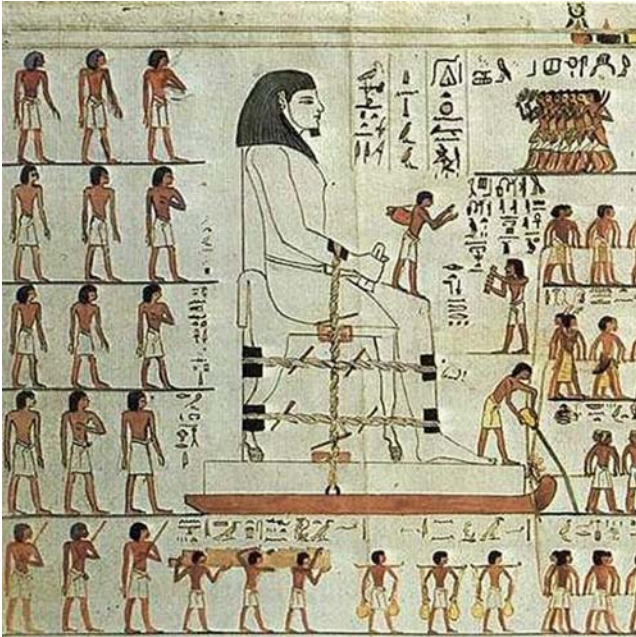




CODESYS

"GOTO OOP"

The logical path from functional to object-oriented programming

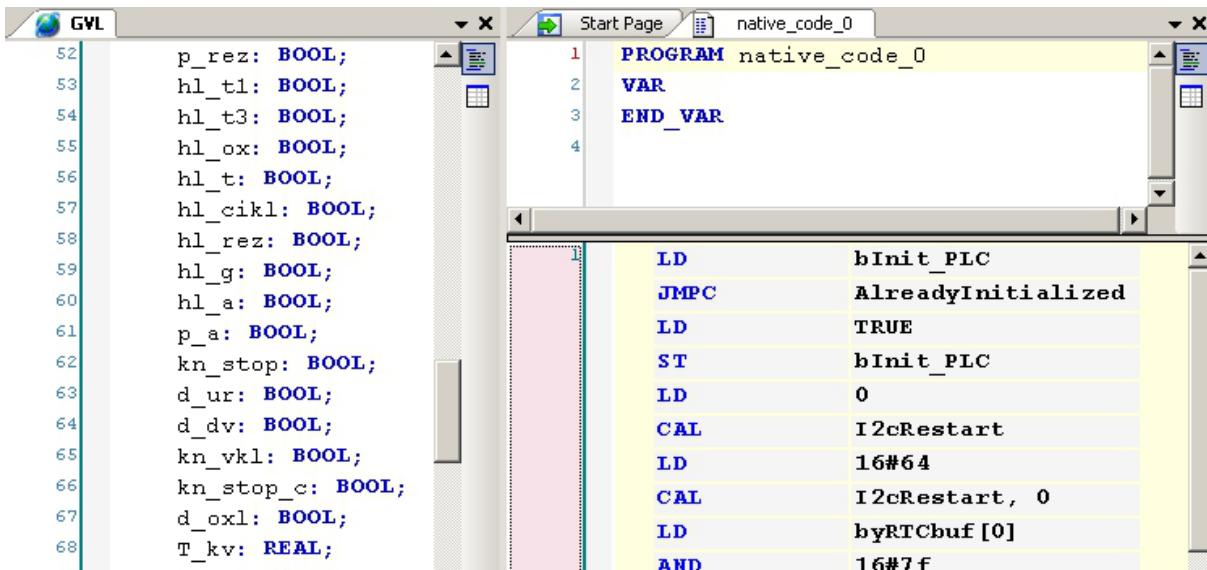


The description of processes with the aid of programs is nothing new. Even the ancient Egyptians' tomb inscriptions contained "program code", e.g. for the movement of a statue. Admittedly their programming language is neither standardized nor easy to understand... in comparison with that it is easy for programmers of industrial controllers to code their wishes - functional programming in the graphic or textual languages of IEC 61131-3 is cleanly standardized.

Nevertheless, a constant development of programming is taking place – mostly in logical steps that optimize the workflow in the programming of applications. This article explains to you, taking the example of a simple timer application, how the "natural" path leads from functional to object-oriented programming.

