



TABLE OF CONTENTS

Quick Start

Building Your Own Strategy 2
Building Your Own Indicator..... 2
Building Your Own Function..... 3

Modules

Function 5
Indicator 6
Strategy..... 7

Variables

Number 8
Bool 8
Series 8
String 8
Declaration of Variables 8
Naming of Variables..... 9

Constants

Immediate Constants 10
External Constants 10

Operators

Assignment 12
Operator [] 12
Arithmetical 12
Logical..... 13
Relational..... 13
Operators Precedence 14

Statements

Conditional Statements 15
Loop Statements..... 15
BREAK Statement..... 15
CONTINUE Statement..... 16
Function and Indicator Call 16
RETURN Statement..... 16

Appendix A: Internal Function Reference

Series Functions 18
Math Functions..... 19
String Function 20
Time Functions..... 20
Trading Functions 21

Appendix B: Library

..... 23

Appendix C: Samples

..... 55



OVERVIEW

Chart Studio uses the Common Technical Analysis Language (CTL). CTL is an easy to understand, Pascal-like, programming language. CTL enables you to write your own indicators and trading strategies.

QUICK START

Building Your Own Strategy

Let us assume that you want to create your own strategy, based on MACD indicator. Steven B. Achelis in his book "Techinical Analysis from A to Z" writes: *The basic MACD trading rule is to sell when the MACD falls below its signal line. Similarly, a buy signal occurs when the MACD rises above its signal line.*

Strategy, that implements this idea in CTL, has the following code:

```
//////////  
{ Basic MACD trading rule }  
strategy sample_macd;  
input lots = 1 ;  
begin  
MACD();  
if crossup (MACD.res, MACD.signal) then buy(lots); { MACD rises above its signal line }  
if crossdown(MACD.res, MACD.signal) then sell(lots); { MACD falls below its signal line }  
end.  
//////////
```

Text enclosed in the curly braces is a comment. It does not affect the result and is written only to make text clearer to the human reader. We recommend using comments everywhere to expand shortened names, for references, where the code is not simple and everytime you want to note something.

Keyword **strategy** denotes that the following code is a strategy. Keywords case is not important. You may type **Strategy**, **STRATEGY** or **sTrAtEgY**. **sample_macd** is the strategy name. A semicolon must be put after every statement or declaration (**strategy sample_macd** is a declaration).

After declarations the code begins. It is enclosed between **begin** and **end**. (with a fullstop) keywords.

First we must calculate MACD indicator. As we are not sure about parameters of the indicator, or prefer using default ones, we do not put anything within the parenthesis after **MACD**.

Then we check if MACD crosses MACD signal line, so that MACD rises above its signal line (**crossup()** command). This is buy signal. **Buy()** command has one parameter. Sell condition is similar, according to Achelis's book.

Building Your Own Indicator

Though we have written over 70 indicators, there are still many others, new indicators appear every month, or you may want to modify the existing one. Let us assume that you want to build MACD (Moving Average Convergence/Divergence) indicator.

Steven B. Achelis in his book "Techinical Analysis from A to Z" writes: *The MACD is the difference between a 26-day and 12-day exponential moving average. A 9-day exponential moving average, called the "signal" (or "trigger") line, is plotted on top of the MACD to show buy/sell opportunities.*

In CTL the code will be:

```
//////////  
{Moving Average Convergence/Divergence}  
indicator MACD;  
input src = close,  
first_period = 12,  
second_period = 26,  
signal_period = 9;  
draw res("MACD"), signal("MACD sig");  
begin  
res := ema(src, first_period) - ema(src, second_period);  
signal := ema(res, signal_period);  
end.  
//////////
```

Indicator keyword declares a new indicator with the name MACD. Then declaration of parameters of the indicator follows. Strategies and functions may have parameters too. The declaration starts with a keyword **input**. Declarators are separated with commas. Each declarator is a new (unique) name, called identifier. Each identifier acts as a value. There are 4 types of values in CTL: **bool** (that is **true** or **false**), **number**, **string** and **series**. **Series is a series of numeric values, which can be accessed by their indexes**. Declarator is used to associate identifier with a type. This may be done by explicit setting of initial value. In the sample identifier **src** is initialized by predefined series **close**, which performs values of close prices. 3 periods declared further are numeric values. When drawing indicator these parameters may be changed with the help of windows controls.

Because it is an indicator we must declare lines that it draws. Hence parameters declaration is followed by **draw** declaration. Each declarator is an identifier (which automatically has **series** type) and line name in parenthesis. Line name is a string, which must be enclosed in double quotes (e.g. "name"). MACD indicator consists of 2 lines: **res** (from 'result') and **signal**.

After declarations, execution code follows in **begin ... end** 'brackets'. As Steven Achelis says, MACD is the difference between a 26-day and 12-day exponential moving average. So we assign (using operator ':=') to **res** (which is declared as series, with the name MACD) the result of subtraction of exponential moving average (**EMA**) of source series (**src**) with period 26 (**second_period**, which in the example may be changed from default value 26) from exponential moving average of **src** with period 12. To calculate signal line, we assign exponential moving average of **res** to **signal**.

Building Your Own Signal

In indicator sample we used exponential moving average by calling `ema()` function. This function takes 2 parameters: one - of series type and the other - of number type; and returns a series value as a result. You may create your own functions each time, when you need some code several times, or want to make you indicator or strategy code clearer by hiding some code in functions. MACD indicator uses `ema` 3 times, thus making it's code clearer and shorter. Here comes the explanation how to build such a function.

Again from Achelis's book we read: *An exponential (or exponentially weighted) moving average is calculated by applying a percentage of today's closing price to a percentage of yesterday's moving average value. ... The exponential percentage can be calculated from a specific number of days using the following formula: Exponential Percentage = 2 / (Time Periods + 1).*

CTL code for this function is:

```
//////////  
{Exponential Moving Average}  
function EMA;  
input s(series), period(number);  
result res(series);  
vars i(number), f = 1, l(number), cnt(number), k(number), tmp(number);  
begin  
f := front(s);  
{round() is needed if period is not an integer, e.g. in DiNapoli MACD}  
l := round(f + period - 1);  
cnt := back(s);  
if l <= cnt then begin
```

```
k := 2 / (period + 1);  
tmp := s[f];  
for i := f + 1 to l do  
    tmp := tmp * (1 - k) + s[i] * k;  
res[i] := tmp;  
for i := l + 1 to cnt do begin  
    tmp := tmp * (1 - k) + s[i] * k;  
    res[i] := tmp;  
end;  
end;  
end.  
////////
```

This is rather sophisticated code, which uses many language features. For detailed explanations see reference. First of all this is a **function**, so the code starts with respective keyword. Function result is declared in the **result** declaration, which may consist only of a single declarator. Result type in parenthesis follows the identifier.

Then local variables declaration goes. Sometimes you need to store intermediate result somewhere. These values are stored in local variables. The declaration start with a keyword **vars** and is followed by declarators, which may be identifiers either with explicit type in the parenthesis or with implicit type which is taken from initializing value type. *Local variables declaration may be also present in indicators and strategies.*

There are some important things in this sample. In MACD trading strategy there was an **if ... then** statement. The function uses more complicated **if** statement: to distinguish which operations must be executed if the condition is true, we have to enclose these operations in the **begin...end** operation parenthesis. The same idea is used for the **for ... to ... do** statement. Statement **for a := 3 to 8 do smth**; is something like "for each value of a, changing from 3 to 8 by 1 do smth". That means that smth will be executed 6 times, and **a** will take values from 3 to 8. Square brackets are used to select a value with specified in the brackets number (this number is called index) from series. **front()** and **back()** are used to calculate first and last indices of a series. For detailed description see reference.

And the last advice. Use text formatting for easy code reading: use one statement per line, use spaces between operators, fill identical number of spaces for identical level of enclosure of begin...end operator brackets, usually 2 or 4 spaces per level. See sample sources and library as examples.

MODULES

Module is a basic unit that can be compiled and either called from another module or executed separately. There are three types of modules: Function, Indicator and Strategy.

Function

In many programs, certain routines are frequently repeated. So you can define commonly used routines as separate functions, thus reducing program size and complexity. Function receives data in input parameters and returns a result of execution to the module that has requested it. An indicator, a function or a strategy can call an existing function by its name.

Every function consists of the following parts (required are marked by asterisk):

```
//////////  
[*Header]  
function Name;  
[Input Parameters Declaration]  
input Parameters;  
[*Result Declaration]  
result Result;  
[Local Variables Declaration]  
vars Variables;  
[*Code]  
begin  
Statements;  
end.  
//////////
```

Name - is a unique name of a function.

Parameters -> **Parameter {, Parameter}** - is a list of comma separated parameters of a function.

Parameter -> Name(Type)

Parameter -> Name = Constant

Result -> Name(Type)

Result -> Name = Constant

Variables -> **Variable {, Variable}** - is a list of comma separated variables of a function.

Variable -> Name(Type)

Variable -> Name = Constant

Statements -> **Statement {; Statement}** - is a list of semicolon separated statements. See CTL Statements.

Each input parameter allows a value to be transferred to the function.

The result allows the function to transfer the result of its execution to a caller.

Variables are used in the function to store results of intermediate calculations.

Variables declared in input, result and vars sections can be treated as locally defined variables, therefore they can be used within a function code.

If the type of transferred to the function variable is not the same as the type appointed in input section, CTL compiler reports an error.

Note: Name length is not limited. The first letter of Name must be ALPHABETIC. Compiler does not distinguish lowercase and uppercase characters, so FOO is the same as foo. Use meaningful names, which will help to make the program easier to read and follow.

Here is an example:

```
//////////  
function PlusDI;  
input period(number);  
result res(series);  
vars i(number), pdm(series), tmp(number);  
begin  
  for i := front(high) + 1 to back(high) do begin
```



```
tmp := high[i] - high[i - 1];  
if tmp < 0 then  
    tmp := 0;  
    pdm[i] := tmp;  
end;  
res := 100 * mma(pdm, period) / mma(truerange(), period);  
end.  
////////
```

Indicator

For each tech-analysis indicator there is a certain mathematical model. Generally it is a set of formulas. Therefore your task is to implement these formulas using CTL. Indicator accepts data in input parameters and returns a result as a set of series. Runtime system, an indicator, a function or a strategy can call an existing indicator by its name.

```
[*Header]  
indicator Name;  
[Input Parameters Declaration]  
input Parameters;  
[*Result Declaration]  
draw Lines;  
[Local Variables Declaration]  
vars Variables;  
[*Code]  
begin  
Statements;  
end.
```

Name - is a unique name of an indicator.

Parameters -> *Parameter {, Parameter}* - is a list of comma separated parameters of an indicator.

Parameter -> *Name(Type)*

Parameter -> *Name = Constant*

Lines -> *Line {, Line}* - is a list of comma separated lines of an indicator.

Line -> *Name("Title")*

Line -> *Name("Title", Style, Color, Thickness)*

Line -> *Name("Title", Style, Color)*

Line -> *Name("Title", Style)*

Line -> *Name("Title") = Value*

Line -> *Name("Title", Style, Color, Thickness) = Value*

Line -> *Name("Title", Style, Color) = Value*

Line -> *Name("Title", Style) = Value*

Variables -> *Variable {, Variable}* - is a list of comma separated variables of an indicator.

