Using standard streams of a child process to implement plug-ins.

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

Keywords:

Application development toolbox, Inter-process Communication, Communication Middleware, Communication over pipe

Abstract:

OpcDbGateway enables to process data collected from different external processes and to write data to them using different means of inter-process communication. This article describes possibilities and one concrete implementation of inter-process communication based on piping of standard output of an external process to OpcDbGateway runtime process. Ready-made or custom console applications communicating over pipes or parameters of command line interface string can considerably enhance ***configurable*** *and/or* ***programmed functionality*** *of integrated applications based on OpcDbGateway.*

Introduction

OpcDbGateway **is application software development product** consisting of **IDE** (GUI) for creating and debugging applications and a **runtime engine/application** without IDE. The OpcDbGateway configuring/programming methodology is alike as used in programmable logic controllers (PLC). It is based on using of ***Function Blocks*** (FB). They consist of ***configurable commands***. Runtime application executes the configurable commands that are saved in configuration created by IDE. There are different types of commands like arithmetic, logical, database, comparison, database statistic, generating user messages. FBs can be configured in a hierarchy started with FB *Main*. Configuring of applications using configurable commands **does not necessary require knowledge of a programming language**. User can choose different types of commands and their operands using select boxes in GUI.

Runtime engine provides two main modes/threads of execution - a **cyclic synchronous and** an **asynchronous one**[1]. The synchronous mode provides PLC-like periodic implicit communication with external data sources and processing data within FB *Main and its nested FBs.* Execution of FB *Main* is repeated with defined period. Except this, there is also processing based on prioritised ***events*** initiated by ***triggers*** of the type ***value*** **or** ***time***. Event in this context means an activity that has to be fulfilled like the start of an external program or call of FB. The trigger is a condition for executing of an event. Event based processing can be executed either in synchronous thread where triggers that initiate events are evaluated always only at the beginning of synchronous cycle, or in the asynchronous thread, where they are evaluated as fast as possible. Runtime engine provides near real-time functionality.

[1] There are also special threads under the hood for evaluating of triggers and for run of very slow activities. Other threads can also run within in-process plug-ins.

OpcDbGateway IDE enables creating of applications functionally equivalent with PLC applications and at the same time also as PAC (programmable automation controller) applications that are created by standard programming languages as C++. It enables e.g. access to databases, logging and alarming functionality and **communication with external processes.** An important feature of the OpcDbGateway GUI is **automatic mapping/configuring** of different types of interfaces with external processes like OPC servers/clients, DDE servers and different kinds of databases. There are also tools for different ways of logging, historical data and alarms processing. GUI offers also checker of configuration consistency and a graphical view of the application structure, logfiles and alarm viewers.

A central point of OpcDbGateway runtime engine is the ***process image memory*** (PIM). It contains **process values** (PV) that represent **input and output data from/to external processes data points and also data for internal processing** according to the configured functionality. Every PV contain its ID, a data value of the type VARIANT[2], time stamp, quality and type of time stamp – gained from an external process, or internal – created by the runtime engine. PVs are placed in PIM within one-dimensional array and can be addressed using their index (ID) in the array.

PIM status is persisted in configuration database when runtime application stops. After the restart, it is renewed and **one-shot FB named *Restart*** is called. By the first start of runtime with new or changed configuration, another **one-shot FB named S*tart*** is called instead.

Within configuration, PVs are referenced by **memory operands** (MO). Configurable commands use MOs as arguments and return values[3]. MOs have their names, that can be chosen for usage by configurable commands by the select box in GUI. MOs can be organised and shown in GUI in tree-like structures of folders. It is useful for the structuring of applications' data model. For the MO, address that represents ID of PV and also data type that specifies basic data type within the VARIANT value of the PV have to be configured. MOs can be used as triggering values **for triggers of type value**. Most used interface used by OpcDbGateway for the inter-process communication is the OPC server's interface. OpcDbGateway GUI application has built-in OPC client. There is a possibility to **configure** the **mapping of every MO to OPC items of the runtime engine's OPC server**. Using OPC client of the GUI offers then one of the means to monitor MOs, to write to them and to debug runtime functionality.