"The entry", or "how we (almost) all began"

Declaring all data in a large global variable list, accessing it briskly in the program code, perhaps even from a single program block – that is a programming style that leads pragmatically to the goal and is therefore still not completely "extinct" today. The fact that many PLC applications have been successfully realized in this way speaks for it. It becomes problematic, however, if errors have to be tracked down in such a program code or if the application is to be adapted. In addition to that it is laborious to re-use parts of the program code for other applications. That is when, at the latest, PLC programmers start thinking about how they can improve their application.



```

52  p_rez: BOOL;
53  hl_t1: BOOL;
54  hl_t3: BOOL;
55  hl_ox: BOOL;
56  hl_t:  BOOL;
57  hl_cikl: BOOL;
58  hl_rez: BOOL;
59  hl_g:  BOOL;
60  hl_a:  BOOL;
61  p_a:  BOOL;
62  kn_stop: BOOL;
63  d_ur:  BOOL;
64  d_dv:  BOOL;
65  kn_vkl: BOOL;
66  kn_stop_c: BOOL;
67  d_oxl: BOOL;
68  T_kv:  REAL;

1  PROGRAM native_code_0
2  VAR
3  END_VAR
4
1  LD      bInit_PLC
2  JMP    AlreadyInitialized
3  LD      TRUE
4  ST     bInit_PLC
5  LD      0
6  CAL    I2cRestart
7  LD      16#64
8  CAL    I2cRestart, 0
9  LD     byRTChuf [0]
10 AND   16#7f

```

Fig. 1: "Quick and dirty": unstructured programming with a global variable list, which the program code accesses "briskly"

1st improvement step: structuring of the data and its use

Powerful aids for structuring are defined in IEC 61131-3: e.g. functions (FUN), which the programmer can call at will from main routines for delimited functions with a return value. Or structures (STRUCT) for summarizing variables. These means implement the initial requirements for structured programming as formulated by the Dutch computer scientist **Edsger Dijkstra** (1930-2002) ("*Go To Statement Considered Harmful*", 1968, and "*Structured Programming*", 1972).

In the example of our task, various timers are declared as an instance of a structure. Beyond that processing takes place with the aid of special functions, e.g. MyClockGetTime, MyClockSetTime, MyClockStartTime, MyClockStopTime. The use of different timers is thus already considerably simplified, because each function call gets its own data handed over or returns it. However, the data transfer must take place with the exact data type. It is also not possible to use alternative code during the execution.

2nd improvement step: summarization of data and code in function blocks (FB)

In the first step, data and code were structured still without a recognizable relationship to one another. With function blocks according to IEC 61131-3 the application programmer can create his data structure together with its execution. The actual data allocation takes place with the instancing of an FB. With each instance an independent data record is thus created which is processed by the identical program code. By the declaration of input variables (VAR_INPUT) and output variables (VAR_OUTPUT) it is possible to access the data within the instance. Instances can be created individually or as arrays and thus structured further.



Disadvantage of the FB: in order to be able to carry out the processing flexibly, numerous VAR_INPUTs are sometimes necessary that in turn have to correspond to the exact type of the VAR_INPUT variables. If alternative program sequences are to be run through within the FB, then additional program code is required for the branching that does not belong to the actual function.

3rd improvement step: FBs with actions

Actions of FBs eliminate a part of these disadvantages. Actually, they were intended by the fathers of IEC 61131-3 for objects that are created in sequential function chart. However, their extension to all FBs has proven itself over many years in the market-leading IEC 61131-3 programming system CODESYS: in addition to the standard properties of FBs described above, the application can directly call these actions from the FB or higher-level blocks. This means that the action code is executed on the data of the FB instance, but independent of other actions or the code in the body of the FB. With the available actions it is thus possible to define different execution variants of an object. Hence, a decision is made with the call regarding which action and thus which variant is to be executed. Branches for different tasks are no longer necessary in the action.

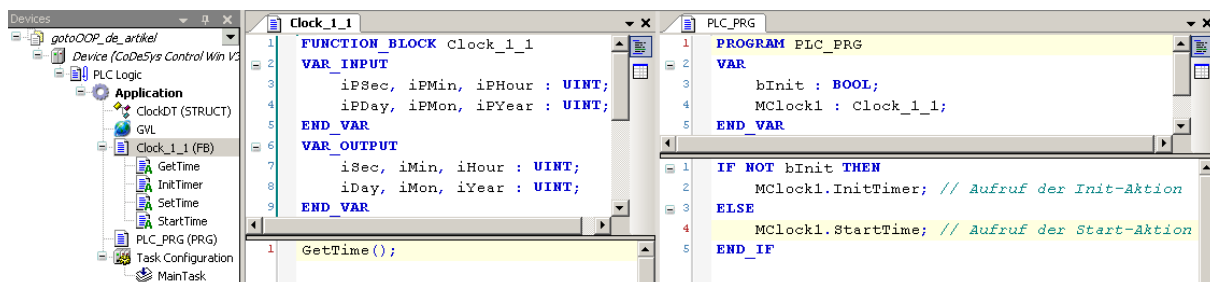


Fig. 2: Actions, as children of an FB, can be called from the body of the FB as well as from the outside and can work on the data of the FB

However, actions do not enable input data (VAR_INPUT). In addition, it is not possible to use identical code within actions for different FBs.

