braces {}. That is the common structure of an if condition followed by its statements. In this case, the if condition is if(Digits==3 || Digits==5), keeping in mind that the double equal sign (==) stands for equals and the double vertical lines (||) stand for "or". Yes, you have to be aware of how your diction gets translated into machine language: while it would be convenient if we could just say "and" or "or," the program will not understand you if you use these words. Instead, you have to use the double vertical lines (||) for "or" and the double ampersand (&&) for "and".

NoteWhile it is easy to type in the double ampersand (&&) for "and," it is hard to type the double vertical lines (| |) for "or" so a quick short cut for this is just to copy and paste it.
Lastly, the first statement that falls in brackets{ vPoint=Point*10; vSlippage=Slippage*10; } has actually two statements separated by a semicolon: one statement defining what vPoint means and another statement defining what vSlippage means. When the condition is not met, there is the interlocking *else* function that points to an alternative compound statement in brackets { vPoint=Point; vSlippage=Slippage; }.

Section 3: The Start () Function

This section is the most important and longest, and it is best to divide this section into separate chuncks, which I have lettered as 3A, 3B etc.

Within the beginning of this start() function I have included the following lines:

```
if(Bars<100) { Print("Bars less than 100");
return(0); }
```

Translation: If Bars are less than 100, do not trade, and print on the screen that bars are less than 100. This is useful code to include in order to prevent a trade from occurring with insufficient bars loaded onto the chart.

Learn Syntax, Language and Structure. Here is another if condition (Bars < 100) set within parenthesis after "if". Now note that the expression that follows the if condition must be set within braces {} if it contains two or more compound statements, and each statement within the braces must be separated by a semicolon. In this example, we have two statements that follow the if condition. In the first statement, *Print* is a resident function that needs the have a description within quotes and surrounded by parenthesis. It will print that description on the screen when the condition is met. The semicolon completes that expression. In the second statement, *return (0)* means that no trades will occur, if there is less than 100 bars.

NoteEvery left brace must have a matching right bracket, or it will not compile, so we close the two statements with a right brace.

Section 3A: Define Short Tags To Common Trading Functions

Here I have defined a number of short tags that represent common trading functions and outfitted these tags to work with MagicNumbers.

Why would I want my trading functions to work with MagicNumbers?

The MagicNumber is the fingerprint of your EA, allowing the program to differentiate this EA from other EAs (or trades) operating on the same currency and timeframe. For instance, if I would like the program to track my open buy positions for this EA only, and not the open buy positions of the platform itself. Thus, when I reference any of my trade information functions, I would like to have them affiliated with the MagicNumber.
For a complete list and definitions of these trade information functions, click

here: http://docs.mql4.com/trading

I make my trading functions work with magicnumbers by placing them under
the **OrderSelect()** Function:

```
for(int Counter=1; Counter<=OrdersTotal(); Counter++)
{ if (OrderSelect(Counter-1,SELECT_BY_POS, MODE_TRADES)==true)
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
{
tag names = common trading functions;
}
}}
```

Translation: If there is any open or pending trades that have my MagicNumber, the following tag names will stand for common trading functions

Lean Syntax, Language and Structure. You will see some variant of the *OrderSelect* function quite often in different EAs, often within the first lines of any custom function (I have used it in my own two functions, 4A and 4B). The OrderSelect function selects an order for further processing, returning true if the function succeeds and false if it fails.

Because the **OrderSelect()** function is so important, I have creating its own article which you can reference called OrderSelect() Function: Retrieving Order Information

For my purposes, I am using the OrderSelect function to select by trades, **MODE_TRADES** (which means open and pending orders) and **MagicNumber**. In other words, I want the trading functions that I have subsumed under this function to work with open and pending orders that are of my magicnumber. The third line, particularly the part that says OrderMagicNumber() == MagicNumber, represents the condition for subsuming the trading functions within the MagicNumber. Of all the trading functions I have subsumed under this function, the one that I use later on in the code is OType=OrderType(), which I use in section 3D, Close Functions.

I should also mention the relevance of the first line in that block of code:

```
 for(int Counter=1; Counter<=OrdersTotal(); Counter++)
```

This is called a **for** operator, and it used to loop through a block of code a predetermined number of times. The first expression, **int = Counter =1**, initializes our **Counter**variable with a value of 1. The second expression, **Counter <=OrdersTotal()**, is the condition, if true, will execute the code within braces (if there were 3 open orders, it will execute the loop three time). The third expression, Counter++, means "increment the value of Counter by one." Every time the loop completes, the counter is incremented by 1 in this example, until eventually all the open orders are accounted for.

Section 3B: Indicator Calling

Here I declare four different moving averages:

```
FastMACurrent = iMA(NULL, FastMATime, FastMAPeriod, FastMAShift, FastMAType,
FastMAPrice, Current + 0);

FastMAPrevious = iMA(NULL, FastMATime, FastMAPeriod, FastMAShift, FastMAType,
FastMAPrice, Current + 1);

SlowMACurrent = iMA(NULL, SlowMATime, SlowMAPeriod, SlowMAShift, SlowMAType,
SlowMAPrice, Current + 0);

 SlowMAPrevious = iMA(NULL, SlowMATime, SlowMAPeriod, SlowMAShift, SlowMAType,
SlowMAPrice, Current + 1);
```

Each one refers to the moving average indicator that is native to MT4 and which has its own particular syntax:

```
double iMA (string Symbo), int Timeframe, int MAPeriod, int MAShift, int MAMethod, int
MAPrice, int Shift)
```

I like to imagine the structure in parenthesis that follows the iMA indentifier as a bus with a number of designated seats. Each seat in the bus is separated by a comma and is called a parameter. The iMA indicator has seven parameters. Each parameter in turn holds a default value that can be customized (or personalized, to keep with the bus metaphor). It is useful to know what the function of each parameter, the default values taking up each parameter, how they can be customized, and what parameters really drive the bus.

Below is table of the Describing each of the Parameters of the Moving Average:

MA Parameters	Description
Symbol	Symbol for trading, such as EURUSD. Symbol() represents the currency chart's pair
TimeFrame	The time period of the chart to apply moving average, usually set to 0, meaning the symbol of the chart EA is attached to.

MAPeriod	The look-back period of the moving average. This is the most important variable.
MAShift	The forward shift of the moving average line, in bars, usually set to 0.
MAMethod	The calculation method of the moving average, with choices including simple, exponential, smoothed or linear weighted. The second most important variable.
MAPrice	The price array to use when calculating the moving average, either close, open, high, low or some type of average. Usually the default of 0, or close, is used.
Shift	The backward shift of the bar to return the calculation for. A value of 0 returns the indicator value of the current bar, and a value of 3 will return the indicator value from 3 bars ago. This is the third most important variable, as we shall see.

A handy quick reference to MA parameters (as well as the parameters to all 20 native indicators) can be found here: http://docs.mql4.com/indicators/iMA

For our immediate purposes, we will be working with the default parameter values, and the most important parameter for our purposes is the MAPeriod, the length of the moving average, which I have defaulted as 2 for the FastMAPeriod and 30 for the SlowMAPeriod. It is the MAPeriod that drives the bus, because it differentiates the fast from the slow moving average.

MAMethod is also important, particularly the Simple (Integer=0) and Exponential (Integer=1). For the fast moving average, I am defaulted with 0 or Simple, and for the slow moving average, I am defaulted with 1 or Exponential. I will thus want the 30 period exponential moving average to cross the 2 period simple moving average in order to trigger a buy signal.

MAShift and MAPrice are usually left at 0, and changing these parameters has little effect.

The last parameter, Shift, bears no relationship to the fourth parameter, MAShift,
so do not confuse the two. Actually, this last parameter an important parameter for locating the moving average in time. It especially becomes important for differentiating the previous bar from the current bar, which is integral to our entry and exit logic.

Remember, all these parameters have been placed as external variables in order to easily modify or optimize them at a later stage.

Quick Question: If I am using just a fast and slow moving average for the Dual MA Crossover, why did I have to declare four moving averages?

When I want to make a dual MA Crossover, it is necessary to indicate what takes place before and after the crossover. As we shall see, the instance of the buy crossover is when the current fast MA is over the current slow MA while the previous fast MA was under the previous slow MA. That makes four moving averages that must be defined: FastMACurrent, FastMAPrevious, SlowMACurrent, SlowMAPrevious. What differentiates a current from a previous moving average? The only thing that differentiates the Current from the Previous moving averages is the last parameter, the Shift parameter: Shift is at 0 for current, and 1 for previous.

Section 3C: Entry Logic:

In the first part of the entry logic, we will be coding up the **EnterOpenBar** logic that we alluded to earlier. Interestingly, this is a short but important piece of code often overlooked by many expert advisors. The code looks like this:

```
bool openbar=true;
if(EnterOpenBar) if(iVolume(NULL,0,0)>1) openbar=false;
```

How does the program find the open of new bar? It has to find the first tick, which occurs on the new bar. Thus, in the above code, I am checking for volume and delaying the entry of the trade till it detects the first tick of the new bar found. You can see that I have two if conditional statements back to back.

The first one, "if (enteronopenbar)," refers to the bool variable, which I have earlier defaulted as true. When true it passes on to the next if conditional statement, "if (iVolume(NULL,0,0)>1)." This second if condition checks to see if the volume is 1, in which case **openbar** becomes true (anything greater than 1 is false) because it has found the first tick of the new bar. Since this checking for openbar is a simple but crucial component for any new system, I have discusses it a bit more in length in its own article.

Next, we move on to the brains of the EA, the strategy conditions for entry and exit.

Here are the buy and sell conditions that I intend to code:

Buy Condition1:
if 3 period moving average *crosses above* 30 period moving average,
buy at market (also close open sell position);

Sell Condition1:
if 3 period moving average *crosses under* 30 period moving average,
sell at market (also close open buy position).

How are these two conditions converted into MQL4 code?

There are many possible ways to code up crossover conditions, but for teaching purposes, we will go with the simplest. MT4 does not have a built in crossover function so we are going to build a common two step work around. We are going to indicate the buy crossover condition by watching if previous bar's fast moving average was previously below the slow moving average and now the current bar bar's moving average is above the slow moving average. Ergo, it has crossed over. Perhaps you can now better understand the following code.

```
if (
FastMACurrent > SlowMACurrent &&
FastMAPrevious < SlowMAPrevious
&& openbar)

{
OpenBuy=true;
if (OppositeClose) CloseSell=true;
}
if (
FastMACurrent FastMAPrevious > SlowMAPrevious && openbar)
{
OpenSell=true;
if (OppositeClose) CloseBuy=true;
}
```

In terms of syntax, when you declare an **if condition,** you must put the logic within braces{}, particularly if contains two or more statements. In order to suggest that the FastMACurrent must be greater than SlowMACurrent, I use the > sign, which is called an operator. We are basically going to say, if the previous bar's 20 period moving average was below the 200 period moving average, and if the current bar's 20 period moving average is now above the current bar's 200 period moving average, then buy at market. Since operators are so important to trading conditions, I wrote a brief article on them here

After the if condition, there are two statements enclosed in braces {} and separated by semicolons: the one statement sets OpenBuy as true if the moving average conditions have been met, and it is simple enough to understand. The second statement is a bit more nuanced. It sets CloseSell as true if the preceding moving average condition has been met, and the extern bool OppositeClose is also true. Notice how it requests its own internal if condition, in this case the OppositeClose bool = true or false, before it can activate its CloseSell or CloseBuy bools. I like to imagine these internal if bool conditions as key and lock mechanisms, enabling the end user to easily turn on and off mechanism (in this case, OppositeClose) from within the expert properties tab.

Section 3D: Close Conditions

We start off this section with the **while** operator. It is a simple method of looping in MQL4, similar to the for loop discussed above, but more appropriate if you are unsure of the number of iterations.

Our **while** loop basically looks like this:

```
while (true) {
// loop code
}
```

Next we place our conditions to close our buy and sell orders:

```
if (OType==0 && CloseBuy==true)
{
close (OP_BUY); // Close Buy
return;
}
```

OType is a variable that stands for trade information function called OderType(), and the each of the trade order types have a corresponding integer. OrderType == 0 refers to OP_BUY, which is buy position, while OrderType = 1 refers to OP_SELL, a sell position.

Here are various order types and their corresponding integer values:

OrderType	Integer	Description
OP_BUY	0	Buy Position
OP_SELL	1	Sell Position
OP_BUYLIMIT	3	Buy Limit Pending
OP_BUYSTOP	4	Buy Stop Pending
OP_SELLLIMIT	5	Sell Limit Pending
OP_SELLSTOP	6	Sell Stop Pending

If there is a current buy position (OType==1) and the bool CloseBuy is (==) true, then my custom close function, close (OP_BUY) can be executed. To learn about my custom close function, click here.

I end this section with a **break** operator. The operator 'break' stops the execution of the nearest external operator of 'while', 'for' or 'switch' type. The execution of the operator 'break' consists in passing the control outside the compound operator of 'while', 'for' or 'switch' type to the nearest following operator.

Section 3E: OrderPlacement

After the break operator of the previous section, I begin this section with another **while** operator. The **break** of the previous section has passed the control or flow over to this second **while** loop.

Here in this section the program is going to be placing its buy and sell orders.

It starts with the following condition:

```
if (OrdersTotalMagicOpen()==0 && OpenBuy==true)
```

OrdersTotalMagicOpen() is a custom program that we will visit shortly that counts the numbers of Open Orders subsumed within the magic number of the EA. If the order total is 0 (==0), then we can proceed. And (&&) if the bool OpenBuy is true, we can proceed.

Next comes the code for determining the values of the stop losses and profit targets:

```
if(StopLoss>0){SL=Bid - StopLoss*vPoint;}else{SL=0;}
if(TakeProfit>0){TP=Bid+TakeProfit*vPoint;}else{TP=0;}
```

Here we are using two **if-else** sequenced statements back to back. Let us go over them a bit. Keep in mind that an else condition evaluates an alternative condition, providing that the previous if statements false. We are combining else and if to create alternative conditions that will only be executed if true. In this case, we are saying that if StopLoss > 0, then we can determine the value of the StopLoss (Bid-StopLoss *vPoint). If it is instead 0 that is our StopLoss, we do not determine the value of the StopLoss and this alternative scenario is that the StopLoss remains as 0. The same logic is duplicate for the TakeProfit. The values that are determined from these if-else conditions are what turn up in the StopLoss and TakeProfit parameters of the **OrderSend()** function.

Next we are wanting to continue to refresh rates and push our orders through until they are filled:

```
ticket=0;number=0;
while(ticket<=0 && number<100){
RefreshRates();
```

We are saying that if ticket is 0 (order not yet filled), and if the number of attempts to fill is less than 100, then we will continue to refresh rates and attempt a fill. This is a useful code to insert if there are requotes in a quickly moving market and you want to be filled regardless. Again, you see the **while** operator in action, this time looping through the tickets and number of attempts to place the order.

Next comes the **OrderSend()** function. It is such a multifaceted function in its own right that I have wrote a brief article for it here.

Section 4A: Close Function

see the article on Custom Close function.

Section 4B: OrdersTotalMagicOpen Function

see the article on Order Counting Functions
Function.

Working With Price Data

Often when developing an EA you will work with bar price data, such as the *high*, *low*, *open*, or *close* of a particular bar.

In this article, we will explore four levels of price data:

Level	Price Data Constants / Functions	Functionality
1	High, Low, Open, Close, also: Bid, Ask	The simplest level. The price data refers to the current data price of the chart symb bar.
2	High[], Low[], Open[], Close[]	The next level above. The price data refers to the current data price of the chat sy flexible as to which historical bar it can refer to. The number in brackets refers to t Example: Close[0] = close of current bar, Close[1] = close of previous bar.
3	iHigh(), iLow(), iOpen(),iClose()	A very flexible level of functionality. The price data function refers to the price data time frame and/or historical bar. **Example #1:** iClose(NULL,0,0) = close of chart symbol, time frame, current bar. **No** Close or Bid or even iClose[0].

