Structuring Program Development with IEC 61131-3

General trends

The role of software has changed. It plays an ever increasing factor in the quality of the product. Software errors can have dramatic effects, even ruining the overall investment of money and time sense. The requirements of industrial control have grown, extending the software code by factors from 100 lines of code to 10,000 now. This is not only prone to errors but makes 100% testing impossible. Creating this software is not a one-man-job anymore: the conventional programmer is now part of a multidisciplinary team.

With the ever increasing requirements, installation, maintenance, upgrades and improvements have become an essential part of the life cycle of the controls, and software plays the crucial role in it.

Introduction

Modern programming methods provide tools to improve the intrinsic quality of software, i.e. its correctness in the sense of reliability, robustness, integrity, persistence, and safety. The international standard IEC 61131-3 provides such a tool, dealing with the programming, installation and maintenance phases of software development projects in industrial control.

Basically, IEC 61131-3 consists of two parts, i.e. Common Elements and Programming Languages.

The structuring tools within IEC 61131-3 are focused on the common elements, although clearly links to the programming languages are needed.

This article shows that by using IEC 61131-3 in a consistent way, one generates software code that is understandable, reusable, verifiable and maintainable.

The essence of structuring

The above mentioned trends require a different approach. An approach through structuring provides advantages like:

- a better overview of the system, not only important for the original programmers, but also for the installation and maintenance personnel
- a better basis for internal communication within the multidisciplinary development team
- a better focus on the real problem and its possible solutions
- a basis for reusable software
- inherent / automatic documentation

Overall, structuring is done via dividing the problem into smaller parts, which again can be sub-divided. There are limits to this: it is not practical to continue to an endless fine granularity, since the effort is then moved towards the integration of these parts.

Within IEC 61131-3 there are two co-operating ways, which for clarity sake we name:

- Modulariy
- Decomposition

Modularity principles

Within the modern software development methods there are five principles associated with modularity. These are:

1. The programming language should support the modular units
2. The units should be composed in such a way / number that they have few interfaces and few interactions
3. The interfaces should be small, needing little data exchange
4. The module interactions require explicit definition, to increase their re-usability
5. The modules should provide data encapsulation: the application data is partitioned, and each partition should only be accessible by a proper set of functions which hide it from undesired uses.

To support this, IEC 61131-3 has defined Program Organization Units, POU's, consisting of:

- Functions
- Function Blocks
- Programs

These will be explained in more detail below.

Functions

We all know functions like add, square root, sin, cos, Greater Than, etc. IEC has an enormous set of these defined. In addition, you also can create your own functions like this simple_function:

FUNCTION SIMPLE_FUN : REAL

VAR_INPUT

A, B : REAL;

C : REAL := 1.0;

END_VAR

SIMPLE_FUN := A*B/C;

END FUNCTION

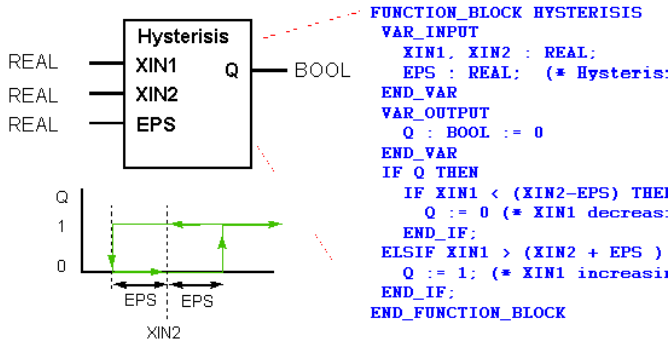
Once defined, you can use it over and over again, within the same program, other programs or even other projects.

Function Blocks, FBs

The same is valid for Function Blocks: there are standard defined Functions Blocks and FB's added by the supplier. You can also create your own Function Blocks and add them to your own Function Block Library. All these Function Blocks are highly re-usable within the same program, new programs or even projects. You can use them with any of the IEC programming languages, giving you a clear separation between different levels of programmers, or maintenance people.

This re-usability increases your efficiency, and reduces the number of errors.

Let's look at an example:



The Function Block above (on the left side) is represented here in the programming language Function Block Diagram. The Function Block has the name Hysterisis. It has three inputs on the left, named XIN1, XIN2 and EPS, all of datatype REAL. It has one output, on the right, called Q, of type BOOL.

Internally, the FB contains body code, as shown on the right side. In this example, the body code is written in the Structured Text, ST, language. The first part deals with the data structure, the second part with the algorithm, which uses the inputs, does some calculation, and sets the outputs. The algorithm is hidden to the user of the Function Block, who sees only the functionality of the block as shown on the left. This creates a different level of access, showing the encapsulation as referred to in point 5 of the modularity principles.

