AmiBroker Development Kit

ADK Revision 2.10a. 4 August, 2010

* added VS2005 x64 configuration for 64-bit AFL 'Sample' plugin
* added #pragma pack statement to plugin.h to force AmiVar packing on x64 systems

ADK Revision 2.10. 10 April, 2010

* struct PluginNotification has new field
StockInfo * pCurrentSINew
and old field pCurrentSI type has changed to StockInfoFormat4 *

ADK Revision 2.00. 6 August, 2009

added support for
* 64-bit date time format
* float volume/open int
* 2 user fields (Aux) in Quotation structure
* 100 new user fields (fundamentals) in StockInfo
* proper alignment for 64-bit platforms (8 byte boundary)

ADK Revision 1.10. 20 November, 2002

Initial release

ADK Source Code license:

1 BASICS

1.1 INTRODUCTION

AmiBroker plug-in DLLs are regular Win32 dynamic link libraries. Currently two types of plugins are supported by AmiBroker plugin interface:

- AFL plugins

- Data plugins
- Optimizer plugins

The AFL plugins can expose unlimited number of functions to the AFL engine. The functions provided by the plugin are integrated tightly with AFL engine so there is no difference in performance or functionality between built-in functions and the ones provided by the plug-in.

Data plugins allow to link any external data source. The interface allows to use fast, local data sources like file-based databases (Metastock files, ASCII files, FastTrack, Quotes Plus) as well as internet-based feeds (eSignal, myTrack, QuoteTracker). Both end-of-day and intraday modes are supported. Data plugins also support various notifications so plugin can notify AmiBroker that new data just arrived and AmiBroker can notify the plugin about settings change and/or user action.

The plug-ins can be created in any language that meets the following requirements:

- ability to build regular API-like 32bit or 64 bit Windows DLL (non ActiveX/COM)
- support for user defined datatypes, structures and unions
- support for pointer data type, pointers to functions, calling functions by pointer
- support for _cdecl calling convention

Visual C++ 6.0 was used to create all source code examples in the AmiBroker Development Kit. However you can use other development platform and/or language. To demonstrate this AFL plugin examples include also project (.dev) files for Bloodshed DevC++ package (http://www.bloodshed.net). DevC++ is a FREE development package that can be used to build plugins for AmiBroker. It produces slightly larger binaries than VisualC++ but other than that it is fully compatible. To compile AFL samples just install DevC++ package and then double click on supplied .dev file, then choose Execute->Compile from DevC++ IDE.

Note to Delphi users: Delphi does not support returning 8 byte structures by register so this is an obstacle in developing plugins in Delphi. This could be solved by little stub assembly code. Please contact me if you are interested adk@amibroker.com

1.2 INTERFACE ARCHITECTURE

Plugin interface is designed to be as simple as possible. AFL plugins need to export just 5 functions to be fully functional. The simplest data plugin needs only 4 functions. The full definition of interface is included in Plugin.h header file and below you will find all necessary information to get going with the development.

1.2.1 Background

To provide maximum flexibility AmiBrokers plug-in interface must provide the methods for two-way communications between AmiBroker and plugin. AmiBroker as the process that loaded the DLL can call the functions exported by the DLL but also the DLL has to have a way to call back AmiBroker functions. For this purpose AmiBroker provides a "site interface" which is a structure containing function pointers that can be used to call back internal AmiBroker code. Data plugins have also ability to send messages to AmiBroker main window notifying about the updates.

Each plugin DLL **must** export at least one function called **GetPluginInfo()**. DLLs without this function are ignored by AmiBroker. GetPlugin info function provides basic information about the plugin needed by AmiBroker to access all remaining functions.

When AmiBroker starts (or when a "Load" button is pressed in the Plugins window) the following things happen:

- "Plugins" folder is scanned for the files with .DLL extension
- each DLL file is examined if it exports **GetPluginInfo()** function. If there is no such function, this DLL is not a valid AmiBroker plugin. If **GetPluginInfo()** function is found, it is called to examine the name and properties of the plug-in. After successful check the DLL is added to the AmiBroker's internal plugin table. At this stage the type of the plugin is determined. From then on AFL plugins are treated differently than data plugins (different functions are called)

After this step DLL is loaded but waits for the second initialization phase.

For AFL plugins this second initialization phase happens when AFL engine starts for a very first time initializing its function table. Then AmiBroker performs the following operations:

- AmiBroker calls **SetSiteInterface()** function exported by the plug in. The site interface is used later in DLL for calling back various AmiBroker functions (including allocating/freeing memory, reading/writing AFL variables, calling back AFL functions)
- AmiBroker calls **Init()** function from the plug in DLL that should be used for initializing working variables/allocating extra memory if necessary
- AmiBroker calls **GetFunctionTable()** from the plug in DLL. In this step the AFL functions provided by the DLL are added to the internal AmiBroker dispatch tables allowing futher calls of these functions.

Note that this is done only once after loading DLLs in the newer versions of AmiBroker (4.10 and up).

For Data plugins the second initialization phase happens when given data source is selected for the very first time in current AmiBroker session. Then AmiBroker just calls **Init()** function from the plugin that should be used for initializing working variables/allocating extra memory if necessary. No other function is called for data plugins at this time.

After such initialization process the plugin is ready to be used. Next actions depend on type of the plugin.

For AFL plugins if any external function call is included in the formula being parsed by AFL engine, AmiBroker finds appropriate pointer to the function in its dispatch table and calls either internal code or the code found in one of the plug-in DLLs.

For Data plugins AmiBroker may call different functions descibed in Data plugin section of this document.

The final stage is performed when AmiBroker exits:

- for each plug-in DLL Release() function is called which should release all the resources allocated via Init() call in the second phase.

2 AFL PLUGINS

2.1 FEATURES

AFL plug-in DLLs have the following features:

- ability to extend built-in function set with a new functions that run at the compiled code speed (the same as built-in functions), functions can have unlimited number of arguments
- ability to overwrite built-in functions
- ability to call back any built-in AFL function
- ability to call back the functions defined in other plugins

- ability to read, write and create the AFL variables
- easy access to stock arrays (open, high, low, close, volume and open interest)
- automatic syntax highlighting of all functions exposed by the plugin
- dynamic loading/unloading the plugins at run time without the need to restart AmiBroker
- unlimited number of plugins can be loaded

2.2 INTERFACE DEFINITION

2.2.1 Data types

One of the most important structures is AmiVar structure. It is used for holding different types of values. This single structure can hold floating point number, the array of floating point numbers, a string or IDispatch interface pointer. Each AFL function receives its arguments as an array of AmiVar values. The AmiVar structure looks like this:

```
typedef struct AmiVar
{
          int type;
          union
          {
                    float val;
                    float *array;
                    char *string;
                    void *disp;
          };
} AmiVar;
```

The first member of the structure - **type** - holds the information about data type which can be one of the following:

```
// the list of AmiVar types
enum { VAR_NONE, VAR_FLOAT, VAR_ARRAY, VAR_STRING, VAR_DISPATCH };
```

VAR_NONE represents the AmiVar that does not have any value. If **type** is VAR_FLOAT it means that **val** member is valid and hold the floating point number. If **type** equals VAR_ARRAY it means that **array** member is valid and points to the array of floating point values (the size of the array is the same for all arrays during single function call and can be obtained from site interface - which will be described later). If **type** equals to VAR_STRING it means that **string** member is valid and points to null-terminated C-style character string. The VAR_DISPATCH type is provided for calling COM objects and will not be covered here.

Proper usage of AmiVar structure looks like this:

```
AmiVar myvar;
myvar.type = VAR_FLOAT; // set the type
myvar.val = 10.5f; // assign floating point number
```

Please note that assigning arrays and strings usually require allocating memory. These allocations must be performed using special allocator functions provided by the site interface in order to enable AmiBroker to track these allocations and free the memory when it is no longer needed.