		Example #2: iClose ("EURUSD",30,1) = close of EURUSD symbol, of 30 minute tim use of it is a threefold increase in functionality than the previous price data functic
4	iHighest(), iLowest(),	Price data function to find the maximum price data values over a period range. Gc data values of a specific range. Example: Low[iLowest(Symbol(), 0, MODE_LOW, 3, bars.

Let us go over each of these four levels of price data.

Level 1 Price Data: The Price Data Limited To The Chart Symbol, Chart Time Frame And Current Bar.

If you simply want to refer to the price data of the current bar of the current chart and time frame, you can use any of the following price data methods: **High**, **Low**, **Open**,**Close**, **Bid**, **Ask**.

Keep in mind all must have an initial capital letter to be read by the program (Bid is valid, bid is not). If you want to refer to the current close price, you can use Close, but you can also use Bid, because the MT4 closing prices are defaulted as Bid prices. For instance, if you wanted your EA to look for a condition whereby the current closing price must be greater than the 200 period moving average, you can state it with:

```
Close > iMA (NULL,0,200,0,MODE_EMA,PRICE_CLOSE,1)
```

or, alternatively, as:

```
Bid > iMA (NULL,0,200,0,MODE_EMA,PRICE_CLOSE,1);
```

Note: Keep in mind that this simple level of price data can only be used for the chart symbol, chart time frame, and current bar. If you want to have the flexibility of working with different bars, you must turn to level 2 or level 3 price data. If you want to have the flexibility of working with different chart symbols, or different time frames, as well as different bars, you must work with level 3 price data.

Level 2 Price Data: The Price Data Flexible Enough To Work With Different Bars

If you want to work with price data for the current chart, but you want to be in control of the bar you are referring to, you can use the predefined series arrays: **High[]**,**Low[]**, **Open[]**, and **Close[]**.

An array is a variable that stores multiple values. They act like the list tables, wherein you can group items in the table and access them by row number, called Indexes. The numbered rows or indexes start from from 0, then proceed to 1, 2, 3 etc. These numbered indexes are contained in the brackets, and in the particular case above, each numbered index refers to a particular bar in time and changing the number changes the bar in time you want to refer to. For example, Close [0] is the open price of the current bar, where 0 is the index, and by changing it, we can get the close price of other bars: Close [1] refers to the bar previous to the current bar, Close [2] refers to the bar 2 bars back from the current, etc.

In most of the EAs we create we will be using either the current bar or the previous bar's price values. However, if you desire even greater flexibility of working with price data of not only different historical bars, but also different symbols other than your current chart, or different time frames other than your current chart, then you must turn to level 3 Price data.

Level 3 Price Data: The Most Flexible Price Data Using A Function That Works With Customizable Symbols, Customizable Time Frames, And Customizable Historical Bars.

If you want to work with price data (high, low, open, close) for a symbol other than the current chart, or if you need price data for a period other than the current chart period, you can use the following price data functions: **iHigh()**, **iLow()**, **iOpen()**, and **iClose()**. These will give you the price data value over a single period or bar.

The table below illustrates the syntax of the **iClose()** function:

```
double iClose (string Symbol, int Period, int Shift)
```

Parameters	Description
Symbol	The Symbol of the currency pair in use
Timeframe	Timeframe. Can be any Timeframe enumeration values.Choose (0) to display the current timeframe displayed on OR choose one of the following: (1, 5, 15, 30, 60, 240, 1440, 10080, 43200) {all in MINUTES}
Shift	The backward shift, relative to current bar

For instance, say for instance you have open a 1 hour chart but you want to check the close of the price of the previous bar on a daily chart.

Here is yesterday's highest high and lowest low sample:

```
double high = iHigh(NULL, PERIOD_D1, 1);
double low = iLow(NULL, PERIOD_D1, 1);
```

The Parameters signify:

- NULL = current chart symbol.
- PERIOD_D1 = D1 or daily chart period, also could be 1440.
- 1 = shift, 1 standing for previous bar.

Note: The **iHigh()** and **iLow()** functions gives you the shift of the maximum value of a single bar. If you want the shift of the maximum value over a range of bars, you must use the **iHighest()** and **iLowest()** functions, discussed further below.

Timeframe Enumeration Table

There are two ways to put in time frames: one by period constants and one by integer values, as in the example below.

```
double high = iClose (NULL,Period_D1,1);
double high = iClose (NULL, 1440,1);
```

You see that PERIOD_D1 can be substituted with 1440 -- the minutes comprising the day period.

Here is a full table of the constant values and their integer equivalents:

Constant	Value	Description
PERIOD_M1	1	1 minute
PERIOD_M5	5	5 minutes
PERIOD_M30	30	30 minute
PERIOD_H1	60	1 hour
PERIOD_H4	240	4 hour
PERIOD_D1	1440	Daily
PERIOD_W1	10080	Weekly
PERIOD_MN1	43200	Monthly
0 (zero)	0	Timeframe of chart

These minute value substitutions are easy enough to remember, and once memorized, I find them easier to type into their designated parameter box than constants. I also find that these integers can be more readily used with extern variables. For instance, what if we wanted to refer to a previous Close different from our own H1 chart but we did not know exactly which time frame to use. We can then can construct the timeframe parameter as an extern variable, as in the example below:

```
// put in extern variables section
extern int TimeFrame = 30;

// placed somewhere within your code
double PreviousClose = iClose (NULL, TimeFrame, 1);
```

TimeFrame is my identifier that refers to the timeframe parameter, and my default value is 30, which refers to the previous M30 bar. I know it is the previous bar because I put in a 1 in the shift parameter. If I want to refer to a current M5 close, I simply put in 5 in the second parameter, and if I wanted to refer to the current bar I simply put in 0 in the third parameter.

Now you have an easy way of making your EA reference multiple time frames, and you can even set the strategy tester to optimize between the timeframes if you so wanted.

NoteYou can use the alternative time period method (PERIOD_M30 = 30) and extern int method (extern int CloseTime = 30) for not only any price data function, but for all indicators and custom indicators; they all have the second parameter that refers to the time frame, which is usually defaulted to 0, or the time frame of your chart, but can be changed to any of the above time frames.

Alternative Currency Substitution Methodology

What is cool is that, not only can you easily reference multiple time frames, you can also easily reference multiple currency pairs. NULL stands for the symbol of the current chart, but it can be replaced with any currency symbol, even the currency outside the EA's own chart. To do so is simple: you just replace NULL with the symbol you want to use (it must appear in your market window) and wrap it in quotation markets.

Sample if you wanted to reference EURUSD symbol:

```
double PreviousClose = iClose ("EURUSD", 30,1);
```

Moreover, you have the flexibility of constructing an extern variable for this parameter, as in the example below:

```
// put in extern variables section
extern string CurrencyName = "EURUSD";
extern int TimeFrame = 30;

// placed somewhere within your code
double PreviousClose = iClose (CurrencyName, TimeFrame, 0);
```

As you can see we substituted NULL with an extern string variable, we identified as CurrencyName, and we defaulted to "EURUSD" (though it could be any currency pairthat you can see in your market window, so long as you put it within quotation marks). Also note that all currency names that you want to attach to an extern variable must use the extern string variable.

Why would you need to refer to a currency pair that is not the same as your chart?

I can think of many reasons, but the foremost that comes to mind is when you are trading by means of correlation. For instance, suppose that you are trading the GBPUSD, but that you consider EURUSD as a leader currency for most of the majors, and thus you want to first check to see the trend direction of the EURUSD as a prior condition for opening up trades on the GBPUSD.

NoteYou can use the currency substitution method (NULL = EURUSD), or the extern string method (extern string CurrencyName = "EURUSD") for not only price data functions, but all indicator and custom indicator functions; they all have the first parameter referring to the currency pair. The default is usually NULL, which stands for the current chart symbol, but it can be changed to any currency symbol that appears in your market watch window.

Level 4 Price Data: The Function That Works With Maximum Price Data Values Over Period Ranges

In addition to working with price data functions like **iHigh()** and **iLow** of a single period, there is the ability to work with price data functions of maximum value over a range of periods: **iHighest()** and **iLowest()**. With each of these functions, you can work with the h/l of a number of bars in a range.

These two functions have their own parameter set:

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

Parameters	**Description**
Symbol	The symbol used. NULL= current chart
TimeFrame	Timeframe. It can be any timeframe innumeration. 0= current timeframe
Type	Series array identifier. It can be any of the Series array identifier enumeration values
Bar Count	Number of bars you want to test, in direction from the start bar to the back one on which the calculation is carried out.
Start Bar	Shift showing the bar, relative to the current bar, that the data should be taken from.

Series array identifier table:

Constant	Value	Description
MODE_OPEN	0	Open Price
MODE_LOW	1	Low Price
MODE_HIGH	2	High Price
MODE_CLOSE	3	Close Price
MODE_VOLUME	4	Volume, used in iLowest() and iHighest() functions.
MODE_TIME	5	Bar open time, used in ArrayCopySeries() function.

Note: It appears as if you have the option to use any Type above, but in general practice, you will be using MODE_HIGH with **iHighest()** and MODE_LOW with **iLowest()**.

Sample of lowest low and highest high of the last 3 bars for a stop loss:

```
// stop = lowest Low of the last 3 bars
if(dir == OP_BUY)
{
stopLoss = Low[iLowest(Symbol(), 0, MODE_LOW, 3, 1)];
}

else if(dir == OP_SELL)
{
stopLoss = High[iHighest(Symbol(), 0, MODE_HIGH, 3, 1)];
}
```

The above is a clever technique of using a different type of stop than the regular one.

What if you wanted to get the lowest value between bars 10 to 20?

```
// calculating the lowest value on the 10 consequtive bars in the range
// from the 10th to the 19th index inclusive on the current chart
double val=Low[iLowest(NULL,0,MODE_LOW,10,10)];
```

Order Counting Functions

Introduction

It is useful to discover how many orders our EA has open and what type these open orders are, such as a buy or sell. It is thus useful to create a order counting function that can count the current number of open orders based on the order type.

MT4 Snippet

```
int OrdersTotalMagicOpen() {
int OrderCount = 0;
for (int l_pos_4 = OrdersTotal() - 1; l_pos_4 >= 0; l_pos_4--) {
OrderSelect(l_pos_4, SELECT_BY_POS, MODE_TRADES);
if (OrderSymbol() != Symbol() || OrderMagicNumber() != MagicNumber) continue;
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
if (OrderType() == OP_SELL || OrderType() == OP_BUY) OrderCount++;
}
return (OrderCount);
}
```

Explanation

We have named our order counting function OrdersTotalMagicOpen(). It will return an integer value of how many orders are currently opened on the specified chart symbol matching the magic number that we have passed as a function argument.

We start by declaring the OrderCount variable, the intial value of which is 0. We use the for operator to loop through the block of code, and the OrderSelect() function to examine the pool of currently open positions, checking for ones that match our order symbol and magic number.

If **OrderMagicNumber()** matches our MagicNumber, we can be confident that this order was placed by our EA. Note: uncertainty about EA identification would only arise if the user is running two EA's on the same currency symbol with the same magic number.

If **OrderMagicNumber()** does not match, we use the **!=** sign, which means "not equals", the program can **continue**, that is, it discards unaffiliated trades and moves on to trades that match up.

We next see if the order matches the order types **OP_BUY** or (||) **OP_SELL**. **OP_BUY** is a constant that indicates a buy market order. To count other types, simply replace**OP_BUY** or OP_SELL with the appropriate order type constant.

If the order matches both our magic number and chart symbol and order type, the value of the **OrderCount** will be incremented by one. After we have looped through all the orders in the order pool, we return the value of **OrderCount** to the calling function.

MT4 Usage

```
if (OrdersTotalMagicOpen() > 0 && CloseOrders == true){ // Close all orders }
```

If there are orders opened by this EA, and the value of CloseOrders is true, then the code inside the braces will run, which will close all the open orders.

Counting Only Buy Or Sell Orders

Sometimes it is necessary to just count the open buy orders or open sell orders, in which case we would have to split out the above function into two.

Counting Buy Orders

```
int BuyTotalMagicOpen() {
int OrderCount = 0;
for (int l_pos_4 = OrdersTotal() - 1; l_pos_4 >= 0; l_pos_4--) {
OrderSelect(l_pos_4, SELECT_BY_POS, MODE_TRADES);
if (OrderSymbol() != Symbol() || OrderMagicNumber() != MagicNumber) continue;
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
if (OrderType() == OP_BUY) OrderCount++;
}
return (OrderCount);
}
```

Counting Sell Orders

```
int SellTotalMagicOpen() {
int OrderCount = 0;
for (int l_pos_4 = OrdersTotal() - 1; l_pos_4 >= 0; l_pos_4--) {
OrderSelect(l_pos_4, SELECT_BY_POS, MODE_TRADES);
if (OrderSymbol() != Symbol() || OrderMagicNumber() != MagicNumber) continue;
if (OrderSymbol() == Symbol() && OrderMagicNumber() == MagicNumber)
if (OrderType() == OP_SELL) OrderCount++;
}
return (OrderCount);
}
```

You can see that the above two pieces of code are almost identical to what we had before, except that we have replaced the condition that checks for the order type being a **OP_BUY** or (||) **OP_SELL** with the the condition that checks for just the one type.

It is pretty easy to create a order counting function for every order type, replacing the constant **OP_BUY** for **OP_BUYLIMIT** or **OP_BUYSTOP**, for instance, and renaming the functions to differentiate the order types.

Retrieving Order Information With The OrderSelect() Function

Once we have successfully placed an order, we will need to gather information about the order, particularly if we want to modify or close it. All this is done through the OrderSelect() function. To use the OrderSelect(), we can ether use the ticket number of the order, or we can loop through the pool of open orders and select each of them in order.

Here is the syntax for the OrderSelect() function:

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

For a handy reference, a description of the above parameters can be found in the table below:

Params	**Description**

Index	Either the ticket number of the order we want to select, or the position in the order pool. The Select parameter indicates which one.
Select	A constant indicating whether the Index parameter is a ticket number or an oder 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_POOL**- By default, refers to the pool of currently opened orders • **MODE_HISTORY**- Examines the closed order pool (the order history)

Here is an example of an OrderSelect() function using the an order ticket number. It is set up for the modification of the of stop loss and take profit after a buy order has taken place:

```
if (OrdersTotalMagicOpen()==0 && OpenBuy==true)
{
ticket = OrderSend(Symbol(),OP_BUY,NormalizeDouble(Lots,LotDigits),
Ask,vSlippage,0,0,EAName, MagicNumber, 0, Green);
return (ticket);
if(ticket>0)
{
OrderSelect(ticket,SELECT_BY_TICKET);
OrderModify(OrderTicket(),OrderOpenPrice(),Bid - Stop_Loss * vPoint, Ask+TakeProfit *
vPoint,0,Green);
}
}
```

In the above example, we used the OrderSelect() to select by ticket number, and then conjoined it with the OrderModify() function, so we could modify the StopLoss and TakeProfit. This example is particularly useful for ECN brokers. In an ECN broker you cannot place your stoploss and takeprofit values in their corresponding parameters within the OrderSend() function. Instead, these parameters must remain as 0. Only after the trade has been placed, can the order stoploss and takeprofit be modified via the OrderSelect() and OrderModify() functions, as in the illustration above.