OpcDbGateway runtime application can run either as **out of process server started by an OPC client** (e.g. the one built into the GUI application) or as a **Windows service**.

Custom plug-ins and extensions

OpcDbGateway functionality can be fundamentally enhanced **using plug-ins and extensions.** The enhancements can be so broad that they provide core business logic or user interface for integrated application or it can be a implementation of an easy new configurable command.

Plug-ins and extensions can be implemented as **in-process** or **out of process** program modules. In-process modules run in one of the threads (synchronous or asynchronous) of the runtime core or in other threads executed within **custom DLLs** but remaining in the same process as the runtime core. Out of process modules are **external programs and scripts** that can be invoked either by using one of

[2] In fact, the VARIANT is encapsulated in CComVariant

[3] There are also other types of operands e.g. constants, database operands that enables addressing of a database table cell, table column or even whole table. There is also operand that represents configured name of custom DLL.

the **inter-process interfaces** (like OPC or DDE) provided by OpcDbGateway or as programs started using **command line interface (CLI) strings parameterised with actual values of some PVs**.

Plug-ins can function as a **one-shot** or **long time running**. Transfer of data between runtime and plug-in can be provided either by **input and output arguments of a function** called from one of runtime threads and implemented within plug-in or trough **PVs of PIM shared by runtime core and plug-in.**

To provide the functionality of custom configurable command, they need to have a **C-style interface to the runtime core application** and have to have a **non-blocking functionality** as not to block threads of the runtime core. The functionality of the one-shot in-process plugin can be invoked from runtime core by **configurable command *CALL DLL***. It provides calling of C-interface function ***DoProcessIO*** where can be implemented most of the plug-ins' functionality. Before calling it, the runtime core calls C-interface function ***GetCountOfIO***[4] to find out the number of input and output arguments for ***DoProcessIO***. Input arguments and placeholders for output arguments has to be placed **in a continuous array of PVs.** Index of the first PV of this array is used as the first input argument of configurable command *CALL DLL,* and the configured name of custom DLL as the 2nd one. Within *CALL DLL,* input arguments are copied from the continuous array of PVs before calling ***DoProcessIO***. After return from *DoProcessIO,* the *CALL DLL* writes values of output arguments to PVs. It means that working with PVs is in this case completely provided on runtime core site. This feature is important because **it enables to build custom DLLs in a different version of development environment as used for the runtime core**.

The custom DLLs, except functionality invoked by *CALL DLL* configurable command, can be affected directly through PVs. The DLLs controlled directly over PVs need to provide minimally a part of their functionality within an **individual thread that is started and eventually runs all the time parallel with other threads of runtime core.** Another possibility is using of *DoProcessIO* to start a new thread, to read periodically values from that thread and also to stop it using the same function. In such case, one of the arguments for *DoProcessIO* has to contain information which activity (start, read/write or stop) have to be fulfilled within the actual call. A disadvantage of this approach is a necessity of pooling data from the custom DLL thread by repeated calling of *DoProcessIO*. This problem can be avoided by **direct access to PIM from the custom DLL**. It is an effective way in case that the custom **DLL can be built in the same environment like the runtime core**. Unfortunately, as we are speaking about **custom** DLLs, it is often not possible. This problem can be solved by **out of process plug-ins** or by external applications. OpcDbGateway offers a few powerful client and server interfaces to provide inter-process communication like OPC client plus server and DDE client. Using them, it is possible not only to communicate with different external data sources and sinks but also to create plug-ins for OpcDbGateway. Unfortunately, these technologies are for most users relatively complicated. On the other hand, **console applications with their standard streams STDIN, STDOUT, STDERR are easier comprehensible, can be programmed in different programming languages and environments and so more users can find an easy own way to implement plug-ins**.