The next important structure is the SiteInterface:

```
struct SiteInterface
{
                                        int                             nStructSize;
                                        int                             (*GetArraySize) (void);
                                        float *                 (*GetStockArray)( int nType );
                                        AmiVar                  (*GetVariable) ( const char *pszName );
                                        void                    (*SetVariable) ( const char *pszName, AmiVar
newValue );
```

```
                                        AmiVar              (*CallFunction) ( const char *szName, int
nNumArgs, AmiVar *ArgsTable );
                                        AmiVar              (*AllocArrayResult) (void);
                                        void *              (*Alloc) (unsigned int nSize);
                                        void                (*Free) (void *pMemory);
};
```

The site interface is provided to the DLL to enable calling back AmiBroker's internal routines from the plug in side. The site interface provides a set of function pointers that

- give you an access to the stock arrays (GetArraySize, GetStockArray)
- allow reading and writing AFL variables (GetVariable, SetVariable)
- allow calling back AFL built-in functions (CallFunction)
- manage memory allocation so it is tracked correctly by AFL engine (AllocArrayResult, Alloc, Free)

The pointer to the site interface is set during SetSiteInterface() function call during second stage of the initialization process (described above). It happens before call to Init() so you can use all these pointers even in the Init() function.

A detailed description on how to use the functions provided by the site interface is given later on in this document.

Also important is a function descriptor structure - FunDesc:

```
// FunDesc structure
// holds the pointer to actual user-defined function
// that can be called by AmiBroker.
// It holds also the number of array, string, float and default arguments
// for the function and the default values
//
typedef struct FunDesc
{
    AmiVar (*Function)( int NumArgs, AmiVar *ArgsTable );
    UBYTE   ArrayQty;       // number of Array arguments required
    UBYTE   StringQty;      // number of String arguments required
    SBYTE   FloatQty;       // number of float args
    UBYTE   DefaultQty;     // number of default float args
    float   *DefaultValues; // the pointer to defaults table
} FunDesc;
```

This structure is used to create a function table that is retrieved by GetFunctionTable. Each entry of this table contains a function name (visible in AFL) and the function descriptor structure shown above. The function descriptor contains a function pointer (**Function** member) which is used to call actual function when it is referenced from AFL formula. It works in the following way:

- AFL engine during parsing detects a function call
- the name of the function is searched in the symbols table
- if it is found and the symbol represents existing function (either built-in or external defined in plugin) the FunDesc structure linked to the symbol is used first for checking if arguments are correct and then to call the appropriate function via the pointer stored in FunDesc structure

In addition to the function pointer, the numbers of arguments of different types are also stored in FunDesc structure for checking argument count and types at run time. Note that AFL allows unlimited number of arguments but the order is fixed: first come the array arguments, then string arguments, then numeric arguments with no defaults and at the end - numeric arguments with default values. The number of arguments of each kind is defined in **ArrayQty**, **StringQty**, **FloatQty**, **DefaultQty** members respectively. The pointer to the array of default values is stored in **DefaultValues** member (this can be NULL if **DefaultQty** is zero

2.2.2 Interface functions

A valid AmiBroker AFL plug-in DLL **must** export the following functions:

```
PLUGINAPI int GetPluginInfo( struct PluginInfo *pInfo );
```

PLUGINAPI int Init(void);
PLUGINAPI int Release(void);

PLUGINAPI int GetFunctionTable(FunctionTag **ppFunctionTable);
PLUGINAPI int SetSiteInterface(struct SiteInterface *pInterface);

The GetPluginInfo() function is used for obtaining the information about the plugin (the name, vendor name, type, min allowed AmiBroker version) - you should provide accurate information in your DLL for easy identification of your plugin in the "Plugins" window in AmiBroker.

Init() and Release() functions are provided to allow extra memory allocation/other resource initialization in the DLL.

SetSiteInterface() function is called by AmiBroker to set the pointer to the SiteInterface structure in your DLL. Using this pointer you can call back various internal AmiBroker functions.

GetFunctionTable() function is called by AmiBroker to retrieve the table of function names/descriptors describing the AFL functions exposed by your DLL.

2.3 CREATING YOUR OWN AFL PLUGIN DLL

Creating your own plug-in DLL is quite simple. If you are using Visual C++ 6 you should do the following:

1. Choose File->New from the menu.
2. From the list of available projects choose "Win32 Dynamic-Link Library" and type the project name, for example "MyPlugin", then click "OK"
3. In the page "Win32 Dynamic-Link Library - Step 1 of 1" choose "A simple DLL project" - this will create a project file and three source code files - MyPlugin.cpp, StdAfx.h, StdAfx.cpp
4. Now copy "Plugin.cpp", "Plugin.h" and "Functions.cpp" files from the Sample plugin DLL source code folder to your project folder
5. Choose Project->Add to project->Files... menu. From the file dialog please choose "Plugin.cpp", "Plugin.h" and "Functions.cpp" files and click OK. Now these files are added to the project and you can build it.

After these steps you have functional copy of a Sample project with your own name (MyPlugin). From now you can modify project files.

The only file you really need to modify is "Functions.cpp" file that actually implements the functions that your plug in will expose to AFL. You should leave "Plugin.cpp" and "Plugin.h" files untouched (with one exception). The exception is that you should modify your plugin name, vendor and version information defined in lines 23-25 of Plugin.cpp:

```
#define PLUGIN_NAME "MyPlugin - enter here real name of the plugin"
#define VENDOR_NAME "Your name"
#define PLUGIN_VERSION 010000
```

The information defined here is displayed by the AmiBroker in the Plugins window so it is important to give the user correct information. Please do not forget to do that.

2.4 IMPLEMENTING YOUR OWN AFL FUNCTIONS

Now the only work which is left to do is to implement your functions in "Functions.cpp" file. It is quite good idea to use already written code supplied with a Sample DLL as a starting point for your modifications.

2.4.1 Defining function table

Function exposed by your plug-in must be listed in the functions table that is retrieved during plug-in intialization using GetFunctionTable call. The function table looks like this:

```
// Each entry of the table must contain:
// "Function name", { FunctionPtr, <no. of array args>, <no. of string args>, <no. of float args>, <no. of
default args>, <pointer to default values table float *>
FunctionTag gFunctionTable[] = {
   "ExampleMACD", { VExampleMACD, 0, 0, 0, 0, NULL },
   "ExampleMA",  { VExampleMA, 1, 0, 1, 0, NULL },
   "ExampleEMA", { VExampleMA, 1, 0, 1, 0, NULL }
   };
```

Each entry in this table contains a string that defines the name of the function as seen by AFL engine and the FunDesc structure that defines pointer to the function itself and the arguments required by the function:

"NameOfYourFunction", { ptrToYourFunction, num_of_array_args, num_of_string_args, num_of_float_args, num_of_default_args, ptr_to_default_values },

AFL engine uses this information during parsing of your AFL formula to check if the function with given name exists, to check, parse and cast arguments to appropriate types and finally to call the function.

If you want to create a function that accepts 1 array and 1 float argument with no default value the function table entry will look like this:

```
FunctionTag gFunctionTable[] = {
"MyFunction", { MyFunction, 1, 0, 1, 0, NULL },
...
```

On the other hand if you want to have a default value of 15 for float argument you would need to write:

```
float myfunction_defaults[] = { 15.0f };
FunctionTag gFunctionTable[] = {
   "MyFunction", { MyFunction, 1, 0, 0, 1, myfunction_defaults },
   ...
```

2.4.2 Defining functions

Every function exposed by your plugin must have the following prototype:

```
AmiVar MyFunction( int NumArgs, AmiVar *ArgsTable )
```

It means that it gets the pointer to the arguments table (*ArgsTable), the number of elements in this array (NumArgs) and returns the AmiVar value.