Variable -> *Name(Type)*

Variable -> *Name = Constant*

Statements -> *Statement {; Statement}* - is a list of semicolon separated statements. See CTL Statements.

Value enable you to draw a horizontal line at specified level. It may be useful to mark top and bottom limits.

Here is an example:

```
indicator Directional_Movement;  
input period = 14;  
draw pdi("+DI"), mdi("-DI");  
begin  
    pdi := plusDI(period);  
    mdi := minusDI(period);  
end.
```

NOTE 1: Before getting access to the result of a certain indicator you have to call it.

NOTE 2: To get access to the result of the call to the indicator use the following syntax: IndicatorName.LineName;

Here is an example:

```
strategy sample_dmi;
input period = 14;
vars lst = 1, pdi(series), mdi(series);
begin
    directional_movement(period); /* calling directional_movement indicator */
    pdi := directional_movement.pdi; /* using the result of the call */
    mdi := directional_movement.mdi; /* using the result of the call */
    lst := back(pdi);
    if lst < front(pdi) + 1 then return;
    if pdi[lst] > mdi[lst] then buy();
    if pdi[lst] < mdi[lst] then sell();
end.
```

Indicator

"Technical analysis is the process of analyzing a security's historical prices in an effort to determine probable future prices."

Steven B. Achelis Technical-Analysis from A to Z

An ability to write your own indicator is a very useful feature. But it is not always enough. Should you buy or sell today? To answer the question you may want to perform analysis based on various indicators. In CTL this problem can be solved by means of strategy. When writing your own strategy you can use such system functions as buy(), sell(), exitlong(), exitshort(), limit_buy(), limit_sell(), stop_buy(), stop_sell(), alert(). Strategy can not be called from another strategy, function or indicator. Strategy does not return a result. To use a certain strategy you have to attach it to a chart.

Every strategy consists of the following parts (required are marked by asterisk):

```
[*Header]
strategy Name;
[Input Parameters Declaration]
input Parameters;
[Local Variables Declaration]
vars Variables;
[*Code]
begin
Statements;
end.
```

Name - is a unique name of a strategy.

Parameters -> *Parameter {, Parameter}* - is a list of comma separated parameters of a strategy.

Parameter -> *Name(Type)*

Parameter -> *Name = Constant*

Variables -> *Variable {, Variable}* - is a list of comma separated variables of a strategy.

Variable -> *Name(Type)*

Variable -> *Name = Constant*

Statements -> *Statement {; Statement}* - is a list of semicolon separated statements. See CTL Statements.

Here is an example:

```
strategy sample_dmi;
input period = 14, lots = 1;
vars lst = 1, pdi(series), mdi(series);
begin
    directional_movement(period);
    pdi := directional_movement.pdi;
    mdi := directional_movement.mdi;
```

```
lst := back(pdi);  
if lst < front(pdi) + 1 then return;  
if pdi[lst] > mdi[lst] then buy(lots);  
if pdi[lst] < mdi[lst] then sell(lots);  
end.
```

Variables And Types

Variables store numbers, strings, etc. Variables allow programs to perform calculations and store data for later use. A program variable can assume a value from a certain range. This range is called the type of a variable. Variable must be declared before using. When you declare a variable you indicate its type.

CTL supports four basic types: Number, Bool, Series and String.

Number

Variables of this type may contain positive or negative numbers in the range 1.7E-308 ... 1.7E+308
Here is an example: 100, -200, 23.4567, -0.45645, 9999999999

Bool

Variables of this type, also called logical variables, may only contain two possible values True or False

Series

In most cases series is used to store prices (Close, High, etc) and results of indicators execution. This type acts like a dynamic array of numbers. It means that you do not have to manage size of any series. Runtime system automatically adjusts the size of series if necessary. Elements' numbers start from 1. You can add or subtract two series, multiply or divide a series by number, negate a series, get an element of a series using operator []. See Operators for details. In general a series may contain invalid numbers at the beginning or at the end. For example when you add or subtract two series their sizes may be different, so the resulting series may contain not valid numbers. We strongly recommend to use front() and back() functions to get a position of the first valid number and the last valid number respectively. See The Library of indicators and functions for examples.

String

String is a sequence of character. Runtime environment automatically adjusts the size of string if necessary.

Declaration of Variables

Variable can only be declared within a certain module. There are two formats that can be used for variable declaration.

The basic format is:

```
vars Name(Type) {,Name(Type)};
```

where *Name* is the unique name of the variable being declared, and *VarType* is one of the basic types of CTL.

The other way is:

```
vars Name=Initializer {,Name=Initializer};
```

where *Name* is the unique name of the variable being declared, and *Initializer* is a constant.

Here are examples:

```
function PlusDI;  
input period(number);  
result res(series);  
vars i(number), pdm(series), tmp(number);  
begin  
  for i := front(high) + 1 to back(high) do begin  
    tmp := high[i] - high[i - 1];  
    if tmp < 0 then  
      tmp := 0;  
    pdm[i] := tmp;  
  end;  
  res := 100 * mma(pdm, period) / mma(truerange(), period);  
end.
```

In this function three variables are declared, namely: *i* has type Number, *pdm* has type Series, *tmp* has type Number. indicator Price_Volume_Trend;

```
input src = close;
draw pvt("PVT");
vars i(number), r = 0;
begin
  for i := front(close) + 1 to back(close) do begin
    r := r + (close[i] / close[i - 1] - 1) * volume[i];
    pvt[i] := r;
  end;
end.
```

In this function two variables are declared, namely: *i* has type Number, *r* is initialized by 0 and has type Number.

Naming of Variables

The length of the *Name* of variable is not limited. The first letter of the name must be ALPHABETIC.

Compiler does not distinguish lowercase and uppercase characters, so *VAR1(Number)* is the same as *var1(number)*. It is better to give variables meaningful names, that will help to make the program easier to read and follow.

CONSTANTS

Each constant is associated with a certain **type**.

Immediate constants

Number constants

Numeric constant is the set of digits (preceded with a possible sign) with possible dot, separating integer and fractional parts of the number. PI is also a number constant, equal to 3.14159265358979. Examples: 0, 7.6, -234, +9834.2334, -PI.

Boolean constants

There are only two immediate boolean constants: true and false.

String constants

String immediate constant is a set of any characters enclosed within double quotes. Examples: "", "a word", "\$", "The price is steadily going up! Maybe we should buy?"

External constants

These constants describe data of the chart, to which an indicator or strategy is attached, system global constants, current trade position state, etc.

Line styles

default_style, solid_line, dash_line, dot_line, dash_dot_line, dash_dot_dot_linelong, histogram, points, crosses, invisible

Line colors

default_color, black, dark_blue, dark_green, dark_cyan, dark_red, dark_magenta, dark_yellow, dark_gray, gray, blue, green, cyan, red, magenta, yellow, white

Line thickness

1, 2, 3, ... and so on



Boolean Constants

Long

True, if current position is long (i. e., after buy command), false if there is no position, or current position is short. See buy and related functions.

Short

True, if current position is short (i. e., after sell command), false if there is no position, or current position is long. See sell and related functions.

String Constants

Version

Current version of CTL. Current value is "0.1".

Symbol

Symbol name of a chart, to which indicator or strategy is attached.

Series Constants

In the following constants index of the element is the number of the period on the chart. That is **high[i]**, **low[i]**, **timestamp[i]**, etc describe the same time period number **i**.

Open

Series of **OPEN** price field, which is the price of the first trade for the period (e.g. first trade of the day for daily chart).

High

Series of **HIGH** price field, which is the highest price that a security was traded for during the period.

Low

Series of **LOW** price field, which is the lowest price that a security was traded for during the period.

Close

Series of **CLOSE** price field, which is the last price that a security was traded for during the period.

Timestamp

Series of time stamps of time periods' starts. Time stamp is expressed as the number of seconds, elapsed since midnight of Jan 1, 1970. See **time** and related functions.

OPERATORS

Assignments

Once you have declared a variable, you can store values in it. This is called *assignment*.

The basic format is:

Variable := Expression;

where Expression can either be a variable or a constant or an arithmetic sequence or a function or an indicator result or a combination of everything above.

An arithmetic sequence is:

37573.5 * 37593 + 385.8 / 367.1

Here are some examples:

a := 385.385837;

b := (**close** + **open**) / 2;

c := "Trading symbol " + symbol + " now is a bad idea.";

d := routine();

e := Macd.signal;

f := a * 3 + 345645;

Operator []

Use this operator to get access to an element of a series at a specified position. Brackets [] enclose the position. Position must be a positive integer. We recommend using front() and back() functions to determine first and last valid number indices.

Here are some examples:

a := s[50];

s[j] := b;

Arithmetical

Op	Description	Left Operand	Right Operand	Result
-	Unary Negation	-	Number	Number
-	Unary Negation (to negate each element of series)	-	Series	Series
*	Multiplication	Number	Number	Number
*	Multiplication (to multiply each element of series by number)	Number	Series	Series
*	Multiplication (to multiply each element of series by number)	Series	Number	Series
/	Division	Number	Number	Number
/	Division	Series	Number	Series
+	Addition	Number	Number	Number
+	Addition	Series	Series	Series
+	Addition (string concatenation)	String	String	String
-	Subtraction	Number	Number	Number
-	Subtraction	Series	Series	Series

Arithmetic operators are used to build arithmetic expressions.

Here are examples of arithmetic expressions:

$(n + 100) * m / (k - 1)$

$(-n + 5) / 14$

Op	Description	Left Operand	Right Operand	Result
NOT	Logical NOT	-	Bool	Bool
AND	Logical AND	Bool	Bool	Bool
OR	Logical OR	Bool	Bool	Bool

Logical operators are used to build logical expressions.

Here is an example of a logical expression:

a **and not** b **or** c **and** d

Let us suppose we have two variables of type Bool: a = True, b = False

not is a unary operator - it is applied to only one value and inverts it:

not a = False

not b = True

and yields True only if both expressions are True:

a **and** b = False

a **and** a = True

or yields True if either expression is True, or if both are:

a **or** a = True

a **or** b = True

b **or** a = True

b **or** b = False

Relational

Op	Description	Left Operand	Right Operand	Result
<	Less than	Number	Number	Bool
>	Greater than	Number	Number	Bool
<=	Less Or Equal	Number	Number	Bool
>=	Greater or Equal	Number	Number	Bool
<>	Not Equal	Number	Number	Bool
<>	Not Equal	Bool	Bool	Bool
=	Equal	Number	Number	Bool
=	Equal	Bool	Bool	Bool

Relational operators are used to build relational expressions.

Here are examples of relational expressions:

a + 100 <= b * 2 - 7

(a >= b) **and not** flag

Operators Precedence

The table below lists the CTL operators precedence. The highest precedence level is at the top of the table.

Unary +, -
*, /
+, -
NOT
AND
OR
<, >, <=, >=, <>, =

For example, the compiler works with arithmetic expressions according to the following rules:

1. Look for all expressions in parentheses, starting from the innermost set of parentheses and proceeding to the outermost.
2. Look for all unary minuses and pluses.
3. Look for all multiplication and division from left to right.
4. Look for all addition and subtraction from left to right.