While the above OrderSelect() conjoins with the OrderModify() function, there is actually a range of order information functions that one can deploy to retrieve information about an order. There is a complete listing of these functions in the MLQ Reference. Here is a list of the commonly used order information functions:

Functions	Description
OrderSymbol()	The symbol of the instrument that the order was placed on.
OrderType()	The type of order: buy or sell; market, stop or limit.
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 of the selected order.
OrderTicket()	The ticket number of the selected order.
OrderMagicNumber()	The magic number of the selected order.

Market Market Orders With The OrderSend() Function

Every new trader wanting to code in MQL4 should be aware of how to use the OrderSend () function to place orders, whether they be market orders or pending stop or limit orders.

The OrderSend() function has the following syntax:

```
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, col Arrow = CLR_NONE);
```

For a handy reference, a description of the above parameters can be found in the table below:

Parameters	Description
symbol	Symbol for trading, such as EURUSD. Symbol() represents the currency chart's pair
Type	The type of order to place: buy or sell, which can be market, stop or limit There is an integer va • 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 number of lots to trade. You can specify mini lots (0.1) or micro lots (0.01) if your broker su
Price	The price at which to open the order. Generally at the Ask for a buy market order, at the Bid fo orders it will be at any valid price above or below the current price.
Slippage	The maximum slippage in points for the order to go through.
StopLoss	The stop loss price, below the opening price for a buy, and above for a sell. If set to 0, no stop
TakeProfit	The take profit price, above the opening price for a buy, and below for a sell. If set to 0, no tak
Comment	An optional string that will serve as an order comment. Comments are shown under the Trade them in th terminal, you right click on any of the open or closed trades, and in the box that op Comments. They serve as an order identifier.
MagicNumber	This is an optional integer value that will identify the order as being placed by a specific EA. W
Expiration	An optional expiration time for pending orders.
Arrow	An optional color for the arrow that will be drawn on the chart, indicating the opening price ar arrow will not be drawn.

The OrderSend() function returns the ticket number ('ticket' is the unique number of an order) of the placed order. We can save these order tickets to static variables for later use.

If no order was placed, due to an error condition, the return value wil be -1. This allows us to analyze the error and take appropriate action based on the error code. In order to get information about the reasons for rejection of the trade request, you should use the function GetLastError().

The Market Order

There are three types of orders that can be placed in MetaTrader: market, stop and limit orders. Being the most common, a market order opens a position immediately at the nearest Bid or Ask price.

Note on Bid and AskThe Bid price is what you see on the MT4 charts, and the Ask price is just a few pips above the bid price, with the difference between them being the spread. We open buy orders and close sell orders on the Ask price; we open sell orders and close buy orders on the Bid price.
Here is an example of a buy market order:

```
OrderSend (Symbol(), OP_BUY, Lots, Ask, Slippage, Bid-StopLoss
*Point, Bid+TakeProfit*Point, "EAName", MagicNumber, 0, Blue)
```

Here is an example of a sell market order:

```
OrderSend (Symbol(), OP_SELL, Lots, Bid, Slippage, Ask+StopLoss *Point, Ask-
TakeProfit*Point, "EAName", MagicNumber, 0, Blue)
```

The symbol() function returns the current chart symbol, and most of the time we will be placing orders on the symbol of the current chart. OP_BUY refers to the buy market order, just as OP_SELL would refer to a sell market order. Ask is a predefined variable in MLQ that stores the latest known seller's price (ask price) of the current symbol.

We generally set the slippage parameter with an external variable (example: extern slippage = 3). The slippage parameter (and corresponding variable) refers to the number of points to allow for price slippage. In other words, it represents the maximum difference in pips for the order to go through. Choosing the right slippage number can be a fine balance: you want to choose a small enough pip value that gives a good price, but at the same time you want to choose a large enough pip value so as not be requoted and miss your price altogether. If you set the slippage to 0, chances are you will be requoted often, and you might miss your price. If you set it to 3 or 4, you will be more likely filled.

Note on 5 digit brokers regarding SlippageIf your broker uses 4 digits quotes (or 2 for Yen pairs), 1 point = 1 pip; however, if your broker uses 5 digit quotes (or 3 for Yen pairs), then 1 point =0.1 pips, in which case you would need to add an additional zero the end of your Slippage setting. It is thus useful for the code to auto-check for 4 or 5 digit brokers and make the appropriate adjustments.
The rule for Stoploss and TakeProfit in regards to OB_BUY:

- Ask - StopLoss = you set the StopLoss Below (-) the Ask price
- Ask + TakeProfit = you set the TakeProfit Above (+) the Ask price

The rule for StopLoss and TakeProfit in regards to OB_SELL:

- Bid + StopLoss = you set the StopLoss Above (+) the Bid price
- Bid - TakeProfit = you set the TakeProfit Below (-) the Bid price

For the above rules, we are using external variables for our stop loss and take profit settings, for instance:

```
extern int StopLoss = 50;
extern int TakeProfit = 100;
```

In the above example, we want the stop loss to be 50 pips below the Ask for a buy market order, and the take profit to be 100 pips above. However, recently, with the addition of 5 digit brokers, there has been a problem in how the pips for the stop loss and take profit is calculated, as per note below.

Note on 5 Digit Brokers Regarding Point ValueIn order to convert the above integers into their appropriate fractional value, we need to multiply our external StopLoss and TakeProfit variable by the Point. Point is a predefined variable in MQL that returns the smallest price unit of a currency, depending on the number of decimal places, making a Point = 0.001 for 4 decimal quotes. But recently, many brokers have adopted fractional pip price quotes, with 3 and 5 decimal places, making a Point = 0.00001 for 5 decimal quotes. The problem in this case is that the above Stoploss would be calculated as 5 pips from opening price instead of 50 pips. That is not what we want. It is thus useful for the code to**auto-check for 4 or 5 digit brokers** and make the appropriate adjustments.

In the comment parameter (8th parameter of the OrderSend function) of our example buy and sell orders above, we typed in "EAName" with the idea that you can put the name of your EA in this field. It is thus one way of identifying your EA from the others. In the external MT4 platform, if you look at your terminal that shows your open and closed orders, you can see that the last field displays the comment field. If you do not see it, you can right click anyway on the terminal and put a check mark on Comments. If you are using multiple EAs on the same account, you should give a distinct name for each of your Eas in the comment field, and so when they all start to generate trades, you can differentiate which ones did which, by looking at the comment field in the terminal.

While the comment field helps you to visually differentiate your EAs from each other, the Magic Number that is deployed for its own parameter (9th parameter of the OrderSend function), helps the program differentiate your EAs from each other. In this parameter goes an integer value, such as "1234". We recommend that you construct an "extern double MagicNumber = 1234" instead, and then place the variable, MagicNumber, into this parameter. The external variable allows you to easily modify the Magic Number. Basically, the Magic Number is a unique number you assign to your orders for the program to distinguish orders opened by your expert advisor and orders that are opened by another expert advisor.

Note on Magic NumbersJust indicating a unique integer value in the Magic Number parameter field is not enough, by itself, for your program to differentiate open orders. Your magic number must also be included with the OrderSelect() function and OrderMagicNumber() function combination,whenever your code tries to reference your open trades.

The combination looks like this:

```
int total = OrdersTotal();
for (int cnt = 0 ; cnt < total ; cnt++)
{
OrderSelect(cnt,SELECT_BY_POS,MODE_TRADES);
if (OrderMagicNumber() == MagicNumber)
```

A good example of this combination code in context is to look at the custom closed function.

Pending Orders With The OrderSend() Function

Pending Stop Order

A *buy stop* order is placed above the current price, while a sell stop order is placed below the current price. The trader expects that the price will rise to the buy stop level or fall to the sell stop level and continue in that direction in profit.

In MQL4, there are two parameters that must be changed within the OrderSend() function to adjust it for stop orders, as we see in the following buy stop example:

```
OrderSend(), OP_BUYSTOP, Lots, Ask + PriceLevel * Point, Slippage, Bid-StopLoss
*Point, Bid+TakeProfit*Point, "EAName", MagicNumber, 0, Blue)
```

As you can see, the first and third parameters have been changed: the first parameter, representing the type of order, has been changed to OP_BUYSTOP, and the third parameter, representing price, now indicates Ask+PriceLevel instead of Ask. We want a plus sign (+) here because we will be adding pips to the ask price to represent a level above the market in which we want to enter into with the buy stop.

It is easy enough to create an external variable called

```
extern PriceLevel = 20
```

This external variable can be later modified, but the default of 20 means that I am expecting the market to reach up 20 pips before I enter into my trade: Ask + PriceLevel (defaulted as 20 my extern variable) * Point.

Let us examine an example of a sell stop:

```
OrderSend(), OP_SELLSTOP, Lots, Bid - PriceLevel * Point, Slippage, Ask+StopLoss
*Point, Ask-TakeProfit*Point, "EAName", MagicNumber, 0, Red)
```

As you can see once again, only the first and third parameters are different from the standard sell market order: the first parameter, representing the type of order, has been changed to OP_SELLSTOP, and the third parameter, representing price, now indicates Bid-PriceLevel instead of Bid.

We want a negative sign (-) here after Bid because we will be subtracting pips from the Bid price to represent a level below the market in which we want to enter into with the sell stop.

Pending Limit Order

Limit Orders are the opposite of the stop order. A buy limit order is placed below the current price, while the sell limit order is placed above the current price. The trader expects that the price will lower to the buy limit level, or rise to the sell limit level, trigger the order, then reverse direction for profit.

Here is an example of a buy limit order:

```
OrderSend(), OP_BUYLIMIT, Lots, Ask - PriceLevel * Point, Slippage, Bid-StopLoss
*Point, Bid+TakeProfit*Point, "EAName", MagicNumber, 0, Blue)
```

Again, only the first and third parameters are modified. We use OP_BUYLIMIT to indicate a buy limit order. We then indicate that the pending price must be less than the current price by modifying the price parameter: we use Ask - PriceLevel * Point, in order to represent that we want to take the trade if market drops, minus (-), to the the PriceLevel, defaulted to 20 pips below the current Ask price. Here is an example of a Sell Limit Order:

```
OrderSend(), OP_SELLSTOP, Lots, Bid + PriceLevel * Point, Slippage, Ask+StopLoss
*Point, Ask-TakeProfit*Point, "EAName", MagicNumber, 0, Red)
```

As you can see once more, the first parameter of order type has been modified to OP_SELL, and the third parameter of price has been modified to indicate that the entry price rests at the PriceLevel, defaulted to 20 pips above the current Bid price.

Expiration Parameter (Second To Last)

Suppose you want to put in an expiration for the pending buy stop order. This expiration time is located in the second to last parameter, and it needs to be in seconds.

For instance, suppose I had an EA that worked on an hourly chart, and I wanted it to expire in 6 hours. Well then, I would have to work how many seconds is in 6 hours, something like this: 60*60*6 = 21600

```
OrderSend(), OP_BUYSTOP, Lots, Ask + PriceLevel * Point, Slippage, Bid-StopLoss
*Point, Bid+TakeProfit*Point, "EAName", MagicNumber, TimeCurrent()+21600, Blue)
```

That order will then expire in 6 hours.

Sometimes I like to optimize this expiration time, and thus I like to put it in an extern variable. Suppose I want to optimize the expiration in hours.

Then I create an extern variable called PendingExpirationHours and a double that calculates the hours:

```
extern double PendingExpirationHours = 6;// Place this at top of source code with the
other extern variables

double expiration = 60*60*PendingExpirationHours;

OrderSend(), OP_BUYSTOP, Lots, Ask + PriceLevel * Point, Slippage, Bid-StopLoss
*Point, Bid+TakeProfit*Point, "EAName", MagicNumber, TimeCurrent()+expiration, Blue)
```

Now, if I want to, I can go ahead and optimize PendingExpirationHours from 0-50, with a step of 1, to see which hour works best for an expiration time of the pending order.

Closing Orders With The OrderClose() Function Or A Custom Close Function

When we close a market order, we are exiting the trade at the current market price, which would be the current Bid price for buy orders and the current Ask price for sell orders.

We close market orders using the **OrderClose()** function, which has the following syntax:

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

The table below describes each of these parameters:

Parameter	Description
Ticket	The ticket number of the market order to close
Lots	The number of lots to close.
Price	The preferred price at which to close the trade.
Slippage	The allowed pip slippage from closing price.
Color	A color constant for the closing arrow.

The example below closes a buy market order, borrowed from the MACD Sample:

```
for(cnt=0;cnt<total;cnt++)
 {
OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
if(OrderType()<=OP_SELL && // check for opened position
OrderSymbol()==Symbol()) // check for symbol
{
if(OrderType()==OP_BUY) // long position is opened
{
// should it be closed?
if(MacdCurrent>0 && MacdCurrentSignalPrevious &&
MacdCurrent>(MACDCloseLevel*Point))
{
OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet); // close position
return(0); // exit
}
</total;cnt++)
```

Explanation

Here the close code starts with the for operator, continuously looping through the block of code to calculate the closing conditions.

The function **OrderSelect()** is used to examine the pool of opened orders, looking for the matching symbol, as well as the order types (O**P_BUY** or **OP_SELL**).

The last, and most important condition for this block of code, is to watch for the strategy conditions for exiting, which in the example above is the MACD exit conditions.

Once the open position symbol and order types are identified, and the MACD exit conditions are met, then the **OrderClose()** function can be deployed, the parameters of which are discussed in the table above.

Custom Close Function

It is can be useful to work with a custom close function because then you can can invoke it easily from within your **Start()** function whenever you have the need to close an order by a any set of conditions. Your custom close function should examine the pool of currently open orders, identifying your order types along with their magic numbers.

MT4 Code Snippet

```
void close(int type){
if(OrdersTotal()>0){
for(Counter=OrdersTotal()-1;Counter>=0;Counter--){
OrderSelect(Counter,SELECT_BY_POS,MODE_TRADES);

if(type==OP_BUY && OrderType()==OP_BUY){
if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber) {
RefreshRates();
OrderClose(OrderTicket(),OrderLots(),NormalizeDouble(Bid,Digits), vSlippage);
} }

if(type==OP_SELL && OrderType()==OP_SELL){
if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber) {
RefreshRates();
OrderClose(OrderTicket(),OrderLots(),NormalizeDouble(Ask,Digits),vSlippage);
}}
}}}
```

MT4 Usage

```
if (OType==0 && CloseBuy==true)
{
close (OP_BUY); // Close Buy
```

Explanation

We have chosen **close()** as the name of our custom close function, and whenever we want to invoke it, we simply have to insert that one word after our close conditions, as in the usage example above.

The second line initiates the function if there is more than zero total open positions. The third line examines the pool of currently open orders, counting them. The fourth line invoke the **OrderSelect()** function in order to select these counted orders for further processing.

We have constructed only one parameter for the custom close function, and that is the variable **int type** in parenthesis (). This parameter is going to be the identifier of our order type. In the custom function, we indicate that type will equal (==) either **OP_BUY** or **OP_SELL**, and so we when we invoke **close()**, we must indicate if **close()** is close (OP_BUY) or close (OP_SELL), as in the usage example above. In the function, we associate type==OP_BUY and OrderType==OP_BUY together with the ampersand (&&).

We also want to make sure that we are selecting buy orders of the correct symbol and magic number, as you can in line 6:

```
if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber) {
```

Refresh Rates() is a function that refreshes the rates of the currency pair, so at to get filled at the most current rates.

Lastly, we deploy the **OrderClose()** function, the parameters of which can be found in the table above.

Constructing Trade Conditions

We use the conditional operators if and else to construct our trading conditions.