In case of functions that don't need any argument NumArgs is zero and ArgsTable has no allocated elements.

In our example MyFunction will multiply elements of the array (first argument) by the numeric value given in second argument.

First we will write the beginning of our function:

```
AmiVar MyFunction( int NumArgs, AmiVar *ArgsTable )
{
          int i;
          AmiVar result;
          result = gSite.AllocArrayResult();
          int nSize = gSite.GetArraySize();
```

As you can see after standard prototype we define the counter variable (i) and the variable that will hold the result of our function (of AmiVar type). Since our function returns an array we need to allocate the memory for its elements (for float return values it is of course not needed, but it *is* needed if you want to return strings). Allocation of the array is easy by calling AllocArrayResult from site interface. You may also use simply Alloc() function from site interface, but this function requires byte size of memory to be allocated so it is more useful to allocate strings (for example: gSite.Alloc(strlen(string) + 1)). At the end of this block we retrieve the size of the arrays used by AFL engine using GetArraySize function of site interface.

Please note that we could write also:

```
          int nSize = gSite.GetArraySize;
          result.type = VAR_ARRAY;
          result.array = gSite.Alloc( sizeof( float ) * nSize );
```

but it is longer than using AllocArrayResult().

Now it is the time for main loop in which we will calculate the values of the resulting array

```
 for( i = 0; i < nSize; i++ )
 {
    result.array[ i ] = ArgsTable[ 0 ].array[ i ] * ArgsTable[ 1 ].val;
 }
```

In this loop we simply multiply each element of the array stored in the first argument by the numeric value stored in the second argument. In function implementation we don't need to check argument types - once we defined them in the function table - AFL engine takes care about type checking and implict conversions, so we can be sure that **ArgsTable[0]** holds the array (therefore **array** member of the union is valid) and **ArgsTable[1]** holds floating point value (therefore **val** member of the union is valid).

Now the only thing left is to return the result from the function:

```
          return result;
}
```

2.4.3 Calling internal AmiBroker functions

You can call internal AmiBroker function using CallFunction() method of site interface. To do so you should prepare argument table first. Argument table should define all parameters needed by the function you are calling (even the default ones).

```
          AmiVar args[ 2 ];
          args[ 0 ].type = VAR_FLOAT;
          args[ 0 ].val = 12;
          args[ 1 ].type = VAR_FLOAT;
          args[ 1 ].val = 26;
          gSite.CallFunction("macd", 2, args );
```

2.4.4 Reading and writing AFL variables

You can read (get) and write (set) the contents of any AFL variable using GetVariable/SetVariable methods of site interface.

To read variable just call:

```
AmiVar value = gSite.GetVariable( "buyprice" );
// value.array holds buy price (array of floats)
```

To store numeric value into variable use:

```
AmiVar myvar;
myvar.type = VAR_FLOAT;

myvar.val = 7;

gSite.SetVariable("myownvariable", myvar );
```

The following example illustrates how to set string variable (we get current time and format it to string then call SetVariable):

```
time_t ltime;
time( &ltime ); // we get current time

AmiVar myvar;
myvar.type = VAR_STRING;
myvar.string = gSite.Alloc( 100 ); // allocate memory for string

sprintf( myvar.string, "The time is %s", ctime( &ltime ) ); // print to allocated buffer

gSite.SetVariable("currenttime", myvar );
```

Third example shows how to set array variable:

```
AmiVar myvar = gSite.AllocArrayResult();

int nSize = gSite.GetArraySize();
for( int i = 0; i < nSize; i++ )
{
          myvar.array[ i ] = sin( 0.1 * i );
}


gSite.SetVariable("sinetable", myvar );
```

Please note that ability to set AFL variables from the plugin level allows you to return ANY number of results from your function. Simply call SetVariable as many times you wish inside your function and you will be able to get the values of all those variables from AFL side.

3 DATA PLUGINS

3.1 FEATURES

Data plug-in DLLs have the following features:

- support for end-of-day, hourly, 15-, 5-, 1-minute, 15-, 5-second and tick base intervals
- auto-refresh of charts/commentaries/interpretation
- streaming quotes display in real-time quote window
- data-on-demand
- build-up intraday histories from streaming data
- support for non-quotation data (fundamentals, etc)

- status display, customisable configuration dialogs, custom context menus
- support for synchronous and asynchronous (event-driven) operation model
- dynamic loading/unloading the plugins at run time without the need to restart AmiBroker
- unlimited number of plugins can be loaded

3.2 INTERFACE DEFINITION

3.2.1 Data types

ATTENTION: The date/time format has changed from 32bit to 64 bit in this release of ADK. Also the layout of Packed Date has changed. Also changed is the laout of Quotation and StockInfo structures. Please read the information below carefully.

In addition to already descibed AmiVar structure, data plugin may use the following data types:

```
struct PackedDate {
                                                        // lower 32 bits
                                                        unsigned int IsFuturePad:1;      // bit
marking "future data"
                                                        unsigned int Reserved:5;         // reserved
set to zero
                                                        unsigned int MicroSec:10;        //
microseconds          0..999
                                                        unsigned int MilliSec:10;        //
milliseconds          0..999
                                                        unsigned int Second: 6;                     //
0..59
                                                        // higher 32 bits
                                                unsigned int Minute : 6; // 0..59 63 is reserved as EOD
marker
                                                unsigned int Hour : 5; // 0..23 31 is reserved as EOD
marker
                                                unsigned int Day : 5; // 1..31
                                                unsigned int Month : 4; // 1..12
                                                unsigned int Year : 12;          // 0..4095

   };


union AmiDate
{
                                                        unsigned __int64      Date;
                                                        struct PackDate       PackDate;
};
```

AmiDate is a structure that AmiBroker uses to store date and time of quotation. To conserve memory, its date/time fields are bit-packed so they span only 64 bits (the same as long integer). Supported date range is Jan 1st, year 0 (zero) upto and including 31 Dec, 4095. Time is stored in one microsecond resolution which is sufficient for everything including tick charts. The time stamp MUST be unique in new format. In tick mode where data source may not provide sub-second resolution, it is plugin responsibility to make unique timestamps for subsequent ticks by using MicroSec and MilliSec fields. AmiBroker's own plugins count consecutive ticks occuring within given second and store this number in MicroSec field. End-of-day records are marked with 63 in Minute field and 31 in hour field. Years are stored without offset so 0 in Year field represents 0 and 2009 represents year 2009.

```
// 40-bytes 8-byte aligned
struct Quotation {
                                                        union AmiDate DateTime;          // 8 byte
                      float   Price;
                      float   Open;
                      float   High;
                      float   Low;
                                                        float   Volume;
                                                        float      OpenInterest;
                                                        float      AuxData1;
                                                        float      AuxData2;
                };
```

Quotation structure holds single bar data. DateTime field holds bar date and time and is an AmiDate structure equivalent to unsigned 64 bit int (change from previous release of ADK) Price field is actualy Close price. The Open, High, Low fields are self-explanatory, these are single precision floating point numbers. Please note that Volume and OpenInterest fields are currently floating point numbers (change since previous release of ADK). There are two new fields: AuxData1, and AuxData2 for storing auxilliary data such as Bid, Ask, or Traded Volume or user-defined numbers. The meaning of those two fields can be dynamic and depends on data source.