OpcDbGateway has not actually implemented interfaces over standard streams within runtime core but it is possible to implement them within semi-custom DLL (like in Figure 1) that will be to disposal as open source and also delivered with OpcDbGateway. In this case, it is supposed that DLL is built in the same development environment like the runtime core and so we can use already mentioned more effective data exchange between DLL and the runtime core using direct access to PVs in PIM. **To enhance OpcDbGateway runtime functionality by a user, it is then enough to choose a**

[4] It means that number inputs and outputs has to be hard coded in the custom DLL.

readymade or create an own console application. It enables even to enhance OpcDbGateway with applications for another platform and packaged as a Docker container.

The *Child process* in Figure 1 can be started using *CLI string* either from Runtime core as a configured event or from the *DLL* using programmed WINAPI function *CreateProcess*.

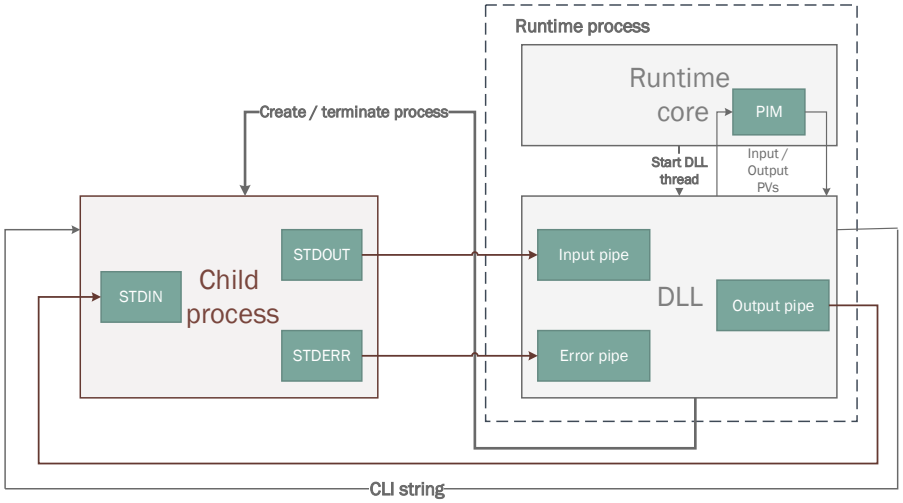

Figure 1 Custom DLL to connect STDIN, STDOUT and STDERR to OpcDbGateway runtime[5]

From the *DLL,* the process can be also stopped using the *TerminateProcess* programmed function. In both cases (from *Runtime core* or from *DLL*), the *CLI string* can be used as one communication input channel from OpcDbGateway runtime to the external child process because, except application path and eventually other parameters, **it can contain parameters that can be substituted by a set of actual PVs**. This channel is useful only if it is enough to send parameters only one time by the start of the child process. For the case that **the application has to run continuously and actual PVs has to be sent repeatedly**, it is necessary to send data from the *Output pipe* in *DLL* to the applications' *STDIN*. This is the 2nd input channel. There are 2 possible output channels for data from the *Child process* to *DLL* – from *STDOUT* to *Input pipe* and from *STDERR* to Error pipe.

Communication using CLI, STDIN, STDOUT.

Using plug-ins implemented as console applications, it is possible to implement not only new custom easy **configurable commands** and **pipelined filters** but also **communication drivers** for different types of devices.

There are following possibilities to configure or to program communication over CLI, STDIN and STDOUT with external processes in OpcDbGateway:

1. Outbound communication from OpcDbGateway to a data sink by periodic or one-shot start of the console application by **configurable event** using **command line interface string (CLI) parameterized by actual values of PVs** from PIM. This solution is described in the white paper „*Connecting OpcDbGateway to MQTT broker to monitor a home/building automation system*"[6] and it does not require a custom DLL.

2. Outbound communication based on writing to the standard input of the console application. This solution requires not only configuring but also implementing a custom DLL for OpcDbGateway

[5] Horizontal flowlines denote data flows and vertical control flows.
[6] http://www.saeautom.sk/download/OpcDbGatewayMQTT.pdf

runtime. It is used mainly when we need a continuous stream of output data without periodic starting of the console application.