Here are some examples:

A := 3.5 * (2 + 3); /* the value of A will be 17.5 */

B := 3.5 * 2 + 3; /* the value of B will be 10 */

To write sophisticated logic expressions use parentheses:

Here are some examples:

b := a > 1 **and** a <= m - 1000; /* WRONG. Compiler will report about error */

b := n < 100; /* RIGHT. */

b := (n < 100) **or** (m > 1000); /* RIGHT. */

if (n < 2) **and** (m > 10) **then** /* RIGHT. */

begin

/* do something */

end;

STATEMENTS

Statements make CTL not a simple formula calculator, but a real programming language. They allow using standard algorithmic constructions, such as conditions and loops.

Conditional Statements

IF-THEN and IF-THEN-ELSE statements

These statements are self-explanatory. After **if** keyword condition comes, which is an expression of type **bool**. Then keyword separates condition from the action that must be taken if the condition is true. If the action is not finished with a semicolon, but **else** keyword is present, then the action after the **else** will be taken in case of false result of condition expression calculation. In many cases it is required to take multiple actions if the condition is true or false. In this case **begin ... end** operation parentheses must be used. See Library for samples.

Loop Statements

FOR-TO-DO and FOR-DOWNTO-DO statement

This is basic statement for loops. The **for** keyword is followed by assignment operator, which sets the initial value for loop variable. Expression after the **to** or **downto** keyword is last value of this variable. The loop action, that follows the **do** keyword will be executed for each value of this variable from the range (increasing the value by one after every run of the loop action for **to** version and decreasing for **downto** version).

Here are the samples:

```
for i := 1 to 10 do
  sum := sum + i;
```

```
i := 1;
for m := back(close) downto front(close) do begin
  res[i] := close[m];
  i := i + 1;
end;
```

The first sample is very simple, it calculates the sum of numbers from 1 to 10. It means: for each *i* from 1 to 10 (changing by one) add *i* to sum and store the result in sum.

The second sample is more complex. It copies the close series to the res in reverse order. There are two actions in the loop body (that is the action after the **do** keyword). So these actions are enclosed in the **begin ... end** operation parentheses.

The loop may be interrupted with the **break** statement or explicitly set for the next iteration with the **continue** statement.

NOTE: The expression after **to** or **downto** is evaluated only once before executing loop body.

WHILE-DO Statement

This is another loop statement. The **while** keyword is followed by a condition, that is an expression of type **bool**. The action (that may be a multiple actions in **begin...end** operation parenthesis) follows the **do** keyword. The action is executed each time the condition is true. Here is an example:

```
i := back(close) - 1;
while close[i] < close[i + 1] do begin
  i := i - 1;
if i < front(close) then
  break;
end;
```

This code finds the position when the close prices started the last steady growth. It also uses the **break** statement to interrupt the loop if the price has been growing since the very beginning.

BREAK Statement

This statement causes current **for ... to ... do** or **while ... do** statements terminate without any conditions. See sample.

CONTINUE statement

This statement causes current loop to make next step or terminate (depending on the loop condition).

Function and Indicator Call

Function and Indicator can be called (executed) using the following syntax:
module_name(paramater1, parameter2, etc)

Function call allows implementing recurring formula calculation. For example, the Fibonacci numbers are defined by the formula:

$$f(i) = f(i - 1) + f(i - 2)$$

And it is known that $f(0) = 0$, $f(1) = 1$. So the recursive function in CTL will be:

```
function fibonacci;  
input n(number);  
result r(number);  
  
begin  
  if n <= 0 then  
    r := 0  
else  
  if n <= 1 then  
    r := 1  
else  
  r := fibonacci(n - 1) + fibonacci(n - 2);  
end.
```

This is only a sample, because this method is not effective in this case (for example to calculate fibonacci(5), the function will be executed 15 times).

Use recursion with caution. If the comparison of the parameter n with one and zero were not present in the sample, this function would cause your PC to go into an infinite loop, until the memory is run out (this is called Stack Overflow Error).

Here is Euclid's algorithm to find the greatest common divisor (GCD) of two integers in CTL:

```
function gcd;  
input a(number), b(number);  
result r(number);  
  
begin  
  if b <= 0 then  
    r := a  
else  
  r := gcd(b, a - int(a / b));  
end.
```

The expression $a - \text{int}(a / b)$ returns the remainder of division a by b.

Return Statement

This statement causes your module to finish execution. Current result value is returned. Usually this statement is used to terminate function, indicator or strategy if input parameters are not correct or there is not enough data in series. See Library sources for samples.



34th Floor (CGC 34-03) 25 Canada Square London E14 5LQ
international | 44 (0) 20 7170 0770
freephone | 0800 358 0864
fax | 44 (0) 20 7170 0788

APPENDIX A: Internal Functions Reference

Reference by Category

Series Functions

back
crossdown
crossup
displace
front
movmax
movmin
makeseries

Math Functions

abs
atan
cos
exp
frac
int
ln
log
max
min
pow
round
sin
sqrt
tan

String Functions

str

Time Functions

datetimestr
day
hour
minute
month
second
time
weekday
year

Trading Functions

alert
buy
sell
exitlong
exitshort
limit_buy
limit_sell
stop_buy
stop_sell
equity
fpl

Detailed Description

Series Functions

front

Input: src(series);

Result: res(number);

Returns index of the first series element, that is a valid number.

back

Input: src(series);

Result: res(number);

Returns index of the last series element, that is a valid number.

displace

Input: src(series), shift(number);

Result: res(series);

Returns series that is same as **src**, but displaced by **shift** to higher indices (if **shift** is positive) or to lower indices (if **shift** is negative). In the latter case, if series elements' indices become negative or zero, these elements will be lost. In other words, the result is a series with the same elements that **src** has, but **front(res) = max(1, front(src) + shift)** and **back(res) = back(src) + shift**.

crossup

Input: first(series), second(series);

Result: res(boolean);

Returns true, if the **first** series rises above **second** series, and false otherwise. This means that the last element of the **first** series is greater than the last element of the **second** series, and the last but one is less or equal.

crossdown

Input: first(series), second(series);

Result: res(boolean);

Returns true, if the **first** series falls below **second** series, and false otherwise. This means that the last element of the **first** series is less than the last element of the **second** series, and the last but one is greater or equal.

movmax

Input: src(series), position(number), period(number);

Result: res(number);

Returns index of maximum series element among **period** elements from **position** back to front element. In other words, this function returns number from **position - period + 1 to position**.

movmin

Input: src(series), position(number), period(number);

Result: res(number);

Returns index of minimum series element among **period** elements from **position** back to front element. In other words, this function returns number from **position - period + 1 to position**.

makeseries

Input: front_index(number), back_index(number), value(number);

Result: res(series);

Returns a new series where all elements in the range **front_index..back_index** are initialized to **value**



Math Functions

abs

Input: x(number);

Result: res(number);

Returns absolute value of a number. That is, if **x** is negative, returns positive part-**x**. If **x** is positive, returns it unchanged.

max

Input: x(number), y(number);

Result: res(number);

Returns the maximum of two numbers.

min

Input: x(number), y(number);

Result: res(number);

Returns the minimum of two numbers.

sin

Input: x(number);

Result: res(number);

Returns the sine of a number.

cos

Input: x(number);

Result: res(number);

Returns the cosine of a number.

tan

Input: x(number);

Result: res(number);

Returns the tangent of a number.

atan

Input: x(number);

Result: res(number);

Returns the arctangent of a number.

sqrt

Input: x(number);

Result: res(number);

Returns the square root of a number.

pow

Input: x(number), y(number);

Result: res(number);

Returns **x** raised to the power of **y**.

exp

Input: x(number);

Result: res(number);

Returns the exponential value of a number.

ln

Input: x(number);

Result: res(number);

Returns the natural logarithm of a number.

log

Input: x(number);

Result: res(number);

Returns the decimal logarithm of a number.

round

Input: x(number);

Result: res(number);

Returns the closest integer of a number.

int

Input: x(number);

Result: res(number);

Returns the integer part of a number.

frac

Input: x(number);

Result: res(number);

Returns the fractional part of a number.

String Functions**str**

Input: x(number), digits(number);

Result: res(string);

Returns the string representation with the number of digits equal to parameter digits.

Time Functions**datetimestr**

Result: res(string);

Returns the string representation of current time in the form: Wed Jan 02 02:03:55 1980.

time

Result: res(number);

Returns current time as the number of seconds elapsed since midnight (00:00:00), January 1, 1970.

year

Input: timepoint = 0;

Result: res(number);

Returns year for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current year.

month

Input: timepoint = 0;

Result: res(number);

Returns month for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current month.

day

Input: timepoint = 0;

Result: res(number);

Returns day of month for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current day of month.

weekday

Input: timepoint = 0;

Result: res(number);

Returns day of week (1 - for Sunday, 7 - for Saturday) for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current day of week.

hour

Input: timepoint = 0;

Result: res(number);

Returns hours since midnight for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current hour.

minute

Input: timepoint = 0;

Result: res(number);

Returns minute after hour for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current minute.

second

Input: timepoint = 0;

Result: res(number);

Returns seconds after minute for specified number of seconds elapsed since midnight, January 1, 1970 (received either with time function or element of timestamp series). If timepoint is zero (or no parameter was passed to the function), returns current second.

Trading Functions**alert**

Input: message(string);

This function can be used only in strategies to attract trader's attention for special states of the market according to strategy idea.

buy

Input: lots(number);

This function has no result. It can be used only in strategies as buy command to enter long position. Function issues a market buy order.

sell

Input: lots(number);

This function has no result. It can be used only in strategies as buy command to enter short position. Function issues a market sell order.

exitlong

This function has no parameters and result. It can be used only in strategies to exit long position. It has no effect if current position is short or flat. Function affects only position entered by this strategy.

exitshort

This function has no parameters and result. It can be used only in strategies to exit long position. It has no effect if current position is long or flat. Function affects only position entered by this strategy.

limit_buy

Input: lots(number),price(number);

This function has no result. It can be used only in strategies as command to issues limit buy order with specified number of lots and price.

limit_sell

Input: lots(number),price(number);

This function has no result. It can be used only in strategies as command to issues limit sell order with specified number of lots and price.



34th Floor (CGC 34-03) 25 Canada Square London E14 5LQ
international | 44 (0) 20 7170 0770
freephone | 0800 358 0864
fax | 44 (0) 20 7170 0788

CHART STUDIO USER GUIDE

Revised: March 2007

stop_buy

Input: lots(number),price(number);

This function has no result. It can be used only in strategies as command to issues stop buy order with specified number of lots and price.

stop_sell

Input: lots(number),price(number);

This function has no result. It can be used only in strategies as command to issues stop sell order with specified number of lots and price.

equity

Result: res(number);

Returns current account equity.

fpl

Result: res(number);

Returns current account fpl.