Flags and PercentReduc fields present in previous versions are removed.

```
#define MAX_SYMBOL_LEN 48
struct StockInfo {
                        char    ShortName[MAX_SYMBOL_LEN];
                                                            char       AliasName[MAX_SYMBOL_LEN];
                                                            char       WebID[MAX_SYMBOL_LEN];
                        char    FullName[128];
                        char    Address[128];
                                                            char       Country[64];
                                                            char       Currency[4];                    /*
ISO 3 letter currency code */
                                                            int                  DataSource;
                        int     DataLocalMode;              /* local mode of operation - 0 - accept
workspace settings, 1 - store locally, 2 - don't store locally */
                                                            int                  MarketID;
                                                            int                  GroupID;
                                                            int                  IndustryID;
                                                            int                  GICS;
                        int     Flags;          /* continuous etc.*/
                        int     MoreFlags;          /*  */
                                                            float      MarginDeposit;                  /*
new futures fields - active if SI_MOREFLAGS_FUTURES is set */
                                                            float      PointValue;
                                                            float      RoundLotSize;
                                                            float      TickSize;            /* new
futures fields - active if SI_MOREFLAGS_FUTURES is set */
                                                            int                  Decimals;
/* number of decimal places to display */
                                                            short      LastSplitFactor[ 2 ];  // /* [ 0 ]
- new, [ 1 ] - old */
                                                            DATE_TIME_INT LastSplitDate;      // at 16-
byte boundary
                        DATE_TIME_INT DividendPayDate;
                        DATE_TIME_INT ExDividendDate;         /* div date */
                                                            float      SharesFloat;//
int     Code;
                                                            float      SharesOut; //int     StockQty;
                                                            float   DividendPerShare;       //
                                                            float   BookValuePerShare;
                                                            float   PEGRatio;                           //
PE Growth ratio
                                                            float   ProfitMargin;
                                                            float      OperatingMargin;
                                                            float      OneYearTargetPrice;
                                                            float      ReturnOnAssets;
                                                            float      ReturnOnEquity;
                                                            float      QtrlyRevenueGrowth;    /* year over
year */
                                                            float      GrossProfitPerShare;
                                                            float      SalesPerShare;           // ttn
Sales Revenue
                                                            float      EBITDAPerShare;
                                                            float      QtrlyEarningsGrowth;
                                                            float      InsiderHoldPercent;
                                                            float      InstitutionHoldPercent;
                                                            float      SharesShort;
                                                            float      SharesShortPrevMonth;
                                                            float      ForwardEPS;           // from
Forward P/E
                                                            float      EPS;       // ttm EPS
                                                            float      EPSEstCurrentYear;
                                                            float      EPSEstNextYear;
                                                            float      EPSEstNextQuarter;
                                                            float   ForwardDividendPerShare;
                                                            float      Beta;
```

```
                                                    float      OperatingCashFlow;
                                                    float      LeveredFreeCashFlow;
                                                    float      ReservedInternal[ 28 ];
                                                    float      UserData[ 100 ];
};
```

StockInfo structure has been expanded significantly. Existing fields are now wider and there are now fundamental fields exposed via Information window as well as 100 new user-definable fields. StockInfo structure holds basic symbol information like full name (FullName field), ticker symbol (ShortName field) and a number of others. This structure is returned by AddStock method of InfoSite structure and allows you to set-up initial values for symbol that was added using AddStock call. See Quotes Plus plugin source code for example usage.

```
struct InfoSite
                                {
                                          int                               nStructSize;
                                          int                               (*GetStockQty)( void );
                                          struct StockInfo * (*AddStock)( const char *pszTicker );
                                          int                               (*SetCategoryName)( int nCategory,
int nItem, const char *pszName );
                                          const char *(*GetCategoryName)( int nCategory, int nItem );
                                    int                           (*SetIndustrySector) ( int nIndustry, int
nSector );
                                          int                               (*GetIndustrySector) ( int
nIndustry );
                                };
```

InfoSite structure is a similar concept to SiteInterface found in AFL plugins. It provides function pointers that allow to call-back AmiBroker internal functions and perform various operations on AmiBroker stock database. See Quotes Plus plugin source code for example usage.

```
struct RecentInfo
{
          int                   nStructSize;
          char       Name[ 64 ];
          char       Exchange[8];
          int                   nStatus;
          int                   nBitmap;   // describes which fields are valid
          float      fOpen;
          float      fHigh;
          float      fLow;
          float      fLast;
          int                   iTradeVol;
          int                   iTotalVol;
          float      fOpenInt;
          float      fChange;
          float      fPrev;
          float      fBid;
          int                   iBidSize;
          float      fAsk;
          int                   iAskSize;
          float      fEPS;
          float      fDividend;
          float      fDivYield;
          int                   nShares; // shares outstanding
          float      f52WeekHigh;
          int                   n52WeekHighDate;
          float      f52WeekLow;
          int                   n52WeekLowDate;
          int                   nDateChange; // format YYYYMMDD
          int                   nTimeChange; // format HHMMSS
          int                   nDateUpdate; // format YYYYMMDD
          int                   nTimeUpdate; // format HHMMSS
          float      fTradeVol; // NEW 5.27 field
          float      fTotalVol; // NEW 5.27 field
};
```

RecentInfo structure is used by real-time plugins to store information about most recent streaming update (last bid/ask/trade and a number of other data). Name is a full name of symbol. There is no ticker symbol in this structure because it is specified by argument passed separately. Please note that

if given data source does not provide all these data then not all those fields have to be valid. To mark which fields are valid you should store combination of RI_*** bit flags defined in Plugin.h into nBitmap field. For example storing RI_LAST | RI_OPEN into nBitmap field will mean that fLast and fOpen fields contain valid values. nDateChange and nTimeChange fields provide the information about date/time of price fields. For example you may start AmiBroker on Sunday and these fields will show Friday's date/time because last price change was on Friday. nDateUpdate and nTimeUpdate fields store the date/time of last update of any field. In most cases this is current minute/second. So in previous example it would be the Sunday date/time. It is very important to update nDateUpdate and nTimeUpdate fields each time you update any value in this structure otherwise real-time quote window will not work properly. Note about new fTradeVol and fTotalVol fields - these are added in AmiBroker version 5.27. If you want to support both old and new AmiBroker versions your plugin must fill both floating and integer counterparts: fTradeVol and iTradeVol as well as fTotalVol and iTotalVal fields.

RecentInfo structure is used in two cases: (1) as a return value of **GetRecentInfo()** function which is called by AmiBroker's real-time quote window on each display refresh, and (2) passed in LPARAM in WM_USER_STREAMING_UPDATE message sent by plugin to AmiBroker. See QuoteTracker plugin source code for example usage.

```
struct PluginStatus
{
	int				nStructSize;
	int				nStatusCode;
	COLORREF	clrStatusColor;
	char			szLongMessage[ 256 ];
	char			szShortMessage[ 32 ];
};
```

Plugin status structure is returned by GetStatus function and provides both numeric and text status information. Text information is provided in long and short version. Short version is displayed in the plugin status area of AmiBroker status bar, long version is displayed in the tooltip. The highest nibble (4-bit part) of nStatus code represents type of status: 0 - OK, 1 - WARNING, 2 - MINOR ERROR, 3 - SEVERE ERROR that translate to color of status area: 0 - green, 1 - yellow, 2 - red, 3 - violet. See QuoteTracker plugin source for sample usage.

```
struct _Workspace {
	int			DataSource;		/* 0 - use preferences, 1 - local, ID of plugin */
	int			DataLocalMode;		/* 0 - use preferences, 1 - store locally, 2 - don't */
	int			NumBars;
	int			TimeBase;
	// the place were OLD intraday settings were located
	int			ReservedB[ 8 ];
	BOOL		AllowMixedEODIntra;
	BOOL		RequestDataOnSave;
	BOOL		PadNonTradingDays;
	int			ReservedC;
	struct	_IntradaySettings IS;
	int			ReservedD;
};


struct PluginNotification
{
	int			nStructSize;
	int			nReason;
	LPCTSTR pszDatabasePath;
	HWND		hMainWnd;
	struct	StockInfo	*pCurrentSI;
	struct	_Workspace	*pWorkspace;
};
```

PluginNotification structure is filled up by AmiBroker and passed to plugin as an argument of Notify() call. nReason field describes the "reason" of notification, that could be the fact that database is loaded, unloaded, settings are changed or user has clicked with right mouse button over plugin status

area of AmiBroker status bar. This last value (REASON_STATUS_RMBCLICK) is used to display context menu that can offer some choices to the user like "Connect", "Diconnect", etc. Possible values are defined in Plugin.h file. See QuoteTracker plugin source for sample usage.

