Legal Disclaimer

The currency markets can do ANYTHING at ANY TIME.

No one can guarantee or forecast how these results will perform or behave in future markets.

Anyone who uses this product or this information is responsible for deciding If, Where, When and How this product and information are used.

Anyone who uses this product or this information is responsible and liable for any outcomes that might result from the use of this product or this information.

There is no warranty or guarantee provided or implied for this product or this information for any purpose.

Who am I ?

繁荣

Prosperity

Biography

Don Baechtel
Cleveland, Ohio USA
dbaechtel@gmail.com
Ph: 216-288-5200

BS degree in Computer Science and Mathematics 1977 from NMIMT, Socorro, New Mexico USA.
45 years Software Development Engineer.
Specialist in Industrial Automation, Robotics, Embedded Control, Digital Signal Processing and Artificial Intelligence.
12 years' investing experience Stocks, Options, and Forex Trading.
2+ years with Apiary Fund
Experienced in Alveo custom Scripts, Indicators and Expert Advisors (Automated trading)
Experienced in Automated Trading platforms in the Cloud using Windows Azure Virtual Machines (VM)

Passionate about writing & coaching the art and science of developing reliable and maintainable code for automating routine Forex trading tasks.

What the Class Is:

- Exchange of Information needed to develop Alveo custom Indicators
- Light coverage of C# programming needed for Alveo Indicators.
- Light coverage of Visual Studio needed to develop Alveo software.
- Hands-on examples to Build Alveo custom Indicators
- Debugging Strategies for Alveo custom Indicators
- Understanding Alveo programming limitations to have a perspective of what can or cannot be done.

What the Class Is Not:

- Extensive detail about Programming, C#, or Visual Studio
- Extensive detail about the Alveo API
- Extensive coverage of existing Alveo Indicators or ExpertAdvisors
- Details and discussions of Automated Trading
- A critique of the Apiary Fund or the Alveo trading platform

What you will receive:

1. Audio/Visual recording of the live Class.
2. A PDF Copy of the Class Notes.
3. A Visual Studio Solution that include several example custom Alveo Indicators.
4. Limited startup Support from me via email.

Goals:

- Overview of Alveo
- Review of C# programming relevant to the Alveo platform
- Review Visual Studio usage to support program development and debugging for Alveo software.
- Introduction to Alveo Code Editor
- Create a template Alveo Indicator
- Create a Visual Studio Project and Solution
- Add the template Indicator to Visual Studio Project
- Add Alveo References to Visual Studio Project
- Build Visual Studio Solution
- Transfer from Visual Studio to Alveo
- Build Indicator with Alveo Code Editor
- Run custom Indicator
- Debugging Indicators in Alveo
- Alveo Hints and Secrets
- Additional Discussions (if time permits)

Goals of Programming

- Reliability
- Readability and Maintainability
- Defensive Programming
- Separation of Concerns and Encapsulation
- Reusability
- Error Handling
- Ease of Debugging
- Design Simplicity
- Efficiency and Performance
- Documentation
- Automating repetitive tasks
- Providing a Profitable Result
- Professional Style