APPENDIX B: Library*Reference by Name***Functions**

AccDis
Accumulate
DirMovDX
EMA
Ishimoku_Func
LinReg
LinRegSlope
MinusDI
MMA
PlusDI
RelVol
SMA
StdErr
StdDev
Swing
TrueHigh
TrueLow
TrueRange
WMA

Indicators

Accumulation_Distribution
Accumulation_Swing
Aroon
Average_True_Range
Bollinger_Bands
Chaikin_Money_Flow
Chaikin_Oscillator
Chande_Momentum_Oscillator
Chikou_Span
Commodity_Channel
Commodity_Selection
Day_Open_Close
DEMA
Detrend
DiNapoli_Detrend
DiNapoli_MA_3x3
DiNapoli_MA_7x5
DiNapoli_MA_25x5
DiNapoli_MACD
DiNapoli_Oscillator_Predictor
DiNapoli_Preferred_Stochastics
Directional_Movement
Directional_Movement_ADX
Directional_Movement_ADXR
Directional_Movement_DX
Dynamic_Momentum
Ease_Of_Movement
Envelope
Forecast_Oscillator
FX_Trend
Indicator_75

Inertia
Intraday_Momentum
Ishimoku
Kairi
Keltner_Channel
Kijun_Sen
Klinger
Linear_Regression
Linear_Regression_Slope
MACD
Market_Facilitation
Mass_Index
Median_Price
Momentum
Money_Flow
Mov_Avg_Exponential
Mov_Avg_Modified
Mov_Avg_Simple
Mov_Avg_Triangular
Mov_Avg_Weighted
MT_Custom
Negative_Volume_Index
On_Balance_Volume
Parabolic_SAR
Percent_Change
Percent_Of_Resistance
Percent_R
Positive_Volume_Index
Price_Channel
Price_Oscillator
Price_Volume_Trend
Rate_Of_Change
Relative_Strength
Relative_Volatility
Senkou_Span
Standard_Deviation
Standard_Error_Bands
Stark_Bands
Stochastics
Swing
TEMA
Tenkan_Sen
Time_Series_Forecast
TRIX
Typical_Price
Ultimate_Oscillator
Volatility_Chaikins
Weighted_Close
Williams_AD
ZigZag

Aroon Indicator

```
/* Aroon indicator */  
indicator Aroon;  
input period = 14;  
draw u("Aroon Up"), d("Aroon Down");  
vars i(number), j(number), maxpos(number), minpos(number);  
begin  
  for i := front(high) + period to back(high) do begin  
    maxpos := i - period;  
    minpos := maxpos;  
    for j := i - period + 1 to i do begin  
      if high[maxpos] < high[j] then  
        maxpos := j;  
      if low[minpos] > low[j] then  
        minpos := j;  
    end;  
    u[j] := 100 * (maxpos - i + period) / period;  
    d[j] := 100 * (minpos - i + period) / period;  
  end;  
end.
```

Average True Range Indicator

```
/* Average true range indicator */  
indicator Average_True_Range;  
input period = 14;  
draw atr("ATR");  
begin  
  atr := mma(truerange(), period);  
end.
```

Bollinger Bands Indicator

```
/* Bollinger bands indicator */  
indicator Bollinger_Bands;  
input src = close, period = 20, deviations = 2;  
draw mid("Mid BB"), upper("Upper BB"), lower("Lower BB");  
vars tmp(series);  
begin  
  mid := sma(src, period);  
  tmp := deviations * stddev(src, period);  
  upper := mid + tmp;  
  lower := mid - tmp;  
end.
```

Chaikin Money Flow Indicator

```
/* Chaikin money flow indicator */  
indicator Chaikin_Money_Flow;  
input period = 21;  
draw cmf("Chaikin MF");  
begin  
  cmf := sma(accdis(), period) / sma(volume, period);  
end.
```

Chaikin Oscillator Indicator

```
/* Chaikin oscillator indicator */
indicator Chaikin_Oscillator;
draw co("Chaikin Osc");
vars ad(series);
begin
  ad := accumulate(accdis());
  co := ema(ad, 3) - ema(ad, 10);
end.
```

Chande Momentum Oscillator Indicator

```
/* Chande momentum oscillator indicator */
indicator Chande_Momentum;
input src = close, period = 20;
draw cmo("CMO");
vars i(number), u(series), d(series), au(series), ad(series), dif(number), f(number);
begin
  f := front(src);
  for i := f + 1 to back(src) do begin
    dif := src[i] - src[i - 1];
    if dif > 0 then begin
      u[i] := dif;
      d[i] := 0;
    end else begin
      u[i] := 0;
      d[i] := -dif;
    end;
  end;
  au := sma(u, period);
  ad := sma(d, period);
  cmo := 100 * (au - ad) / (au + ad);
end.
```

Chiko Span Indicator

```
indicator Chikou_Span;
draw res("Chikou Span");
begin
  res := displace(close, 25);
end.
```

Commodity Channel Index Indicator

```
/* Commodity channel index indicator */
indicator Commodity_Channel;
input period = 14;
draw cci("CCI");
vars i(number), j(number), mean(series), avgmean(series), dev(number);
begin
  mean := (high + low + close) / 3;
  avgmean := sma(mean, period);
  for i := front(avgmean) to back(avgmean) do begin
    dev := 0;
    for j := i - period + 1 to i do
      dev := dev + abs(mean[j] - avgmean[i]);
    cci[i] := (mean[i] - avgmean[i]) * period / dev / 0.015;
  end;
end.
```

Commodity Selection Index Indicator

```
/* Commodity selection index indicator */
indicator Commodity_Selection;
input period = 14, margin = 2500, commission = 25, point = 100;
draw csi("CSI");
vars i(number), k(number), atr(series), adx(series), adxr(series);
begin
  k := 100 * point / sqrt(margin) / (150 + commission);
  atr := sma(truerange(), period);
  adx := mma(dirmov_DX(period), period);
  adxr := (adx + displace(adx, period)) / 2;
  csi := k * adxr * atr {/ close};
end.
```

Day Open / Close Indicator

```
/* Day Open/Close indicator */
indicator Day_Open_Close;
draw dayopen("day Open"), dayclose("day Close");
vars daystamp(number), iday(number), i(number), j(number), curopen(number), lastclose(number), cnt(number), fst(number);
begin
  cnt := back(close);
  fst := front(close);
  if cnt < fst then
    return;
  curopen := open[fst];
  dayopen[fst] := curopen;
  daystamp := day(timestamp[fst]);
  for i := fst + 1 to cnt do begin
    iday := day(timestamp[i]);
    if iday = daystamp then
      dayopen[i] := dayopen[i - 1]
    else begin
      dayopen[i] := open[i];
      j := i - 1;
      lastclose := close[j];
      while day(timestamp[j]) = daystamp do begin
        dayclose[j] := lastclose;
        j := j - 1;
        if j < fst then
          break;
        end;
      daystamp := iday;
    end;
  end;
  j := cnt;
  lastclose := close[j];
  dayclose[j] := lastclose;
  j := j - 1;
  if j >= fst then
    while day(timestamp[j]) = daystamp do begin
      dayclose[j] := lastclose;
      j := j - 1;
      if j < fst then
        break;
      end;
    end;
  end.
```



Double Exponential Moving Average Indicator

```
/* Double exponential moving average indicator */  
indicator DEMA;  
input src = close, period = 21;  
draw res("DEMA");  
vars tmp(series);  
begin  
    tmp := ema(src, period);  
    res := 2 * tmp - ema(tmp, period);  
end.
```

Detrend Price Indicator

```
/* Detrend price indicator */  
indicator Detrend;  
input src = close, period = 20;  
draw res("DPO");  
vars i(number), shift(number), avg(series), st(number);  
begin  
    avg := sma(src, period);  
    st := front(avg) - shift;  
    if st < front(src) then  
        st := front(src);  
    for i := st to back(avg) - shift do  
        res[i] := src[i] - avg[i + shift];  
end.
```

DiNapoli Detrend Price Indicator

```
/* DiNapoli Detrend price indicator */  
indicator DiNapoli_Detrend;  
input src = close, period = 7;  
draw res("DDPO");  
begin  
    res := src - sma(src, period);  
end.
```

DiNapoli Moving Average Indicator

```
/* DiNapoli Moving Average indicator */  
indicator DiNapoli_MA_3x3;  
input src = close, period = 3, shift = 3;  
draw res("DiNapoli MA");  
begin  
    res := displace(sma(src, period), shift);  
end.
```

Dinapoli Moving Average Indicator

```
/* DiNapoli Moving Average indicator */  
indicator DiNapoli_MA_7x5;  
input src = close, period = 7, shift = 5;  
draw res("DiNapoli MA");  
begin  
    res := displace(sma(src, period), shift);  
end.
```



DiNapoli Moving Average Indicator

```
/* DiNapoli Moving Average indicator */  
indicator DiNapoli_MA_25x5;  
input src = close, period = 25, shift = 5;  
draw res("DiNapoli MA");  
begin  
    res := displace(sma(src, period), shift);  
end.
```

DiNapli Moving Average Convergence/Divergence Indicator

```
/* DiNapoli Moving Average Convergence/Divergence indicator */  
indicator DiNapoli_MACD;  
input src = close;  
draw res("DMACD"), signal("DMACD sig");  
vars  
    first_period = 8.3896,  
    second_period = 17.5185,  
    signal_period = 9.0503;  
begin  
    res := ema(src, first_period) - ema(src, second_period);  
    signal := ema(res, signal_period);  
end.
```

DiNapoli Oscillator Predictor

```
/*DiNapoli Oscillator Predictor */  
Source code is not available
```

DiNapoli Preferred Stochastics Indicator

```
/* DiNapoli Preferred Stochastics indicator */  
indicator DiNapoli_Preferred_Stochastics;  
input src = close, k_period = 8, k_slow_period = 3, d_period = 3;  
draw k("DNP Fast %K"), pk("DNP %K"), pd("DNP %D");  
vars i(number), lo(number), dif(number);  
begin  
    for i := front(src) + k_period - 1 to back(src) do begin  
        lo := movmin(low, i, k_period);  
        dif := movmax(high, i, k_period) - lo;  
        if dif > 0 then  
            k[i] := 100 * (src[i] - lo) / dif  
        else  
            k[i] := 0;  
    end;  
    pk := mma(k, k_slow_period);  
    pd := mma(pk, d_period);  
end.
```

Directional movement +DI and -DI indicator

```
/* Directional movement +DI and -DI indicator */  
indicator Directional_Movement;  
input period = 14;  
draw pdi("+DI"), mdi("-DI");  
begin  
    pdi := plusDI(period);  
    mdi := minusDI(period);  
end.
```

Directional Movement ADX Indicator

```
/* Directional movement ADX indicator */
indicator Directional_Movement_ADX;
input period = 14;
draw adx("ADX");
begin
    adx := mma(dirmov_DX(period), period);
end.
```

Directional Movement ADXR Indicator

```
/* Directional movement ADXR indicator */
indicator Directional_Movement_ADXR;
input period = 14;
draw adxr("ADXR");
vars adx(series);
begin
    adx := mma(dirmov_DX(period), period);
    adxr := (adx + displace(adx, period)) / 2;
end.
```

Directional movement DX Indicator

```
/* Directional movement DX indicator */
indicator Directional_Movement_DX;
input period = 14;
draw dx("DX");
begin
    dx := dirmov_DX(period);
end.
```

Directional Movement DX Function

```
/* Directional movement DX function */
function Dirmov_DX;
input period(number);
result dx(series);
vars pdi(series), mdi(series), i(number);
begin
    pdi := plusDI(period);
    mdi := minusDI(period);
    for i := front(pdi) to back(pdi) do
        dx[i] := 100 * abs(pdi[i] - mdi[i]) / (pdi[i] + mdi[i]);
end.
```

Dynamic Momentum Index Indicator

```
/* Dynamic momentum index indicator */
indicator Dynamic_Momentum;
input src = close;
draw dymi("DyMI");
vars period(number), dev(series), adev(series), i(number), st(number), j(number), dif(number), up(number), dn(number);
begin
    dev := stddev(close, 5);
    adev := sma(dev, 10);
    for i := front(adev) to back(adev) do begin
        period := 30;
        if dev[i] <> 0 then begin
            period := int(14 * adev[i] / dev[i]);
            if period < 3 then period := 3;
        end;
    end;
```

```

  if period > 30 then period := 30;
end;
up := 0;
dn := 0;
st := i - period + 1;
if st < front(src) + 1 then st := front(src) + 1;
for j := st to i do begin
  dif := src[j] - src[j - 1];
  if dif > 0 then
    up := up + (+dif - up) / period
  else
    dn := dn + (-dif - dn) / period;
end;
if up + dn = 0 then
  dymi[j] := 0
else
  dymi[j] := 100 * up / (up + dn);
end;
end.