3. Inbound communication from the STDOUT of the console application to the OpcDbGateway runtime, either continuous or one-shot. It always must be implemented using custom DLL. More universal kind of inbound communication is the continuous one, and so we describe a solution for this type of communication.

The test case (Figure 2) described within this article provides inbound and also outbound communication between OpcDbGateway and MQTT broker.

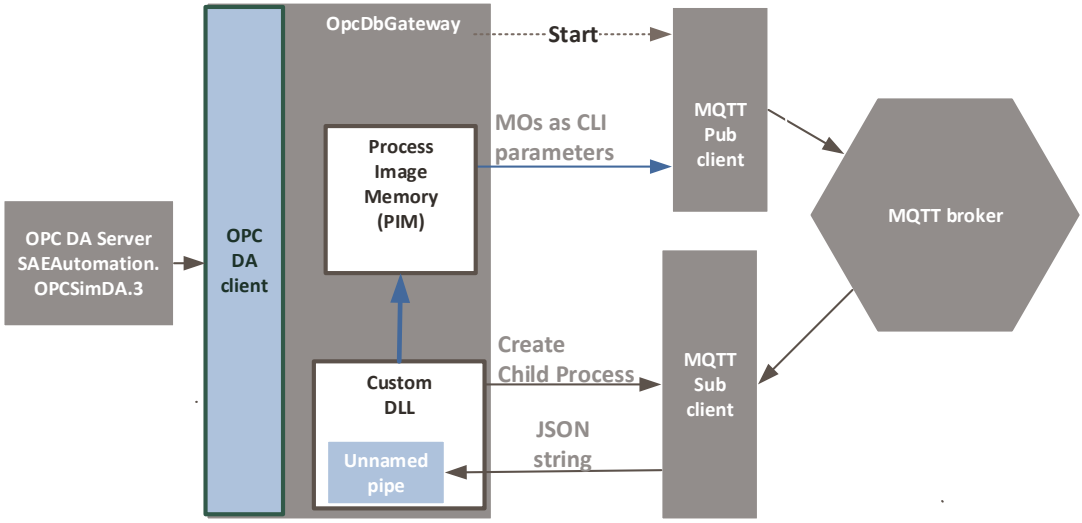

Figure 2 Test case - communication with MQTT broker

It combines the 1st approach where external application is started as configured event of the type "Call External Program" invoked periodically by the trigger of the type *time*. This way, **MQTT publish client** periodically sends messages with defined topic to MQTT broker. The messages are formatted as **JSON strings and parameterised by PVs** that are actualised from simulation OPC server[7], and with the system PV containing a time stamp. The event is started on the asynchronous thread.

In the same configuration, the **second part of the application** is implemented. It provides **receiving of messages from MQTT broker** that have the same topic as used for above described sending. For that purpose, a **continually running** custom DLL is started. From the DLL, a new child process – the **MQTT subscription client** is started. DLL provides reading of JSON messages from STDOUT of the child process using unnamed pipe and saves them to defined PV. To keep this approach generic enough, it is supposed that the JSON string is then parsed in another custom DLL.

Parameterizing of continuously running custom DLLs

Continuously running custom DLL uses data exchange with OpcDbGateway runtime core over shared PVs. It is started just after starting runtime core and need not be activated by the configurable

[7] SAEautomation OPCSimDa OPC server is delivered together with OpcDbGateway and can be used for testing.

command *CALL DLL*. To be able to communicate with runtime core, the indexes of shared PVs must be known by the custom DLL. The easiest way would be to hard code them in DLL. But, it has a few disadvantages. Firstly, it can be a problem for using the same semi-custom DLL more times in the same application when a different placement of a set of input and output parameters for every DLL usage is necessary. Secondly, the algorithms for automatic proposing of PV indexes for related MOs in GUI has not information about such in DLL hard-coded indexes and so duplicity in PVs configuring can happen. Because of this, we need to find a way to transfer information where (in which PVs) can the semi-custom DLL find information from the applications' configuration. It can be provided by already mentioned configurable command *CALL DLL* that provides invocation of the function *DoProcessIO* within custom DLL. Contrary to the standard usage of the function *DoProcessIO* when it is used for implementation of one-shot functionality of the custom DLL, it provides only transfer of configured parameters for usage by continuously running threads of the custom DLL. The thread where the main functionality of custom DLL is executed has to wait in a loop till necessary parameters are transferred.