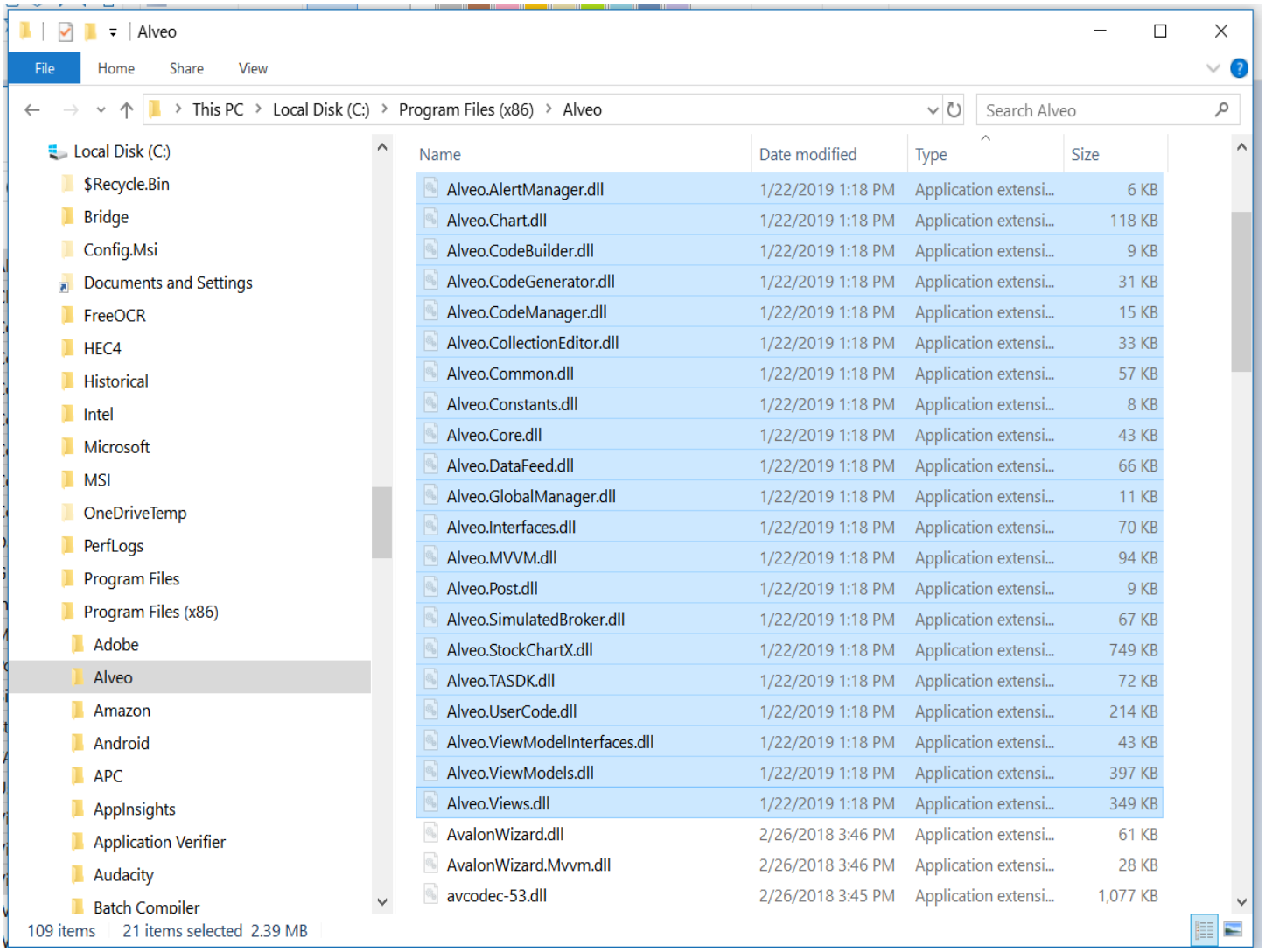
Defensive Programming

- Avoiding errors
- Initialize all variables
- Check validity of Data and Inputs
- Check for null references to objects that have not been instantiated
- Avoid Division by Zero
- Avoid Square Root of negative numbers
- Be wary of comparison for equality of floating-point values
  Instead of **if (double) A == (double)B)**
  use **if (Math.Abs(A-B) < 1e-10)**
- Add Error and Exception Handling
- Test each function
  ***** In Forex, check logic for both Long and Short sides
- Read through code with a critical eye.
  Ask yourself: What might go wrong with this code?
- Simplify Coding and Logic
- Dump data files to verify calculations and logic
- Use Visual Studio Unit Testing
- In Alveo:
  - Use Alveo Print function to save information about important events and decisions
  - Use Alveo GetLastError function to check for error returns.

What are DLLs?

- Not all your code is located in your application,
  most of the code must be dynamically linked in using various libraries of **compiled** code.
- Microsoft Dynamic Link Libraries
  - See en.wikipedia.org/wiki/Dynamic-link library
- **Dynamic-link library** (or **DLL**) is Microsoft's implementation
  of the shared library concept in the Microsoft Windows
- DLLs: Windows, .Net Framework (System), Alveo, and other sources