```

Ease of Movement Indicator

```

/* Ease of movement indicator */
indicator Ease_Of_Movement;
input period = 14;
draw emv("EMV");
vars i(number), s(series);
begin
  for i := front(volume) + 1 to back(volume) do begin
    if volume[i] = 0 then
      s[i] := 0
    else
      s[i] := (high[i] - low[i]) * (high[i] - high[i - 1] + low[i] - low[i - 1]) * 5000 / volume[i];
    end;
    emv := sma(s, period);
  end.
end.

```

Exponential Moving Average Function

```

/* Exponential moving average function */
function EMA;
input s(series), period(number);
result res(series);
vars i(number), f(number), l(number), cnt(number), k(number), tmp(number);
begin
  f := front(s);
  {round() is needed if period is not an integer, e.g. in DiNapoli MACD}
  l := round(f + period - 1);
  cnt := back(s);
  if l <= cnt then begin
    k := 2 / (period + 1);
    tmp := s[f];
    for i := f + 1 to l do
      tmp := tmp * (1 - k) + s[i] * k;
    res[l] := tmp;
    for i := l + 1 to cnt do begin
      tmp := tmp * (1 - k) + s[i] * k;
      res[i] := tmp;
    end;
  end;
end.
end.

```



Envelope Indicator

```
/* Envelope indicator */  
indicator Envelope;  
input src = close, period = 25, vshift_percent = 5;  
draw up("Env Up"), dn("Env Dn");  
vars avg(series);  
begin  
  avg := sma(src, period);  
  up := avg * (1 + vshift_percent / 100);  
  dn := avg * (1 - vshift_percent / 100);  
end.
```

Forecast Oscillator Indicator

```
/* Forecast oscillator indicator */  
indicator Forecast_Oscillator;  
input src = close, period = 5, signal_period = 3;  
draw res("FO"), sig("FO sig");  
vars tsf(series), i(number);  
begin  
  tsf := linreg(src, period) + linregslope(src, period);  
  for i := front(tsf) + 1 to back(tsf) do  
    res[i] := 100 * (src[i] - tsf[i - 1]) / src[i];  
    sig := ema(res, signal_period);  
end.
```

FX Trend Indicator

```
/* FX Trend indicator */  
indicator FX_Trend;  
input first_period = 7, second_period = 14, third_period = 28;  
draw fxt("FX Trend");  
vars i(number), fst(number), lst(number), hh1(number), hh2(number), hh3(number),  
  ll1(number), ll2(number), ll3(number), dif1(number), dif2(number), dif3(number), mul(number);  
begin  
  fst := max(max(max(first_period, second_period), third_period), front(close));  
  lst := back(close);  
  mul := 100 / (1 / first_period + 1 / second_period + 1 / third_period);  
  for i := fst to lst do begin  
    hh1 := movmax(high, i, first_period);  
    hh2 := movmax(high, i, second_period);  
    hh3 := movmax(high, i, third_period);  
    ll1 := movmin(low, i, first_period);  
    ll2 := movmin(low, i, second_period);  
    ll3 := movmin(low, i, third_period);  
    dif1 := hh1 - ll1;  
    dif2 := hh2 - ll2;  
    dif3 := hh3 - ll3;  
    if (dif1 <> 0) and (dif2 <> 0) and (dif3 <> 0) then  
      fxt[i] := mul * ((close[i] - ll1) / dif1 / first_period +  
        (close[i] - ll2) / dif2 / second_period +  
        (close[i] - ll3) / dif3 / third_period)  
    else  
      fxt[i] := 0;  
  end;  
end.
```

Indicator 75 Indicator

```
/* Indicator75 indicator */
indicator Indicator_75;
input src = close;
draw i75("I75");
begin
    i75 := sma(src, 9) - displace(sma(src, 18), 3);
end.
```

Inertia Indicator

```
/* Inertia indicator */
indicator Inertia;
input period = 14, regression_period = 20;
draw ine("Inertia");
begin
    ine := linreg(relvol(period), regression_period);
end.
```

Intraday Momentum Index Indicator

```
/* Intraday momentum index indicator */
indicator Intraday_Momentum;
input period = 14;
draw cmo("IMI");
vars i(number), u(series), d(series), au(series), ad(series), dif(number), f(number);
begin
    for i := front(open) to back(open) do begin
        dif := close[i] - open[i];
        if dif > 0 then begin
            u[i] := dif;
            d[i] := 0;
        end else begin
            u[i] := 0;
            d[i] := -dif;
        end;
    end;
    au := sma(u, period);
    ad := sma(d, period);
    for i := front(au) to back(au) do begin
        dif := au[i] + ad[i];
        if dif = 0 then
            cmo[i] := 0
        else
            cmo[i] := 100 * au[i] / dif;
        end;
    end;
end.
```

Ishimoku Function

```
/* Ishimoku function */
function Ishimoku_Func;
input period(number);
result res(series);
vars i(number);
begin
    for i := front(high) + period - 1 to back(high) do
        res[i] := (movmax(high, i, period) + movmin(low, i, period)) / 2;
    end.
```

Ishimoku Indicator

```
/* Ishimoku indicator */
indicator Ishimoku;
draw cs("Chikou Span"), ts("Tenkan-sen"), ks("Kijun-sen"), sa("Senkou Span A"), sb("Senkou Span B");
begin
  cs := displace(close, 25);
  ts := ishimoku_func(9);
  ks := ishimoku_func(26);
  sa := displace((ts + ks) / 2, 26);
  sb := displace(ishimoku_func(52), 26);
end.
```

Kairi Indicator

```
/* Kairi indicator */
indicator Kairi;
input src = close, period = 12;
draw res("KRI");
vars i(number), avg(series);
begin
  avg := sma(src, period);
  for i := front(avg) to back(avg) do
    res[i] := 100 * (src[i] - avg[i]) / avg[i];
end.
```

Keltner channel indicator

```
/* Keltner channel indicator */
indicator Keltner_Channel;
input period = 12, factor = 1;
draw mid("Mid KC"), upper("Upper KC"), lower("Lower KC");
vars tmp(series);
begin
  mid := mma(close, period);
  tmp := factor * mma(truerange(), period);
  upper := mid + tmp;
  lower := mid - tmp;
end.
```

KijunSen Indicator

```
/* KijunSen indicator */
indicator Kijun_Sen;
draw res("Kijun-sen");
begin
  res := ishimoku_func(26);
end.
```

Klinger Oscillator Indicator

```
/* Klinger oscillator indicator */
indicator Klinger;
input signal_period = 13;
draw kvo("KVO"), kvosig("KVO sig");
vars cm(number), vf(series), i(number), hlc(series), dm(series), trend(number), newtrend(number);
begin
  hlc := high + low + close;
  dm := high - low;
  cm := 0;
  trend := 0;
```

```

for i := front(hlc) + 1 to back(hlc) do begin
  newtrend := trend;
  if hlc[i] > hlc[i - 1] then
    newtrend := 1;
  if hlc[i] < hlc[i - 1] then
    newtrend := -1;
  if trend = newtrend then
    cm := cm + dm[i]
  else
    cm := dm[i - 1] + dm[i];
  trend := newtrend;
  if cm = 0 then
    vf[i] := 0
  else
    vf[i] := volume[i] * abs(2 * dm[i] / cm - 1) * trend * 100;
end;
kvo := ema(vf, 34) - ema(vf, 55);
kvosig := ema(kvo, signal_period);
end.

```

Linear Regression Indicator

```

/* Linear regression indicator */
indicator Linear_Regression;
input src = close, period = 14;
draw res("LRI");
begin
  res := linreg(src, period);
end.

```

Linear Regression Slope Indicator

```

/* Linear regression slope indicator */
indicator Linear_Regression_Slope;
input src = close, period = 14;
draw res("LR Slope");
begin
  res := linregslope(src, period);
end.

```

Linear Regression Function

```

/* Linear regression function */
function LinReg;
input s(series), period(number);
result res(series);
vars i(number), sumx(number), sumx2(number), sumxy(number), sumy(number), a(number), b(number), l(number),
cnt(number), f(number);
begin
  f := front(s);
  l := f + period - 1;
  cnt := back(s);
  if l <= cnt then begin
    sumx := 0;
    sumy := 0;
    sumx2 := 0;
    sumxy := 0;
    for i := f to l do begin
      sumx := sumx + i;
      sumy := sumy + s[i];
      sumx2 := sumx2 + i * i;

```

```

    sumxy := sumxy + i * s[i];
  end;
  b := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
  a := (sumy - b * sumx) / period;
  res[] := a + b * i;
  for i := l + 1 to cnt do begin
    sumx := sumx + period;
    sumy := sumy + s[i] - s[i - period];
    sumx2 := sumx2 + 2 * i * period - period * period;
    sumxy := sumxy + i * s[i] - (i - period) * s[i - period];
    b := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
    a := (sumy - b * sumx) / period;
    res[i] := a + b * i;
  end;
end;
end.

```

Linear Regression Function

```

/* Linear regression function */
function LinRegSlope;
input s(series), period(number);
result res(series);
vars i(number), sumx(number), sumx2(number), sumxy(number), sumy(number), l(number), cnt(number), f(number);
begin
  f := front(s);
  l := f + period - 1;
  cnt := back(s);
  if l <= cnt then begin
    sumx := 0;
    sumy := 0;
    sumx2 := 0;
    sumxy := 0;
    for i := f to l do begin
      sumx := sumx + i;
      sumy := sumy + s[i];
      sumx2 := sumx2 + i * i;
      sumxy := sumxy + i * s[i];
    end;
    res[] := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
    for i := l + 1 to cnt do begin
      sumx := sumx + period;
      sumy := sumy + s[i] - s[i - period];
      sumx2 := sumx2 + 2 * i * period - period * period;
      sumxy := sumxy + i * s[i] - (i - period) * s[i - period];
      res[i] := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
    end;
  end;
end.

