The Plugin proxy plugin
PRINTED MANUAL

Plugin proxy plugin

Printed: 11/2/2024

Publisher

AGG Software

Production

©2018-2024 AGG Software
http://www.aggsoft.com

Table of Contents

1 Introduction

The "Plugin proxy" module is the adapter between the main program and modules written in other programming languages. Unlike "Python plugin" module, which also provides access to API, this API interface has more features, but developing it is also more complicated.

The module uploads the DLL developed by you and calls API functions from it.

After installation, in the module folder, you can find an example for Visual C++ 2015 that implements the functions of data filter, data parser, and data export modules.

2 System requirements

The following requirements must be met for "Plugin proxy" to be installed:

Operating system: Windows 2000 SP4 and above, including both x86 and x64 workstations and servers. The latest service pack for the corresponding OS is required.

Free disk space: Not less than 5 MB of free disk space is recommended.

Special access requirements: You should log on as a user with Administrator rights in order to install this module.

The main application (core) must be installed, for example, Advanced Serial Data Logger.

3 Installing Plugin proxy

1. Close the main application (for example, Advanced Serial Data Logger) if it is running;
2. Copy the program to your hard drive;
3. Run the module installation file with a double click on the file name in Windows Explorer;
4. Follow the instructions of the installation software. Usually, it is enough just to click the "Next" button several times;
5. Start the main application. The name of the module will appear on the "Modules" tab of the "Settings" window if it is successfully installed.

If the module is compatible with the program, its name and version will be displayed in the module list. You can see examples of installed modules on fig.1-2. Some types of modules require additional configuration. To do it, just select a module from the list and click the "Setup" button next to the list. The configuration of the module is described below.

You can see some types of modules on the "Log file" tab. To configure such a module, you should select it from the "File type" list and click the "Advanced" button.