Alveo DLLs



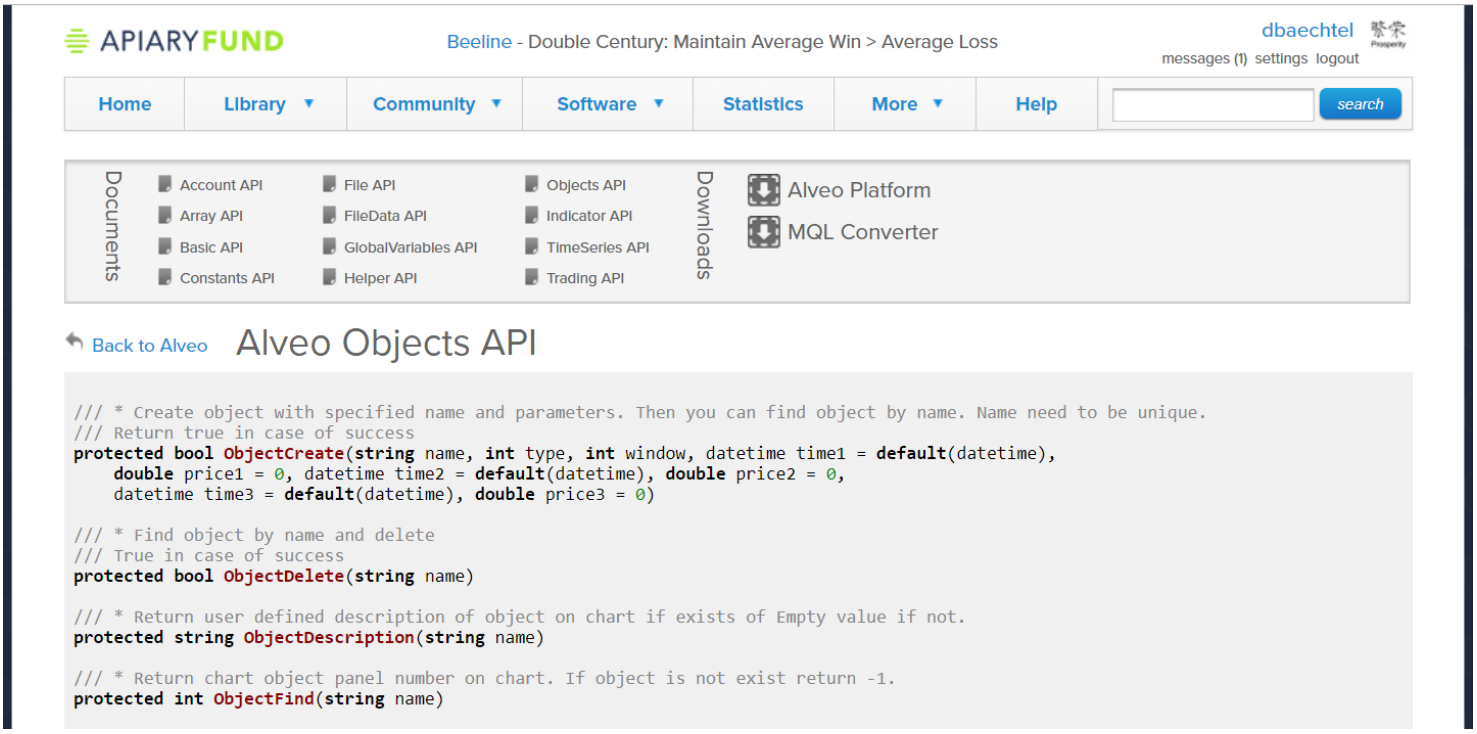
Overview of Alveo

Alveo and MT4

- MT4(MQL4) by MetaTrader is a trading platform and was designed using C++
- Alveo is an incomplete port of the MQL4 API using C# programming language
    - Many MQL4 features are missing (approximately half are missing)
    - Several functions are incomplete in implementation
    - Some functions have limited or partially changed functionality
    - Still, MQL4 API documentation can be used as a guide for Alveo functionality
    - Some MT4 Indicators, ExpertAdvisors can be migrated to Alveo
      depending on the functionality used, not all MT4 components can be ported to Alveo.
    - Several Alveo functions use Protected access and cannot be used.

Where to find Alveo API Details:

- Alveo DLLs (Visual Studio Object Browser an IntelliSense)
- apiaryfund.com/alveo/codebase/account - incomplete Alveo API doc
- github.com/marlais/Alveo – some sample code
- github.com/dbaechtel/Alveo - some sample code
- github.com/marlais/Alveo/wiki - Alveo Wiki (incomplete)
- docs.mql4.com/ - MQL4 API reference



Alveo software components

Scripts

What an Alveo Scripts does and is used for:

- An Alveo Script is a Short running function to perform some Alveo task
  Such as: Deleting all unfilled Pending Orders
- Scripts are terminated if the Chart is Restarted.

Indicators

What an Alveo Indicators does and is used for:

- Long running programs that use Market data to display information on a currency Chart.
  Such as: HMA, ATR, MACD, etc.
- Indicators are reinitialized if the Chart is Restarted.

ExpertAdvisors

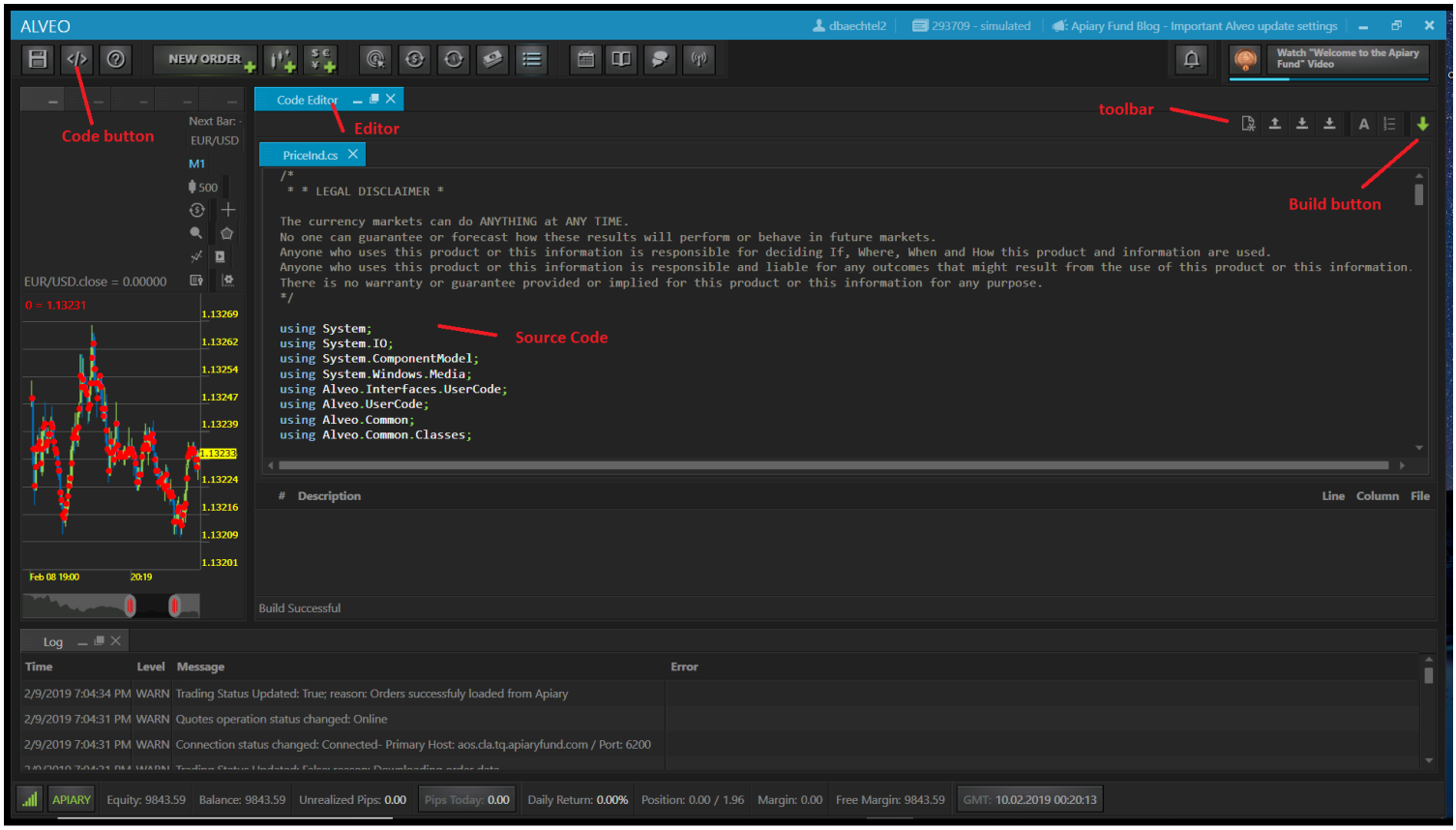
What an Alveo ExpertAdvisors does and is used for:

- Long running programs that use Market data to perform trade automation on the user's behalf.
  Such as: Automated Trading, custom Automated Trailing StopLoss, etc.
- ExpertAdvisors are reinitialized if the Chart is Restarted.

Comparison	Alveo Script	Alveo Indicator	Alveo Expert Adviser
Relative Size	short	medium	long
Complexity	simple	simple to complex	complex
Number of Tasks	1 or few	1	few to many
Parameter Settings	yes	yes	yes
Usual Lifetime	few seconds or less	continuous	continuous
Example Usage	simple function or macro	Drawing data on Chart	complex program. automated trading
Selected from	Scripts list	Indicators list	Expert Advisors list
Restarted by Alveo	no	yes	yes

Alveo Code Editor

- Designed to Load, Edit, and Build Alveo software components
- All Alveo software are compiled using the same namespace. ☹
- When one Alveo component is Built, all the components are rebuilt.

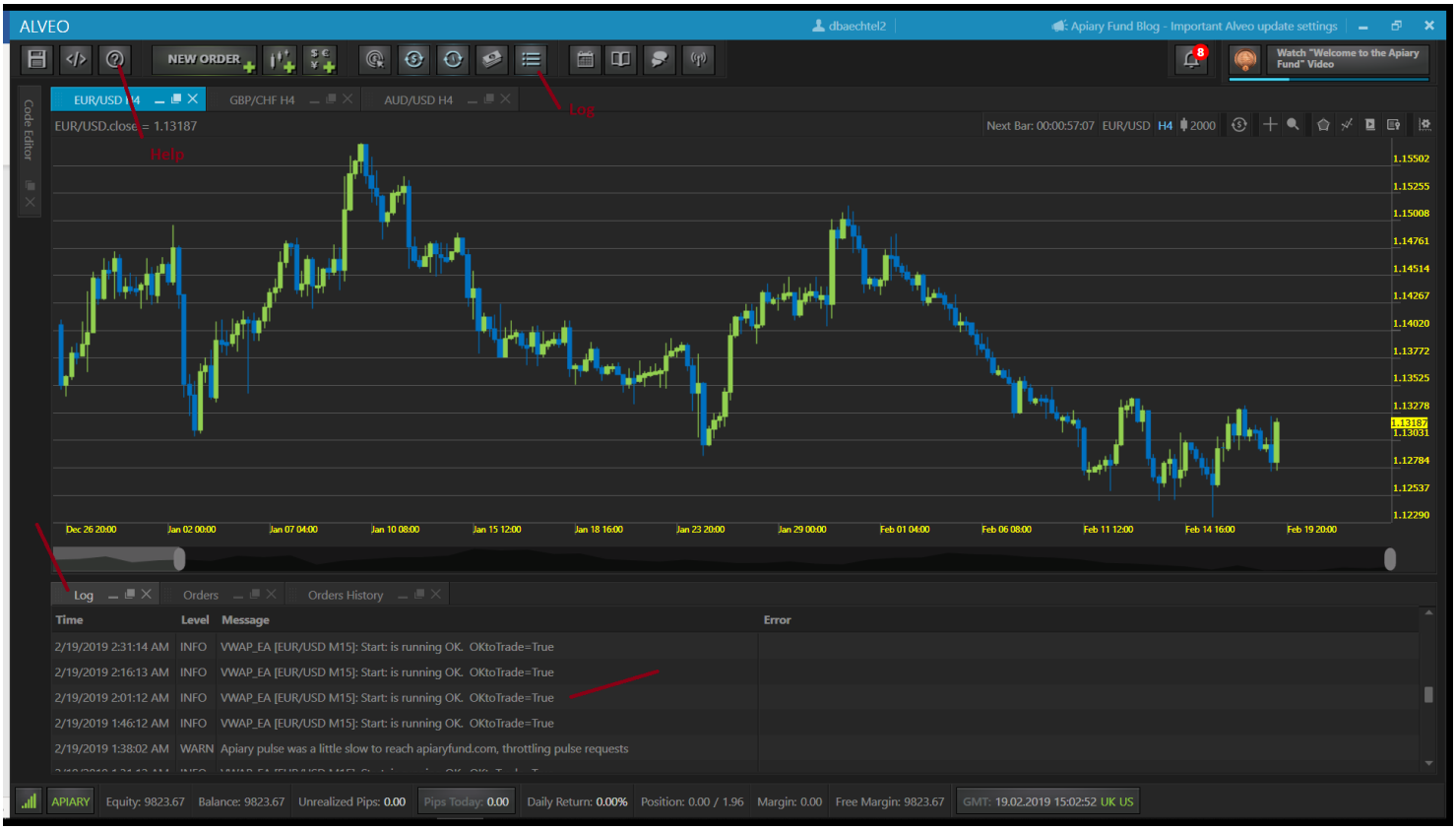


Alveo Indicator Debugging Techniques in Alveo

- Alveo Log and Log files
- Print function to Alveo Log

  public **void Print**(**string** *message*)