3.2.2 Interface functions

A valid AmiBroker data plug-in DLL **must** export the following functions:

```
PLUGINAPI int GetPluginInfo( struct PluginInfo *pInfo );
PLUGINAPI int Init(void);
PLUGINAPI int Release(void);
PLUGINAPI int GetQuotesEx( LPCTSTR pszTicker, int nPeriodicity, int nLastValid, int nSize, struct Quotation
*pQuotes, GQEContext *pContext  )
PLUGINAPI int GetQuotes( LPCTSTR pszTicker, int nPeriodicity, int nLastValid, int nSize, struct
QuotationFormat4 *pQuotes );
```

The **GetPluginInfo()** function is used for obtaining the information about the plugin (the name, vendor name, type, plugin ID, min allowed AmiBroker version) - you should provide accurate information in your DLL for easy identification of your plugin in the "Plugins" window in AmiBroker.

Init() and **Release()** functions are provided to allow extra memory allocation/other resource initialization in the DLL.

GetQuotesEx() function is a basic function that all data plugins must export and it is called each time AmiBroker wants to get new quotes. The main idea behind GetQuotesEx function is very simple: ticker symbol, bar interval and pre-allocated quotation array of nSize elements are passed as arguments to this function. **GetQuotesEx()** should simply fill the array with the quotes of given symbol and given interval. It is that simple. Detailed implementations vary depending on underlying data source (detailed description later in this document).

GetQuotes() function - it is legacy version of GetQuotesEx() function. It is called by AmiBroker versions prior to 5.27. It is called by newer versions of AmiBroker when GetQuotesEx() is NOT exposed by the plugin. This legacy compatibility layer guarantees that old plugins continue to work with new versions of AmiBroker without need to rewrite, although for best performance and to get new features it is advised to update plugins to support new ADK.

Functions listed below are **optional** for the data plugin and they may be exported/defined only when additional functionality is needed:

```
PLUGINAPI AmiVar GetExtraData( LPCTSTR pszTicker, LPCTSTR pszName, int nArraySize, int nPeriodicity, void*
(*pfAlloc)(unsigned int nSize) );
PLUGINAPI int Configure( LPCTSTR pszPath, struct InfoSite *pSite );
PLUGINAPI int SetTimeBase( int nTimeBase );
PLUGINAPI int Notify( struct PluginNotification *pNotifyData );
PLUGINAPI int GetPluginStatus( struct PluginStatus *status );
```

GetExtraData() is used for retrieving non-quotation data. It is called when the GetExtraData() AFL function is used in the formula. pfAlloc parameter is a pointer to AFL memory allocator that you should use to allocate memory so it can be later freed by AmiBroker itself. Source code for QuotesPlus plugin shows sample implementation of **GetExtraData()** that retrieves fundamental data.

Configure() function is called when user presses "Configure" button in AmiBroker's File->Database Settings window. The path to the database and InfoSite structure are passed as arguments. This allows the plugin to display its own dialog box that provides plugin-specific settings and allows to store the settings either in registry (global settings) or in the file stored in database path (local per-database settings). InfoSite allows the plugin to automatically setup AmiBroker's database out of

information retrieved from external data source. The source code for QuotesPlus plugin shows real-life example (getting symbols and setting up entire sector/industry tree). If **Configure()** function is not exported by the plugin AmiBroker displays message that "plugin does not require further configuration" when "Configure" button is clicked.

SetTimeBase() function is called when user is changing base time interval in File->Database Settings window. It is not required for plugins that handle only daily (EOD) interval. It has to be defined for all intraday plugins. The function takes bar interval in seconds (60 for 1-minute bars, 86400 for daily bars) and should return 1 if given interval is supported by the plugin and 0 if it is not supported.

Notify() function is called when database is loaded, unloaded, settings are changed, or right mouse button in the plugin status area is clicked. It supersedes SetDatabasePath (which is now obsolete) that was called only when database was loaded. **Notify()** function is optional however it is implemented in almost all plugins because it is a good place to initialize/deinitialize plugin state. For example implementation of **Notify()** function please check QuoteTracker plugin source.

GetPluginStatus() function is optional and used mostly by real-time plugins to provide visual feedback on current plugin status. It provides a way to display status information in the AmiBroker's status bar. For example implementation of **GetPluginStatus()** function please check QuoteTracker plugin source.

The following two functions are implemented **only** by real-time plugins:

```
PLUGINAPI struct RecentInfo * GetRecentInfo( LPCTSTR pszTicker ); // RT plugins    only
PLUGINAPI int GetSymbolLimit( void ); // RT plugins only
```

GetRecentInfo() function is exported only by real-time plugins and provides the information about the most recent trade, bid/ask, days high/low, etc (updated by streaming data). It is called by AmiBroker's real-time quote window on each window refresh (occurs several times a second). The function takes ticker symbol and returns the pointer to RecentInfo structure described above.

GetSymbolLimit() function is exported only by real-time plugins. It returns the maximum number of streaming symbols that plugin and/or external data source can handle. The result of this function is used to limit the number of symbols that are displayed in real-time quote window.

3.3 CREATING YOUR OWN DATA PLUGIN DLL

Creating your own plug-in DLL is quite simple. If you are using Visual C++ 6 you should do the following:

1. Choose File->New from the menu.
2. From the list of available projects choose "Win32 Dynamic-Link Library" and type the project name, for example "MyPlugin", then click "OK"
3. In the page "Win32 Dynamic-Link Library - Step 1 of 1" choose "A simple DLL project" - this will create a project file and three source code files - MyPlugin.cpp, StdAfx.h, StdAfx.cpp
4. Now copy "Plugin.cpp", "Plugin.h" files from the Data_Template plugin DLL source code folder to your project folder
5. Choose Project->Add to project->Files... menu. From the file dialog please choose "Plugin.cpp", "Plugin.h" files and click OK. Now these files are added to the project and you can build it.

After these steps you have functional copy of a Sample project with your own name (MyPlugin). From now you can modify project files.

The only file you really need to modify is "Plugin.cpp" file that actually implements the functions that your plug in will expose to AFL.

You have to modify your plugin name, vendor and version information and plugin ID code defined in lines 23-26 of Plugin.cpp:

```
#define PLUGIN_NAME "MyPlugin - enter here real name of the plugin"
#define VENDOR_NAME "Your name"
#define PLUGIN_VERSION 010000
#define PLUGIN_ID PIDCODE( 'T', 'E', 'S', 'T')
```

The information defined here is displayed by the AmiBroker in the Plugins window so it is important to give the user correct information. Please do not forget to do that.

It is **EXTREMELY IMPORTANT** to use PLUGIN_ID to uniquely identifies your data source. AmiBroker uses the plugin ID to distinguish between data sources. For testing purposes you may use PIDCODE('T', 'E', 'S', 'T'), but for release to the public you should contack us at adk@amibroker.com to receive unique plugin identifier for your data plugin. Already reserved plugin IDs are: QTRK, MSTK, eSIG, myTK, TC2K, FTRK, CSI, QCOM, DTNI.