```

Moving Average Convergence/Divergence Indicator

```

/* Moving Average Convergence/Divergence indicator */
indicator MACD;
input src = close,
  first_period = 12,
  second_period = 26,
  signal_period = 9;
draw res("MACD"), signal("MACD sig");

```

CHART STUDIO USER GUIDE

Revised: March 2007

```
begin
  res := ema(src, first_period) - ema(src, second_period);
  signal := ema(res, signal_period);
end.
```

Market Facilitation Index Indicator

```
/* Market facilitation index indicator */
indicator Market_Facilitation;
draw res("MFcl");
vars i(number);
begin
  for i := front(volume) to back(volume) do
    if volume[i] = 0 then
      res[i] := 0
    else
      res[i] := (high[i] - low[i]) / volume[i];
    end.
end.
```

Mass Index Indicator

```
/* Mass index indicator */
indicator Mass_Index;
input period = 25;
draw mass("Mass");
vars ahl(series), aahl(series), i(number), sum(number), en(number);
begin
  ahl := ema(high - low, 9);
  aahl := ema(ahl, 9);
  sum := 0;
  en := front(aahl) + period - 1;
  if en <= back(aahl) then begin
    for i:= front(aahl) to en do
      if aahl[i] <> 0 then
        sum := sum + ahl[i] / aahl[i];
    mass[en] := sum;
    for i := en + 1 to back(aahl) do begin
      if aahl[i] <> 0 then
        sum := sum + ahl[i] / aahl[i];
      if aahl[i - period] <> 0 then
        sum := sum - ahl[i - period] / aahl[i - period];
      mass[i] := sum;
    end;
  end;
end.
```

Median Price Indicator

```
/* Median price indicator */
indicator Median_Price;
draw res("Median");
begin
  res := (high + low) / 2;
end.
```

Direction Movement -DI Function

```
/* Directional Movement -DI function */
function MinusDI;
input period(number);
result res(series);
vars i(number), mdm(series), tmp(number);
begin
  for i := front(low) + 1 to back(low) do begin
    tmp := low[i - 1] - low[i];
    if tmp < 0 then
      tmp := 0;
    mdm[i] := tmp;
  end;
  res := 100 * mma(mdm, period) / mma(truerange(), period);
end.
```

Modified Moving Average Function

```
/* Modified moving average function */
function MMA;
input s(series), period(number);
result res(series);
vars i(number), tmp(number), l(number), cnt(number), f(number);
begin
  f := front(s);
  l := f + period - 1;
  cnt := back(s);
  if l <= cnt then begin
    tmp := 0;
    for i := f to l do
      tmp := tmp + s[i];
    res[l] := tmp / period;
    for i := l + 1 to cnt do
      res[i] := res[i - 1] + (s[i] - res[i - 1]) / period;
    end;
  end;
end.
```

Momentum Indicator

```
/* Momentum indicator */
indicator Momentum;
input src = close, period = 12;
draw res("Momentum");
begin
  res := 100 * src / displace(src, period);
end.
```

Money Flow Index Indicator

```
/* Money flow index indicator */
indicator Money_Flow;
input period = 14;
draw mf("Money Flow");
vars pmf(series), apmf(series), nmf(series), anmf(series), i(number), hlc(series);
begin
  hlc := high + low + close;
  for i := front(volume) + 1 to back(volume) do begin
    pmf[i] := 0;
    nmf[i] := 0;
    if hlc[i] > hlc[i - 1] then
```

```

    pmf[i] := hlc[i] * volume[i];
  if hlc[i] < hlc[i - 1] then
    nmf[i] := hlc[i] * volume[i];
  end;
  apmf := sma(pmf, period);
  anmf := sma(nmf, period);
  mf := 100 * apmf / (apmf + anmf);
end.

```

Exponential Moving Average Indicator

```

/* Exponential moving average indicator */
indicator Mov_Avg_Exponential;
input src = close, period = 25;
draw res("EMA");
begin
  res := ema(src, period);
end.

```

Modified Moving Average Indicator

```

/* Modified moving average indicator */
indicator Mov_Avg_Modified;
input src = close, period = 25;
draw res("MMA");
begin
  res := mma(src, period);
end.

```

Simple Moving Average Indicator

```

/* Simple moving average indicator */
indicator Mov_Avg_Simple;
input src = close, period = 25;
draw res("SMA");
begin
  res := sma(src, period);
end.

```

Triangular Moving Average Indicator

```

/* Triangular moving average indicator */
indicator Mov_Avg_Triangular;
input src = close, period = 25;
draw res("TMA");
vars period1(number);
begin
  period1 := int(period / 2) + 1;
  res := sma(sma(src, period + 1 - period1), period1);
end.

```

Weighted Moving Average Indicator

```

/* Weighted moving average indicator */
indicator Mov_Avg_Weighted;
input src = close, period = 25;
draw res("WMA");
begin
  res := wma(src, period);
end.

```

MT Custom Indicator

```
/* MT Custom indicator */
indicator MT_Custom;
input src = close;
draw mtc("MT Custom");
begin
    mtc := sma(src, 25) - displace(sma(src, 66), 3);
end.
```

Moving Maximum Function

```
/* Moving maximum function */
function movmax;
input s(series), pos(number), period(number);
result res(number);
vars i(number), fst(number);
begin
    fst := pos - period + 1;
    if fst < front(s) then
        return;
    res := s[fst];
    for i := fst + 1 to pos do
        if s[i] > res then
            res := s[i];
end.
```

Moving Minimum Function

```
/* Moving minimum function */
function movmin;
input s(series), pos(number), period(number);
result res(number);
vars i(number), fst(number);
begin
    fst := pos - period + 1;
    if fst < front(s) then
        return;
    res := s[fst];
    for i := fst + 1 to pos do
        if s[i] < res then
            res := s[i];
end.
```

Negative Volume Index Indicator

```
/* Negative Volume Index indicator */
indicator Negative_Volume_Index;
input src = close;
draw res("NVI");
vars i(number), tmp(number);
begin
    tmp := 1000;
    res[front(src)] := tmp;
    for i := front(src) + 1 to back(src) do begin
        if volume[i] < volume[i - 1] then
            tmp := tmp * src[i] / src[i - 1];
        res[i] := tmp;
    end;
end.
```

On Balance Volume Indicator

```
/* On Balance Volume indicator */
indicator On_Balance_Volume;
input src = close;
draw res("OBV");
vars i(number), tmp(number);
begin
  tmp := 0;
  for i := front(src) + 1 to back(src) do begin
    if src[i] > src[i - 1] then
      tmp := tmp + volume[i];
    if src[i] < src[i - 1] then
      tmp := tmp - volume[i];
    res[i] := tmp;
  end;
end.
```

Parabolic SAR Indicator

```
/* Parabolic SAR indicator */
indicator Parabolic_SAR;
input af_inc = 0.02, af_max = 2.0;
draw res("Parabolic");
vars i(number), islong(bool), af(number), extreme(number), sar(number);
begin
  if back(close) >= front(close) then begin
    islong := true;
    af := af_max;
    extreme := high[front(close)];
  sar := low[front(close)];
  res[front(close)] := sar;
  for i := front(close) + 1 to back(close) do begin
    sar := sar + af * (extreme - sar);
    if islong then
      if low[i] < sar then begin
        islong := false;
        af := af_inc;
        sar := extreme;
        extreme := low[i];
      end else begin
        if extreme < high[i] then begin
          extreme := high[i];
          af := af + af_inc;
          if af > af_max then
            af := af_max;
          end;
        end
      else {if not islong}
        if high[i] > sar then begin
          islong := true;
          af := af_inc;
          sar := extreme;
          extreme := high[i];
        end else begin
          if extreme > low[i] then begin
            extreme := low[i];
          end
        end
      end
    end
  end
end.
```

```
af := af + af_inc;
if af > af_max then
af := af_max;
end;
end;
res[i] := sar;
end;
end;
end.
```

Percentage Change Culmative Indicator

```
/* Percent Change Cumulative indicator */
indicator Percent_Change;
draw res("PCC");
vars i(number), r = 0;
begin
for i := front(close) + 1 to back(close) do begin
r := r + close[i] / close[i - 1] - 1;
res[i] := r;
end;
end.
```

Percent of Resistance Indicator

```
/* Percent of resistance indicator */
indicator Percent_Of_Resistance;
input src = close, period = 12;
draw res("POR");
vars i(number), lo(number), dif(number);
begin
for i := front(src) + period - 1 to back(src) do begin
lo := movmin(low, i, period);
dif := movmax(high, i, period) - lo;
if dif > 0 then
res[i] := 100 * (src[i] - lo) / dif
else
res[i] := 100;
end;
end.
```

Williams %R Indicator

```
/* Williams' %R indicator */
indicator Percent_R;
input src = close, period = 14;
draw res("%R");
vars i(number), hi(number), dif(number);
begin
for i := front(src) + period - 1 to back(src) do begin
hi := movmax(high, i, period);
dif := hi - movmin(low, i, period);
if dif > 0 then
res[i] := -100 * (hi - src[i]) / dif
else
res[i] := 0;
end;
end.
```

Directional Movement +DI Function

```
/* Directional Movement +DI function */
function PlusDI;
input period(number);
result res(series);
vars i(number), pdm(series), tmp(number);
begin
  for i := front(high) + 1 to back(high) do begin
    tmp := high[i] - high[i - 1];
    if tmp < 0 then
      tmp := 0;
    pdm[i] := tmp;
  end;
  res := 100 * mma(pdm, period) / mma(truerange(), period);
end.
```

Positive Volume Index Indicator

```
/* Positive Volume Index indicator */
indicator Positive_Volume_Index;
input src = close;
draw res("PVI");
vars i(number), tmp(number);
begin
  tmp := 1000;
  res[front(src)] := tmp;
  for i := front(src) + 1 to back(src) do begin
    if volume[i] > volume[i - 1] then
      tmp := tmp * src[i] / src[i - 1];
    res[i] := tmp;
  end;
end.
```

Price Channel

```
/* Price Channel */
indicator Price_Channel;
input period = 10;
draw top("PC Top"), bot("PC Bot");
vars i(number);
begin
  for i := front(high) + period to back(high) do begin
    top[i] := movmax(high, i - 1, period);
    bot[i] := movmin(low, i - 1, period);
  end;
end.
```

Price Oscillator Indicator

```
/* Price oscillator indicator */
indicator Price_Oscillator;
input src = close, short_period = 1, long_period = 25, percent_mode = true;
draw res("Price Osc");
vars avg1(series), avg2(series);
begin
  avg1 := sma(src, short_period);
  avg2 := sma(src, long_period);
  if percent_mode then
    res := 100 * (avg1 - avg2) / avg2
  else
    res := avg1 - avg2;
end.
```

Price Volume Trend Indicator

```
/* Price Volume Trend indicator */
indicator Price_Volume_Trend;
input src = close;
draw pvt("PVT");
vars i(number), r = 0;
begin
  for i := front(close) + 1 to back(close) do begin
    r := r + (close[i] / close[i - 1] - 1) * volume[i];
    pvt[i] := r;
  end;
end.
```

Rate of Change Indicator

```
/* Rate of change indicator */
indicator Rate_Of_Change;
input src = close, period = 12, percent_mode = false;
draw res("ROC");
vars dis(series);
begin
  dis := displace(src, period);
  if percent_mode then
    res := 100 * (src - dis) / dis
  else
    res := src - dis;
end.
```