The if operator evaluates a true and false condition. If the condition is true, the code immediately after the if statement is executed. If the condition is false, it will skip ahead to the code following the if block:

```
if (BuyCondition=true)
{OpenBuyOrder(...);
}
```

If there is only one statement folowing the if operator, it can be written like this:

```
if (BuyCondition == true) OpenBuyOrder(...);
```

Multiple statements must be enclosed in braces.

The else operator evaluates an alternative condition, provided that the previous if statement(s) are false. You can combine else and if to create an alternative condition that will only be executed if it is true.

For example, this code evaluates three conditions in order. If one of them is true, only that block of code will be executed. If none of them are true, none of them will be executed:

```
if (Condition1 == true) // execute condition1
else if (Condition2 = true) // execute condition2
else if (Condition3 = true) // execute condition3
```

The else operator can be used by itself at the end of an if-else sequence to indicate a condition that will be executed by default if all of the other if operators are false.

Building Strategy Conditions With Indicators

The majority of trading systems use indicators to determine trading signals. Metatrader includes over 20 common indicators, including moving average, MACD, RSI, and stochastics. MQL has built in functions for the stock indicators. You can also use custom indicators in your expert advisor.

Trend Indicators

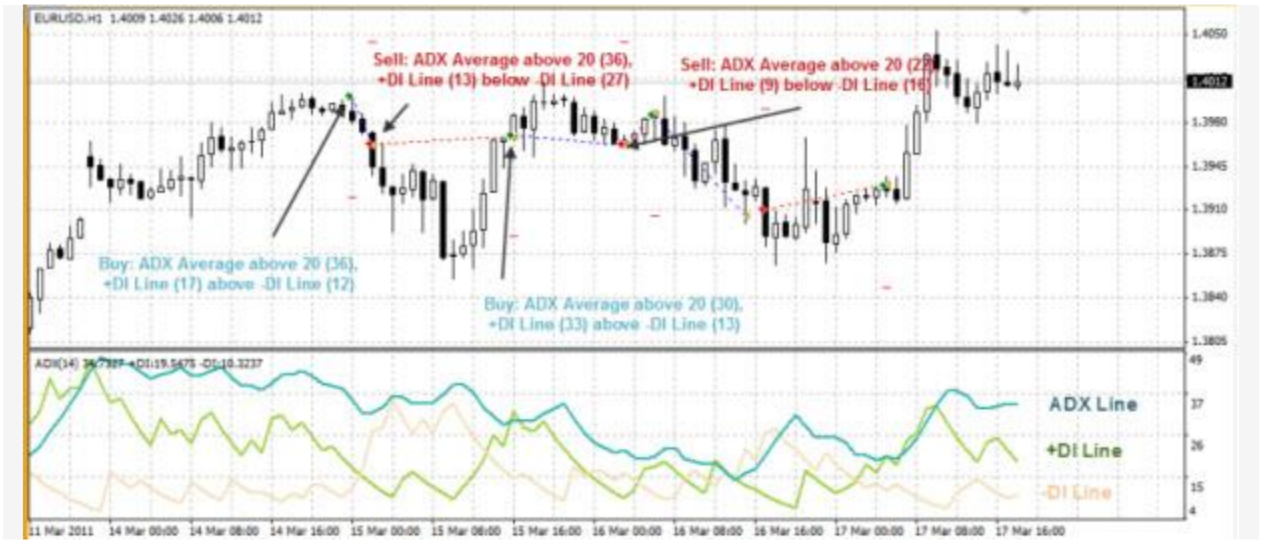
The moving average is most well known of the trend indicators. It shows whether the price has moved up or down over the indicator period. We have seen how to contruct the conditions of a moving average crossover. Let us examine the conditions of other trend indicators for a strategy entry and exit.

If you go to your MT4 console and you click **Insert / Indicators / Trend**, you get the following list of indicators:

- Average Directional Index
- Bollinger Bands
- Commodity Channel Index
- Moving Average
- Parabolic Sar
- Standard Deviation

All of these can be converted into buy and sell conditions within your expert advisors, as you shall see below.

Note on Usageif you are interested in inserting any of the code in your custom EA, you can copy and paste the extern variables into your extern variables section. Then you copy and paste the indicator calling variables somewhere within your **start()** function, using my templates as guide, and just below these, you can copy and paste your buy and sell conditions. Alternatively, you download and use the EAs that I have constructed around each indicator.



Params	Average Directional Index

Intent	//Buy: +DI line is above -DI line, ADX is more than a certain value and grows (i.e. trend strengthens)
	//Sell: -Dline is above +DI line, ADX is more than a certain value and grows (i.e. trend
	strengthens)

Extern	extern int adx=0; //Indicator period
	extern int adu=14; //Period of averaging for index calculation
	extern double minadx=20; //Minimal threshold value of ADX

Indicator Calling	PosDLine =iADX(NULL,adx,adu,PRICE_CLOSE,MODE_PLUSDI,0);
	NegDLine =iADX(NULL,adx,adu,PRICE_CLOSE,MODE_MINUSDI,0);
	ADXCurrent =iADX(NULL,adx,adu,PRICE_CLOSE,MODE_MAIN,0);
	ADXPrevious =iADX(NULL,adx,adu,PRICE_CLOSE,MODE_MAIN,1);

BuyCond	if (PosDLine > NegDLine && ADXCurrent >= minadx && ADXCurrent > ADXPrevious)
SellCond	if (NegDLine > PosDLine && ADXCurrent >= minadx && ADXCurrent > ADXPrevious)

Params	Bollinger Band
Intent	//Buy: price crossed lower line upwards (returned to it from below) //Sell: price crossed upper line downwards (returned to it from above)

Extern	extern int bandp=0; //Indicator period
	extern int bandpx=20; //Period of averaging for indicator calculation
	extern int banddev=2; //Deviation from the main line

Indicator Calling	BBLowCurrent=iBands(NULL,bandp, bandpx,
	banddev,0,PRICE_CLOSE,MODE_LOWER,0);
	BBLowPrevious=iBands(NULL,bandp, bandpx,
	banddev,0,PRICE_CLOSE,MODE_LOWER,1);
	BBUpCurrent=iBands(NULL,bandp, bandpx,
	banddev,0,PRICE_CLOSE,MODE_UPPER,0);
	BBUpPrevious=iBands(NULL,bandp, bandpx,
	banddev,0,PRICE_CLOSE,MODE_UPPER,1);
	BBCurrentClose = iClose (NULL, 0,0);
	BBPreviousClose = iClose (NULL, 0,1);

BuyCond	if (BBLowPrevious<BBPreviousClose && BBLowCurrent>=BBCurrentClose)
SellCond	if (BBUpPrevious>BBPreviousClose && BBUpCurrent<=BBCurrentClose)

	Commodity Chanel Index
Intent	//Buy: 1. indicator crosses +100 from below upwards. 2. Crossing -100 from below upwards. 3. //Sell: 1. indicator crosses -100 from above downwards. 2. Crossing +100 downwards. 3.

Extern	extern int CCp=0; //Indicator period
	extern int CCpx=14; //Period of averaging for indicator calculation
	extern int CCLine = 100;

Indicator Calling	CCCurrent = iCCI(NULL,CCp,CCpx,PRICE_TYPICAL,0);
	CCPrevious = iCCI(NULL,CCp,CCpx,PRICE_TYPICAL,1);
	CCCrossLinePos = CCLine;
	CCCrossLineNeg =-CCLine;

BuyCond	if ((CCPrevious<CCCrossLinePos && CCCurrent >= CCCrossLinePos) || (CCPrevious <=CCCrossLine CCCurrent>=CCCrossLineNeg))
SellCond	if ((CCPrevious>CCCrossLinePos && CCCurrent <= CCCrossLinePos)|| (CCPrevious >=CCCrossLineNeg&& CCCurrent<=CCCrossLineNeg))

	Parabolic Sar
Note	//Buy: Parabolic SAR crosses price downwards //Sell: Parabolic SAR crosses price

	upwards

Extern	extern int sar=0; //Indicator period
	extern double sarstep=0.02; //Stop level increment
	extern double sarstop=0.2; //Maximal stop level
	extern int sar2=0; //Price period

Indicator Calling	sarcurrent = iSAR(NULL,sar,sarstep,sarstop,0);
	sarprevious = iSAR(NULL,sar,sarstep,sarstop,1);
	closecurrent = iClose(NULL,0,0);
	closeprevious = iClose(NULL,0,1);

BuyCond	if (sarprevious>closeprevious&&sarcurrent<=closecurrent)
SellCond	if (sarprevious<closeprevious&&sarcurrent>=closecurrent)

	MA Rising or Falling

Intent	//Buy: MA grows
	//Sell: MA
	falls

Extern	extern int maperiod=14; //Period of averaging for indicator calculation
	extern int mamode = 1; // Type of moving average, 1= Exponential

Indicator Calling	macurrent = iMA(NULL,0,maperiod,0,mamode,0);
	maprevious = iMA(NULL,0,maperiod,0,mamode,1);
	maprevious2 =iMA(NULL,0,maperiod,0,mamode,2);

BuyCond	if(maprevious2 > maprevious && maprevious > macurrent)
SellCond	if (maprevious2 < maprevious && maprevious < macurrent)

Building Strategy Conditions With Oscillators

The other type of indicator is an oscillator. Oscillators are drawn in a separate window and they oscillate between high and low price extremes. They are centered around a neutral axis (generally 0), or they are bound by an upper or lower extreme (such as 0 and 100). Examples of oscillators include momentum, stochastics and RSI.

Oscillators indicate overbought and oversold levels. While they can be used as an indicator of trends, they are generally used to locate areas of pending reversal. These are used to produce counter-trend signals.

Below is a number of occillators:

	MACD(1)
Note	//Buy: MACD rises above the signal line //Sell: MACD falls below the signal line

Extern	int fastmacd=12; //Averaging period for calculation of a quick MA
	extern int slowmacd=26; //Averaging period for calculation of a slow MA
	extern int signalmacd=9; //Averaging period for calculation of a signal line

Indicator Calling	macdmaincurr=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_MAIN,0);
	macdmainprev=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_MAIN,1);
	macdsigcurr=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_SIGNAL,0);
	macdsigprev=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_SIGNAL,1);

BuyCond	if (macdmainprev < <macdsignalprevious&&macdmaincurrent style="outline: none;">macdsignalprev && macdmaincurr >= macdsignalcurr) </macdsignalprevious&&macdmaincurrent>
SellCond	if (macdmainprev > macdsignalprev && macdmaincurr <=macdsignalcurr)

	MACD(2)

Note	//Buy: crossing 0 upwards //Sell: crossing 0 downwards

Extern	int fastmacd=12; //Averaging period for calculation of a quick MA
	extern int slowmacd=26; //Averaging period for calculation of a slow MA
	extern int signalmacd=9; //Averaging period for calculation of a signal line

Indicator Calling	macdmaincurr=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_MAIN,0);
	macdmainprev=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_MAIN,1);
	macdsigcurr=iMACD(NULL,0,fastmacd,slowmacd,signalmacd,0,MODE_SIGNAL,0);
	macdsigprev=iMACD(NULL,0,fastmacd,slowmacd,signalmacd0,MODE_SIGNAL,1);

BuyCond	if (macdmainprev < 0<macdsignalprevious&&macdmaincurrent style="outline: none;"> && macdmaincurr >= 0) </macdsignalprevious&&macdmaincurrent>
SellCond	if (macdmainprev > 0 && macdmaincurr <= 0)

	RSI
Note	//Buy: crossing 30 upwards //Sell: crossing 70 downwards

Extern	rsiu = 14; // averaging period
	lowerband = 30;
	upperband = 70;

Indicator Calling	rsicurrent=iRSI(NULL,0,rsiu,PRICE_CLOSE,0);
	rsiprevious=iRSI(NULL,0,rsiu,PRICE_CLOSE,1);

BuyCond	if (rsiprevious<lowerband&&rsicurrent>=lowerband)
SellCond	if (rsiprevious>upperband&&rsicurrent<=upperband)

	Stochastics Occilator(1)
Note	//Buy: main line rises above 20 after it fell below this point //Sell: main line falls lower than 80 after it rose above this point

Extern	stok = 5; // Period(amount of bars) for the calculation of %K line
	stod = 3; //Averaging period for the calculation of %D line
	stslow =3; // Value of slowdown
	mamode = 1;
	lowerband = 20;
	upperband = 80;

Indicator Calling	stocurrent=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_MAIN,0);
	stoprevious=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_MAIN,1);

BuyCond	if (stoprevious<lowerband&&stocurrent>=lowerband)
SellCond	if (stoprevious>upperband&&stocurrent<=upperband)

	Stochastics Occilator(2)
Note	//Buy: main line goes above the signal line //Sell: signal line goes above the main line

Extern	stok = 5; // Period(amount of bars) for the calculation of %K line
	stod = 3; //Averaging period for the calculation of %D line
	stslow =3; // Value of slowdown
	mamode = 1;
	lowerband = 20;
	upperband = 80;

Indicator Calling	stomaincurr=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_MAIN,0);
	stomainprev=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_MAIN,1);
	stosignalcurr=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_SIGNAL,0);
	stosignalprev=iStochastic(NULL,0,stok,stod,stslow,mamode,0,MODE_SIGNAL,1);

BuyCond	if (stomainprev<stosignalprev&&stomaincurr>=stosignalcurr)
SellCond	if (stomainprev>stosignalprev&&stomaincurr<=stosignalcurr)

	Williams % Range
Note	//Buy: crossing -80 upwards //Sell: crossing -20 downwards

Extern	willperiod = 14; // averaging period
	lowerband = -20;
	upperband = -80;

Indicator Calling	willcurrent=iWPR(NULL,0,willperiod,0);
	willprevious=iWPR(NULL,0,willperiod,1);

BuyCond	if (willprevious<upperband>willcurrent>= upperband)
SellCond	if (willprevious>lowerband&&willcurrent<=lowerband)

	Force Index
Note	/Flag 14 is 1, when FI recommends to buy (i.e. FI<0) //Flag 14 is -1, when FI recommends to sell (i.e. FI>0)
Extern	forceu = 2; // averaging period
Indicator Calling	forcecurrent=iForce(NULL,0,forceu,MODE_SMA,PRICE_CLOSE,0);
BuyCond	if (forcecurrent<0)

SellCond	if (forcecurrent>0)

Preparing Custom Indicators For Strategy Deployment: Coverting Into ICustom Functions (Example: NonLagMA)

An incredible advantage in using MT4 is the use and integration of hundreds of custom indicators available online. Many can easily use a custom indicator by dragging and dropping it from the custom indicators window into your chart.

In order to use a custom indicator for an expert advisor, there are a few things you need to know. The custom indicator that you are wanting to use for your EA must be located in the indicators directory (C:\Program Files\MetaTrader 4\experts\indicators for standard installations of MT4). When an EA is executed, it looks for the compiled indicators (.ex4 file) in the indicators directory. There is a built-in function for working with custom indicators called the **iCustom()**. The **iCustom()** is a MQL4 function that enables you to use external indicators in your expert advisor or custom indicator code without re-writing the code from scratch.