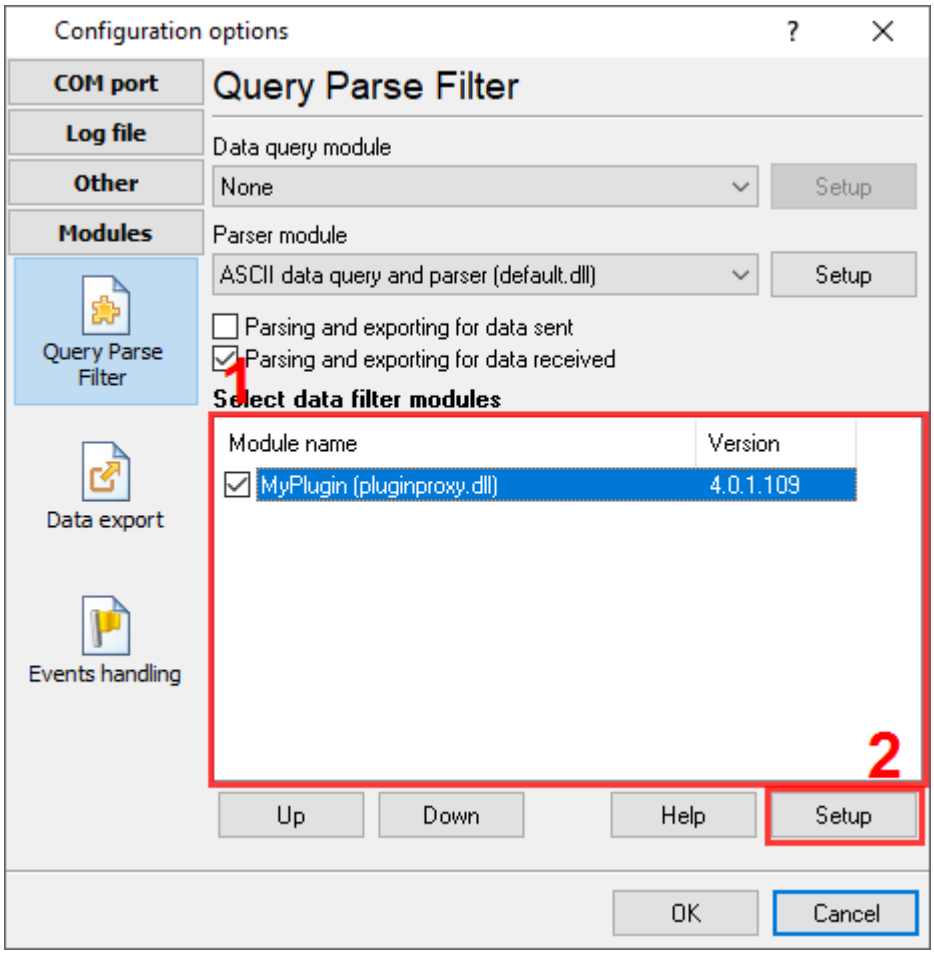

Fig. 1. Example of installed module

4 Glossary

Main program - it is the main executable of the application, for example, Advanced Serial Data Logger and asdlog.exe. It allows you to create several configurations with different settings and use different plugins.

Plugin - it is the additional plugin module for the main program. The plugin module extends the functionality of the main program.

Parser - it is the plugin module that processes the data flow, singling out data packets from it, and then variables from data packets. These variables are used in data export modules after that.

Core - see "Main program."

5 Setup

Module "Plugin proxy" does not have its own settings window. When you click the "Settings" button in the main program window, a call through API is redirected to your developed module. Your module may not have dialog boxes at all.

Follow these steps to connect your module:

1. Based on the example, you need to create a DLL with your implementation of the module.
2. Place your developed module in the folder next to pluginproxy.dll (in the program folder\plugins\pluginproxy).

6 API

After installation, in the module folder, you can find an example for Visual C ++ 2015 that implements the functions of data filter, data parser, and data export modules. You can use one of the examples as a prototype.

Note:

1. When developing your module, please keep in mind that all API functions can be called from different threads (excluding aggPluginShowOptionsDialog). Therefore, all code must be thread-safe.
2. A DLL of your module can be loaded in different configurations with different settings.
3. Do not let exceptions leave your DLL. Everyone must use try-catch constructions.

The following functions may be present in the code of your module.

aggPluginCreate

void __stdcall aggPluginCreate(_di_IaggPluginW Sender)

Sender - specifies API interface module. This parameter is also present in other API calls.

The main program calls this function when it creates a new module instance in one of configurations. Here you can initialize data structures of your module, as well as specify the following parameters:

1. Module name that will be displayed in the list of modules.
2. One or more module types that are implemented in this DLL. Depending on the type of module, it will appear in the appropriate list in the main window of the program (some functions are called in the context of a certain type of module).

The function that is called next is aggPluginLoad.

aggPluginLoad

void __stdcall aggPluginLoad(_di_IaggPluginW Sender, const PPluginCallRec pCallRec, int &Result)

pCallRec - module call parameters.

The main program calls this function as soon as the module has been created. Calling this function means that the module is supposed to read the settings from data storage (if required) and start working.

Result - the result of the module loading function. If the module returns mrtSuccess, that means the main program considers that the module has been loaded successfully and is ready to proceed. If a different result is returned, the main program will report problems and stop the execution of the module.

aggPluginUnload

void __stdcall aggPluginUnload(_di_IaggPluginW Sender, const PPluginCallRec pCallRec, int &Result)

Result - the result of the module unloading function.

The main program calls this function before removing the module instance. Before returning from this function, the module must stop all processes, all long-running operations, and save the data.

aggPluginDestroy

void __stdcall aggPluginDestroy(_di_IaggPluginW Sender)

The main program calls this function before removing the module instance. Immediately after this call, DLL may be unloaded from the data memory. Therefore, to avoid memory leaks, the module must delete all created objects in the data memory.

aggPluginLoadSettingsNow

void __stdcall aggPluginLoadSettingsNow(_di_IaggPluginW Sender)

The main program can call this function in the following ways:

1. After calling aggPluginShowOptionsDialog in one of module instance and changing settings.
2. After restoring a copy of configuration from a file.

When calling this function, the module must read all saved settings, and, if necessary, apply them in the working module. When creating a module, you must take into account that settings may be missing in the data repository. In this case, default settings should be used.

aggPluginSaveSettingsNow

void __stdcall aggPluginSaveSettingsNow(_di_IaggPluginW Sender)

The main program can call this function when creating a backup copy of the configuration. When calling this function, the module must save all settings to data storage (registry or INI configuration file).

aggPluginShowOptionsDialog

void __stdcall aggPluginShowOptionsDialog(_di_IaggPluginW Sender, const PPluginCallRec pCallRec

The main program calls this function when the user clicks the "Configure" button in the main program settings window. The module can display a window with settings or immediately return control to the main program.