Relative Strength Indicator

```
/* Relative strength indicator */
indicator Relative_Strength;
input src = close, period = 14;
draw rsi("RSI");
vars i(number), u(series), d(series), au(series), ad(series), dif(number), f(number);
begin
  f := front(src);
  u[f] := 0;
  d[f] := 0;
  for i := f + 1 to back(src) do begin
    dif := src[i] - src[i - 1];
    if dif > 0 then begin
      u[i] := dif;
      d[i] := 0;
    end else begin
      u[i] := 0;
      d[i] := -dif;
    end;
  end;
  au := mma(u, period);
  ad := mma(d, period);
  rsi := 100
end.
```

RelativeVolatility Index Indicator

```
/* Relative volatility index indicator */
indicator Relative_Volatility;
input period = 14;
draw rvi("RVI");
begin
  rvi := relvol(period);
end.
```

Realative Volatility Function

```
/* Relative volatility function */
function RelVol;
input period(number);
result rv(series);
vars i(number), devhi(series), devlo(series), uhi(series), ulo(series), dhi(series), dlo(series),
  auhi(series), aulo(series), adhi(series), adlo(series);
begin
  devhi := stddev(high, 10);
  devlo := stddev(low, 10);
  for i := front(devhi) to back(devhi) do begin
    if high[i] > high[i - 1] then begin
      uhi[i] := devhi[i];
      dhi[i] := 0;
    end else begin
      uhi[i] := 0;
      dhi[i] := devhi[i];
    end;
    if low[i] > low[i - 1] then begin
      ulo[i] := devlo[i];
      dlo[i] := 0;
    end else begin
      ulo[i] := 0;
      dlo[i] := devlo[i];
    end;
  end;
  auhi := mma(uhi, period);
  aulo := mma(ulo, period);
  adhi := mma(dhi, period);
  adlo := mma(dlo, period);
  rv := 50 * (auhi / (auhi + adhi) + aulo / (aulo + adlo));
end.
```

Senkou Span Indicator

```
/* Senkou Span indicator */
indicator Senkou_Span;
draw sa("Senkou Span A"), sb("Senkou Span B");
vars ts(series), ks(series);
begin
  ts := ishimoku_func(9);
  ks := ishimoku_func(26);
  sa := displace((ts + ks) / 2, 26);
  sb := displace(ishimoku_func(52), 26);
end.
```

Simple Moving Average Function

```
/* Simple moving average function */  
function SMA;  
input s(series), period(number);  
result res(series);  
vars i(number), tmp(number), l(number), cnt(number), f(number);  
begin  
  f := front(s);  
  l := f + period - 1;  
  cnt := back(s);  
  if l <= cnt then begin  
    tmp := 0;  
    for i := f to l do  
      tmp := tmp + s[i];  
    res[] := tmp / period;  
    for i := l + 1 to cnt do begin  
      tmp := tmp + s[i] - s[i - period];  
      res[i] := tmp / period;  
    end;  
  end;  
end.
```

Standard Deviation Indicator

```
/* Standard deviation indicator */  
indicator Standard_Deviation;  
input src = close, period = 5, deviations = 1;  
draw res("StdDev");  
begin  
  res := deviations * stddev(src, period);  
end.
```

Standard Error Bands Indicator

```
/* Standard Error Bands indicator */  
indicator Standard_Error_Bands;  
input src = close, regress_period = 21, standard_errors = 2, smooth_period = 3;  
draw mid("SE mid"), top("SE top"), bot("SE bot");  
vars se(series);  
begin  
  se := sma(stderr(src, regress_period), smooth_period);  
  mid := sma(linreg(src, regress_period), smooth_period);  
  top := mid + standard_errors * se;  
  bot := mid - standard_errors * se;  
end.
```

Stark Bands Indicator

```
/* Stark_Bands indicator */  
indicator Stark_Bands;  
input factor = 1;  
draw mid("Mid SB"), upper("Upper SB"), lower("Lower SB");  
vars tmp(series);  
begin  
  mid := sma(close, 6);  
  tmp := factor * mma(truerange(), 15);  
  upper := mid + tmp;  
  lower := mid - tmp;  
end.
```

Standard Deviation Function

```
/* Standard deviation function */
function StdDev;
input s(series), period(number);
result res(series);
vars i(number), sum(number), sumsq(number), l(number), cnt(number), f(number);
begin
  f := front(s);
  l := f + period - 1;
  cnt := back(s);
  if l <= cnt then begin
    sum := 0;
    for i := f to l do begin
      sum := sum + s[i];
      sumsq := sumsq + s[i] * s[i];
    end;
    res[l] := sqrt(abs(sumsq - sum * sum / period) / period);
    for i := l + 1 to cnt do begin
      sum := sum + s[i] - s[i - period];
      sumsq := sumsq + s[i] * s[i] - s[i - period] * s[i - period];
      res[i] := sqrt(abs(sumsq - sum * sum / period) / period);
    end;
  end;
end;
```

Standard Error Function

```
/* Standard error function */
function StdErr;
input s(series), period(number);
result res(series);
vars i(number), sumx(number), sumx2(number), sumxy(number), sumy(number), sumy2(number), a(number), b(number),
l(number), totalratio(number), cnt(number), f(number);
begin
  if period <= 2 then
    return;
  f := front(s);
  l := f + period - 1;
  cnt := back(s);
  if l <= cnt then begin
    sumx := 0;
    sumy := 0;
    sumx2 := 0;
    sumxy := 0;
    sumy2 := 0;
    for i := f to l do begin
      sumx := sumx + i;
      sumy := sumy + s[i];
      sumx2 := sumx2 + i * i;
      sumxy := sumxy + i * s[i];
      sumy2 := sumy2 + s[i] * s[i];
    end;
    b := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
    a := (sumy - b * sumx) / period;
    totalratio := (sumy2 - a * sumy - b * sumxy) / (period - 2);
    if totalratio > 0 then
      res[l] := sqrt(totalratio)
  end;
end;
```

```

else
  res[] := 0;
for i := 1 + 1 to cnt do begin
  sumx := sumx + period;
  sumy := sumy + s[i] - s[i - period];
  sumx2 := sumx2 + 2 * i * period - period * period;
  sumxy := sumxy + i * s[i] - (i - period) * s[i - period];
  sumy2 := sumy2 + s[i] * s[i] - s[i - period] * s[i - period];
  b := (period * sumxy - sumx * sumy) / (period * sumx2 - sumx * sumx);
  a := (sumy - b * sumx) / period;
  totalratio := (sumy2 - a * sumy - b * sumxy) / (period - 2);
  if totalratio > 0 then
    res[i] := sqrt(totalratio)
  else
    res[i] := res[i - 1];
  end;
end;
end.

```

Stochastics Indicator

```

/* Stochastics indicator */
indicator Stochastics;
input src = close, k_period = 5, k_slow_period = 3, d_period = 3;
draw k("Fast %K"), pk("%K"), pd("%D");
vars i(number), lo(number), dif(number);
begin
  for i := front(src) + k_period - 1 to back(src) do begin
    lo := movmin(low, i, k_period);
    dif := movmax(high, i, k_period) - lo;
    if dif > 0 then
      k[i] := 100 * (src[i] - lo) / dif
    else
      k[i] := 0;
    end;
    pk := sma(k, k_slow_period);
    pd := sma(pk, d_period);
  end.
end.

```

Swing Index Function

```

/* Swing index function */
function Swing;
input limit_move(number);
result sw(series);
vars hl(number), hc(number), lc(number), co(number), r(number), k(number), i(number);
begin
  for i := front(close) + 1 to back(close) do begin
    hl := abs(high[i] - low[i]);
    hc := abs(high[i] - close[i - 1]);
    lc := abs(low[i] - close[i - 1]);
    co := abs(open[i - 1] - close[i - 1]);
    if hc >= lc then begin
      k := hc;
      if hc >= hl then
        r := hc - lc / 2 + co / 4
      else
        r := hl + co / 4;
    end else begin

```

```

k := lc;
if lc >= hl then
  r := lc - hc / 2 + co / 4
else
  r := hl + co / 4;
end;
if r = 0 then
  sw[i] := 0
else begin
  r := 50 * ((close[i] - close[i - 1]) + (close[i] - open[i]) / 2 + (close[i - 1] - open[i - 1]) / 4) * k / r / limit_move;
  if r > 100 then
    r := 100;
  if r < -100 then
    r := -100;
  sw[i] := r;
end;
end;
end.

```

Swing Index Indicator

```

/* Swing index indicator */
indicator Swing_Index;
input limit_move = 3.0;
draw res("Swing");
begin
  res := swing(limit_move);
end.

```

Triple Exponential Moving Average Indicator

```

/* Triple exponential moving average indicator */
indicator TEMA;
input src = close, period = 21;
draw res("TEMA");
vars tmp1(series), tmp2(series);
begin
  tmp1 := ema(src, period);
  tmp2 := ema(tmp1, period);
  res := 3 * tmp1 - 3 * tmp2 + ema(tmp2, period);
end.

```

TenkanSen Indicator

```

/* TenkanSen indicator */
indicator Tenkan_Sen;
draw res("Tenkan-sen");
begin
  res := ishimoku_func(9);
end.

```

Time Series Forecast Indicator

```

/* Time series forecast indicator */
indicator Time_Series_Forecast;
input src = close, period = 14;
draw res("TSF");
begin
  res := linreg(src, period) + linregslope(src, period);
end.

```

TRIX Indicator

```
/* TRIX indicator */
indicator TRIX;
input src = close, period = 12;
draw res("TRIX");
vars triple(series), dis(series);
begin
  triple := ema(ema(ema(src, period), period), period);
  dis := displace(triple, 1);
  res := (triple - dis) / dis * 100;
end.
```

Calculates True High

```
/* calculates true high */
function TrueHigh;
result r(series);
vars i(number), th(number), f(number);
begin
  f := front(high);
  if back(high) >= f then
    r[f] := high[f];
  for i := f + 1 to back(high) do begin
    th := close[i - 1];
    if th < high[i] then
      th := high[i];
    r[i] := th;
  end;
end.
```

Calculates True Low

```
/* calculates true low */
function TrueLow;
result r(series);
vars i(number), tl(number), f(number);
begin
  f := front(low);
  if back(low) >= f then
    r[f] := low[f];
  for i := f + 1 to back(low) do begin
    tl := close[i - 1];
    if tl > low[i] then
      tl := low[i];
    r[i] := tl;
  end;
end.
```

Calculates True Range

```
/* calculates true range */
function TrueRange;
result r(series);
vars truelow(number), truehigh(number), i(number), f(number);
begin
  begin
    f := front(close);
    if back(close) >= f then
      r[f] := high[f] - low[f];
    for i := f + 1 to back(close) do begin
```

```

truehigh := close[i - 1];
if truehigh < high[i] then
  truehigh := high[i];
truelow := close[i - 1];
if truelow > low[i] then
  truelow := low[i];
r[i] := truehigh - truelow;
end;
end.

```

Typical Price Indicator

```

/* Typical price indicator */
indicator Typical_Price;
draw res("Typical");
begin
  res := (high + low + close) / 3;
end.

