Customized CANopen tools made easy

Instead of inventing the wheel again and again, standardized, CANopen compliant API may be used independently of programming environments and applications.

There are many kinds of CANopen equipped applications on the market. Most of them are test and commissioning tools for individual devices and entire systems. Traditional implementation approach has been either to implement separately only some CANopen services or link an entire CANopen stack into an application as a source code. The first approach has been typically frame-oriented, and thus lead into very constrained functionality and regular tool updates, according to the updates in the corresponding device. Latter approach typically has not to be updated so often, but the use of a CANopen stack as a source code with all possible extensions in the application-programming interface (API) may lead into unnecessary complexity. Both alternatives do not help in the development of next products, because of the constrained or complicated API usage. Moreover, application specific implementation leads into testing the same basic CANopen features from application to another.

However, sufficient CANopen API for any kind of purposes can be kept very simple. Only the standardized communication services may be considered. In commissioning tools service data objects (SDO) play a primary role, optionally in conjunction by the heartbeat consumer. Commissioning tools may additionally require process data object (PDO) support, because fine tuning of some runtime parameters often need to be adjusted. Furthermore, an emergency consumer is needed in order to provide comprehensive state management in conjunction with the other services.

This article presents an approach with CANopen implemented as highly re-usable library external to the application. Python is used as an example application programming language. A concept, how the external CANopen library concept works is presented after details behind selection of Python and before the details. The first detailed section presents main differences between IEC 61131-3 and Python. The second detailed section introduces object access mechanisms. Next detailed sections describe, how signals differ from parameters and how signals and parameters may be accessed consistently. The last section sets concluding remarks and proposes some further development.

Python

Python was chosen as an application development environment, because of its numerous advantages. It is available for free and there exists a large community providing support and knowledge pool in various discussion forums. Furthermore, it is a very productive high level language. Due to the large community, the large number of free libraries for various purposes, increase the development efficiency further. The operating system (OS) independence provides not only support for several operating systems, but also an easier upgrade path between operating system versions, which has been a major problem especially between various Microsoft Windows versions.

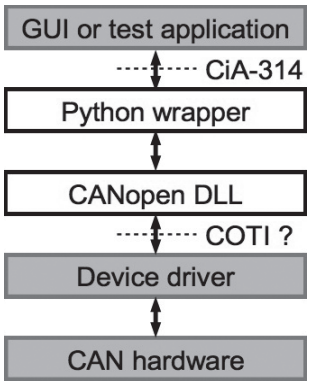

Figure 1: Software layers

Python also supports OS-independent GUI implementations. There are various graphical user interface (GUI) toolkits available. Tcl/Tk based Tkinter is built-in into the Python environment. There are also additional widgets available for Tkinter, which are supported by Tcl/Tk, but not by Tkinter by default. There are also QT-based toolkits for those who prefer QT as a GUI environment. When OS-independence is important, special attention shall be paid on finding out the potential issues with QT among the operating system.

CANopen library concept

The entire CANopen implementation has been encapsulated into an independent library with a standardized API. It was found, that API according to CiA 302-4 [5] and CiA 314 [6] works extremely well, independent of the programming language. The main advantage of the library implementation is, that it works like a CANopen interface of a programmable logic controller (PLC), which enables a direct use by PLC programming experts, without learning lots of new things. Newcomers need to learn only a single API, which applies for both supporting tools and PLCs. Another significant advantage of such library is, that due to a re-use of it with various applications, it will be continuously tested. It has been proved, that the use with various applications rapidly reveals the problems, due to the large number of use cases.

Everything with the external library works in a CANopen way – through the object dictionary. In addition, due to the standardized API [6], everything may be adjusted also from the application – through the object dictionary. In the case of various testing and commissioning tools, there do not exist fixed configurations and applications that shall scan the system and adjust available services accordingly.

```
(* IEC 61131-3 structured text ------------------------------ *)
VAR
  getStateX: CIA314_Get_State;          (* Create FB instance *)
END_VAR
  getStateX(ENABLE := gsEna, KERNEL := krnlNum,
    DEVICE := 16#0, TIMEOUT := 16#3E8); (* Execute the FB *)
  gsCon := getStateX.CONFIRM;           (* Use FB outputs *)
  gsErr := getStateX.ERROR;
  gsStat := getStateX.STATE;

# Python ------------------------------------------------------------
gsRtn = CIA314_GetState(krnlNum, 0x0, 0x3E8) # Call function
gsErr = gsRtn['ERROR']                       # Use return values
gsStat = gsRtn['STATE']
```

Figure 2: Example call of IEC 61131-3 FB (top) and corresponding Python function (bottom)

On the contrary, the utilization of fixed configurations apply for the GUI applications.