Right after than you should add (if it does not already exist) the following line:

```
// IMPORTANT: Define plugin type !!!
#define THIS_PLUGIN_TYPE PLUGIN_TYPE_DATA
```

This defines that the plugin you are writing is data plugin.

3.4 IMPLEMENTING DATA PLUGIN

A very basic data plugin requires just modification of **GetQuotesEx()** function supplied with the template. Before we will dig into details a little background is needed.

Each time AmiBroker needs quotes for particular symbol it calls **GetQuotesEx()** function. Please note that AmiBroker caches response received from **GetQuotesEx()** and will not ask for quotes again for the symbol until: (a) user chooses "Refresh" from View menu, (b) plugin notifies AmiBroker that new data arrived using WM_USER_STREAMING_UPDATE message, (c) old data were removed from the cache. AmiBroker cache maintains the list of most recently accessed symbols and may remove the data of the least recently accessed symbols. The size of AmiBroker's cache is controlled by "in-memory cache" setting in Tools->Preferences->Data.

External data sources could be divided into two categories (a) local databases (b) remote databases. Local databases have all data stored on computer hard disk or CD-ROM and available for immediate retrieval. Remote databases (also known as on-demand data sources) do not store all data locally, instead they retrieve the data on-demand from remote computer (usually via Internet). These two kinds of data sources have to be handled differently.

In the first case (local sources) quotes can be retrieved in synchronous way: we ask for data and block the calling application until data are collected. This is acceptable because data are available locally and can be delivered within few milliseconds. This is the case for all file-based sources like: Metastock, Quotes Plus, TC2000, FastTrack. During **GetQuotesEx()** call you should simply read requested number of bars from the data source and fill provided Quotation array.

In the second case (remote sources) quotes have to be retrieved in asynchronous way. When **GetQuotesEx()** function is called for the first time, request for new data has to be send to the data source. As data from remote source arrive usually after a few seconds (or more) we can **not** block calling application (AmiBroker). Instead control should be returned to AmiBroker. Depending on architecture of your data source you should either setup a window or another thread that will wait for the message sent back by the data source when data is ready. When such message is received the plugin should send WM_USER_STREAMING_UPDATE message that will nofity AmiBroker that it should ask for quotes. In the response to this message AmiBroker will call **GetQuotesEx()** function again.

This time you should fill Quotation array with the data you received from remote source. To avoid repeated requests for historical intraday data, once it is retrieved, real-time plugins begin to collect streaming time/sales data and build-up intraday bars. Each successive **GetQuotesEx()** call receives bars that were build-up inside plugin from streaming upates. This mode of operation is used by eSignal and myTrack real-time plugins.

Now we will show the simplest form of **GetQuotesEx()** function that will read the quotes from the local ASCII file. We will be reading 1-minute intraday ASCII files.

We assume that data files are stored in 'ASCII' subfolder of AmiBroker directory and they have name of <SYMBOL>.AQI and the format of Date (YYMMDD), Time (HHMM), Open, High, Low, Close, Volume (actually these are AQI files produced by AmiQuote) and quotes inside file are sorted in ascending rder (the oldest quote is on the top)

```
PLUGINAPI int GetQuotesEx( LPCTSTR pszTicker, int nPeriodicity, int nLastValid, int nSize, struct Quotation
*pQuotes, GQEContext *pContext  )
{

           // we assume that intraday data files are stored in ASCII subfolder
           // of AmiBroker directory and they have name of .AQI
           // and the format of Date(YYMMDD),Time(HHMM),Open,High,Low,Close,Volume
           // and quotes are sorted in ascending order - oldest quote is on the top

           char filename[ 256 ];
           FILE *fh;
           int  iLines = 0;

           // format path to the file (we are using relative path)
           sprintf( filename, "ASCII\\%s.AQI", pszTicker );

           // open file for reading
           fh = fopen( filename, "r" );

           // if file is successfully opened read it and fill quotation array
           if( fh )
           {
                      char line[ 256 ];

                      // read the line of text until the end of text
                      // but not more than array size provided by AmiBroker
                      while( fgets( line, sizeof( line ), fh ) && iLines < nSize )
                      {
                                 // get array entry
                                 struct Quotation *qt = &pQuotes[ iLines ];

                                 // parse line contents: divide tokens separated by comma (strtok) and
interpret values

                                 // date    and time first
                                 int datenum = atoi( strtok( line, "," ) );  // YYMMDD
                                 int timenum = atoi( strtok( NULL, "," ) );  // HHMM

                                 // unpack datenum and timenum and store date/time
                                 qt->DateTime.Date = 0; // make sure that date structure is intialized with
zero
                                 qt->DateTime.PackDate.Minute = timenum % 100;
                                 qt->DateTime.PackDate.Hour = timenum / 100;
                                 qt->DateTime.PackDate.Year = 2000 + datenum / 10000;
                                 qt->DateTime.PackDate.Month = ( datenum / 100 ) % 100;
                                 qt->DateTime.PackDate.Day = datenum % 100;

                                 // now OHLC price fields
                                 qt->Open = (float) atof( strtok( NULL, "," ) );
                                 qt->High = (float) atof( strtok( NULL, "," ) );
                                 qt->Low  = (float) atof( strtok( NULL, "," ) );
                                 qt->Price = (float) atof( strtok( NULL, "," ) ); // close price

                                 // ... and Volume
                                 qt->Volume = (float) atof( strtok( NULL, ",\n" ) );

                                 iLines++;
                      }

                      // close the file once we are done
                      fclose( fh );
```

```
        }

        // return number of lines read which is equal to
        // number of quotes
        return iLines;
}
```

For simplicity the example does not do any serious error checking.

A few comments about arguments of **GetQuotesEx()** function. First argument pszTicker is a null-terminated ticker symbol, nPeriodicity is bar interval (in seconds). Third parameter nLastValid requires some more description. When AmiBroker calls **GetQuotesEx()** function it may already have some data for given symbol (stored for example in its own files when local data storage is ON). However, before **GetQuotesEx()** is called AmiBroker always allocates quotation array of size defined in File->Database Settings: default number of bars. This size is passed to the plugin as nSize argument. If AmiBroker has already some data for given symbol it fills initial elements of quotation array passed to **GetQuotesEx()** function. The index of last filled element of quotation array is passed as nLastValid argument. This allows to update just a few new bars without the need to fill entire array inside the plugin. This technique is used for example in QuotesPlus plugin that does not update entire array if it finds that last valid quote is the same as last quote available from Quotes Plus database. The last argument is pQuotes array which is array of Quotation structures. The array is allocated by AmiBroker itself and it has the size of nSize elements.