  example: Print("Bartime=" + b.BarTime);
- Writing Data files



Example Routine to Write messages to a Log file

```
internal void WriteLog(string header = null, string msg = null, bool clear = false)
{
    // Delete Log file if clear = true
    // Append header and msg to Log file
    string dataFileDir = "C:\\temp\\";                  // file location
    string LogFilename = this.ToString() + ".Log.csv";  // file name
    if (!System.IO.Directory.Exists(dataFileDir))       // create directory if !Exists
        System.IO.Directory.CreateDirectory(dataFileDir);
    var path = dataFileDir + LogFilename;
    if (clear)                                          // Delete log file if clear=true
        System.IO.File.Delete(path);
    if (!string.IsNullOrEmpty(header)                   // if header or msg has contents
      || !string.IsNullOrEmpty(msg))
    {
        // use StreamWriter to append header and msg lines to file and then Close the file
        System.IO.StreamWriter tradeLogFile = new System.IO.StreamWriter(path, append: true);
        if (!string.IsNullOrEmpty(header))
            tradeLogFile.WriteLine(header);
        if (!string.IsNullOrEmpty(msg))
            tradeLogFile.WriteLine(msg);
        tradeLogFile.Close();
    }
}
```

Example invocations of WriteLog Method

`WriteLog`(msg: “This will be the first line in the file.”, clear: **true**);

`WriteLog`(msg: “The current Price=” + curPrice;

`WriteLog`(msg: “The current ATR=” + ATR.value.ToString(“F5”);

Alveo Hints

- Indicator Filenames **must** be the same as Indicator name
- Copy (missing) pdf_js.pak file to C:\Program Files (x86)\Alveo directory.
- Alveo symbol and timeframe arguments not implemented. ☹
    - This means that Alveo Indicators and EAs cannot access data from other symbols or timeframes. ☹
- Alveo Arrays
    - ArrayIsSeries = true means indexing is **Reverse,** Array[0] is most recent.
    - See: https://docs.mql4.com/array
- Test snippets of Alveo code before developing too much.
    - Some things don't work the way that you expect.
    - Do not invest too much effort in code without testing it.
- Some Windows functionality is not supported by Alveo
    - Example: Http, XML, Serialization and others
- Not all MQL4 MarketInfo types are implemented in Alveo
- Not all MQL4 Account Information is implemented in Alveo.
- MQL4 Global Variables are not implemented in Alveo
- MQL4 Chart Operations are not implemented in Alveo
- Not all MQL4 Object Functions are implemented in Alveo
- Not all Windows features work in Alveo.
- Not all C# features work in Alveo.

Unmanaged and Managed Code

- Unmanaged code is the good old C or C++ with no CLR support, therefore unmanaged code does not have a garbage collector and you will have to **keep track of all your memory allocations to avoid memory leaks**. Also, when you build an unmanaged project in Visual Studio, the resulting library or executable is written directly on machine code, therefore it doesn't need the **.NET Framework** to run.
- The managed C# language has CLR support this **allows you to use the Garbage Collector and the .NET Framework classes**.
- Also, when you build a managed project in Visual Studio, the resulting library or executable is written in CLR code (which is then translated to machine code by the CLR), therefore it needs the **.NET Framework** installed to run.
- Writing in managed C# code is **easier** and **has more functionality** than unmanaged code.

Review of C# Programming

Namespace

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc.) inside it.
- Namespaces are used to organize code into logical groups and to **prevent name collisions** that can occur especially when your code base includes multiple libraries.
- All identifiers at namespace scope are visible to one another without qualification.
- Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.
- Example: System.Console.WriteLine("Hello World!");
- The **using** keyword can be used so that the complete name is not required, as in the following example:

```
#using System;
…
Console.WriteLine("Hello");
```

C# preprocessor directives

- See: docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/#using
- As a directive, when #using is used to create an alias for a namespace
  or to import types defined in other namespaces
- #region, #endregion used for outlining

Attributes

- An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program.

```
[Serializable]

[Description("description of stuff")]

[Category("Settings")]

[DisplayName("name")]
```

About Variables, Global and Local

- Scoping rules
- Type casting
  Like: decimal <-> double

Example:

```
double closePrice = (double)ChartBars[1].Close;
```

Object creation and garbage collection

- new statement to create a new instance of an object
  Example: Queue<double> myQ = **new** Queue<double>();
- Garbage collection

Statements

- Variable declarations
- Assignment statements
- Control Flow
  - If/then
  - For loop
  - Foreach
  - While Loop
  - Switch
- Code Blocks with braces { }
- Class and Class declarations
  - Variables
  - Methods
- Exception Handling
  - try, catch, finally, throw

Function Calls, Arguments and return

- Named arguments
  Example: `public double GetThePrice(PriceTypes type, Bar );`
- Default argument values
  Example: `internal void WriteLog(string header = null, string msg = null, bool clear = false)`
  Example Invocation: `WriteLog(msg: "Hello there.", clear: true);`

Type and member definition Modifiers

- See: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/modifiers

System.Collections.Generic

- See: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netframework-4.7.2
  - Such as: List<T>, Queue<T>, Dictionary<T,T>, Stack<T>, etc.