4th improvement step: FBs with methods

Methods are blocks that were not originally defined in this form in IEC 61131-3. Like actions, methods are nothing other than functions that are assigned to an FB. However, they can now have their own parameters in order to make their execution even more flexible and more independent from one another.

5th improvement step: FB-implemented interface(s)

Even methods are new objects in the IEC 61131-3, whose use is quite simple to comprehend. With the next new object, "interface", we are now definitely entering object-oriented programming territory. As opposed to programs, function blocks, functions, actions and methods, interfaces do not contain any code. So what do we need them for? As the name



already suggests, it concerns the description of an interface for calling methods. This description is uniformly defined in one place. This means that an interface defines a set of methods and their call parameters. The methods in the interface itself are empty – no code is to be defined, only the interface.

If an FB now implements one or more interfaces, then all methods defined for this FB in the interface must be programmed-out. There are several advantages to this:

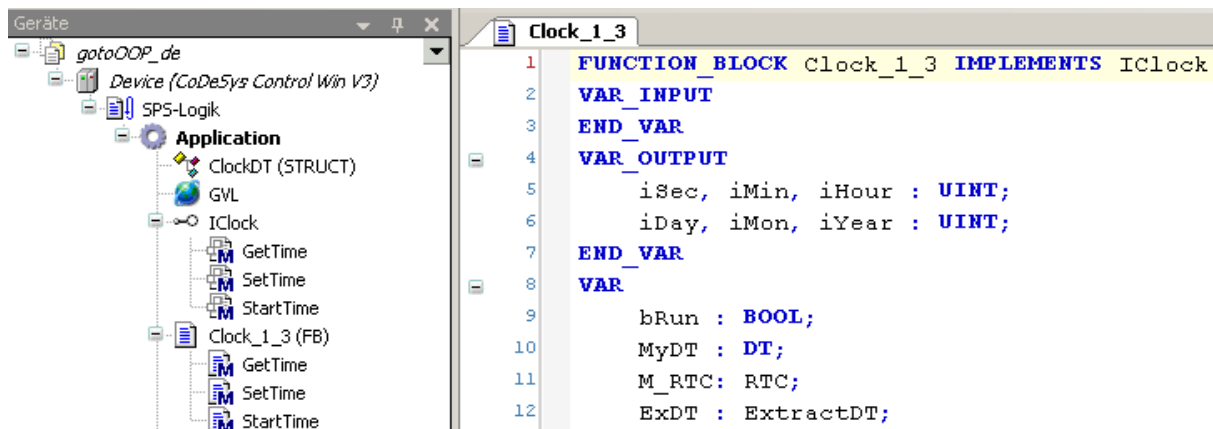


Fig. 3: The interface IClock is implemented by the FB; the methods of the interface are programmed-out in the FB

On the one hand one can uniformly define the call interface for an FB in one place by defining the interface. This is particularly interesting if there are several different units in an application that are called in an identical way. There are plenty of examples of this in automation:

- All drives (independent of manufacturer, bus connection, power class) support functions such as 'Homing', 'Reset' or 'Drive to target position'.
- Each fieldbus has functions for sending and receiving data.
- Most mechatronic units must be enabled by means of software and return a diagnostic signal.

With the central definition of an interface the programmer can ensure that no function is forgotten for the FB. If the interface should have to be changed or extended, this is done in a central place. Also, the compiler monitors whether the implementation, for example of an added method, was actually done.

An even greater advantage: through the implementation of interfaces in different units, the implementations of the methods can naturally take place entirely individually. Nevertheless, the call of the methods remains identical. "Hey, wait a minute!" the expert reader may well exclaim at this point, "I have to call the methods as part of an FB instance! Although the method call does not change, it contains the instance name!" This objection is naturally justified. Unlike the normal FB, however, access can now take place via the implemented interface, for example in arrays. This means that in the actual call of the methods it no longer matters from which instance and with which concrete implementation it is carried out. This allocation takes place with the filling of the array as a field of the "data type" interface.



CODESYS

And that is naturally an extreme simplification if a larger number of methods of similar type, but different concrete implementation is to be called.

Further steps

OOP is by no means exhausted with interfaces and methods, however. Why shouldn't one re-use methods in other FBs? What about the properties of FBs? Which further possibilities can be derived from this concept? To be continued...

Conclusion

The logical path taken in the optimization of the programming style leads to OOP – and it can be taken even further without having to leave the familiar IEC 61131-3 interface! After all, the properties described will already all be included in the third revision of the standard.