3.5 LEGACY FORMAT SUPPORT (PRE-5.27 VERSIONS)

The data interface for pre-5.27 AmiBroker versions had only **GetQuotes** function exported that used old quotation structure (QuotationFormat4). To ensure backward compatibility with old versions of AmiBroker (pre 5.27), any data plugin MUST export GetQuotes function. The simplest way to do this is to copy-paste the following code that does entire format translation back and forth and uses new GetQuotesEx format described earlier. Old format structures (QuotationFormat4, StockInfoFormat4) as well as ConvertFormat4Quote() and ConvertFormat5Quote() helper functions are defined in the Plugin_Legacy.h.

```
// GetQuotes wrapper for LEGACY format support
// convert back and forth between old and new format
//
// WARNING: it is highly inefficient and should be avoided
// So this is left just for maintaning compatibility,
// not for performance
//
PLUGINAPI int GetQuotes( LPCTSTR pszTicker, int nPeriodicity, int nLastValid,   int nSize, struct
QuotationFormat4 *pQuotes )
{
 AFX_MANAGE_STATE( AfxGetStaticModuleState() );
 Quotation *pQuote5 = (struct Quotation *) malloc( nSize * sizeof( Quotation   ) );
 QuotationFormat4 *src = pQuotes;
 Quotation *dst = pQuote5;
 int i;
 for( i = 0; i <= nLastValid; i++,     src++, dst++ )
 {
   ConvertFormat4Quote( src, dst );
 }
 int nQty = GetQuotesEx( pszTicker, nPeriodicity, nLastValid, nSize, pQuote5,   NULL );
 dst = pQuote5;
 src = pQuotes;
 for( i = 0; i < nQty; i++, dst++,     src++ )
 {
   ConvertFormat5Quote( dst, src );
 }
 free( pQuote5 );
 return nQty;
}
```

3.6 CONVERTING/UPGRADING EXISTING DATA PLUGINS (ADK 1.10) TO USE NEW API (2.00)

If you already have existing data plugin written with ADK 1.10 (for pre-5.27 versions of AmiBroker), here is the list of things to do/check in order to update your plugin to new ADK and make it AmiBroker 5.27+ compatible. Please note that altough AmiBroker 5.27 has its own compatibility layer for old plugins (will call old-style GetQuotes() if plugin does not export new-style GetQuotesEx()), this layer has certain computational cost. This cost is insignificant for end-of-day plugins but may be serious if you use real-time data source with large number of bars (500K+). So it is highly recommended, where possible, to support new format using this new version of ADK.

Here are the steps required / check list for conversion to new ADK:

1. Use new header (just copy Plugin.h / Plugin_Legacy.h files from ADK Include folder)

2. Examine sources for all occurences of "Tick" member of AmiDate structure and replace it with 'Second'. Do not forget about removing multiplication/division by 5 applied previously to 'Tick' member.

3. Examine sources for all occurrences of Volume and OpenInt fields of quotation structure. Replace 'int' for 'float' data type in those places.

4. Remove any usage of Flags and PercReduc fields in Quotation structure

5. Change your original GetQuotes function to GetQuotesEx function, do not forget to add new parameter (GQEContext *) at the end of parameter list. This parameter is for future expansion, can be NULL and currently can be ignored.

6. To support legacy format copy the legacy GetQuotes() wrapper listed above. It will convert from old format to new, call new GetQuotesEx and convert back to new format.

7. If you are using AddStock() function from InfoSite make sure that you call new function ONLY if structure size (nStructSize) is equal to 32 or more (on 32 bit platforms).

```
PLUGINAPI int Configure( LPCTSTR pszPath, struct InfoSite *pSite )
{
   if( pSite->nStructSize >= sizeof( struct InfoSite ) )
   {
    /// YOU CAN CALL pSite->AddStockNew here !
    // and you can access new StockInfo data members
   }
   else
   {
    /// otherwise you can _only_ call pSite->AddStock here !

    // NOTE THE DIFFERENCES between returned StockInfo and StockInfoFormat4
   }
}
```

If you fail to take these precaution steps your code will generate crash when user presses Configure button

8. RecentInfo has been extended to support float trade vold and total volume. Old fields are left in place for legacy support, so you plugin should fill BOTH
old (integer) and new (float) volume and set the structure size correctly so new versions of AmiBroker can know that float fields are available

9. Now Year member of AmiDate structure holds FULL year (0000 upto 4095) (as compared to previously used offset 1900 year) so you need to find
all occurrences where 1900 constant is used in your code and adjust the code (usually be removing 1900 constant) so FULL year is used in new format. When removing 1900 constant from

"everywhere", do NOT do that for Plugin_Legacy.h file as it contains conversion functions between old and new format where using 1900 constant is required.

10. Make sure to use 64 bit integers (unsigned __int64) wherever new datetime comparisions are used.

11. Examine source codes for mask such as 0x007FFF that was previously used as " EOD" mask. With current 64-bit date/time this has to be replaced by
0x000007FFfffffFFC0i64 constant (defined as DAILY_MASK in the header)

4 OPTIMIZER PLUGINS

4.1 Getting Started

Optimizer plugins are very simple to implement. You just need to get skeleton code (MonteCarlo random optimizer sample is good as a starting point) and add your bits to it.

As every AmiBroker DLL plugin, optimizer plugins require 3 core functions: GetPluginInfo, Init(), Release() that are standard part of AmiBroker plugin interface. They are very staightforward (single-liners in most cases). You can just copy / paste the functions below. The only 2 things that you must change:

- the name of the plugin (see #define PLUGIN_NAME ...), and
- the plugin ID code (see PIDCODE inside PluginInfo structure)

The plugin ID code MUST BE UNIQUE. Otherwise it will conflict with other plugins. For tests I suggest using `PIDCODE( 't', 'e', 's', '1')` changing the last digit if you want to have more than one test plugin. For list of already used IDs please see : http://www.amibroker.com/plugins.html . Before releasing your plugin to the public, you must request unique plugin ID from support at amibroker.com. The plugin ID will be later used to specify optimizer in AFL code via OptimizerSetEngine() function.

```
// These are the only two lines you need to change
#define PLUGIN_NAME "Monte Carlo Optimizer plug-in"
#define VENDOR_NAME "Amibroker.com"
#define PLUGIN_VERSION 10001


#define THIS_PLUGIN_TYPE PLUGIN_TYPE_OPTIMIZER

////////////////////////////////////////
// Data section
////////////////////////////////////////
static struct PluginInfo oPluginInfo =
{
       sizeof( struct PluginInfo ),
       THIS_PLUGIN_TYPE,
       PLUGIN_VERSION,
       PIDCODE( 'm', 'o', 'c', 'a'),
       PLUGIN_NAME,
       VENDOR_NAME,
      13012679,
      387000
};


////////////////////////////////////////////////////////////
// Basic plug-in interface functions exported by DLL
////////////////////////////////////////////////////////////

PLUGINAPI int GetPluginInfo( struct PluginInfo *pInfo )
{
    *pInfo = oPluginInfo;

   return True;
}


PLUGINAPI int Init(void)
{

   return 1;
}

PLUGINAPI int Release(void)
{
```

```
   return 1;      // default implementation does nothing
}
```

4.2 Optimizer Interface

The optimizer interface consists of 4 simple functions:

- int OptimizerInit(struct OptimizeParams *pParams);
- int OptimizerRun(struct OptimizeParams *pParams, double (*pfEvaluateFunc)(void *), void *pContext);
- int OptimizerFinalize(struct OptimizeParams *pParams);
- int OptimizerSetOption(const char *pszParam, AmiVar newValue);

And two data structures:

```
struct OptimizeItem
{
    char   *Name;
    float   Default;
    float   Min;
    float   Max;
    float   Step;
    double   Current;
    float   Best;
};

#define MAX_OPTIMIZE_ITEMS 100

struct OptimizeParams
{
   int      Mode;            // 0 - gets defaults, 1 - retrieves settings from formula (setup phase), 2 - optimization phase
   int      WalkForwardMode;   // 0 - none (regular optimization), 1-in-sample, 2 - out of sample
   int      Engine;            // optimization engine selected - 0 means - built-in exhaustive search
   int      Qty;             // number of variables to optimize
   int      LastQty;
   BOOL   CanContinue;      // boolean flag 1 - means optimization can continue,
                           //0 - means aborted by pressing "Cancel" in progress dialog or other error
   BOOL   DuplicateCheck;      // boolean flag 1 - means that AmiBroker will first
                        // check if same param set wasn't used already
                       // and if duplicate is found it won't run backtest, instead will return previously stored value
   int      Reserved;
   char   *InfoText;         // pointer to info text buffer (providing text display in the progress dialog)
   int      InfoTextSize;     // the size (in bytes) of info text buffer
   __int64   Step;         // current optimization step (used for progress indicator) - automatically increased with each iteration
   __int64   NumSteps;         // total number of optimization steps (used for progress indicator)
   double  TargetCurrent;
   double  TargetBest;
   int      TargetBestStep;      // optimization step in which best was achieved
   struct   OptimizeItem   Items[ MAX_OPTIMIZE_ITEMS ]; // parameters to optimize
};
```

4.2.1 Data structures

The OptimizeParams structure holds all information needed to perform optimization. The most important part is **Items** array of OptimizeItem structures. It holds the array of all parameters specified for optimization using AFL's Optimize() function. The number of valid parameters is stored in Qty member of OptimizeParams structure.