Names used within a function block are local to that FB. No matter which name is used in a FB for local data, there will be no conflict if the same name is used in a different manner in another function block or elsewhere in the program.

Programs: hierarchical network

With these Functions and Function Blocks, you can look at a program as a network of these basic building blocks. In this way complex programs can be broken down into function blocks, which can again be broken down into smaller function blocks. This helps you increase your efficiency.

Decomposition: how does it look in IEC 61131-3?

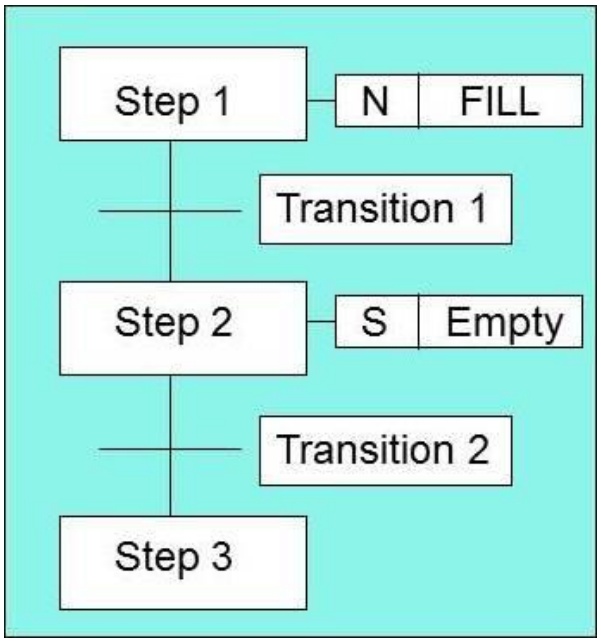
As decomposition tool, IEC 61131-3 provides Sequential Function Charts (SFC). SFC describes graphically the sequential behavior of a control program. In this way it structures the internal organization of a program by decomposing the control problem into manageable parts, while maintaining the overview. This makes it very suitable for diagnostic purposes.

SFC consists of Steps, linked with Action blocks and Transitions (see picture below).

Each step represents a particular state of the system. Steps are linked to Actions, performing a certain control function.

A transition is coupled to a condition, which, when true, causes the previous step to be de-activated and the next step to be activated.

Each action block or transition can be programmed in any of the IEC languages, Ladder Diagram, Function Block Diagram, Instruction List and Structured Text, and even including SFC itself for further decomposition.



SFC supports alternative sequences and parallel sequences, such as commonly required in batch applications. For instance, one sequence is used for the primary process, and a second for monitoring the overall operating constraints. And this occurs within the same overview and structure.

Structuring: 7 steps to success

Structuring within IEC 61131-3 is presented here as the combination of Modularity and Decomposition. The following 7 steps provide a road to success for the structuring of software:

1. Identification of the external interfaces to the control system
2. Definition of the main signals exchanged between the control system and the rest of the plant
3. Definition of all operator interactions, overrides and supervisory data
4. Analysis of the control problem broken down from the top level into the logical partitions
5. Definition of the required POU's, i.e. Program & Function Blocks
6. Definition of scan cycle time requirements for the different parts of the application
7. Configuration of the system by defining resources, linking programs with physical inputs and outputs and assigning programs and function blocks to tasks

IEC 61131-3 helps you especially in the last steps 4 - 7, where the translation into software occurs.

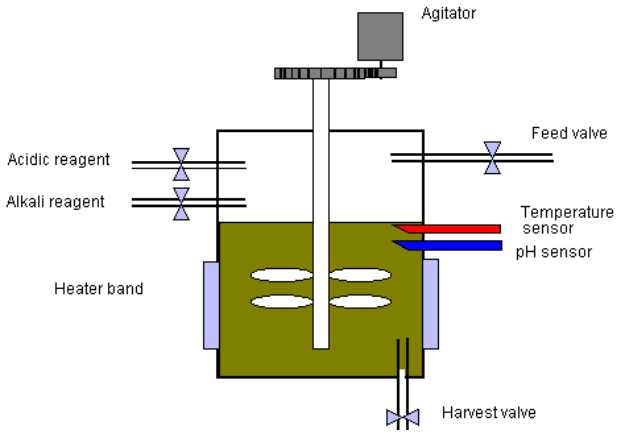
Besides these 7 steps, there are some overall principles which should be used to optimize the structuring method. These principles are:

- Work purely symbolic: no absolute addressing (this only in declaration part). Advantages: easy adaptation to changing environment, higher level of re-usability of code, fewer side effects
- Program parts belonging to each other should be joined in the source code also
- Do not use jumps. Advantages: higher transparency, higher level of re-usability, fewer side effects
- Consistent naming of variables and function blocks increases transparency and overall readability

An Example: Fermentation control system (courtesy of Omron Electronics)

An example tells more than 1000 words…, so let us look at a fermentation process and its control, as shown below.