Taking into account above considerations, the custom continuously running DLL for the inbound communication can be implemented following way (Figure 3):

- Just after starting of OpcDbGateway, runtime application core calls function *OnStart* implemented within DLL and by that new thread is created.
- New thread continuously runs within call back function *IOThread*. At the beginning, **it waits in the loop for sending of information** from the OpcDbGateway runtime like e.g. a placement of PVs for command line string for starting of the console application and the placement of the from MQTT broker received message.
- Above mentioned parameters are transmitted by calling the DLL function *DoProcessIO* from the OpcDbGateway runtime. It is configured using *CALL DLL* configurable command.
- After receiving parameters, the console application is started from the function *IOThread* using WINAPI function `CreateChildProcess`. Within that, unnamed pipe for STDOUT of the console application is created.

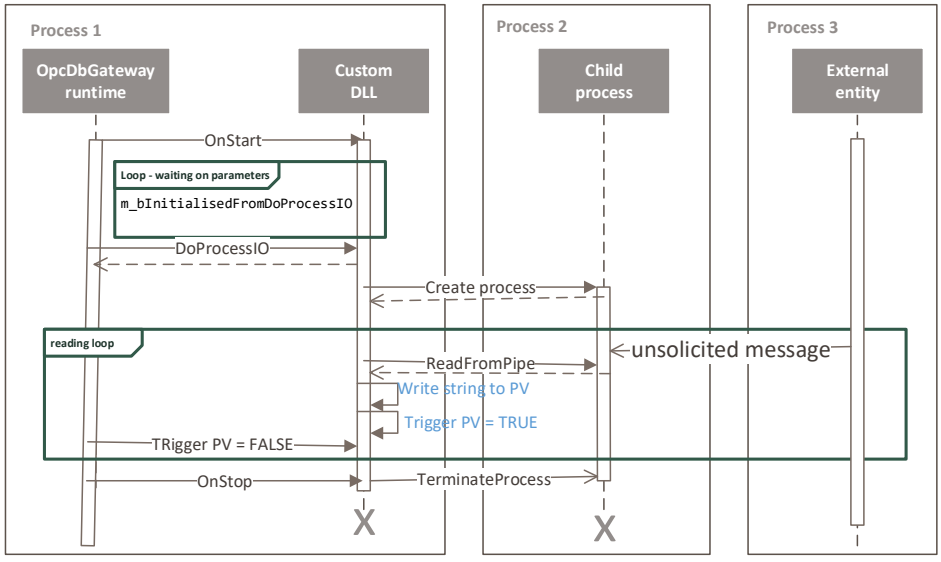

Figure 3 Sequential diagram receiving of unsolicited messages by OpcDbGateway runtime application from external entity using a child process started from custom DLL

- Within *IOThread,* **new loop with reading data from the pipe and saving them directly to PV** is started.
- Above mentioned loop can be disrupted for one of following reasons:
    - OpcDbGateway runtime got request to stop and so calls the DLL interface function *OnStop,*
    - By calling *DoProcessIO* with STOP request,
    - By receiving a request to stop from an external entity with that the console application communicates.
- After receiving a kind of STOP request, the loop is interrupted and the DLL thread is stopped.

It is also important to mention how messages from child process are processed within OpcDbGateway runtime core according to the configured functionality. The messages are coming asynchronously and so it is necessary **to inform the OpcDbGateway runtime core when new message comes and to provide that newer message does not overwrite the older** one without noticing it. Because of this, any time when a message arrives also a **PV used as a triggering value** is set to TRUE in the DLL. Within OpcDbGateway runtime core, it activates configurable trigger of the type *Value* that starts a configurable event of the type *Call Function Block*. It is necessary to configure a function block that provides:

- next processing of received message (e.g. call another DLL to parse JSON string),
- confirming that the message has been processed by the setting of triggering PV to FALSE.