4.2.2 OptimizerInit function

PLUGINAPI int OptimizerInit(struct OptimizeParams *pParams)

This function gets called when AmiBroker collected all information about parameters that should be optimized. This information is available in OptimizeParams structure. The optimization engine DLL should use this point to initialize internal data structures. Also the optimizer should set the value of pParams->NumSteps variable to the expected TOTAL NUMBER OF BACKTESTS that are supposed to be done during optimization.

This value is used for two purposes:
1. progress indicator (total progress is expressed as backtest number divided by NumSteps)
2. flow control (by default AmiBroker will continue calling OptimizerRun until number of backtests

reaches the NumSteps) - it is possible however to
override that (see below)

Note that sometimes you may not know exact number of steps (backtests) in advance, in that case provide estimate. Later, inside OptimizerRun you will be able to adjust it, as tests go by.

Return values:
1 - initialization complete and OK
0 - init failed

4.2.3 OptimizerSetOption function

PLUGINAPI int OptimizerSetOption(const char *pszParam, AmiVar newValue)

This function is intended to be used to allow setting additional options / parameters of optimizer from the AFL level.

It gets called in two situations:
1. When SetOptimizerEngine() AFL function is called for particular optimizer - then it calls OptimizerSetOption once with pszParam set to NULL
and it means that optimizer should reset parameter values to default values
2. When OptimizerSetOption("paramname", value) AFL function is called

Return codes:
1 - OK (set successful)
0 - option does not exist
-1 - wrong type, number expected
-2 - wrong type, string expected

4.2.4 OptimizerRun function

PLUGINAPI int OptimizerRun(struct OptimizeParams *pParams, double (*pfEvaluateFunc)(void *), void *pContext)

This function is called multiple times during main optimization loop

There are two basic modes of operations
1. Simple Mode
2. Advanced Mode

In simple optimization mode, AmiBroker calls OptimizerRun before running backtest internally. Inside OptimizationRun the plugin should simply set current values of parameters and return 1 as long as backtest using given parameter set should be performed. AmiBroker internally will
do the remaining job. By default the OptimizerRun will be called pParams->NumSteps times.
In this mode you don't use pfEvaluateFunc argument.

See Monte Carlo (MOCASample) sample optimizer for coding example using simple mode.

In advanced optimization mode, you can trigger multiple "objective function" evaluations during single OptimizerRun call.
There are many algorithms (mostly "evolutionary" ones) that perform optimization by doing multiple runs, with each run consisting of multiple "objective function"/"fitness" evaluations. To allow interfacing such algorithms with AmiBroker's optimizer infrastructure the advanced mode provides access to pfEvaluateFunc pointer that call evaluation function.

In order to properly evaluate objective function you need to call it the following way:

pfEvaluateFunc(pContext);

Passing the pContext pointer is absolutely necessary as it holds internal state of AmiBroker optimizer. The function will crash if you fail to pass the context.

The following things happen inside AmiBroker when you call evaluation function:
a) the backtest with current parameter set (stored in pParams) is performed
b) step counter gets incremented (pParams->Step)
c) progress window gets updated
d) selected optimization target value is calculated and stored in pParams->TargetCurrent and returned as a result of pfEvaluateFunc

Once you call pfEvaluateFunc() from your plugin, AmiBroker will know that you are using advanced mode, and will NOT perform extra backtest after returning from OptimizerRun

By default AmiBroker will continue to call OptimizerRun as long as pParams->Step reaches pParams->NumSteps. You can overwrite this behaviour by returning value other than 1. See Standard Particle Swarm Optimizer (PSOSample) for coding example using advanced mode.

Return values:
0 - terminate optimization
1 (default value) - optimization should continue until reaching defined number of steps
2 - continue optimization loop regardless of step counter

4.2.5 OptimizerFinalize function

PLUGINAPI int OptimizerFinalize(struct OptimizeParams *pParams)

This function gets called when AmiBroker has completed the optimization. The optimization engine should use this point to release internal
data structures.

Return values:
1 - finalization complete and OK
0 - finalization failed

5 SUPPLIED EXAMPLES

In the ADK archive you will find the following examples of plug in DLLs:

Sample AFL plugins:

- Sample - contains Visual C++ 6/DevC++ source code of sample DLL that shows how to calculate standard and exponential moving average, and how to call AFL built-in functions from the DLL. Also includes a function that skips undefined (empty) values that can be found at the beginning of the array. It also shows how to create variable-period exponential moving average and how to set AFL variable values inside the plugin code.
  Good starting point for writing your own plug-ins. Includes Sample.dev project file for DevC++.
- Candle - contains Visual C++ 6/DevC++ source code of candlestick function plug-in DLL. Includes Candle.dev project file for DevC++.
- JRSample - contains Visual C++ 6/DevC++ source code showing how to write the plugin that uses Jurik Research generic DLLs (JMA in this example). Includes JRSample.dev project file for DevC++.
  For this sample to work you will need Jurik Reseach generic JMA DLL installed properly. Please

note that you should copy **JRS_32.LIB** file included with Jurik's package to the project directory before building the example

Sample Data plugins:

- QT - contains complete Visual C++ 6 source code of actual Quote Tracker plugin DLL. It shows all aspects of programming intraday data plugin
- QP2 - contains complete Visual C++ 6 source code of actual Quotes Plus plugin DLL. Excellent example of end-of-day plugin with support for fundamental data via GetExtraData(). To compile this project you will need to copy the header file (usrdll32.h) and the lib file (qpdll.lib) from Quotes Plus SDK which is available on Quotes Plus CD. You can order trial version from http://www.qp2.com
- ASCII - contains very simple ASCII plugin (contains GetQuotes function described in this document)
- Data_Template - contains template project for data plugin

Sample optimizer plugins:

- MOCASample - Monte-carlo style (random) optimizer driver
- PSOSample - Particle Swarm Optimizer driver
- Tribes - Tribes parameter-less particle swarm optimizer driver

Note that pre-build ready to use DLLs are located in Plugins subfolder of this ADK archive.

6 DEBUGGING PLUGIN DLLs

If you are using Visal C++ as your debugger you can simply, pick "Broker.exe" as "Debug executable" in the project settings and then you just use run it within DEBUGGER. You can setup a breakpoint inside your function or anywhere else. The debugging plugin does not differ at all from debugging any other DLL in the VC.

You may also need to turn off AmiBroker exception trapping by adding the following DWORD key to the registry

HKEY_CURRENT_USER\Software\TJP\Broker\Settings\TrapExceptions = 0

(set it to zero)

Without it AmiBroker traps exceptions and it will display "error recovery" instead of allowing debugger to catch the exception. Use this setting in development environment only. Production environment **MUST NOT** have this registry entry otherwise bugs could not be reported and user-triggered errors/exceptions may crash application instead of being caught and handled.