Fermentation process



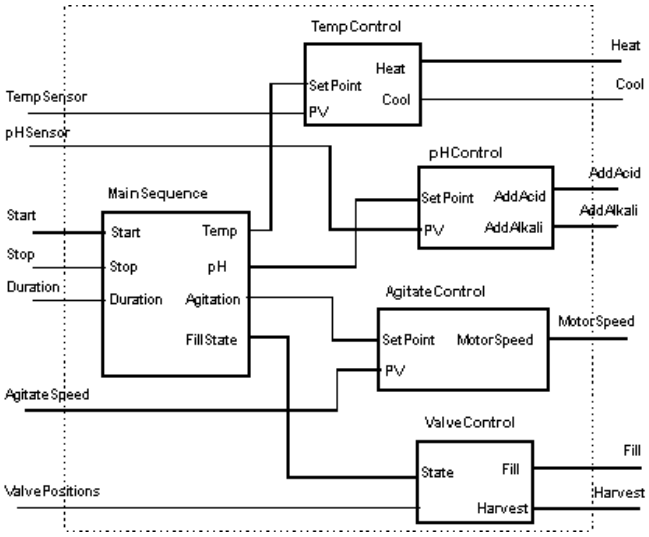
All external interfaces are defined here (step 1). There is a large vessel, which can be filled (Feed Valve) with the liquid, can be heated with the heater band (cooling via convection), can be stirred via the motor, and where acid and alkali fluid can be added into the vessel.

Looking at step 4, the analysis of the control problem broken down from the top level into the logical partitions, one can easily identify 5 functions:

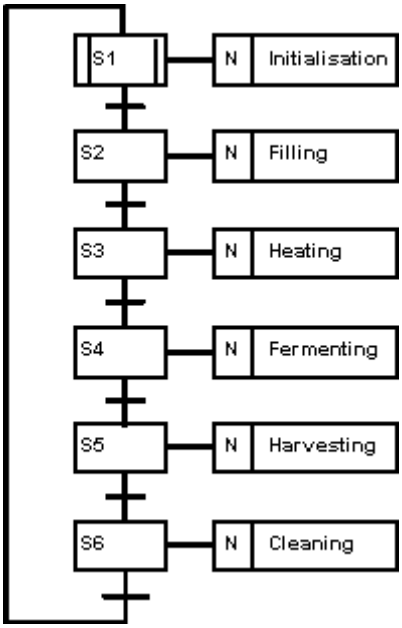
1. **Main Sequence,** d e.g., top level process steps - filling, heating, agitating, fermenting, harvesting, cleaning.
2. **Valve control,** e.g., operating valves used to fill and empty the vessel
3. **Temperature control** for monitoring the temperature of the vessel modulating the heater.
4. **Agitator control** for the agitator motor activated as demanded by the main process sequence.
5. **pH Control** for monitoring the acidity of the fermentation contents, adding acidic or alkali reagents as required.

Step 5: Definition of the required POU's,.e., Program & Function Blocks.

Presenting these in the Function Block Diagram programming language, the overview of the fermentation control program could look like this: (Read from left to right side. On the left are the inputs; on the right the outputs).



If we look closely at the Main Sequence, we could structure it with Sequential Function Charts, SFC, as follows:



We start at the top with the Initialization: since we do not know the status of the system when we first switch it on, we must check the position of the valves, etc.
Then we start filling till the right level has been reached.

Next phase is the heating till the fermentation process starts. When it does, we move to the next phase: the actual fermentation process control part.

After completion, we harvest, and after that clean, and we are ready to restart at the top.

This decomposition gives everybody involved a clear overview which sequences are involved, and further modularization into the function blocks which can be programmed in any of the four languages.

Or, stated differently: our user requirement specification is (nearly) done!

The programming work now to be done is at the level of the action blocks. Those could be divided between different people, with different backgrounds. For this, IEC defined 2 graphical and 2 textual programming languages, i.e., Instruction List, Structured Text, Ladder Diagram and Function Block Diagram, to best suit the needs and the problem at hand. Also, further decomposition of the action blocks can be done via SFC, if needed.

The development system will support you in the two final steps:

6. Definition of scan cycle time requirements for the different parts of the application.
7. Configuration of the system by defining resources, linking programs with physical inputs and outputs and assigning programs and function blocks to tasks.
   Conclusion

The software development process has changed

- more requirements
- more functionality
- more code
- more people involved
- … more requirements wishes

Structuring, Modularity and Decomposition are essential elements in modern software development and IEC 61131-3 offers the right basis to fulfill your requirements.

Please let us know if this article is useful for you!

info@PLCopen.org

www.PLCopen.org