```

Ultimate Oscillator Indicator

```

/* Ultimate oscillator indicator */
indicator Ultimate_Oscillator;
input first_period = 7, second_period = 14, third_period = 28;
draw uo("UltOsc");
vars tr(series), atr1(series), atr2(series), atr3(series), bp(series), abp1(series), abp2(series), abp3(series), i(number),
fst(number), lst(number);
begin
  tr := truerange();
  atr1 := sma(tr, first_period);
  atr2 := sma(tr, second_period);
  atr3 := sma(tr, third_period);
  bp := close - truelow();
  abp1 := sma(bp, first_period);
  abp2 := sma(bp, second_period);
  abp3 := sma(bp, third_period);
  fst := max(max(front(atr1), front(atr2)), front(atr3));
  lst := min(min(back(atr1), back(atr2)), back(atr3));
  for i := fst to lst do
    if (atr1[i] <> 0) and (atr2[i] <> 0) and (atr3[i] <> 0) then
      uo[i] := (abp1[i] / atr1[i] * 4 + abp2[i] / atr2[i] * 2 + abp3[i] / atr3[i]) / 7 * 100;
end.

```

Volatility (Chaikin's) Indicator

```

/* Volatility (Chaikin's) indicator */
indicator Volatility_Chaikins;
input mav_period = 10, roc_period = 10;
draw res("Volatility");
vars ahl(series), i(number);
begin
  ahl := ema(high - low, mav_period);
  for i := front(ahl) + roc_period to back(ahl) do
    if ahl[i - roc_period] = 0 then
      res[i] := 0
    else
      res[i] := 100 * (ahl[i] - ahl[i - roc_period]) / ahl[i - roc_period];
end.

```

Weighted Close Indicator

```
/* Weighted close indicator */
indicator Weighted_Close;
draw res("WC");
begin
res := (high + low + 2 * close) / 4;
end.
```

William's Accumulation/Distribution Indicator

```
/* Williams' Accumulation/Distribution indicator */
indicator Williams_AD;
input src = close;
draw res("WAD");
vars i(number), tmp(number);
begin
tmp := 0;
for i := front(src) + 1 to back(src) do begin
if src[i] > src[i - 1] then
if low[i] < src[i - 1] then
tmp := tmp + src[i] - low[i]
else
tmp := tmp + src[i] - src[i - 1];
if src[i] < src[i - 1] then
if high[i] > src[i - 1] then
tmp := tmp + src[i] - high[i]
else
tmp := tmp + src[i] - src[i - 1];
res[i] := tmp;
end;
end.
```

Weighted Moving Average Function

```
/* Weighted moving average function */
function WMA;
input s(series), period(number);
result res(series);
vars i(number), sum(number), sum2(number), w(number), wsum(number), l(number), f(number), cnt(number);
begin
f := front(s);
l := f + period - 1;
cnt := back(s);
if l <= cnt then begin
wsum := (period + 1) * period / 2;
sum := 0;
sum2 := 0;
w := 1;
for i := f to l do begin
sum := sum + s[i];
sum2 := sum2 + s[i] * w;
w := w + 1;
end;
res[] := sum2 / wsum;
for i := l + 1 to cnt do begin
sum2 := sum2 + s[i] * period - sum;
sum := sum + s[i] - s[i - period];
res[i] := sum2 / wsum;
end;
end.
```

ZigZag Indicator

```

/* ZigZag indicator */
indicator ZigZag;
input src = close, reversal = 5;
draw zz("ZigZag");
vars f(number), i(number), j(number), lastturn(number), hh(number), ll(number), hhpos(number), llpos(number), pos(number),
turnup(bool), turndn(bool);
begin
  f := front(src);
  if f <= back(src) then begin
    reversal := reversal / 100;
    hh := src[f];
    ll := hh;
    zz[f] := hh;
    lastturn := f;
    llpos := f;
    hhpos := f;
    for i := f + 1 to back(src) do begin
      turnup := false;
      turndn := false;
      if lastturn = f then begin
        turnup := (src[i] - ll > reversal * ll) and (src[f] - ll > reversal * ll);
        turndn := (hh - src[i] > reversal * hh) and (hh - src[f] > reversal * hh);
      end else
        if zz[lastturn] > zz[lastturn - 1] then
          turnup := (i = back(src)) or (src[i] - ll > reversal * src[i])
        else
          turndn := (i = back(src)) or (hh - src[i] > reversal * src[i]);
      if (turnup and not turndn) or (not turnup and turndn) then begin
        if turnup then
          pos := llpos;
        if turndn then
          pos := hhpos;
        for j := lastturn + 1 to pos - 1 do
          zz[j] := ((pos - j) * src[lastturn] + (j - lastturn) * src[pos]) / (pos - lastturn);
        lastturn := pos;
        hh := src[pos];
        ll := hh;
        hhpos := pos;
        llpos := pos;
        zz[pos] := ll;
        for j := pos + 1 to i - 1 do begin
          if src[j] < ll then begin
            ll := src[j];
            llpos := j;
          end;
          if src[j] > hh then begin
            hh := src[j];
            hhpos := j;
          end;
        end;
      end;
    end;
  if src[i] < ll then begin
    ll := src[i];
    llpos := i;
  end;
  if src[i] > hh then begin

```



34th Floor (CGC 34-03) 25 Canada Square London E14 5LQ
international | 44 (0) 20 7170 0770
freephone | 0800 358 0864
fax | 44 (0) 20 7170 0788

CHART STUDIO USER GUIDE

Revised: March 2007

```
    hh := src[j];  
    hhpos := i;  
end;  
end;  
pos := back(src);  
for j := lastturn + 1 to pos do  
    zz[j] := ((pos - j) * src[lastturn] + (j - lastturn) * src[pos]) / (pos - lastturn);  
end;  
end.
```



APPENDIX C: Strategy Samples

Samples

Finally we would like to present you several strategy samples.
Each sample is a CTL code of a simple trading strategy.

Reference by Name

BBS
CCI
DMI
DMI_ADXR
EMA1
EMA2
MACD
EMA1_STOP
EMA1_LIMIT
ROC
RSI
STOCHASTICS

BBS

```
strategy sample_bbs;
input period = 20, lots = 1;
begin
  if back(close) < front(close) + period - 1 then return;
  bollinger_bands(close, period, 2);
  if crossdown(close, bollinger_bands.upper) then sell(lots);
  if crossup (close, bollinger_bands.lower) then buy(lots);
  bollinger_bands(close, period, 1);
  if crossup (close, bollinger_bands.upper) then exitlong();
  if crossdown(close, bollinger_bands.lower) then exitshort();
end.
```

CCI

```
strategy sample_cci;
input period = 14, lots = 1;
vars lst = 1;
begin
  lst := back(close);
  if lst < front(close) - 1 + period then return;
  commodity_channel(period);
  if (commodity_channel.cci[lst] > -100) and (commodity_channel.cci[lst - 1] <= -100) then buy(lots);
  if (commodity_channel.cci[lst] < +100) and (commodity_channel.cci[lst - 1] >= +100) then sell(lots);
end.
```

DMI

```
strategy sample_dmi;
input period = 14, lots = 1;
vars
  lst = 1, pdi(series), mdi(series);
begin
  directional_movement(period);
  pdi := directional_movement.pdi;
  mdi := directional_movement.mdi;
  lst := back(pdi);
  if lst < front(pdi) + 1 then return;
  if pdi[lst] > mdi[lst] then buy(lots);
  if pdi[lst] < mdi[lst] then sell(lots);
end.
```

DMI ADXR

```
strategy sample_dmi_adxr;
input period = 14, lots = 1;
vars pdi(series), mdi(series), adxr(series),
      p1(number), p2(number), m1(number), m2(number);
begin
  if (back(close) < 100) or (period <= 0) then return;

  Directional_Movement(period);
  Directional_Movement_ADXR(period);
  pdi := Directional_Movement.pdi;
  mdi := Directional_Movement.mdi;
  adxr := Directional_Movement_ADXR.adxr;

  p1 := pdi[back(pdi) - 1];
  p2 := pdi[back(pdi)];

  m1 := mdi[back(mdi) - 1];
  m2 := mdi[back(mdi)];

  if (adxr[back(mdi)] >= 25) then begin
    if (p1 >= m1) and (p2 < m2) then sell(lots);
    if (p1 <= m1) and (p2 > m2) then buy(lots);
  end;
end.
```

EMA1

```
strategy sample_ema1;
input period = 10, lots = 1;
vars ma(series);
begin
  ma := ema(close, period);
  if crossup(close, ma) then buy(lots);
  if crossdown(close, ma) then sell(lots);
end.
```

EMA2

```
strategy sample_ema2;
input period1 = 18, period2 = 30, lots = 1;
vars ma1(series), ma2(series);
begin
  ma1 := ema(close, period1);
  ma2 := ema(close, period2);
  if crossup(ma1, ma2) then buy(lots);
  if crossdown(ma1, ma2) then sell(lots);
end.
```

MACD

```
strategy sample_macd;
input lots = 1;
begin
  MACD();
  if crossup(MACD.res, MACD.signal) then buy(lots);
  if crossdown(MACD.res, MACD.signal) then sell(lots);
end.
```

EMA1 Stop

```
strategy sample_ema1_stop;
input period = 10, delta=0.01, lots=1;
vars ma(series);
begin
  ma := ema(close, period);
  if crossup (close, ma) then stop_buy(lots,close[back(close)]+delta);
  if crossdown(close, ma) then stop_sell(lots,close[back(close)]-delta);
end.
```

EMA1 Limit

```
strategy sample_ema1_limit;
input period = 10, delta=0.01, lots=1;
vars ma(series);
begin
  ma := ema(close, period);
  if crossup (close, ma) then limit_buy(lots,close[back(close)]+delta);
  if crossdown(close, ma) then limit_sell(lots,close[back(close)]-delta);
end.
```

ROC

```
strategy sample_roc;
input src = close, period = 20, lots = 1;
vars roc(series);
begin
  if (back(src) < 100) or (period <= 0) then return;

  Rate_Of_Change(src, period);
  roc := Rate_Of_Change.res;

  if (roc[back(roc) - 1] <= 0) and
    (roc[back(roc)] > 0)
  then buy(lots);

  if (roc[back(roc) - 1] >= 0) and
    (roc[back(roc)] < 0)
  then sell(lots);
end.
```

RSI

```
strategy sample_rsi;
input period = 20, lots = 1;
vars lst = 1;
begin
  lst := back(close);
  if lst < front(close) - 1 + period then return;
  relative_strength(close, period);
  if (relative_strength.rsi[lst] > 30) and (relative_strength.rsi[lst - 1] <= 30) then buy(lots);
  if (relative_strength.rsi[lst] < 70) and (relative_strength.rsi[lst - 1] >= 70) then sell(lots);
end.
```



34th Floor (CGC 34-03) 25 Canada Square London E14 5LQ
international | 44 (0) 20 7170 0770
freephone | 0800 358 0864
fax | 44 (0) 20 7170 0788

CHART STUDIO USER GUIDE

Revised: March 2007

Stochastics

```
strategy sample_stochastics;  
vars lst = 1, lots = 1;  
begin  
  stochastics();  
  lst := back(stochastics.pk);  
  if lst <= front(stochastics.pk) then return;  
  if (stochastics.pk[lst] > 20) and (stochastics.pk[lst - 1] <= 20) then buy(lots);  
  if (stochastics.pk[lst] < 80) and (stochastics.pk[lst - 1] >= 80) then sell(lots);  
  if (stochastics.pk[lst] < 20) then exitlong();  
  if (stochastics.pk[lst] > 80) then exitshort();  
end.
```