Initializing a continuously operating custom DLL for OpcDbGateway[i]

Vladimir Palacka SAE - Automation, s.r.o., (Slovakia). Trenčianska 19, NOVA DUBNICA 01851, email: vladimir_palacka@saeautom.sk

Keywords:

Custom DLL, non-blocking call, shared process image memory

Abstract:

OpcDbGateway enables to enhance base configurable functionality using custom DLLs. Data exchange between core runtime application and the custom DLL can be provided over a C-style functional interface or over shared process variables of the process image memory. Advantages and problems of both methods are discussed in this article.

Custom DLLs for OpcDbGateway runtime application have two basic modes of operation:

1. Single shot - which represents a ***DoProcessIO*** **call from a synchronous or asynchronous thread** of the OpcDbGateway runtime application that is **started by the configurable *CALL DLL* command**.

2. Continuous - after the start of the runtime application, the main application thread of the DLL will start (and eventually also other threads) that work all the time when the main executive application runs.

The **first mode** has a few advantages. **First**, there is C-style interface between OpcDbGateway runtime and custom DLL consisting from two functions[1] that need customizing: *`GetCountOfIO`*, *`DoProcessIO`*:

```
#define DllExport extern "C" __declspec (dllexport)
// The function retrieves a specified count of inputs/outputs.
DllExport void  WINAPI GetCountOfIO (LPWORD lpInputs, LPWORD lpOutputs);

// The function processes all values from input buffer and retrieves the processed values into output
buffer.
DllExport bool  WINAPI DoProcessIO (const CComVariant* lpInputs, WORD wInCnt,
                                                CComVariant* lpOutputs, WORD wOutCnt);
```

Using of C-style interface is one of premises to be able to use DLL compiled with a compiler different than the client OpcDbGateway runtime application.

[1] There are also other functions used as interface to runtime application that need not be modified *OnStart(), OnStop(), OnInitMemory(CProcessImageMemory* lpPIM) and functions to provide information about DLL for configuring application `GetProductName`, `GetProductVersion`, `GetCompanyName`, `GetLegalCopyright`, `GetDescription`.*

Second, the *DoProcessIO* function has as parameters pointers to a contiguous array of *CComVariant*[2] type values with an adjustable number of inputs' (*WORD wInCnt*) and outputs' (WORD wOutCnt) operands and, given the *CComVariant* type properties, also with different primitive data types. The first part of the array is used as input and the second as the output buffer. *DoProcessIO* uses data from the input buffer and, after processing them, fills results to the output buffer. This way, high flexibility of the interface for different types of custom DLLs is provided. However, there is a small constraint - the input and output parameter field sizes for a given custom DLLs must be hard-coded in the *GetCountOfIO* function in the DLL. The OpcDbGateway runtime application detects the size of *CComVariant* I/O array by calling *GetCountOfIO* function from runtime application in custom DLL and initializes input buffer from a continuous array of memory operands (MO) in Process Image Memory (PIM). When *DoProcessIO* returns, it updates PIM by values from the output buffer. This way, it is provided that reading /writing access is done **without direct access to PIM from custom DLL**. It means that custom DLL does not need to contain the class *CProcessImageMemory* providing read/write access to PIM.

The *CALL DLL* configurable command (as every configurable command of the OpcDbGateway) e.g. *CALL DLL (Ret1, DllExample, IO)* has two input parameters: the custom DLL name e.g. *DllExample*, the MO name where the I/O parameters array begins[3] e.g. *IO* and one output parameter – an MO for the *DoProcessIO* call result (TRUE/FALSE) e.g. *Ret1*. Experience with compiling the source code of the sample DLL in different versions of Visual Studio shows that there are no start up or run-time problems in this mode. The disadvantage of the first mode is that the function ***DoProcessIO* in the DLL must not be blocking.** Otherwise, the synchronous or asynchronous thread of the runtime application from which *DoProcessIO* is called will be blocked. Since information on the success of the function call is also returned, it is possible to determine whether the output values are correct and eventually to transfer error message from *DoProcessIO* in one of *CComVariant* type variable.