System.Linq

- The System.Linq namespace provides classes and interfaces that support queries that use Language-Integrated Query (LINQ).
  - Such as: Average, Contains, Count, Max, Min, Sum
  - Example:

```
Queue<double> sma = new Queue<double>();
sma.Enqueue(1.23);
sma.Enqueue(2.34);
sma.Enqueue(4.56);
 var count = sma.Count;   // count = 3
var smaSum = sma.Sum();   // smaSum = 1.23 + 2.34 + 4.56 = 8.13
var smaValue = sma.Average();   // smaValue = (1.23 + 2.34 + 4.56) / 3 = 2.71
var min = sma.Min();   // min = 1.23
```

Naming conventions

In computer programming, a **naming convention** is a set of rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation.

See: en.wikipedia.org/wiki/Naming convention_(programming)

Choose a naming convention and try to stick with it.

Code Indenting, Edit > Advanced > Format Document

In computer programming, an **indentation** style is a convention governing the **indentation** of blocks of **code** to convey program structure. ... **Indenting** is not a requirement of most programming languages, where it is used as secondary notation. Rather, **indenting** helps better convey the structure of a program to human readers.

See: https://en.wikipedia.org/wiki/Indentation_style

Code Comments

Not all code lines need a Comment.

Comments should be used to help readability and aid understanding

.Net Framework

See: https://en.wikipedia.org/wiki/.NET_Framework

.NET Framework (pronounced as "*dot net*") is a software framework developed by Microsoft that runs primarily on Microsoft Windows.

It includes a large class library named Framework Class Library (FCL) and provides language interoperability (each language can use code written in other languages) across several programming languages.

Review Visual Studio usage

Solutions

Collection of one or more Projects

Projects, Project Types, Project Properties

Collection of one or more elements to build executable items.

Solution Explorer

- Folders
- Add > New > Item
- Add > Reference

Create new Visual Studio Solution, Project, and C# Class

- Visual Studio Installer, Launch
- File > New > Project > Visual C# > Console App (.Net Framework)
- Project Name, Solution Name
- Solution Explorer > Project > Add > New Folder > Rename
- Folder > Add > New Item > Visual C# > Class > Name (Indicator Name)

Object Browser

- See: https://en.wikipedia.org/wiki/Object_browser
- An **Object Browser** is tool that allows a user to examine the components involved in a software package, such as Microsoft Word or software development packages.
- An **object browser** will usually display the hierarchy of components; the properties and events associated with the objects; and other pertinent information; it also provides an interface for interacting with objects.

Source Code Navigation

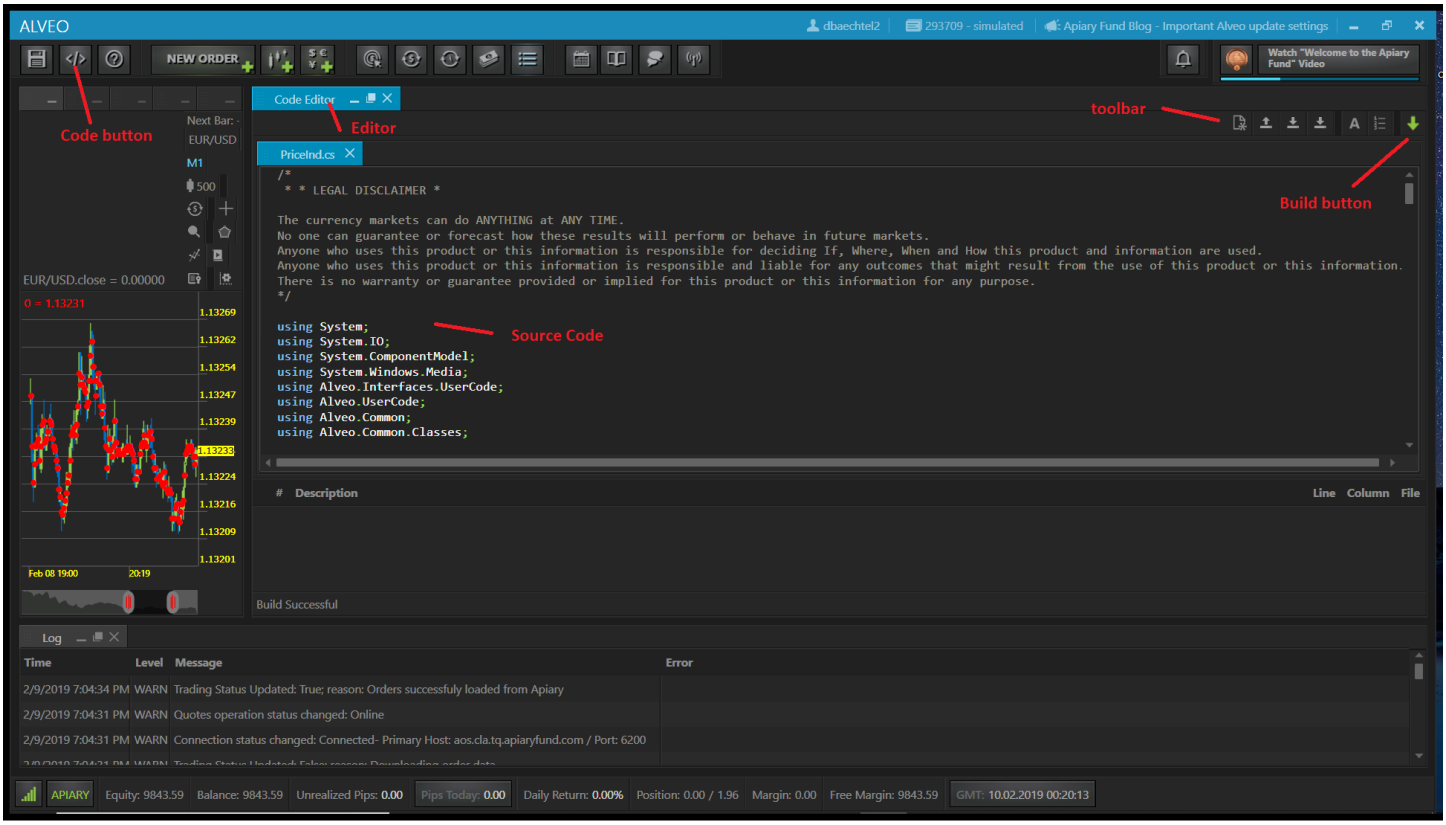
- Bookmarks, Find and Replace, Find In Files
- Find All References, Go To Definition, Peek Definition