Signal usage complies with PLCs, where application contains global variables for storing the temporary values over a single application cycle. Output variables are written to the process image [5] at the end of each cycle and input variables read from it at the begin of each cycle. An additional Python wrapper module has been implemented only in order to keep applications as simple as possible by adapting C-style data types, used in the library, into Python data types in a dedicated module.

IEC 61131-3 vs. Python

Unlike IEC 61131-3, Python does not support function blocks (FB), that is, why functions shall be used instead. The main difference between FBs and functions is, that execution control signals Enable and Confirm are not needed in functions. All other arguments and return values have been kept intact. It has been considered, that the function based API could be used also in other programming languages with minimal changes.

Function arguments are passed in the simplest way, fixed number of arguments in a fixed order. Arguments for Python functions may alternatively be passed as named values, which would allow passing of unsupported arguments without leading into exception. This would also allow passing of subset of arguments without exception, which is not supported in IEC 61131-3. Named arguments are related with the dictionary data type, which is not supported by other common languages. Thus, fixed order and fixed number of arguments are used. When the arguments are provided as a tuple, they can be passed to the appropriate function as a single parameter. Figure 2 clarifies the minor differences.

IEC 61131-3 function blocks may return multiple values, which is not directly supported by functions of more traditional programming languages. Python supports the dictionary data type, which is excellent for the return values – multiple return values can be provided as dictionary members and are accessible by key names, which is almost equal to the IEC 61131-3 function blocks from access point of view. Same principles are used throughout the Python implementation of the API as specified in CiA 314. Data structures may be used accordingly with other languages.

```
01  sdoSt = CIA314_Sdo_Write( intrf, 0x0, 0xA040, 0x02, outByte, 0x1 )
02  tSdoPar = ( 0x0, 0xA4C0, 0x3, inByte, 0x1 )
03  sdoSt = CIA314_Sdo_Read( intrf, *tSdoPar )
```

Figure 3: Example SDO operations with separate arguments and arguments packed into a tuple

A further possibility in Python is, that multiple arguments may be packed into a tuple and be passed to a function as a single group of arguments. Such a mechanism enables the collecting of object access information into a single location, where it is easy to maintain. Majority of the application code remains simple and independent of the system configuration. Example of this exists in Figure 3.

Object access

One of the main tricks with the CANopen library is, that also objects in the local object dictionary are accessed with SDO functions. The primary result is, that such a mechanism complies with accessing any kind of functions from an external library, which conforms the way of working in most operating systems. Direct memory access, which is traditionally used as signal object accesses in PLCs, is not a preferred mechanism in major operating systems. Thus, additional interfaces are not required and a well-supported object dictionary (OD) based interface concept may be used.

A write operation of byte value of outByte into local network variable 0xA040:0x02 with individual arguments is shown in line 1 of Figure 3. A tuple defining read arguments of byte value of local input network variable 0xA4C0:0x03 is composed in line 2. Actual read operation is shown in line 3, after which the inByte variable contains the read value. Type of both outByte and inByte shall be defined according to the object type, c_ubyte in the example.

Especially in applications with GUI or log file, it is useful to use SDO abort code decoding function for converting numeric codes into text descriptions. Standardized abort codes are available in machine understandable format, already translated into several languages. It has been presented, how this is read and translated into a decoding function [3]. The optimal implementation varies, depending on the capabilities of the used programming language.

Signal and parameter objects

From an application point of view, most significant difference between signals and parameters is, that parameters are accessed only on-demand but signals shall be regularly updated between application variables and process image in the OD. Periodically called functions read input values from the library and write output values to the library. Such an approach results a minimum number of updates and maximum performance, when only values of the used objects are read and written. If the application accesses different set of signals in different modes, unused signals need not to be updated.

Parameter objects are typically managed in an application dependent manner, sometimes individually or more often in groups but almost never all at a time, especially in the larger systems. Parameter accesses shall be designed carefully so, that they do not have significant impact on the network schedule [1]. Thus, a general approach for exporting parameter read-and-write functions cannot be found.