The syntax of the function is as follows:

```
double iCustom(string Symbol, int Timeframe, string IndicatorName, int Indicator Parameters,
int Mode, int Shift);
```

Parameters	Description
Symbol	The data symbol used to calculate the indicator. NULL means current symbol.
Timeframe	Timeframe. Can be any Timeframe enumeration values. Choose (0) to display the current time of the following: (1, 5, 15, 30, 60, 240, 1440, 10080, 43200) {all in MINUTES}
IndicatorName	This is the name of the indicator file as it appears in the *Custom Indicators* list in the *Naviga* "NonLagMA_v7.1.1".
Indicator	The Inputs tab in the Custom Indicators Properties window will show the parameters for th parameters and their order must correspond with the declaration order and the type of ext

Parameters	indicator.They must all be separated by a comma.
Mode	Line index. Can be from 0 to 7 and must correspond with the index used by one of the SetI
Shift	Index of the value taken from the indicator buffer (shift relative to the current bar the giver

Of the six parameters above, the easiest to figure out are the first three and last. We have already discussed symbol and timeframe in previous articles. The name of the indicator is also rather easy: it is the name of the indicator as it appears in the Custom Indicator's list in the Navigator window, surrounded by quotation markets. Example: If you see that the indicator is called NonLagMA_v7.1 in the window, you will write it as "NonLagMA_v7.1" in the indicator name parameter.

We have also already discussed in previous articles the last parameter, the Shfit, but here is a recap. Shift indicates the bar that the indicator is being calculated upon. Bars are numbered 0, 1, 2, 3, 4, 5....as you go back in time on the chart. Changing the shift option is similar to moving the indicator line into the future or the past. Shift of 1 obtains the value of the previous bar. Shift of 0 obtains the value of the current bar.

In Sum:

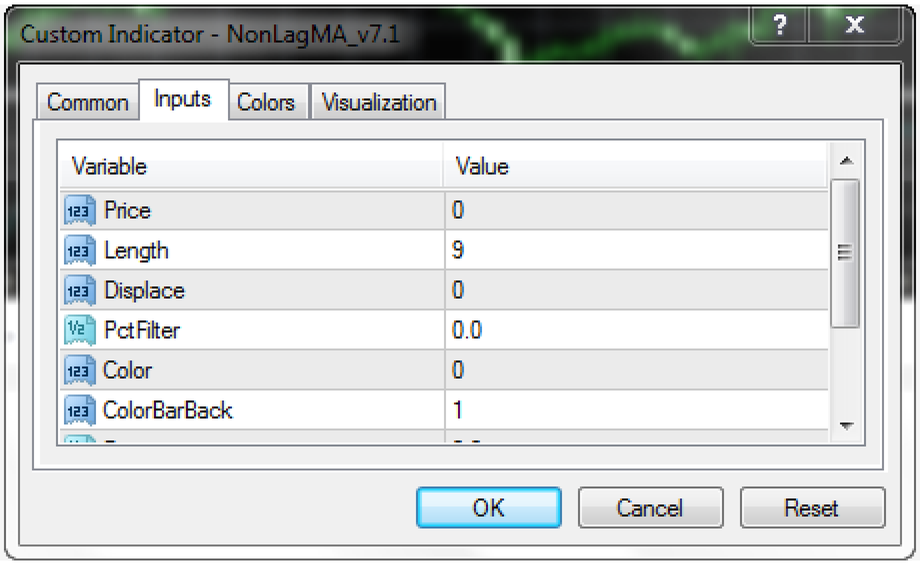
- shift = 0 to get the indicator's value of the current bar (still forming)
- shift = 1 value on the previous bar
- shift = 2 value in a bar before the previous bar

The third parameter, Indicator Parameters, and the fourth parameter, Mode, can be tricky to discover and implement within an EA.

Discovering The Indicator Parameters

First, let us examine how to discover the custom *Indicator Parameters*. Most of the time the custom indicator might use 1 or 2 indicator parameters, but there are some that use 3 or more, making the discovery process more difficult. The custom indicator NonLagMA_7.1 is an example of a more complex indicator that uses 9 indicator parameters. Note: Parameters are all inputs that change the value of the indicators; they are the variables that you play with to get the best set up for your custom indicator. You can download the NonLagMA_7.1 here.

If you go to the Inputs tab in the Custom Indicator Properties window, it will show you the parameters for the NonLagMA_v7.1 custom indicator:



Variable	Value
Price	0
Length	9
Displace	0
PctFilter	0.0
Color	0
ColorBarBack	1

Though you can see 6 inputs in the above picture, there are actually 9 inputs or indicator parameters. There are 9 indicator parameters with different inputs you can change the value of to get the best setup, one of which is essential to the formation of the indicator (Length).

An even better method of discovering the indicator parameters is to copy and paste them from the source code. If you right click on the custom indicator and click *Modify*, you can see the source code of the indicator in the MetaEditor. If the indicator is grayed out it means you do not have the mq4 code, just the ex4, and you have to figure out a way to decompile it, or you can deduce the parameters from the properties window using the method above. If you have the mq4 code and can open it, you are in luck. Now, check the extern variables at the beginning of the indicator source code. The indicator parameters, data types and default values, will be listed here, and you can simply copy and paste the extern code to the extern variables section of your expert advisor.

With the NonLagMA indicator, you will see the following extern variables:

```
//---- input parameters
extern int Price = 0;
extern int Length = 9; //Period of NonLagMA
extern int Displace = 0; //DispLace or Shift
extern double PctFilter = 0; //Dynamic filter in decimal
extern int Color = 1; //Switch of Color mode (1-color)
extern int ColorBarBack = 1; //Bar back for color mode
extern double Deviation = 0; //Up/down deviation
extern int AlertMode = 0; //Sound Alert switch (0-off,1-on)
extern int WarningMode = 0; //Sound Warning switch(0-off,1-on)
```

Bear in mind that though there are 9 parameters, only one, Length, will change the essential nature of the indicator. The others can be left at their default values, which in most cases other than Color, is 0. If we also put in 0 as the default for the last two parameters, mode and shift, we can construct our iCustom

NonLagMA indicator with the above identifiers:

```
iCustom(NULL, 0, "NonLagMA_v7.1",Price,Length,Displace, PctFilter, Color, ColorBarBack,
Deviation, AlertMode, WarningMode, 0,0).
```

For the purposes of optimization, only one of the nine parameters above, Length, is the one that you will change or optimize in order to find the best NonLagMA for your currency symbol and timeframe. The others will remain at their default values. It is often the case that if you see length as a parameter in your custom indicator, it will be the one that will become the most essential to the formation of your strategy, and thus most essential for further manipulation. The above custom indicator should compile effectively, but in order to put it in a strategy you must know how to work with the last two parameters, *mode* and *shift*.

Discovering The Mode (Line /Signals)

Mode is a line index that ranges from 0 to 7. MT4 allows up to 8 indicator lines (buffers) per custom indicator. These lines (buffers) must correspond with the index used by one of SetIndexBuffer lines from within the code of the indicator.

It is helpful to visual this: Your indicator has up to 8 lines/signals (or arrows) with different colors. Each line gives you an output, and each line has a value. With mode you select the line (signal) that you need for your EA (just one out of eight). Most of the time, indicators have only one line/signal so mode=0 should be used.

Note on Mode: Because MT4's index goes from 0 to 7, mode takes values from 0 to 7. So the first signal has a mode=0 (and not 1). If your indicator has three indicator lines/signals and you want to get the value for the first line, you use mode=0, the value for the second line/signal is obtained with mode=1, and mode=2 will get the value for the third signal.

Ok, then, how do you identify the mode to use for your EA?

The mystery of the mode lies within the Indicator Buffers.

Basically, you have to look for the buffers in the code and these will become the lines of your mode. Lets say that you have 3 buffers:

```
iCustom(Symbol(),Timefram e,"CUSOMT INDI NAME",all parameters separated by coma,
BUFFER, SHIFT);
```

If you wish to have values of 3 buffers of current bar you will write :

```
iCustom(Symbol(),Timefram e,"CUSOMT INDI NAME",all parameters separated by coma, 0,
0);
iCustom(Symbol(),Timefram e,"CUSOMT INDI NAME",all parameters separated by coma, 1,
0);
iCustom(Symbol(),Timefram e,"CUSOMT INDI NAME",all parameters separated by coma, 2, 0);
```

Ok, fine, let us see an example of this buffer syntax and where it is located in the code.

Just under the initialization function of the NonLagMA indicator, you will see a number of indicator buffers that look like this:

```
IndicatorBuffers(6);
SetIndexStyle(0,DRAW_LINE);
SetIndexBuffer(0,MABuffer);
SetIndexStyle(1,DRAW_LINE);
SetIndexBuffer(1,UpBuffer);
SetIndexStyle(2,DRAW_LINE);
SetIndexBuffer(2,DnBuffer);
SetIndexBuffer(3,trend);
SetIndexBuffer(4,Del);
SetIndexBuffer(5,AvgDel);
string short_name;
```

SetIndexStyle tells us that there are three lines (DRAW_LINE). Underneath, **SetIndexBuffer** tells us the mode number and name of those three lines. It is these lines that begin with SetIndexBuffer that are the key to finding your mode. Specifically, the contents in parenthesis indicate the number of the mode and its name (0=MA, 1=UP, 2=Dn). There is a fourth signal (mode 3, trend) that does not have a line and is called trend. You can ignore the two buffers relating to delete (4,Del) and (5, AvgDel) as they are not signals.

Here are the 4 lines (buffers) and their signal names put into a table for you to see:

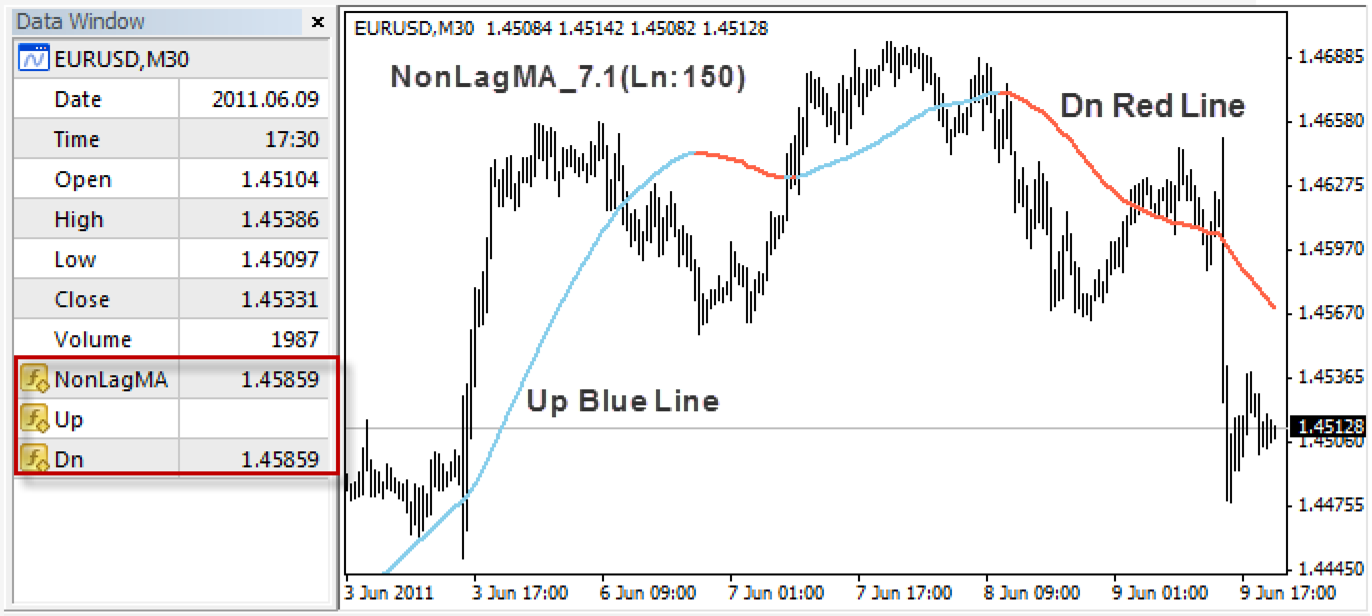
Modes	**Name**
0	MA Signal
1	Up Signal

2	Down Signal
2	Trend Signal

Now that we know the number and names of the lines(signals) of our mode parameter, how do we find out what they do?

One way, of course, is to have a look at the source code. However, as we have not as yet discussed the construction of indicators, I am going to assume that the code appears Greek to you.

The other method is to visually deduce the function of the lines /signals. You need to apply your indicator to the chart and open up the data window (View /Data Window or Ctr +D). At the bottom of your data window you will see many of the signals pertaining to your indicator, along with their corresponding data values, depending on the bar your mouse is hovering over.


Data Window

EURUSD,M30

Date	2011.06.09
Time	17:30
Open	1.45104
High	1.45386
Low	1.45097
Close	1.45331
Volume	1987
NonLagMA	1.45859
Up	
Dn	1.45859

The data table that I highlighted in red provides the clue for deducing the function of the modes (lines / signals). The NonLagMA is the first mode (mode = 0), and we can deduce that it is the line itself. If you hover your mouse over any bar it gives you the data reading of the NonLagMA row for that bar. **Up** is the second mode (mode =1) and we can deduce that it is the blue line. If you hover your mouse over any bar that has a blue line underneath it, you will see that it gives you a data reading for the **Up**row and no data reading for the Dn row. **Dn** is the third mode (mode = 2), and we can deduce that it is the red line. If you hover your mouse over any bar that a red line over it, you will see that it gives you a data reading for the **Dn** row and no data reading for the **Up** row.

The fourth mode of Trend (mode = 3) does not appear in the data window, though I know from looking at the code that reads like this: if current trend signal == 1, the trend is long, and if current trend signal == -1, the trend is short.

Now what you roughly know the four modes of the NonLagMA custom indicator, you can create at least three types of EA conditions:

1. EA that works with the MA line itself, as in a crossover (Mode=0);
2. EA that works with the Up (Blue) and Dn (Red) signals, taking buys when the NonLagMA turns blue and taking sells when it turns red (Mode=1 and Mode=2).
3. EA that works with the Trend signal, buy when trend is up, sell when trend is down (Mode=3).

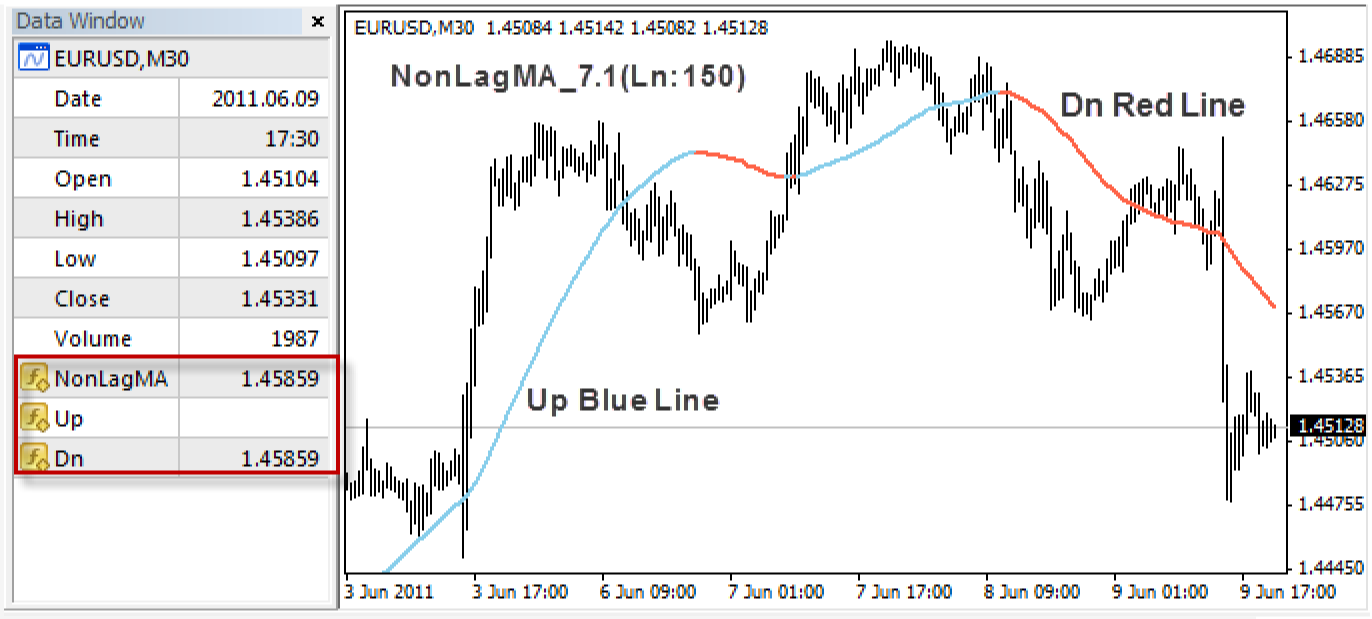
We will examine each of these EA combinations in the next article: Building Strategies with Custom Indicators and the iCustom function (Example: NonLagMA)

Building Strategies With Custom Indicators And The ICustom Function (Example: NonLagMA)

In the previous article, we walked you through the process of converting a custom indicator into an iCustom function as a necessary first step before it can be deployed in an expert advisor. We defined the 6 different parameters of an iCustom function, paying particular attention to the last three: Indicator Parameters, Mode, and Shift. We used the example of converting a complicated custom indicator, the NonLagMA_7.1, into an iCustom indicator function, and we walked you through the deductive discovery process of finding its Indicator Parameters and Mode. In this article we will continue to use the example of the NonLagMA_7.1, and explore the different ways of weaving the iCustom fuction into different expert advisors.