Debugging in Visual Studio

- Run, Pause, End
- Breakpoints
- Continue, Step Into, Step Over, Step Out
- QuickWatch
- Watch Window
- Call Stack
- Writing to Console

Using the Alveo Code Editor

- Alveo Code Editor is used to Edit and Build Alveo software components
- Alveo Code Directories
- Open, Save, Line Numbers, Build
- New, Edit



Parts of an Alveo Indicator

#using statements

Indicator class
declaration

User Settings
and class variables

Class constructor
(initialize class)

Init( )
(initialize variables)

Deinit( )
(termination cleanup)

Start( )
(display computations)

IsSameParameters( )
SetIndicatorParameters( )

Indicator calculations
(internal class object)

Create a template Alveo Indicator for Visual Studio

- In Alveo, Code > Indicator > New
  - Fill out Name
  - Define Series (optional)
- Select Alveo Indicator and use Cntrl-A and Cntrl-C to Copy ALL to Paste Buffer
- In Visual Studio display the Indicator Source file
- Select Indicator file and use Cntrl-A to select All and Cntrl-V to Copy Paste Buffer to file
- Rebuild the VS Solution
- In VS Project, Add > Reference to Alveo DLLs

Alveo Indicator Indexes (Series) and Levels

- **Indexes (Series) are buffered data to be displayed on the Chart**
- **Levels are constant value horizontal lines to be displayed on the Chart**

Anatomy of Alveo custom Indicator

- **#using Directives**
- **namespace**
- **class definition, IndicatorBase**
- **Class Properties, variables**
- **Class Constructor**
- **Alveo Entry Points**
  - *Class Constructor*
  - *Init*
  - *Deinit*
  - *Start*
  - *IsSameParameters*
  - *SetIndicatorParameters*

Important Alveo Indicator functions and variables

IndicatorBuffers - Number of buffers to be allocated.

IndicatorShortName - Sets the "short" name of a custom indicator

SetIndexBuffer - Binds a specified indicator buffer with one-dimensional dynamic array of the type double.

SetIndexLabel - Sets drawing line description.

SetIndexStyle - Sets the new type, style, width and color for a given indicator line.

SetIndexArrow- Sets an arrow symbol for indicators line of the DRAW_ARROW type.

IndicatorCounted - Returns the number of bars not changed after the indicator had been launched last.

indicator_chart_window – Boolean that determines if Indicator is displayed in chart window or separately.

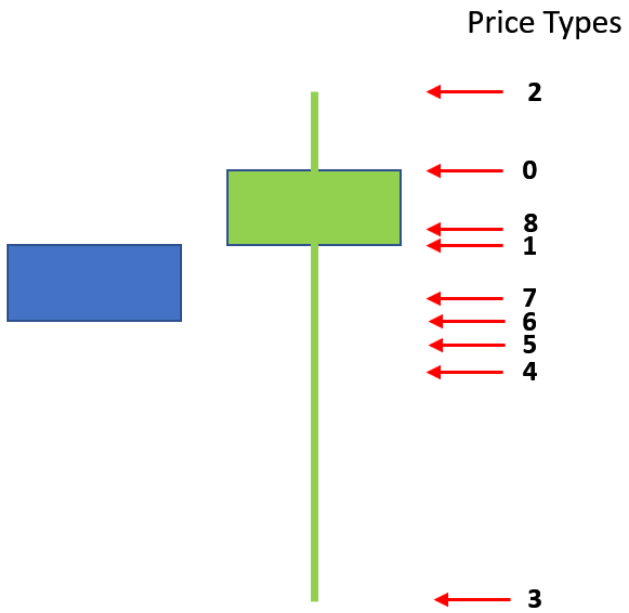
Introduction to Price Type P7

Alveo PriceTypes enumeration

```
public enum PriceTypes
{
    PRICE_CLOSE = 0,
    PRICE_OPEN = 1,
    PRICE_HIGH = 2,
    PRICE_LOW = 3,
    PRICE_MEDIAN = 4,      // (b.high + b.low) / 2 (i.e. Mid Bar)
    PRICE_TYPICAL = 5,     // (b.high + b.low + b.close) / 3
    PRICE_WEIGHTED = 6,    // (b.high + b.low + 2 * b.close) / 4
    PRICE_OHLC = 7,        // (b.open + b.high + b.low + b.close) / 4
    PRICE_P7 = 8           // (b.low + b.high + 2 * b.open + 3 * b.close) / 7
}
```