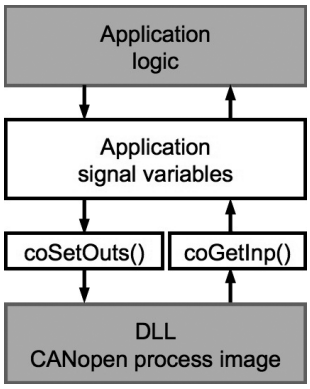

Figure 4: Process image synchronization with periodically called functions coSetOuts() and coGetInp()

Signal consistency

The typical heartbeat transmission interval is several hundreds of milliseconds and thus it may either be executed in an own thread or sequentially read as part of signal synchronization thread one node-ID in each cycle. One should remember, that reading NMT states shall not be dependent on the NMT-state, because problems in network structure are most important use cases for system monitoring. Heartbeat provides an raw operational state of each device. Such information gives indirect validity indication for incoming signals – if the source device is not in operational state or even does not exist, corresponding RPDOs cannot be valid in any case.

```
01  pstst = CIA314_Get_Rpdo_State(intrf, device, pdoNumber )
02  timeOut = pstst['STATE_PDOTIMEOUT']
03  tooLong = pstst['STATE_PDOTOOLONG']
04  tooShort = pstst['STATE_PDOTOOSHORT']
```

Figure 5: Example of getting RPDO status

Status of incoming signals is essential in control systems and importance increases among the increasing safety requirements [2]. RPDO status is a standardized basic service [4] and its results are as easy to be distributed to the mapped signals equally to the heartbeat status from producer devices to the corresponding signals [1]. A standard API does not exist yet, but a call shown in Figure 5 is under development and will be proposed into CiA-314.

Additional cross-reference is required for distributing the PDO status into meta-information of the mapped signal. Such information is already available in the RPDO mapping objects of standard DCF files, when a network project has been completed. Distribution may be located into a dedicated abstraction layer service function with heartbeat consumer information [1].

Parameter consistency

Parameter handling presented in the literature [1] applies also to the Python approach without any changes. A special ▷

Related articles

- Optional structures in CANopen projects
- CANopen device configuration editors
- SI-Unit and scaling management in CANopen
- Exception management in CANopen systems

References

[1] Saha H., Improving development efficiency and quality of distributed IEC 61131-3 applications with CANopen system design, Proceedings of the 13th iCC, CiA, 2012

[2] Hietikko M., Malm T., Saha H., Comparing performance level estimation of safety functions in three distributed structures, Journal of Reliability Engineering and System Safety, issue 134, Elsevier, 2015, pp. 218-229

[3] Saha H., CANopen in the frontline of open data, CAN-Newsletter 3/2014, CiA, 2014, pp. 42- 45

[4] CANopen application layer and communication profile, CiA-301, CiA

[5] Additional application layer functions, Part 4: Network variables and process image, CiA-302-4, CiA

[6] Accessing CANopen services in devices programmable in IEC 61131-3 languages, CiA-314, CiA

[7] COTI, Conformance Test Interface, Appendix to CANopen Device test, CiA, 2008

[8] LIN Specification Package, Revision 2.2A, LIN Consortium, 2010

case is, that there are two or more devices at the same node-ID in the network. Heartbeat reveals unambiguously only missing nodes, not necessarily case, when two or more devices share the same node-ID. Detection of duplicate devices is typical in various commissioning tools, but may also be required in control systems.

In the case of two nodes with the same node-ID, the SDO client receives the first server response normally, but raises an exception if the second SDO server reply is received without a client request. Such an error condition shall be read by function CIA314_Get_CANopen_Kernel_State(), because in such a case the SDO transaction seems to end normally and CIA314_Sdo_Read() or CIA314_Sdo_Write() does not return any error status.

Depending on the implementation, the status may also need to be called before the SDO function, in order to clear the potential error status caused by preceded operations. Meta information earlier presented in the literature [1] have been supplemented by the status attribute, where the SDO status will stored.

Conclusions

Independent CANopen library with Python and Tkinter provides very efficient GUI implementations, mostly thanks to the Tkinter layout manager taking care of many basic widget operations. The CANopen library provides similar easy-to-use approach by implementing system integration interface similar to a PLC with CiA-302-4 compliant process image and CiA-314 compliant CANopen service access functions.

CANopen API specified by CiA-302-4 and CiA-314 is tiny, but still sufficient for interfacing the CANopen system. The use of harmonized API among PLCs and GUIs causes less to learn by programmers. Concise API results in fewer opportunities to make mistakes and less code to be maintained and tested. During two years of development work with various applications, more complex API have not be required.

Cyclic signal update according to the GUI update rate helps in avoiding overloading GUI framework by network reception events, which is a common result with callbacks. The cyclically updated variables approach introduces only a load required by current application state.

While a standardized API exists for CANopen, re-usability of the approach could be improved further by developing a generic hardware abstraction layer (HAL) based on the CANopen conformance test interface (COTI) specification [7]. Such an approach has been used with Local Interconnect Network (LIN) from the very beginning [8]. ◀

Author

Heikki Saha
TK Engineering
www.tke.fi