Success - if the function returns TRUE, that means the main program assumes that settings have been changed, and they must be loaded in other module instances. Therefore, a call to aggPluginLoadSettingsNow in other module instances may follow. Modified settings in this module instance must be applied before exiting this function. The main program expects the module configuration window to exist only while calling this function.

aggPluginEvent

void __stdcall aggPluginEvent(_di_IaggPluginW Sender, const PWideChar Event, const PPluginCallR

Event – the identifier of the "PORT-CLOSE" event.

Result - if the function has handled the event, it must return mrtSuccess.

The main program calls this function when the module "subscribed" to receive events during the module creation step, and such an event was generated in the main program or another module.

aggPluginExecuteCustomCommand

void __stdcall aggPluginExecuteCustomCommand(_di_IaggPluginW Sender, const PWideChar Comm

Command - identifier of command type "MY-COMMAND."

Result - if the function has handled the event, it must return mrtSuccess.

When the module is created, it can declare the availability of its commands. Own commands appear as menu items in the program main window. The main program calls this function when the user clicks on the corresponding menu item.

aggPluginSetParserItems

void __stdcall aggPluginSetParserItems(_di_IaggPluginW Sender)

The main program calls this function when updating the list of parser variables. Typically, the function call occurs before calling aggPluginShowOptionsDialog and is applicable only for the modules of the type "Filter" (mttFilter), "Data export" (mttDataExport), "Events handling" (mttEventsHandling).

aggPluginProcessData

void __stdcall aggPluginProcessData(_di_IaggPluginW Sender, const PPluginCallRec pCallRec, int &Result)

The main program calls this function when the module must process any data. The function is applicable only for modules "Data query" (mttDataQuery), "Parser" (mttDataParser), "Filter" (mttFilter), "Data export" (mttDataExport), "Events handling" (mttEventsHandling).

If the function call is executed for a parser type module, then incoming data in pCallRec contain a pointer to the block of recently received data. In this case, all incoming data are additionally saved in a special buffer. In all other cases, incoming data contain a pointer to the list of values.

Result - if the function has handled the data, it must return mrtSuccess.

aggPluginGetData

void __stdcall aggPluginGetData(_di_IaggPluginW Sender, const PPluginCallRec pCallRec, int &Result)

The main program calls this function when the parser module reports the presence of “pending” data. For example, such data may appear when processing timeout data.

aggPluginStateChanged

void __stdcall aggPluginStateChanged(_di_IaggPluginW Sender, const DWORD ModuleTypes)

The main program calls this function when the module is supposed to initiate one of executable functions. For example, if the module can operate as a data query module and a parser, but the user has selected the module only in the "Parser" list.

ModuleTypes - bitmask of the types of modules that should be enabled.

A module can additionally query the module parameter "ModuleEnabled" to get the mask of all enabled module types.

7 Error codes

While executing the functions an API module can return the following error codes:

mrtSuccess = 0. Successfull completion. No errors.
mrtInvalidParameter = -1. Invalid parameter.
mrtNotSupported = -2. Not supported.
mrtException = -3. Unhandled exception error.
mrtNotInitialized = -4. Module not initialized.
mrtError = -5. Other errors.

Error codes are declared in header file "aggPluginModule.h".

8 Module API

The file "aggPluginModule.h" contains data types and class "CaggPlugin" in C ++ programming language which enables working with API modules.

Types of modules

mttDataQuery = 0x01. Data query module.
mttDataParser = 0x02. Data (handling) parsing module.
mttDataExport = 0x04. Data export module.
mttEventsHandling = 0x08. Events handling module.
mttLogging = 0x10. Data logging module. Available only in Data Logger Suite.
mttFilter = 0x20. Data filter module.
mttDeviceLayer = 0x40. Data source module. Not available via API.
mttMenuExt = 0x80. Program main menu items extension module (user commands).