If you are not familiar with the 6 parameters of the iCustom function, I suggest you go back and read the the article Preparing a Custom Indicator for Strategy Deployment: Converting it into an iCustom Function.

If you have forgotten or have not seen what a NonLagMA custom indicator looks like, here is a screenshot:



Data Window

EURUSD,M30

Date	2011.06.09
Time	17:30
Open	1.45104
High	1.45386
Low	1.45097
Close	1.45331
Volume	1987
NonLagMA	1.45859
Up	
Dn	1.45859

Previously, when discussing the detective work in discovering the mode parameter (lines/signals) of the iCustom function, I pointed out that that the data table highlighted in red provides the visual clues. We deduced that the NonLagMA is the first mode (mode=0) represented by the line itself, Up is the second mode (mode =1) represented by the the blue line, and Dn is the third mode (mode = 2) represented by the red line. Not visually represented here is the fourth mode of Trend (mode = 3), but it is represented in the code itself, with the logic being that if trend == 1, trend is up, if trend = -1, trend is down.

Now what you roughly know the four modes of the NonLagMA custom indicator, you can create at least three types of expert advisors based on the different modes:

1. EA that works with the MA line itself (Mode=0), as in a crossover.
2. EA that works with the Up Blue Line (Mode=1) and Dn Red Line (Mode=2), taking buys when the NonLagMA turns blue and taking sells when it turns red.
3. EA that works with the Trend signal (Mode=3), buy when trend is up, sell when trend is down.

All three EAs will share same extern variables below:

```
extern int Price = 0;
extern int Length = 150;
extern int Displace = 0; //DispLace or Shift
extern double PctFilter = 0; //Dynamic filter in decimal
extern int Color = 1; //Switch of Color mode (1-color)
extern int ColorBarBack = 1; //Bar back for color mode
extern double Deviation = 0; //Up/down deviation
extern int AlertMode = 0; //Sound Alert switch (0-off,1-on)
extern int WarningMode = 0; //Sound Warning switch(0-off,1-on)
extern int Shift = 1;
```

I went over the process of discovering the above indicator parameters in the previous article, but I will recap. The easiest way was to copy and paste the extern variables from the indicator source code over to the expert advisor, and then to populate the iCustom function indicator with the above identifiers, letting them fall AFTER the third parameter (the indicator name) and BEFORE the last two parameters (the mode and shift).

Thus, you would be converting your iCustom indicator from this:

```
double ma = iCustom (NULL,0,"NonLagMA_v7.1",0,0,Shift);
```

to this:

```
double ma = iCustom (NULL, 0, "NonLagMA_v7.1",Price,Length,Displace, PctFilter, Color,
ColorBarBack, Deviation, AlertMode, WarningMode, 0,Shift).
```

Note: While this particular indicator has 9 indicator parameters, it should be pointed out that the most important one is the Length. Just like the moving average, changing the length changes the fundamental structure of the NonLagMA. The other parameters can be left at the default value of 0.

NonLagMA EA #1: Working With The MA Line Signal (Mode = 0)

	NonLagMA Crossover
Intent	//Buy: when close crosses over NonLagMA //Sell: when close crosses under NonLagMA

Indicator Calling	double macurrent = iCustom(NULL, 0, "NonLagMA_v7.1",Price,Length,Displace, PctFilter, Color, Colo
	WarningMode, 0,Shift).
	
	double maprevious = iCustom(NULL, 0, "NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Color, Co
	WarningMode, 0,Shift+1).
	
	double close_current = iClose (NULL,0,Shift);
	double close_previous = iClose (NULL,0,Shift+1);

BuyCond	if (close_current >= macurrent && close_previous <= maprevious)
SellCond	if (close_current <= macurrent && close_previous >= maprevious)

Explanation:

There is not much to explain here. We are using this NonLagMA crossover in much the same way as we conducted a MA crossover. One could represent a fast NonLagMA crossing over/under a slow NonLagMA, but for illustration purposes, I just represented the current close crossing under/over the NonLagMA. The end result will be more or less similar.

In indicator calling, notice that mode parameter of my iCustom indicator functions (the second from last), is set to 0. As we have seen above, 0 refers to the NonLagMA line itself.

Notice that when I represent current close and previous close I use the iClose function. The iClose function is useful if you wanted to work with the closing price of a different time frame, currency symbol, or shift value. If you worked with Close or Bid, you are referring to only current price of the current symbol. In my case, I want to work with a different shift value, the current close (Shift) and the previous close (Shift+1), and so I needed to construct a variable called close_current = iClose (NULL,0,Shift) and a variable called close_previous = iClose (NULL,0,Shift+1).

NonLagMA EA #2: Blue UP And Red Dn Color Change (Mode 1 And 3)

	NonLagMA Line Color Change
Intent	//Buy: when current line is up and blue and previous line is down and red //Sell: when current line is down and red and previous line is up and blue
Indicator Calling	double buy_macurrent = iCustom(NULL, 0, "NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Color, WarningMode, 1,Shift). double buy_maprevious = iCustom(NULL, 0, "NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Colo AlertMode, WarningMode, 1,Shift+1).

	double sell_macurrent = iCustom(NULL, 0, "NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Color, WarningMode, 2,Shift). double sell_maprevious = iCustom(NULL, 0, "NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Colo AlertMode, WarningMode, 2,Shift+1).
BuyCond	if (sell_maprevious!=EMPTY_VALUE && sell_macurrent==EMPTY_VALUE && buy_macurrent!=EMPT
SellCond	if(buy_maprevious!=EMPTY_VALUE && buy_macurrrent==EMPTY_VALUE && sell_macurrent!=EMPT

Explanation:

In the indicator calling section, you will see that created variables for a buy_ma (the blue line) and sell_ma (the red line), and made a current and previous version of each by setting a different in the last parameter, the Shift: current (Shift) and previous (Shift+1). What distinguishes the buy_ma versus the sell_ma is the second to last parameter, the mode parameter. The buy_ma has a mode of 1, which as we have discovered, is the blue (UP) line, and the sell_ma has a mode of 2, which as we have discovered, is the red (Dn) line.

When you see the entry conditions, the first thing that might strike you as odd is these EMPTY_VALUE words and exclamation marks. You would be scratching your head wondering what I am doing with all that EMPTY_VALUE.

Well, internally, in geek-speak, EMPTY_VALUE refers to a function:

```
void SetIndexEmptyValue( int index, double value)
```

This function sets the drawing line to an empty value. Empty values are not drawn or shown in the DataWindow. By default, empty value is EMPTY_VALUE.

In plain English, EMPTY_VALUE is when the data window shows nothing for that Line / Signal/ Arrow. In the case of the NonLagMA, whenever the line is blue, the UP row in the data window shows the indicator data value, and at the same time the Dn row in the data window shows nothing, empty value. Conversely, whenever the line is red, the Dn row in the data window shows the indicator data value, and at the same time the Up row shows nothing, empty value.

Now, when we say that buy_macurrent = EMPTY_VALUE, we are trying to say that we do not want the blue line at all (we want it to be empty of value). If we say buy_macurrent! = EMPTY_VALUE, the exclamation mark (!) indicates "DO NOT WANT"; thus, we do not want the blue line to be empty of value, or, to put it another way, we want the blue line to be represented, to have value.

If we look at the buy condition in plain English, we are saying: we are going to buy when the previous red line did exist (sell_maprevious!=EMPTY_VALUE) and now it does not (sell_macurrent=EMPTY_VALUE) and instead the blue line exists (buy_macurrent!=EMPTY_VALUE).

Conversely, with the sell condition we are saying: we are going to buy when the previous blue line did exist (buy_maprevious!=EMPTY_VALUE) and now it does not (buy_macurrent=EMPTY_VALUE) and instead the red line exists (sell_macurrent!=EMPTY_VALUE).

NonLagMA EA #3: Trend Change (Mode 4)

	NonLagMA Trend
Intent	//Buy: when current trend is up and previous trend is down //Sell: when current trend is up and previous trend is down
Indicator Calling	double trend_macurrent = iCustom(NULL, 0,"NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Colo AlertMode, WarningMode, 3,Shift). double trend_maprevious = iCustom(NULL, 0, NonLagMA_v7.1",Price,FastLen,Displace, PctFilter, Col AlertMode, WarningMode, 3,Shift+1).
BuyCond	if (trend_macurrent==1 && trend_maprevious==-1)
SellCond	if (trend_macurrent==-1 && trend_maprevious==1)

Explanation

The difference between the trend_ma and the previous types versions in the above two EAs is the second to last parameter, the mode. The mode is 3, which as we have discovered, refers to the trend signal. The trend signal does not appear in the data window, like the previous modes, but it does appear in the code of the indicator:

```
trend[shift]=trend[shift+1];
if (MABuffer[shift]-MABuffer[shift+1] > Filter) trend[shift]= 1;
if (MABuffer[shift+1]-MABuffer[shift] > Filter) trend[shift]=-1;
```

The code calculates how the trend is formed. In plain English, if trend = 1, then the trend is long, and if the trend = -1, the trend is short.

To put the above into an entry condition is very simple. For a buy condition, I just need to say that I wanted the current trend to be up (trend_current==1) and previous trend to be down (trend_previous==-1). For sell condition, I just needed to say that I wanted the current trend to be down (trend_current==-1) and the previous trend to be up (trend_previous==1).

Using Trend Filters To Enhance Strategies

Many intraday trend systems can be enhanced with a separate trend filter. This is an indicator that looks at recent price action and determines whether the overall trend is up, down, or neutral.

A trend filter is a way for the system to trade in the direction of the overall trend, filtering out trades that are contrary to the trend with the hope (proven from back testing) that the trend filter is filtering out more losing trades than winning trades. We system traders often find or create an interesting trending system on a smaller time frame chart like the M30, and we can see through back testing that the system can profitably exploit many trend opportunities on this relatively smaller time frame. However, we may also see from visual back testing that the system, though ultimately profitable, takes a lot of unnecessary losses whenever it trades contrary to the larger trend. When we see this, it behooves us to look for a trend filter of the same or alternate indicator, set on a larger time frame or period, that can help the system identify the lager trend of the market and only allow trades to occur in the direction of that larger trend. It helps if we also code in a true / false bool for that trend filter, so that we can easily back test with the filter on or off to see if it is enhanced or not by its addition.

We will look at the construction of a simple trend filter, the Moving Average Trend Filter, in order to give you an idea of the process of creating and implementing a trend filter. Once you know how to code up this one trend filter, you can easily code up many alternative ones based on different indicators, until you finally discover the right set of trending filters to test upon all your trending strategies.

Moving Average Trend Filter

We are going to code up a moving average filter that when set to true only trades longs when the closing price is above the moving average and only trades shorts when the closing price is below the moving average. Simple enough.

Long Positions: When close is above moving average.
Short Positions: When close is below moving average.

Parameters

MAFilter

Bool: Indicate whether or not you want the Moving Average Filter on or off. Trades from primary entry system are only taken in the direction of the moving average: only long when close is above moving average, only short when close is below. Default is false.

MAFilterRev

Bool: Indicate whether or not you want the Moving Average Filter Reversed on or off. Trades from primary entry system are only taken in the opposite direction of the moving average: only long when close is below moving average, only short when close is above. Default is false. Though this reversed filter is not often used, it can be useful if you discovered through back testing that applying the MAFilter above severely diminished your strategy. You may then want to see if applying the MAFilterRev can then super charge it.

MATime

Int: This is the time frame of your moving average. Default is 0, which means same time frame of the chart. You should test the same time frame first, and then move upwards in your tests, testing on each larger time frame. For instance, if your primary strategy was set on a M30 chart, you would test your Moving Average with the default of 0, which would be your M30, then you would test it with 60, which would be H1, then 240, which would be H4, and 14440, which would be D1.

MAPeriod

Int: This is the period of your moving average. Default is 50, which is a common longer period for the Moving Average to determine trend direction. Another common longer period is 200. I would advise setting an optimization between 50 to 200, with a step interval of 25 or 50, to see which of the longer MA periods can best identify the trend for that currency on each of the time frames you test upon.

MAMethod

Int: This is the method or mode of the moving average, which is defaulted to 1, the exponential moving average. Remember: in the method parameter, 0 = Simple, 1 = Exponential, 2 = Smoothed, 3 = Linear Weighted. I think 1 or exponential is the best type of moving average. The other good one to test with is

0, the simple moving average. Both are very popular and effective modes of the moving average.

MT4 Code Snippets

The block of code below is placed in the external variables section.

```
extern bool MAFilter=true;
extern bool MAFilterRev = false;
extern int MATime = 0;
extern int MAPeriod = 50;
extern int MAMethod=1;
```

The block of code below is placed in the indicator calling section.

```
if(mafilter1 || mafilter1rev)
double mafilter=iMA(NULL,MATime,MAPeriod,0,MAMethod,PRICE_CLOSE,shift);
```

MT4 Code Usage Example

Below is an example of a MAFilter being interwoven into the code of the Moving Average Cross strategy. We earlier detailed how to construct a moving average cross strategy, so it should be familiar. One might imagine that a MAFilter added to a moving average cross tehnique would be redundant; however, there is a case to be made for how it can act as a compliment. For instance, if the moving average cross were to take trades based on period of 30 on a M30 time frame, it may be enhanced if it were complimented with a 200 period MAFilter on a H4 time frame. It may benefit by taking trades on the trends of the shorter time frame and period that are ultimately in the direction of the longer time frame and period.

```
bool BuyCondition = false;
bool SellCondition = false;

if (
FastMACurrent > SlowMACurrent && FastMAPrevious < SlowMAPrevious){
BuyCondition=true;
if (OppositeClose) CloseSell=true;
}

if (
FastMACurrent < SlowMACurrent && FastMAPrevious > SlowMAPrevious){
SellCondition=true;
if (OppositeClose) CloseBuy=true;
}

if (BuyCondition
&&
(MAFilter==false || (MAFilter && Ask>mafilter))
&&
(MAFilterRev==false || (MAFilter && Ask< mafilter))
OpenBuy = true;

if (SellCondition
&&
(MAFilter==false || (MAFilter && Bid<mafilter))
 < mafilter))
&&
(MAFilterRev==false || (MAFilter && Bid> mafilter))
OpenSell = true;
</mafilter))
```

Explanation

In the MT4 usage section above, there are 5 blocks of code. In the first block of code I create a bool for BuyCondition and a bool for SellCondition, and default them both to false. I set them to false initially because I only want them to true when the logic I subscribe to them becomes true. The second and third blocks of code set out to identify the buy and sell conditions based on the moving average crossover. When the fast moving average crosses over slow moving average, then BuyCondition=true; when fast moving average crosses under slow moving average, then SellCondition=true. We have already gone over these conditions in the Basic Expert Advisor: MA cross. The only difference is that I have subsumed these conditions into the bools of BuyCondition and SellCondition instead of the bools of OpenBuy and OpenSell, which I make reference to in the fourth and fifth blocks of code.