A new PriceType P7

P7 = MidBar + 2*MidBody + Close = (High + Low + 2*Open + 3*Close)/7



Goals of P7 Price

- Better represents average price movement
- Smoother than Close, Median (MidBar), Typical, OHLC, or Weighted
- Reduce influence of wicks High or Low
- Biased towards recent Close Price.
- Better Price value for Indicators and ExpertAdvisors.

Creating a Price Indicator (PriceInd) from the Indicator Template code

- Create New Indicator Class file for PriceInd
    - Solution Explorer > Indicators > Add New Item > C# > Class > Name
- Transfer Template source to PriceInd file
- Change class name to PriceInd
- Add PriceTypes enum
- Add PriceType User Setting and initialize
- Update IsSameParameters and SetIndicatorParameters functions for User Settings
- Add display buffer array (Series)
- Add Indicator functions
- In Start(), check if display update needed
- In Start(), Calculate value to display
- Update display buffer value(s)
- Add Exception Handling
- Add Comments and Descriptions
- Edit > Advanced > Format Document
- Review entire PriceInd code
- Transfer PriceInd file to Alveo, Build, Run on Chart, and Test
- Check Printed messages in Alveo Log

Comma Separated Variables (CSV) formatted file

- Simple Format: variables separated by commas

Example file: *H1,H2,H3*
2,4,6.6
1,3,7.2

Loaded in Excel

A1 | H1

	A	B	C	D
1	H1	H2	H3	
2	2	4	6.6	
3	1	3	7.2	
4				

- Human and Computer readable
- Can Load automatically in Excel for analysis
- Numbers, Strings, Dates, Time
- Easy to Generate in Code
- Can be used to dump data, results of calculations and logic

Using Embedded C# Class Objects

Advantages:

- Separation of Concerns
  Each Class has a dedicated purpose.
- Encapsulate code necessary for Indicator calculations separate from Alveo Indicator infrastructure.
- Embedded Class can contain variables, methods, and events needed for the embedded Class object.
- Embedded Class can provide access to more that one result data item.
  Examples:
  * internal double value // result
  * internal double prevValue // previous value
  * internal bool isRising // value Slope is Rising
  * internal bool isFalling // value Slope is Falling
  * internal DateTime LastUpdate // date and time of last update
- Embedded Class can contain additional embedded classes.
  Example: Alveo Indicator > ATR Class > SMA Class
- Embedded Classes can be reused for Indicators, Expert Advisors, Scripts

Coding Embedded C# Class Object

```
#using …                                    // libraries needed
namespace Alveo.UserCode                    // namespace
public class VWAP3 : IndicatorBase          // Alveo Indicator declaration
{
    VWAPobj vwap;                           // Internal VWAPobj
    Bar theBar;
    …
    public VWAP3()                          // parameterless (default) constructor for VWAP3 called by Alveo
    {
    }

    protected override int Init()       // Alveo EA Init entry point called when Chart is refreshed
    {
            vwap = new VWAPobj(this, …);  // Instantiate VWAPobj
            …
            return 0;
    }

    protected override int Deinit()     // Alveo Deinit entry point for termination cleanup
    {
            return 0;
    }

    protected override int Start()      // Alveo Start entry point called every Tick & closed Bar
    {
            …
            theBar = ChartBars[1];                      // most recently closed Bar
            double VWAPvalue = vwap.Calc(theBar);       // VWAPvalue to be displayed on the Chart
            return 0;
    }

    internal class VWAPobj              // embedded VWAPobj Class object
    {
            …                           // variables
            VWAP3 Parent;
            internal double value;

            internal VWAPobj()          // parameterless (default) constructor for VWAPobj
            {
                    // initialize VWAPobj variables
                    Parent = null;
                    value = 0;
            }

            internal VWAPobj(VWAP3 parent, …) : this()   // constructor for VWAPobj w/ parameters
            {
                    Parent = parent;
                    …
            }

            internal void Init(ref Bar b)  // Initialize VWAPobj using data from Bar
            {
                    …
                    value = (double)b.Close;
                    Parent.Print("VWAPobj initialized.");
            }

            internal double Calc(ref Bar b)  // Calculate VWAPobj value suing Bar b and return double
            {
                    if (value == 0) Init(ref b);
                    value = …                // VWAPobj Calculations
                    return value;
            }
    }
}
```

Alveo Bar Data type

```
namespace Alveo.Common.Classes
{
    public class Bar
    {
        public Bar();

        public DateTime BarTime { get; set; }
        public decimal Open { get; set; }
        public decimal High { get; set; }
        public decimal Low { get; set; }
        public decimal Close { get; set; }
        public long Volume { get; set; }

        public Bar Clone();
    }
}
```

Example Usage:

Bar b = ChartBars[1]; // most recently Closed Chart Bar
Print("Most recently closed Bar, time=" + b. BarTime + " Close Price=" + b.Close);