DLL thread is waiting in loop till this confirmation comes and only then it reads new message from the pipe. Of course, for some applications also providing a messages buffering can be necessary.

Implementation of test application

The test application is implemented using **configuration** *MQTTPubSubExample3.odg* and **MS Visual Studio C++ project** of the type ***regular DLL dynamically linked to MFC* with support for ATL**. The project was developed modifying original *Example.dll* sources delivered with OpcDbGateway and converted from version for Visual Studio 2005 to Visual Studio 2017. Although testing with debug version of the project was without a problem (except for the big size of DLL about 8 MB) the release version was (as expected) not usable. Because of this, the project was rebuilt in the development environment for OpcDbGateway runtime core VS2005.

The **application configuration** consists of **two parts** – the **part for MQTT publish** identical as described in the above-mentioned white paper and the part for **MQTT subscribe**. The **part of configuration for MQTT subscribe** consists of **parameterization phase** that is implemented within the FB *FBforCallReadStdOutChildDll*[8] called from one of one-shot FBs ***Start*** or ***Restart,*** and from **execution phase** provided on runtime core site by the FB *FBConfirmNotificationFromReadChildStdOut* called as an **asynchronous event** *EvCallFBNotificationFromReadChildStdoutArrived* triggered by **trigger of the type value** *TrNotificationFromReadChildStdOutArrived* that is activated by value *TRUE* of the PV *DataForReadChildStdOutDll/MoIndexForNewStdOutNotification*.

[8] In the *FBforCallReadStdOutChildDll,* an information about indexes of PVs that are used as an interface between runtime core and DLL is initialized according to configured constants and by the CALL DLL configurable command is transferred to DLL.

The **parameterization phase** consists of preparing data that have to be submitted to custom DLL and of the submitting itself provided by configurable command *CALL DLL* that provides calling of C-interface function *DoProcessIO* in the custom DLL. Memory operands for submitted data are configured within folder *DataForReadChildStdOutDll* (Figure 4). When configuring this data, it is important to provide that input data for *DoProcessIO* and output data are in one continuous array[9].

Name	Var...	Bit	Bit Nr.	DataTy...	Description
MoActualStatus	1030	0	0	WORD	Actual status of the status machine
MoCliString	1031	0	0	STRING	CLI string for child process
MoIndexForChildStdoutString	1032	0	0	WORD	PV index for string from stdout
MoIndexForNewStdOutNotification	1033	0	0	WORD	Index of PV for notification by new stdoutstring
MoLogFromDoProcessIO	1034	0	0	STRING	Error / info message from DoProcessIO
MoResultOfCallDllReadChildStdOut	1035	0	0	BOOL	
MoCallFBactivationCondition	1036	0	0	BOOL	
MoStdOutString	1037	0	0	STRING	
MoNotificationNewStdout	1038	0	0	BOOL	
MoCopyOfLogStrFromCallDll	1040	0	0	STRING	

Name: DataForReadChildStdOutDll

Figure 4 Memory operands for interface between OpcDbGateway runtime and custom DLL

The functionality of the custom DLL is implemented as a status machine that is initially stopped. By submitting of parameters from configuration and setting of the MO *MoActualStatus to the value 1,* it is started. It can be also stopped again, in case that *MoActualStatus* is set to 2 and *CALL DLL* is repeated with the same parameters. The MO *MoCliString* contains index of PV from that the custom DLL can read CLI string for the child process that has to be started by DLL. The MO *MoIndexForChildStdoutString* contains an index of PV where DLL will put data that are read from STDOUT of the child process. The *MoIndexForNewStdOutNotification* contains an index of PV that will be set to *TRUE* in case that new data from STDOUT has been read. It can be used as triggering variable to initiate a FB for processing received data. The *MoLogFromDoProcessIO* can contain error message from executing the function *DoProcessIO.* It is useful mainly by creating of configuration to find out if parameters for *DoProcessIO* are not properly initialised.