It is the fourth and fifth blocks of code that implement the MAFilter. Each one starts with an "if ( )" condition that has three conditions set within the brackets. The first condition, the BuyCondition or SellCondition, subsumes the entry logic of the moving average crossover.

The second condition, the MAFilter condition, is a compound statement within parenthesis separated by a || ("or") operator. The first part of the statement indicates that if MAFilter is set to false (MAFilter==false), then the MAFilter is not operational. The second part of the statement (after the ||) indicates that if the MAFilter is set to true (notice it does not have to say == true, for the bool by itself means true), then it can proceed to the MAFilter rule. Under the BuyCondition, the MAFilter rule is that Ask must be greater than

(>) mafilter. **Note:** the variable for mafilter (in lower case) is defined in the MT4 snippet placed in the indicator calling section.

The third condition, the MAFilterRev condition, is also a compound statement within parenthesis separated by a || ("or") operator. The first part of the statement indicates that if MAFilterRev is set to false (MAFilterRev==false), then the MAFilterRev is not operational. The second part of the statement (after the ||) indicates that if the MAFilterRev is set to true, then it can proceed to the MAFilterRev rule. As you can see the MAFilterRev rule is the opposite of the MAFilter rule. If the MAFilter rule indicated that the Ask must be greater than (>) the mafilter, then the MAFilterRev indicates that the Ask must be less than (<) the mafilter.

If all three of the above conditions are met, then the EA can proceed to OpenBuy = True or OpenSell=true, which triggers the buy or sell entry signal in my code.

Conclusion

The above MAFilter is just one of one many possible trend filters one can construct and implement. I have chosen the MAFilter because it is perhaps one of the simplest and most effective for identifying trend direction.

Other good indicators that can act as trend filters are variations of the moving average: JMA, Hull, NonLagMA, and Slope Directional Line. I have used all four indicators as reliable trend filters. One can also experiment with momentum oscillators like RSI and Stochastics. With whatever indicator used, one should experiment with different periods and time frames to see which ones can best filter for the larger trend. Trend is only relevant in conjunction to the time frame, and once you identify the time frame, there is nothing fancy about the concept of trend.

It is not necessary to try to find or construct super fancy, mathematical trend filters. Many traders attempt to over complicate the problem of trend identification. They invent all kinds of fancy mathematical equations and methods of massaging past price action to more precisely determine whether the trend is up or down, and most attempts at such are pointless. The simplest and most popular approaches to trend identification are often the best.

Not every EA can benefit from having a trend filter. Some EAs incorporate trending indicators that work well enough without additional trending filters. Other EAs try to predict trend reversals or counter trend conditions, and therefor enter the trade when the trend is against them at the time.

Bear in mind that all filters prevent trades from occurring and we want the trades prevented to be be mostly losing ones. It is often best to back test one filter at a time and to try not to use too many. Your strategy should be good enough to stand on its own, without the need for a filter. The trending filter applied to a good strategy should be tested out as an optional enhancement. If we discover that the filter is preventing a good number, or even an equal number of winning trades, it is not a good filter to have working for you. It is only acceptable if it can prevent a greater percentage of losing trades under a lengthy 5-10 year back test.

Money Management: Lot Sizing

Introduction

You may have already heard this, but choosing appropriate lot sizing for your trading system is a key ingredient for developing a good system. You can specify a lot size as simply as declaring one in an internal variable as a fixed lot size for every order, but we will explore a simple method that calculates the lot size based on a percentage of your free margin.

There is a little math behind the scenes, but basically, if you choose a custom risk setting of 1, you will trade 0.01 micro lot for every 1K in equity size. Thus, with a custom risk setting of 2 and a 10K account size, you will be starting with 0.2 lots, and it will automatically add/subtract 0.01 lot for every $100 in profit/loss. This auto lot sizing technique is as simple as it gets, but very effective for automatically compounding your profits, de-compounding your losses, or automatically adjusting for deposits and withdrawals in the account.

Parameters

MM

Bool: Whether or not you will use money management.

Risk

Double: Your predefined risk setting.

Lots

Double: If MM is turned off, this is the manual lot size you will use.

LotDigits

Double: This is the number of decimal places for the lots provided by your broker. Most have two decimal places, but some have one.

MT4 Code Snippets

```
extern bool MM = TRUE;
extern double Risk = 2;
extern double Lots = 0.1;
extern double LotDigits =2;




double GetLots()
{
double minlot = MarketInfo(Symbol(), MODE_MINLOT);
double maxlot = MarketInfo(Symbol(), MODE_MAXLOT);
double leverage = AccountLeverage();
double lotsize = MarketInfo(Symbol(), MODE_LOTSIZE);
double stoplevel = MarketInfo(Symbol(), MODE_STOPLEVEL);

double MinLots = 0.01; double MaximalLots = 50.0;

if(MM)
{
double lots = Lots;

double lots = NormalizeDouble(AccountFreeMargin() * Risk/100 / 1000.0, LotDigits);
if(lots < minlot) lots = minlot;
if (lots > MaximalLots) lots = MaximalLots;
if (AccountFreeMargin() < Ask * lots * lotsize / leverage) {
Print("We have no money. Lots = ", lots, " , Free Margin = ", AccountFreeMargin());
Comment("We have no money. Lots = ", lots, " , Free Margin = ", AccountFreeMargin());
}}
else lots=NormalizeDouble(Lots,Digits);
return(lots);
}
```

You will see that we first had to declare a number of extern variables to dermine if we should have turn the management on (true) or off (false), what is going to be our custom risk setting if on, and if not, what is going to be the default lot size.

LotDigits is how many decimal places your broker allows for (for instance, if it allows for micro lots, such as 0.01, it would have 2 digits or decimal places).

GetLots() is the name we have given to our custom function So all of the o(it could have been any name), and all that is subsumed between its brackets is a calculation of this function. You will simply place **GetLots()** in the third parameter of the **OrderSend()** function in order to call upon it, replacing the fixed lots variable that was there before.

We create a variable **minlot** to reference the **MarketInfo()** function. The **MarketInfo()** function is the function we need to retrieve various market data of the given currency, such as the Bid or Ask price, the Swap value, the number of digits, and for our purposes, it can also tell us the minimum lot size for that currency. We want to make sure that whatever lot calculation is conducted, it is greater than the min lot size of the broker, else it is less than minlot, it will be the **minlot**.

The main calculation of the automatic MM lot happens in one line:

```
double lots = NormalizeDouble(AccountEquity() * Risk/100 / 1000.0, LotDigits);
```

AccountEquity() is one of many account information functions that returns the equity value of the current account. We want to return the equity value of the account, as opposed to the **AccountBalance()**, because equity represents a more valid picture of the state of the account (aka, the net account value). We want the equity value to conduct our math on appropriate lot sizing. We are going to multiply this equity value with our risk value, then divide by 100, and then further divide by 1000, in order to determine the appropriate lot size.

The effect is proportional lot sizing, based on the risk setting chosen: it makes a risk setting of 1 trade 0.01 lots per 1K in equity, a risk setting of 2 trade 0.02 lots per 1K in equity, etc. There is a host of possibilities, depending on the risk setting chosen. Lots are added or subtracted to the account as it grows or diminishes in size. For instance, a risk setting of 2 would trade 0.2 lots on a 10K account , and add / subtract by 0.01 lot for every $100 gain or loss in equity. The user can easily adjust a risk setting that is suitable to his or her risk tolerance, EA trading style, and account size.

If **MM** is set to true, we will calculate the lot size based on the equity, and assign that value to the **lots** variable. If MM is false, we simply assign the value of **lots** to the fixed lot size of **Lots**.

You can see that the above code is relatively simple, but it can make a world of difference in auto lot sizing based on a changing equity size. There are more complicated ways in determining lot sizing, but sometimes the simplest methods work best.

Note:The EA developer or end-user must determine the appropriate risk setting for the EA based on a rigorous back test, paying close attention to risk related stats (average losing trade, consecutive losing trades, and max draw down). Usually this risk analysis is first done with a fixed lot size, such as 0.1 for a

5K account. Once all the risk-related stats are compiled for a fixed lot size, one can better determine the risk setting one can be comfortable trading with.

Trailing Stop With Profit Threshold

There might be many different types of trailing stop possibilities out there, but for our purposes, we will be examining the most effective: the trailing stop that will activate after a custom threshold and then trail by a custom level of pips.

This trailing stop moves the stop loss up or down with the order price after it first reaches profit threshold, thereby "locking in" profit and providing loss protection.

For instance, if you set the profit threshold to 50 and the trailing stop to 50, the trade will have to move 50 pips in profit before the trailing stop is activated, at which point the stop will be moved to break even. If you further indicated that your trailing step is 5, then the market would have to move up 5 pips more in profit before your stop would rise 5 pips above breakeven. It would then adjust increasingly upwards on that basis, locking in 5 more pips of profit for each additional 5 pip gain.

We will thus need three external variables for this new trailing stop, one indicating the profit threshold, the other indicating the trailing stop, and the last indicating the stepping distance:

```
extern double TrailingStart = 50;
extern double TrailingStop = 25;
extern double TrailingStep = 5;
```

The variables that the user manipulates are thus the **TrailingStart**, which is the profit threshold that needs to be achieved before the trailing stop can be activated; the**TrailingStop**, which is the amount of pips we will trail from new profit highs; and the **TrailingStep**, the amount of new pips that need to be gained before the stop can be increased by the amount of the gain.

Let us examine the code for this trailing stop:

```
void TrailOrder(double Trailingstart,double Trailingstop){
int ticket = 0;
double tStopLoss = NormalizeDouble(OrderStopLoss(), Digits); // Stop Loss
int cnt,vPoint,vSlippage;

double sl     = OrderStopLoss(); // Stop Loss

  if (Digits == 3 || Digits == 5)
    {vPoint = Point * 10; vSlippage = Slippage * 10;    }
    else
    {vPoint = Point; vSlippage = Slippage;}



RefreshRates();
if(OrdersTotal()>0){
for(cnt=OrdersTotal();cnt>=0;cnt--){
OrderSelect(cnt,SELECT_BY_POS,MODE_TRADES);
if(OrderType()<=OP_SELL && OrderSymbol()==Symbol()
&& OrderMagicNumber()==MagicNumber){

if(OrderType()==OP_BUY){
if(Ask> NormalizeDouble(OrderOpenPrice()+TrailingStart* vPoint,Digits)
 && tStopLoss < NormalizeDouble(Bid-(TrailingStop+TrailingStep)*vPoint,Digits)){
tStopLoss = NormalizeDouble(Bid-TrailingStop*vPoint,Digits);
ticket =
OrderModify(OrderTicket(),OrderOpenPrice(),tStopLoss,OrderTakeProfit(),0,Blue);
if (ticket > 0){
Print ("TrailingStop #2 Activated: ", OrderSymbol(), ": SL", tStopLoss, ": Bid", Bid);
return(0);
}}}

if (OrderType()==OP_SELL) {
if (Bid < NormalizeDouble(OrderOpenPrice()-TrailingStart*vPoint,Digits)
&& (sl >(NormalizeDouble(Ask+(TrailingStop+TrailingStep)*vPoint,Digits)))
|| (OrderStopLoss()==0)){
tStopLoss = NormalizeDouble(Ask+TrailingStop*vPoint,Digits);
ticket = OrderModify(OrderTicket(),OrderOpenPrice(),tStopLoss,OrderTakeProfit(),0,Red);
if (ticket > 0){
Print ("Trailing #2 Activated: ", OrderSymbol(), ": SL ",tStopLoss, ": Ask ", Ask);
return(0);
}}}
}}}}
```

Explanation

We are using void TrailOrder() as our custom function.

You will notice the **for** loop and the **OrderSelect()** function from our order loop code. It will loop through the order pool and examine each order to see if we need to apply a trailing stop to it. If the current market order is a buy market order, as indicated by OP_BUY, and if it matches our chart symbol and magic number, we will proceed with the trailing stop calculation.

There are two conditions that must be determined prior to activating the two trailing stop:

Condition1: Check for **TrailingStart**. We are asking the program to watch if the ask price is greater than the open price plus the **TrailingStart** value. For instance, if the trailing start was set to 50 pips, then the trailing stop can only be activated *only if* the market reaches 50 pips in profit. It makes much more sense to have the trailing stop be activated after a profit threshold than at the onset.

Condition2: Check for **TrailingStop** and **TrailingStep**. This second condition checks to see if the current stop loss is less than the current price minus the trailing stop and trailing step. For instance, if the trailing start were 50 pips, trailing stop 25 pips and the trailing step 5 pips, then when the trade moves in profit by 50 pips, condition1 is activated, the program can proceed to condition2, checking for TrailingStop and TrailingStep. Condition2 then makes sure that the first stop is adjusted 25 pips below the 50 pips profit threshold. If the trade continues to move in profit, the trailing stop would move up an additional 5 pips for each additional 5 pip increase, as defined**TrailingStep** amount.

We determine the stop loss distance by subtracting our trailing stop setting, multiplied by **vPoint()**, from the current bid price. This is stored in the variable **tStopLoss**. We pass the **tStopLoss** variable as our new stop to the **OrderModify()** function.

Note: We use the MQL function **NormalizeDouble()** to round our variables to the correct number of digits after the decimal point. Prices can be quoted up to eight decimal places,
and **NormalizeDouble()** helps to round that down to 4 or 5 digits (2-3 digits for JPY pairs).

We are adding a **Print** function here to our code to give us a print read-out on our journal tab of when this trailing stop was activated.

In general, the trailing stop conditions for the sell orders follow the same logic as the buy orders.

So now, how does this custom function get inserted into our Start() Function so that our code knows that we are working with it?

We simply put a single line anywhere within the start() function, preferably near the beginning of the function:

```
if(TrailingStop>0 && TrailingStart > 0) TrailOrder (TrailingStart, TrailingStop);
```

Basically, I am creating a conditional **if** expression that checks to see if **TrailingStart** and **TrailingStop** are both greater than 0, that is, if the user has defined integer values for these fields. If so, then the custom function, **TrailOrder()**, will be activated. If not, if both or either field remains at the default of 0, the trailing stop remains deactivated.

That is all there is to it. You now have a sophisticated trailing stop mechanism as an extra weapon in your arsenal.

Auto-Detect/Define Slippage And Point Values For 5 Digit Brokers

For the longest time, when most brokers had currency quotes in 4 digits (or 2 digits for yen pairs), the Slippage value and Point values would work as you as you intended. But with the addition of fractional pip brokers, one needs to make an adjustment to these values, or else there will be a problem.

The **Slippage** value, found in the fourth parameter of the **OrderSend()** function, represents the maximum difference in pips for the order to go through. If your broker is a 4 digit broker, then 1 pip = 1 pip. No problem. If you indicate a 3 pip Slippage, you will be making sure that you are filled within 3 pips of the signal price. However, if your broker uses 5 digit quotes (or 3 for Yen pairs), then 1 point =0.1 pips. Now this is a problem, because if you put in 3 for slippage, it would be 0.3, which is like having 0. You risk the danger of being requoted. In order to make the manual adjustment, you would have to add an additional zero to the end of your Slippage setting, turning your 3 into a 30, in order to remain true to your 3 pip slippage intention. It is thus useful for the code to auto-check for 4 or 5 digit brokers and make the appropriate adjustments automatically.

