Object-oriented extensions for IEC 61131-3

Article *in* IEEE Industrial Electronics Magazine · January 2010

DOI: 10.1109/MIE.2009.934795 · Source: IEEE Xplore

1 author:

Bernhard Werner
3S-Smart Software Solutions GmbH

IEC 61131-3 and Object Orientation

1. Introduction

Industrial automation is faced with several challenges today: The complexity of applications is ever increasing as is the number of variants available. Development cycle times are decreasing and more and more tasks are assigned to the controller software which thus plays a more than crucial role.

Object oriented programming (OOP) in languages like C++ or Java is common place in desktop application development and an integral part of university education today. It has proved to be absolutely unbeatable when it comes to elegantly handling complex software development tasks and producing flexible, reusable software components. OOP has clearly reduced the development time of new software and simplified the solution of complex software tasks.
Industrial controllers (PLCs), however, are still mostly programmed in the languages of the IEC 61131-3 standard. In order to meet the challenges of modern industrial automation it is therefore only logical to add object oriented programming to the upcoming third edition of the standard.

At the moment, an international IEC 61131-3 maintenance group is discussing a working draft of this next edition. This draft contains a proposal for object oriented extensions to be added to the IEC 61131-3. The following article refers to this proposal. It is, however, fact that the standardization as concerns OOP is still at an early stage and it is well possible that this article may differ from the final edition.

2. New Language Elements

The IEC 61131-3 already contains a simple class concept, the function block. A function block has an internal state, a routine manipulating this state and may be instantiated several times. All IEC 61131-3- programmers are familiar with these concepts. Thus, the extension of the existing function block by object oriented features is a natural way of introducing object orientation to the IEC 61131-3.

The features proposed in the current working draft are:

1. Methods
2. Inheritance
3. Interface abstraction

With these new language elements the working draft covers all common elements of object oriented programming languages such as classes, objects, methods, inheritance and polymorphism.

2.1 Methods

A traditional function block contains only one routine for manipulating the function block's internal state, although there may be very different tasks to perform on this

state, like e.g. initialisation, error handling or cyclic execution. The only way to control this routine is by assigning specific values to the inputs of the function block when calling this routine.
An object oriented approach would separate the code for the different tasks to separate methods of the function block.
The following example shows a direct comparison of the two concepts:

1. Start the pump
   Pump1(start := TRUE, Direction := Forward, Reset := FALSE); (*classical approach*)
   Pump1.Start(Direction := Forward); (*object-oriented approach*)

2. Reset the pump
   Pump1(start := FALSE, Reset := TRUE); (*classical approach*)
   Pump1.Reset(); (*object-oriented approach*)

There are several advantages of the object oriented approach here:
- Structuring: Whereas the classical implementation must contain all functionality in one body, the object oriented approach allows separation of the code for different tasks.
- Readability: The name of the method clearly shows what the pump is meant to do, whereas in the classic approach the assignment of inputs determines the usage.

Language feature: Methods
Methods, as defined in the current working draft, may be seen as a function declared inside a function block with result type, parameters and local variables. As is the case within functions, these local variables will not keep their state from one call of the method to the next. Methods have an implicit access to the variables of the function block. In addition to traditional object oriented methods, IEC 61131-3-methods may contribute to the instances internal state. This allows the implementation of EDGE-detection inputs or state machines (SFC) within a method.
Thus, all languages of the standard can be used for the implementation of methods (although in the following code samples, the implementation language is always ST).

Language feature: Access specifiers
For fine grained access on methods, the proposal defines the following access specifiers:

- PUBLIC: access without any restriction
- PRIVATE: access is restricted to the function block
- PROTECTED: access is restricted to the function block and its derivations
- INTERNAL: special access in combination with namespaces (also a proposal for the third edition, that cannot be discussed in this article)

2.2 Inheritance

In classical IEC 61131-3-programming variants of a function block are constructed by copying the function block and changing its implementation. There is no way of reusing common parts of the code. Obviously, this approach has a lot of disadvantages for code maintenance.

In object oriented programming, a new function block may be constructed by inheritance. This means, the new function block inherits all variables and methods of the old function block. It may define additional variables and additional methods and it may *override* methods of its parent.

Language feature: Extends
EXTENDS in a function block makes it the subclass of another function block.

2.3 Interfaces

Language feature: Interface
For the declaration of abstract super classes, the *interface construct* is introduced. An interface is similar to a function block, with subordinate method prototypes. The difference is: the interface does not have variables nor an implementation part and its method prototypes in turn do not have local variables nor an implementation. If a function block is derived from an interface, it must contain concrete methods (with implementation) for all method-prototypes defined by the interface.

The introduction of *interfaces* avoids the problems that arise with *multiple inheritance*. A function block may be derived from one other function block (and not more) but from an arbitrary number of interfaces.