In the **second continuous mode**, the customer's DLL communicates with the runtime application directly through PVs. This means that, unlike the previous mode where the functional interface is used, there is a kind of "data interface" – PIM shared between runtime and DLL. Pointer to the 1st PV is sent to DLL using function *OnInitMemory*:

```
// The function is called after the main process application memory is successfully initialized.
DllExport void  WINAPI OnInitMemory         (CProcessImageMemory* lpPIM);
```

This mode is suitable for example for implementing a **communication driver to receive unsolicited messages from a data source** (e.g., from a MQTT broker). PVs can be directly used to parameterize functionality of the communication driver (it means reading PVs initialised within runtime application) or writing of data from external source to PVs, and synchronising threads in a threaded DLL with the runtime application. Competitive access from different threads is made possible by using the critical section synchronization mechanism. The DLL needs a proprietary copy of the functional code for accessing PIM, the same as used by runtime application, instead of functional interface as used in the previous case. Using the mentioned copy of the code is risky in case that this functionality is upgraded in the runtime application. Experiments show that this can function in a general way only in case if custom DLL and runtime application are compiled and linked with the

[2] CComVariant type enables to transfer data in a self-describing data format. For each value transmitted, you send two fields: a code specifying a data type and a value represented in the specified data type. The sender and receiver must agree on the set of possible formats.

[3] As there is only one MO to specify both input and output array they must be configured as one continuous array.

same version of compiler and linker – in the VS2005 environment. However, the main disadvantage of a custom DLL in the continuous mode is that PV indexes for the data interface must be hardcoded in the DLL. It has following implications. Firstly, the OpcDbGateway configuration application allows automatic allocation of free indexes for PVs. As a consequence, overlaps of PVs assigned in the configuration and hard-coded in the DLL may happen[4]. Secondly, as parameters transferred trough PVs with hardcoded indexes can be used, only one instance of given type of custom DLL with one set of parameters can be deployed.

The problem can be solved by one of following ways:

1. At the start of the runtime application, either not to run the continuously running thread of the DLL at all or to run it only in a mode of waiting for parameters. Indexes of PVs used for parameterizing can be transferred within calling *DoProcessIO*. Only after the parameters have been uploaded will the functionality of the special thread in the DLL be started.

2. Run a new continuously running thread from *DoProcessIO.* This method is advantageous in that it is always possible to start a new thread with the use of various parameters and thus have multiple running instances of functionality secured in the custom DLL. Of course, the possibility of stopping individual threads have to be also ensured. This can be done by using a parameter to indicate whether a *DoProcessIO* part of the code to start or to end the thread should be executed.

3. Completely exclude PIM sharing. Instead of transferring data from continuously running thread in DLL to send them in output buffer that is created by runtime application before calling *DoProcessIO* and filled after return *DoProcessIO.* This method is best from the point of view of having the possibility to create custom DLL in different versions of the development environment. It has also disadvantage that *DoProcessIO* need to be called periodically eventually to have also a buffer for the case that between two calls more unsolicited messages arrives.

Conclusions

The function *DoProcessIO* implemented in custom DLL can be used not only to implement new configurable non-blocking commands but also to parameterise continuously running threads within custom DLL, change the status of running threads and to provide periodical inbound and outbound communication between OpcDbGateway and external processes.

[i i] This work was supported in part by the European Commission in the scope of the project Flex4Grid (Prosumer Flexibility Services for Smart Grid Management), grant agreement 646428 — Flex4Grid — H2020-LCE-2014- 2015/H2020-LCE-2014-3

[4] The overlapping can be removed by correcting indexes of PV manually using configuring application, but still a problem persists – information about used PV indexes must be read from DLLs sources or put to the DLL description that the configuring application reads using function *GetDescription*.