The **Point** value has a similar problem. As we have seen when we examined the **OrderSend()** function, the pip integers that we indicate for our StopLoss or TakeProfit need to be multiplied by Point in order to convert them to their appropriate fractional value. Point is a predefined variable in MQL that returns the smallest price unit of a currency, depending on the number of decimal places. Thus, Point = 0.0001 for 4 decimal quotes. But recently, many brokers have adopted fractional pip price quotes, with 3 and 5 decimal places, making a Point = 0.00001 for 5 decimal quotes. The problem is that the Stoploss pip value intended as 50 pips would be then be calculated as 5 pips. In order to make the manual adjustment, you would have to add an additional zero to the end of your StopLoss value, changing 50 into 500, in order to remain true to your 50 pip stop loss intention. It is thus useful for the code to auto-check for 4 or 5 digit brokers and make the appropriate adjustments.

Instead of requiring the user to add an additional zero to their slippage, stop loss and take profit settings every time they trade on a fractional pip broker, we can create a code that auto detects the fractional broker and makes the adjustments to both the Slippage and Point automatically.

```
// Global Variables

double vPoint;
int vSlippage;
int init()

// Detect 3/5 digit brokers for Point and Slippage

if (Point == 0.00001)
{ vPoint = 0.0001; vSlippage = Slippage *10;}
else {
if (Point == 0.001)
{ vPoint = 0.01; vSlippage = Slippage *10;}
else vPoint = Point; vSlippage = Slippage;
}
```

You would then be replacing all multiples by Point with vPoint, and inserting vSlippage in the Slippage parameter of the **OrderSend()** function, as in the following example:

```
OrderSend (Symbol(), OP_BUY, Lots, Ask, vSlippage, Bid-StopLoss
*vPoint, Bid+TakeProfit*vPoint, "EAName", MagicNumber, 0, Blue)
```

Enter On Open Bar

Most expert advisors run in real-time, on every tick, which has its plus side and downside. On the plus side, running on every tick allows the EA to catch the smallest movement in price, and this can be great for a scalping system.

However, on the negative side, executing trades in real-time on every tick can make many systems susceptible to false signals.

Sometimes it is better to check trading conditions only once per bar. By waiting for the bar to close, we can be sure that the condition has occurred and that the signal is valid.

Trading once per bar also means that the results in the Strategy Tester will be more accurate and relevant. Due to the inherent limitations of MT4's Strategy Tester, using "every tick" as the testing model will produce unreliable back testing results, due to the fact that ticks are often modeled from M1 data. The trades that occur in live trading will not necessarily correspond to trade made in the Strategy Tester.

But by placing our trades on the close of the bar and using "Open prices only" as the testing mode we can get testing results that more accurately reflect real-time trades. The disadvantage of trading once per bar is that trades may be executed late, especially if there is a lot of price movement over the course of the bar. It's basically a trade-off between responsiveness and reliability.

There are two ways that I know that can check the trade conditions once per bar: 1) time stamp method; and 2) volume method;

1. Time Stamp Method

Here is the code for the time stamp method to check for the opening of the new bar.

```
extern bool EnterOpenBar = true;
int CurrentTime;

int init(){
CurrentTime= Time[0];
return(0);
}
int start(){

if (EnterOpenBar) = true)
{
if(CurrentTime != Time[0]){
// first tick of new bar found
// buy and sell conditions here
CurrentTime= Time[0];
return(0);
}
}
```

Here in this method, we first declare an external variable named **EnterOpenBar** to turn the feature on and off.

In the **init()** function we will assign the time stamp of the current bar to CurrentTime. This will delay the trade condition check until the opening of the next bar.

Next we compare the value of the **CurrentTime** variable to **Time[0]**, which is the time stamp of the current bar. If the two values do not match, then a new bar is detected and we can proceed to open up trade on new bar.

2. Volume Method

Here is the code for the volume method for checking the opening of new bar.

```
extern bool EnterOpenBar = true;

int start ()
bool OpenBar=true;
if(EnterOpenBar) if(iVolume(NULL,0,0)>1) OpenBar=false;

if (OpenBar) {
// first tick of new bar found
// buy and sell conditions here
return (0);
}
```

I prefer the volume method because it contains much shorter pieces of code.

As in the previous method, we first declare an external variable named EnterOpenBar to turn the feature on and off. We do not need anything in the **init()** function.

We just need to put in that one line of code to check for the openbar condition based on volume. This condition checks to see if the volume is 1, in which case OpenBar=true because it has found the first tick of the new bar. If the volume is greater than 1 (>1) OpenBar = false because it is no longer the new bar.

Multi-Session Time Filter With Auto GMT Offset

Introduction

This is multi period allowance trade time filter. It's usable when you only want to trade in Asian Session, and/or London session, and/or New York Session.

Parameters

AutoGMTOffset
Bool: Indicate if you want auto GMT offset turned on or off. Turn off during back testing.

Manual GMT Offset
Double: If AutoGMTOffset is turned off, indicate what your manual GMT offset would be.

How do you determine your manual GMT offset?Click on the following link to find the

GMT. http://www.greenwichmeantime.com.
Find your brokers time by seeing it displayed in your trading platform under Market Watch (the top left hand side of your MetaTrader4 platform). Now do a little math to find out the hourly difference betweem the GMT and your brokers time. That is the offset. If the market watch time is ahead of the GMT then your setting is a positive number. But don't use a plus sign. Just write as you normally would for a positive number. If your broker's time is behind the GMT put a negative sign in front of the number. I'm using Primebank, and its market watch info says 2:00 am when the GMT says 23:00 PM, which means that Primebank is ahead by +3 hours, so I would indicate 3 in the manual offset.

UseTradingHours
Bool: Whether or not to use the time filter. It is defaulted as off.

TradeAsianMarket
Bool: Whether or not to trade Asian session.

StartTime1
Double: Don't Trade before this time. Defaulted at 21:00, the start of Asian Session, if you include Australia. Sydney starts at 21.00, and Tokyo starts at 23.00 GMT.

EndTime1
Double: Don't Trade after this time. Defaulted at 07.00, the start of the European Session. Note: The Tokyo session continues on for 1 more hour to end at 08.00, so if you are interested in trading this session in its entirety, you should indicate from 23:00 to 08.00.

TradeEuropeanMarket
Bool: Whether or not to trade European session.

StartTime2
Double: Don't Trade before this time. Defaulted at 7.00 GMT, the start of the London Session, though Germany does not open till 08.00.

EndTime2
Double: Don't Trade after this time. Defaulted at 12.00 GMT, the start of the New York Session. Note: The European Session continues till 16.00, so if you are interested in trading this session in entirety, you should have your defaults from 7.00 to 16.00.

TradeNewYorkMarket
Bool: Whether or not to trade the New York session.

StartTime3
Double: Don't Trade before this time. Defaulted at 12:00, the start of the New York Session in GMT, which is 8:00 EST. Note that the NY stock exchange does not open till 9:30 EST, or 13.30 GMT, and the first hour and half (from 9:30 to 11:00 EST, or 13:30 to 15:00 GMT) is heavy trading, lots of liquidity.

EndTime3
Double: Don't Trade after this time. Defaulted at 21.00, the end of the New York session, the closing bell in New York.

How do you fine tune the session times?There are two ways. One way is via research and estimation. You look at the time zone charts themselves, looking for their distinct characteristics for the best possible time zones for your system. For instance, you might want to trade a scalper only during the low liquidity Asian session, or a breakout system in the high liquidity confluence sessions (when the Asian/European sessions overlap, or when the European/New York sessions overlap). You can find these time zone charts in our own time zone page, or at http://www.forexmarkethours.com.

Another, more accurate method, is via optimization. Select a session as true, and the others as false, and optimize the start and end parameters for that session. For instance, if you were wanting to trade only the Asian session, and you wanted to know the best hours to trade with your system, I would keep the StartTime1 at 22.00 and optimize EndTime1 from 22 through 7, and the optimization report will detail which hours work best for my system.

MT4 Code Snippets

Paste This Code Near Top Of Source File

```
#import "Kernel32.dll"
void GetSystemTime(int& a0[]);
```

Paste This Code In Define Variables Section

```
extern string _6 = "--- Trading Hours ---";
extern bool AutoGMTOffset = TRUE;
extern double ManualGMTOffset = 0;
extern bool UseTradingHours = true;
extern bool TradeAsianMarket = true;
extern double StartTime1 = 22.00;
extern double EndTime1 = 07.00;
extern bool TradeEuropeanMarket = true;
extern double StartTime2 = 07.00;
extern double EndTime2 = 12.00;
extern bool TradeNewYorkMarket = true;
extern double StartTime3 = 12.00; // 8:00 EST
extern double EndTime3 = 17.00;
int gmtoffset;
string gs_548 = "";
```

Paste This Code After Start() Function

```
if (!IsTesting() && AutoGMTOffset == TRUE) gmtoffset = GMTOffset();
else gmtoffset = ManualGMTOffset;

string ls_52 = "Your Strategy is Running.";
string ls_60 = "Your Strategy is set up for time zone GMT " + gmtoffset;
string ls_76 = "Account Balance= " + DoubleToStr(AccountBalance(), 2);
string ls_84 = " ";

Comment("\n",
"\n", " ",
"\n", " ",
"\n", " ", ls_52,
"\n", " ", ls_60,
"\n", " ", ls_76,
// "\n", " ", ls_77,
"\n");
```

Paste This Code By Itself, At End Of Source File, Outside Of Start() Function

```
int TradeTime()
{
if (!IsTesting() && AutoGMTOffset == TRUE) gmtoffset = GMTOffset();
else gmtoffset = ManualGMTOffset;

int TradingTime=0;
int CurrentHour=Hour(); // Server time in hours
double CurrentMinute =Minute(); // Server time in minutes
double CurrentTime=CurrentHour + CurrentMinute/100; // Current time
double CurrentTime1 = CurrentTime + gmtoffset;

if (CurrentTime1==0) CurrentTime=00;
if (CurrentTime1<0) CurrentTime1 = CurrentTime1 + 24;
if (CurrentTime1 >= 24) CurrentTime1 = CurrentTime1 - 24;

if (!DaytoTrade()) return(false);
if (UseTradingHours==true)
{
if (TradeAsianMarket==true)
{
if(StartTime1
if(CurrentTime1>=StartTime1 && CurrentTime1<=EndTime1)
TradingTime=1;}

if(StartTime1>EndTime1){
if(CurrentTime1>=StartTime1 || CurrentTime1<=EndTime1)
TradingTime=1;}
}

if (TradeEuropeanMarket==true)
{
if(StartTime2
if(CurrentTime1>=StartTime2 && CurrentTime1<=EndTime2)
TradingTime=1;}

if(StartTime2>EndTime2){
if(CurrentTime1>=StartTime2 || CurrentTime1<=EndTime2)
TradingTime=1;}
}

if (TradeNewYorkMarket==true)
{
if(StartTime3
if(CurrentTime1>=StartTime3 && CurrentTime1<=EndTime3)
TradingTime=1;}

if(StartTime3>EndTime3){
if(CurrentTime1>=StartTime3 || CurrentTime1<=EndTime3)
TradingTime=1;}
}
}

else
TradingTime=1;
```

```
return(TradingTime);
}
```

Lastly, Insert Function TradeTime() Someplace In Code As Part Of Entry Condition

Example:

```
if (BuyCondition1 && TradeTime() )
OpenBuy=true;
```

Trading Days Filter, With NFP And Holidays

Introduction

This is a trading days filter, that allows users to turn on or off particular days of the week. In addition, it allows users to turn on or off the day of, and day prior to, Non-Farm Payrolls, just in case you do not want to trade on this volatile day. Moreover, the user can turn off the Christmas holidays and/or New Years holidays.

Parameters

Sun to Friday
Bool: Indicate whether or not you want to trade each specific day. The default is true, each day is traded.

NFP_Friday
Bool: Whether or not you want to trade on volatile Non-Farm payroll Friday, first Friday of the month. Default is that the day is traded.

NFP_ThursdayBefore
Bool: Whether or not you want to trade on the Thursday before the volatile Non-Farm Payroll Friday. Default is that the day is traded.

ChristmasHoldays
Bool: Whether or not you want to trade during a custom defined period in December, during the

low liquidity of the Christmas Holidays.

XMAS_DayBeginBreak
Double: This is the day in December you will start to filter out, as the beginning of the Christmas Holiday. The default is 15, but it can be any day.

NewYearsHolidays
Bool: Whether or not you want to trade during a custom defined period in the beginng of January, the low liquidity of the New Years Holidays.

NewYears_DayEndBreak
Double: This is the day in January you stop the filter, the end of the New Years holiday period. The default is 3, but it can be any day.

MT4 Code Snippets

```
extern string _7 = "--- Trading Days ---";
extern bool Sunday = true;
extern bool Monday = true;
extern bool Tuesday = true;
extern bool Wednesday = true;
extern bool Thursday = true;
extern bool Friday = true;
extern bool NFP_Friday = true;
extern bool NFP_ThursdayBefore = true;
extern bool ChristmasHolidays = true;
extern double XMAS_DayBeginBreak = 15;
extern bool NewYearsHolidays = true;
extern double NewYears_DayEndBreak = 3;



bool DaytoTrade(){
bool daytotrade = false;

if(DayOfWeek() == 0 && Sunday) daytotrade = true;
if(DayOfWeek() == 1 && Monday) daytotrade = true;
if(DayOfWeek() == 2 && Tuesday) daytotrade = true;
if(DayOfWeek() == 3 && Wednesday) daytotrade = true;
if(DayOfWeek() == 4 && Thursday) daytotrade = true;
if(DayOfWeek() == 5 && Friday) daytotrade = true;
if(DayOfWeek() == 5 && Day() < 8 && !NFP_Friday ) daytotrade = false;
if(DayOfWeek() == 4 && Day() < 8 && !NFP_ThursdayBefore ) daytotrade = false;
if(Month() == 12 && Day() > XMAS_DayBeginBreak && !ChristmasHolidays ) daytotrade =
false;
if(Month() == 1 && Day() < NewYears_DayEndBreak && ! NewYearsHolidays ) daytotrade =
false;

return(daytotrade);}
```

Explanation

We are naming our day filter function **DaytoTrade()**.

We declare a daytotrade (lower case) bool variable, and initiate it as false.

Next, we have a line for each day of the week, indicating if the day of the week should be traded or not. **DayofWeek()** function returns the current day of the week of the last known server time:

0 = Sunday
1 = Monday
2 = Tuesday
3 = Wednesday
4 = Thursday
5 = Friday
6 = Saturday

The logic of each line is as follows. If **DayofWeek()** is (==) Sunday (0), and the Sunday external bool is true (&& Sunday), then **daytotrade** bool is true. If Sunday external bool has been manually set to false, daytotrade reverts to false. The same on through all the days.

Next we filter for Non-Farm Payroll days, which occur the first Friday of every month.

Lastly, we filter for the Christmas and New Years Holidays.

MT4 Code Usage

```
if (OrdersTotalMagicOpen()==0 && OpenBuy==true
&& DaytoTrade() ) {
//Place order
}
```

All you need to do is slip the **DaytoTrade()** function in as a condition for placing order. It will then check to see if you have declared any days not to trade (these days will indicated as false), as well as holidays not to trade (the holiday period will be indicated as false), before placing an order.

We generally use **DaytoTrade()** function in conjunction with the **TradeTime()** discussed here.