Input or output data

Data type:

TDataRecType : unsigned char {
pdtNone, // no data.
pdtParamList, // list of values is passed to pParams.
pdtBuffer, // pointer to data buffer in pData.
pdtObject, // random object in pObject (not used in API).
pdtString, // string of ANSI ASCII characters, pointer which is passed to pStr.
pdtDWORD, // simple numeric value that is passed to dwData.
pdtCommand, // command, the code of which is passed to dwData (see below). Is passed from the module to the data source. Not all data sources support commands.
pdtNotify, // notification, the code of which is passed to dwData (see below). Is passed from data source to modules. Not all data sources pass notifications.
pdtParamListObj // object with a list of parameters (obsolescent).
};

Commands and notifications:

pdtCommand_ClientConnect = 0x0101; // client connected (IP data source => module).
pdtCommand_ClientDisconnect = 0x0102; // client disconnected (must be disconnected) (IP data source <=> module).
pdtCommand_TryRestoreConnection = 0x0103; // restore connection (data source <= module).
pdtCommand_ClientBlock = 0x0104; // client block (IP data source <= module).
pdtCommand_BufferCleanupEnable = $0105; // set an indication that the parser does not want to receive and handle data from the client that has already disconnected (main program <= module-parser).
pdtCommand_BufferCleanupDisable = $ 0106; // remove indication (main program <= module-parser).
pdtCommand_GetQueueSize = $0201; // get the amount of data in the process queue (data source => main program).

struct TDataRec

```
{
	DWORD dwTag; // undefined tag.
	DWORD dwDataSize; // size of input data. Can be zero. It only has value for data type pdtBuffer or pdtString.

	TDataRecType bDataType;
	union
	{
		struct
		{
			DWORD dwData;
		};
		struct
		{
			char *pStr;
		};
		struct
		{
			void* pObject;
		};
		struct
		{
			TaggParamsList* pParamsObj;
		};
		struct
		{
			void *pData;
		};
		struct
		{
			_di_IaggParamsListA* pParams;
		};

	};
};
```

Parameters of API function calls

```
struct TPluginCallRec
{
	DWORD dwSize; // size of data structure TPluginCallRec
	Byte bVersion; // data structure version
	DWORD dwModuleHandle; // unique ID of module.
	DWORD dwModuleIndex; // unique ID of module instance
	DwDataSource DWORD; // unique ID of data source
	DWORD dwClientID; // unique ID of client in the data source. Jointly, dwDataSource and dwClientID uniquely identify source of data.
	TModuleTypes aCallModuleType; // mask of module types in the context of which API function is called. Normally, only one module type is specified.
	TaggParamsList* aParameters_; // pointer to class - list of additional call parameters. Obsolescent. Not in use.
```

```
        TDataRec aDataIn; // input data.
        TDataRec aDataOut; // output data.
        unsigned char aClientName[50]; // data source name for dwClientID. Can be empty and filled with zeros.
        _di_IaggParamsListA aParameters; // interface for calling list of additional call parameters. Read only.
};

typedef TPluginCallRec* PPluginCallRec;
```

Please note that all unique identifiers may change when the main program is restarted.

API module functions

In the case of the successful execution of any functions below, the result is 0 (S_OK).

HRESULT __stdcall Log(const byte Level, const BSTR str)

The function displays the message **str** in the program message log. **Level** can be the following: 0 - error, 1 - warning, 2 - information.

HRESULT __stdcall GetModuleParameter(const BSTR AName, OleVariant* AResult)

The function returns epy module parameter value with the **AName** name.

HRESULT __stdcall SetModuleParameter(const BSTR AName, const OleVariant &Value, bool* AResu

The function changes the module parameter value with **AName** to **Value**. Note that only a limited list of parameters is available for modification.

HRESULT __stdcall NewData(_di_IaggParamsListW* List)

The function returns a pointer to the interface of "new" parameters list. It is only used in modules of type mttDataParser and mttFilter. The parameters extracted from the data packet are added to this list by the parser. Values generated by data filtering module are also added to the list.

HRESULT __stdcall PluginStorageAdd(const BSTR AId, const PVOID AData)
HRESULT __stdcall PluginStorageGet(const BSTR AId, PVOID* AData)
HRESULT __stdcall PluginStorageDel(const BSTR AId, PVOID* AData)

These functions allow us to work with the data storage of the module. The data storage is individual for each module instance. The data in the data store are saved between API functions calls. It is up to the module itself to delete data from the storage.

AId - the identifier of the value in a data store.

AData - a pointer to a data.