The indexes of PV are set from configured constants. E.g. the value of *MoIndexForChildStdoutString* is set by value of constant *CoMoIndexForStdOutString.* The CLI string for the child process is defined in the constant *ConstLIstringForReadChildStdOut.*

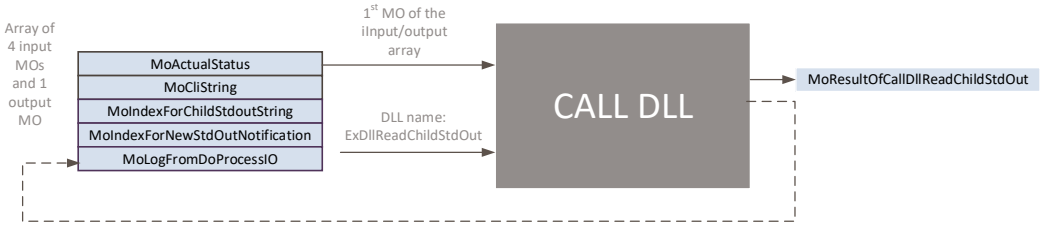

Figure 5 Parameterising custom DLL using configurable command CALL DLL

The *CALL DLL* (Figure 5) configurable command is called with following types of parameters:

[9] The OpcDbGateway runtime is able to find numbers of input and output data that are hard coded in DLL because it calls at the beginning the DLL C-interface function `GetCountOfIO`. It was necessary to do it this way because of constant setting of 2 input parameters and one output parameter of the *CALL DLL* command.

CALL DLL (MO for a result of the call, Name of DLL, MO of the first PV in input/output parameters array).

In our case it is parameterised following way:

CALL DLL (MoResultOfCallDllReadChildStdOut, ExDllReadChildStdOut, MoActualStatus).

The function *DoProcessIO* **checks the validity of data in parameters array** and put them in the instance of the class *CExample1App* for usage within the DLL global *IOThread* function. There is provided start/stop of the external console application and the loop for reading data from the unnamed pipe. All data and methods related to working with console application are implemented in the class *ChildStdoutReader*.

The DLL uses **standard logging functionality** of the OpcDbGateway core that is for the DLL provided by methods of the class *CLogger* and within the configuration by trigger of the type value named *Universal Log Trigger* that is activated by PV *System/UniversalLogTrigger* , by asynchronous event named *Universal Log Event* that activates function block *Write Universal Log Message.* In this FB, the configurable command WRITE_MESSAGE_TO_LOGFILE is called and then the PV *System/UniversalLogTrigger* is set to *FALSE* to signalise to DLL that writing of log message has been executed and logger is prepared for next log message. The principle for notifying and confirming log messages from DLL to runtime is the same as for messages from the unnamed pipe.

The test environment is completed with MQTT broker that can be placed locally or in the cloud, and by MQTT publish and subscription console applications.

Conclusions

The article describes generally how to create plug-ins for connecting an external process – console applications with OpcDbGateway runtime core application over interface consisting of command line, STDIN, STDOUT and STDERR.

It is also shown on example how two readymade external processes the MQTT publishing and the MQTT subscription client can provide outbound and inbound communication to/from OpcDbGateway runtime from/to MQTT broker. The custom DLL developed for this example can be used also more generally for other types of external processes.

In the actual implementation, it is not possible to start more threads for different console applications. Configuring application does not allow to configure the same DLL under different names but It is possible to rename original DLL and to configure it under a different name.

Principally, it would be possible to start more threads within one DLL. Parametrization for all of them would need to be done within one call of the *DoProcessIO*. The same function can be then used for individual starting and stopping of different external processes.

The test application **does not solve next processing of string received from standard output of the console** because it is supposed that the DLL for reading data from STDOUT **is implemented as semi-custom, it means developed in the same development environment** as OpcDbGateway runtime core and can be used generally for different processes connected through their STDOUTs and unnamed pipes. On another hand, the DLL for next processing of received string can be developed really as a custom DLL using different development environments.

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