Language feature: Implements
The keyword IMPLEMENTS in a function block makes it the subclass of one or more interfaces. An IMPLEMENTS declaration requires the function block to have at least all the methods which the named interfaces have (with the same parameter and result types). The new thing is that in the function block the methods must now also have an implementation part!

Language feature: Reference semantics for interface types
Like a function block name, the name of an interface can be used as a type name in the declaration of variables. These declarations have *reference semantics*. This means, this declaration does not provide a new object but only a new reference to a *function block*-instance declared elsewhere (initially NULL). This instance must be of a function block type implementing the interface. The assignment of a function block instance to an interface variable makes that variable refer to the function block instance. Thus, the same interface variable can refer to different instances of different function block types (Polymorphism).
Note that a function block instance may be assigned to any interface variable of interface types it is derived from.

3. Example

The following example illustrates one possible usage of the new language elements. It contains a mixture of the IEC 61131-3-languages Structured Text (similar to PASCAL) and Function Block Diagram (a graphical language).

The application is a simplified building application which could for example be used to control the lighting system of rooms in an office building. The application mainly consist of rooms with so called daytime and nighttime modes and a room control which controls these modes. The room control is to handle *any* function block with a daytime/nighttime mode, therefore we define an abstract interface for these kind of function blocks.

```
INTERFACE ROOM
METHOD DAYTIME : VOID
END_METHOD
METHOD NIGHTTIME : VOID
END_METHOD
END_INTERFACE
```

Figure 1: Abstract interface for rooms

The room control function block can now handle a variable of type *ROOM*:

```
FUNCTION_BLOCK ROOM_CTRL
VAR_INPUT
    RM : ROOM;
END_VAR
VAR_EXTERNAL
    DAYTIME : TOD;  // global time definition
END_VAR
IF (RM = NULL) THEN       // always test on valid reference!
   RETURN;
END_IF
IF DAYTIME >= TOD#20:15 OR DAYTIME <= TOD#6:00 THEN
   RM.NIGHTTIME();
ELSE
   RM.DAYTIME();
END_IF
END_FUNCTION_BLOCK
```

Figure 2: Implementation of room control

This function block is a common function block with one body and no methods. Note that the room control function block can be called with *any* function block implementing the ROOM-interface.

Now we need function blocks which implement the interface. The following code fragment contains two such function blocks,, the second being inherited from the first one:

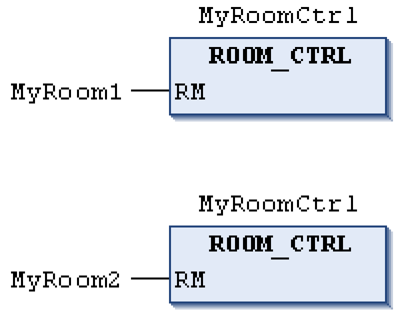
```
FUNCTION_BLOCK LIGHTROOM IMPLEMENTS ROOM
VAR
     LIGHT : BOOL;
END_VAR
PUBLIC METHOD DAYTIME : VOID
     LIGHT := FALSE;
END_METHOD
PUBLIC METHOD NIGHTTIME : VOID
     LIGHT := TRUE;
END_METHOD
END_FUNCTION_BLOCK

FUNCTION_BLOCK LIGHT2ROOM EXTENDS LIGHTROOM
VAR
     LIGHT2 : BOOL;
END_VAR
PUBLIC METHOD DAYTIME : VOID
     SUPER.DAYTIME();          // call of parent implementation
     LIGHT2 := FALSE;
END_METHOD
PUBLIC METHOD NIGHTTIME : VOID
     SUPER.NIGHTTIME();        // call of parent implementation
     LIGHT2 := TRUE;
END_METHOD
END_FUNCTION_BLOCK
```

Figure 3: Example of interface implementation

Now we have all parts of our tiny building application and we can bring them together in a small test program:

```
PROGRAM TEST
VAR
     MyRoom1 : LIGHTROOM;
     MyRoom2 : LIGHT2ROOM;
     MyRoomCtrl : ROOM_CTRL;
END_VAR
 room control with two different types of rooms!
```




```
END_PROGRAM
```

Figure 4: simple building application

The crucial point in this example is the usage of *polymorphism*. One variable of an interface type can be assigned variables of different types at runtime. This offers enormous new opportunities for the design of automation applications. Would this program have been written in a conventional way without using OO it would either have been necessary to define a control function block for every room or to include the control code into every different type of room. In both cases we would have to duplicate code.

4. Conclusion

Even the small example shown above clearly demonstrates the strong advantages of object oriented programming:

- better structured program code with “separation of concerns” and “information hiding”
- flexible extensibility by new types of objects (e.g. software representations of new types of drives)
- reuse of code for defining specialized subclasses (inheritance)
- reuse of code operating on different implementations of an interface (polymorphism)