HRESULT __stdcall GetParserItems(OleVariant* AResult)

The function returns a list of parser variables in the form of a character set (BSTR), divided by CR + LF characters into several strings. Each string contains a description of one variable of the parser in the form:

DESCRIPTION|NAME|DATA_TYPE|DEFAULT_VALUE

DESCRIPTION - user-defined textual description of a parser variable.
NAME - the name of parser variable (usually, only letters of Latin alphabet A-Z, numbers and underscore).
DATA_TYPE - data type, if it is specified in the parser. If it is not, the data type is "String." Possible data types: String, Memo, Bytes, Blob, Boolean, Float, Smallint, Word, Integer, Date, Time, DateTime, DWord, Byte, Shortint, Int64, Currency.
DAFAULT_VALUE - default value if it is specified in the parser.

HRESULT __stdcall GetConfigs(OleVariant* AResult)

The function returns a list of data sources that are specified in the configuration in which the module can be loaded. It is returned in the form of a character set (BSTR), divided by CR + LF characters into several strings. Each string contains a description of one data source in the form:

NAME<TAB>DATA_SOURCE_ID

NAME - name of data source.
<TAB> - tab character ASCII (0x09).
DATA_SOURCE_ID - unique identifier of a data source in the form of HEX string. You can convert it into DWORD number and use in calls.

HRESULT __stdcall SendData(const PPluginCallRec pCallRec, const PBYTE AData, const int ASize, int* ARes

The function sends a set of bytes to the data source.

Note that not all data sources support data submission.

pCallRec - specifies a data sender and a data receiver. This parameter can be NULL. In this case, data are sent to all data sources in this configuration.

AData – the pointer to the data buffer.

ASize - the size of the data buffer in bytes.

AResult - a result of data submission. Error codes:

>= 0 - successfully sent.
-1 - sending is not supported.
-2 - unknown error.
-3 - sending is disabled.
-4 - invalid parameters.

-5 - no data source found.
-6 - configuration is blocked (loading settings, deleting a configuration, etc.) and data sending is impossible.
-7 - invalid sender ID (invalid value specified).
-8 - wrong sender ID (not applicable to this configuration).
-9 - module initialization is not complete
-10 - configuration in the module is not completely initialized.
-11 - unhandled exception.
-12 - data source closed (pause has been pressed in the main program window).
-13 - data source temporarily stopped sending data (output data buffer overflow during the connection process, etc.).
-14 - other, the internal error of the data source module.

HRESULT __stdcall GetArgs(const _di_IaggParamsListW* AArgs)

The function is used to prepare a list of parameters for sending an event or saving a configuration. It returns a pointer to a list of values that you can fill in and pass as a parameter to the next function.

HRESULT __stdcall SendEvent(const BSTR AEventId, const _di_IaggParamsListW AArgs)

The function submits the **AEventId** event with parameters **AArgs** to the main program and other modules. A list of parameters can be prepared using GetArgs. **AArgs** can be NULL. The list of standard events is given in the "Parameters" section.

HRESULT __stdcall LoadSaveConfig(const bool ASave, const _di_IaggParamsListW AArgs)

The function, depending on the **ASave** argument, either loads or saves in the module configuration values from **AArgs**. You can prepare a list of arguments using GetArgs and fill in the necessary values. The name of the value in the list may contain the character "\". In this case, the name of the value will be interpreted as the path and name separated by this character.

9 Module parameters

In this section, you can see the list of parameter identifiers with which you can access module parameters by calling the appropriate API function. The character "[w]" in a parameter description means that the value can be changed.

ModulePath - (string) the path to the folder where the module file is installed.

ModuleName - (string) the name of the module.

ApplicationFullName - (string) the name of the main application.

ModuleRegistryRoot1 - (dword) the registry branch where settings are stored:
HKEY_LOCAL_MACHINE или HKEY_CURRENT_USER

ModuleRegistryRoot2 - (dword) the backup registry branch.

ModuleRegistryPath - (string) the path in the registry where settings are stored.

INIFile - (string) the name of the INI file where settings are stored. If the name is specified, then settings are not stored in the registry but in a file.

INISectionPrefix - (string) if settings are stored in a file, then the parameter stores the prefix of the INI section of the settings file for this configuration and module.

DisplayFullVersion - (string) the version of the module.

IsTemporaryLoad - (bool) indicates that the module is temporarily loaded. For example, to edit settings.

LogTitle - (string) [w] the module title, which is displayed in the log file with program messages before any message from the module. It may be the same as the module description.

Description - (string) [w] the module description that is displayed in module lists in the main program.

EventsSupported - (string) [w] a list of events that a module is ready to receive and handle. Identifiers are separated by commas. By default, this list is empty. Standard events:

LOG-MESSAGE - new text in the program message log.
NEW-LOG-FILE - a new log file with data has been created.
LOG-FILE-DELETE - deletion of the old log file with data.
NEW-DATA-PACKET - a new data packet from the parser.
ERROR-WRITE-FILE - error while writing to log file.
PORT-OPEN - data source is open (started).
PORT-CLOSE - data source is closed.
CONFIG-CHANGE - configuration has been changed by the user.

USER-LOGOOF, USER-LOGON - in service mode, end or beginning of user session.
STOP-SERVICE - in service mode, service stop.

10 Buffer

When the module is operating in parser mode, the API creates a 65 KB input buffer for each data source and client. All received data are automatically written to this buffer. When the aggPluginProcessData API function is called, a pointer to the last portion of the received data is passed to it. Note that this portion of data may not contain the entire data packet. The task of the parser, after data handling, is to clear this buffer. When the buffer overflows, the old data will be replaced by new data.

The full class code for working with the buffer is given in the file "fifobuffer.h." To get access to the buffer object, you need to call the class method "CaggPlugin":

CFifoBuffer GetDataSourceBuffer(const PPluginCallRec pCallRec)

CFifoBuffer class object has the following methods and properties that enable interfacing with the buffer:

Size - returns the number of bytes in the buffer.

Data - returns a pointer to the beginning of the buffer.

void Clear() - Clears the buffer completely.

void Shift(const int ASize) - Removes "ASize" bytes at the beginning of the buffer (shifts the buffer).

int Find(const PBYTE ASign, const byte ASignSize, const unsigned int Offset = 0)

It searches for byte signature "ASign" of the length "ASingSize" bytes in the buffer, starting from the Offset. Returns the offset of the found signature or "-1".

11 List of values

File "aggParamLst.h" contains data types and class "CaggParamsList" in the programming language C++ which enables working with a list of values.

A list of values is a set of items. Each item in the list is an object that has a name and a value. This list can be divided into one or more virtual strings by a special delimiter element, that has a fixed name "NEW_ROW."

An object of the class "CaggParamsList" has the following methods and properties.

Clear() - clears the list completely.

Delete(index) - removes an element with index from the list.

CopyFrom(object [, start_index = 0[, end_index = -1]]) - copies the specified number of items from the object list. Additional **start_index** and **end_index** parameters specify the start and end indexes of items in the source list. If **end_index** is -1, then all items in the list are copied down to the end.

ItemByName(name) - returns a list item by its **name** (string).

ItemIndexByName(name) - returns the index of a list item by its **name**.

ItemValueByName(name) - returns the value of a list item by its **name**. A value can be of any simple type, including Null.

ItemValueByNameDef(name) - returns the value of a list item by its **name**. If the element with this name is not found, it returns the **default** value.

InsertItem(index, name, value) - inserts a new element with **name** and **value** into the list, at the **index** position. Returns the added item.

SetItem(name, value[, canadd = False]) - changes the value of a list item with **name** to a new **value**. If the additional **canadd** parameter is True and the value is not in the list, then a new value is added at the end of the list. Returns the found or new list item.

AddItem(name, value) - adds a new element with **name** and **value** at the end of the list. Returns a new list item.

AddItemCopy(item) - adds a copy of the **item** at the end of the list. Returns a new list item.

FindRow(item, start_idx, is_row_end_sign) - searches for index of the last element of a string starting from start index **start_idx**. Returns True if a string is found (one or more list items after **start_idx**). **end_idx** is a variable whose value returns the index of the last element of the string. **is_row_end_sign** is a variable that returns True if **end_idx** points to a special string separating element.

Example:

```
int nStartIdx = 0;
int nEndIdx = 0;
bool bEndRowSign = false;
while (aData.FindRow(nStartIdx, nEndIdx, bEndRowSign))
{
	//
	// your code here
	//

	nStartIdx = nEndIdx + 1;
}
```

NewRow() - adds a new separator element at the end of the list.

Count is a list property that contains the number of items in the list.

Items[index] - the property that allows getting a list item by its index.

List item

Each list item has two properties:

Name – the name of the element (string value).

Value – the value of the element. A value of arbitrary type, which can also be "Null."