CHAPTER 11

Object-Oriented Industrial Programming

This chapter describes two different interpretations of the term "Object-Oriented Programming", how one interpretation is a subset of the other, and how the subset provides many of the benefits of object-based programming without the complexity. This chapter goes on to provide examples of plant objects, how those objects are used in a control Program, and how this programming technique is a natural and intuitive way to control plant and equipment objects.

Also, this chapter provides some history on the evolution from traditional PLC programming to object-based programming, introduces new techniques for mapping I/O and configuring objects in an object-based environment, describes best-practices for PLC design, and shows how designs can be perfected before being deployed to the field using system-level simulation.

The lab for this chapter is exploring and simulating several completed discrete, batch, and continuous OOIP designs; and building your own OOIP design.

OOP vs OOIP

- Object Oriented Programming (OOP)
  - Utilizes Classes, Inheritance, Overloading, Polymorphism, Interfaces, Dynamic Binding, Events, Dependency Injection, etc.
  - Primarily text-based
  - Typically requires extensive training
  - Often considered too complex to be used and maintained by plant technicians
  - Great for code that is hidden from the end-user
    - Such as compiled libraries and black-box software/firmware
- Object Oriented Industrial Programming (OOIP)
  - Utilizes Encapsulation, Composition, and Abstraction
  - Mixture of graphics and text
  - Can be mastered with minimal training and/or OJT
  - Embraced by controls engineers and technicians
  - Great for white-box code that is accessible to the end-user

The Power of OOP with the simplicity of traditional PLC programming

When asked the meaning of Object-Oriented Programming, the response tends to be very different between members of the Industrial Controls community (OT) versus members of the Computer Science community (IT). To differentiate the two in this book, we will refer to the industrial controls interpretation as Object-Oriented Industrial Programming (OOIP) and describe them each as follows:

Object-Oriented Programming uses the full suite of OOP techniques shown in the graphic, is primarily text based, requires extensive training to generate reliable applications, and is great for black-box code that is only ever viewed or maintained by its developers or other highly trained individuals.

Object-Oriented Industrial Programming utilizes Encapsulation, Composition, and Abstraction, uses both text and graphical languages, can be mastered by controls engineers and plant technicians with minimal training, and is great for white-box code which must be understood and maintained by individuals with a wide range of skill levels.

Industrial controls software engineering has unique requirements for high reliability and for ease of use by a broad spectrum of users. Those are the reasons why graphical languages have been the mainstay of industrial controls programming and industrial controls engineers tend to wait for the latest trends in computer science to mature before adoption (such as symbolic addressing and data structures which both matured for 20 years before entering the industrial controls mainstream).

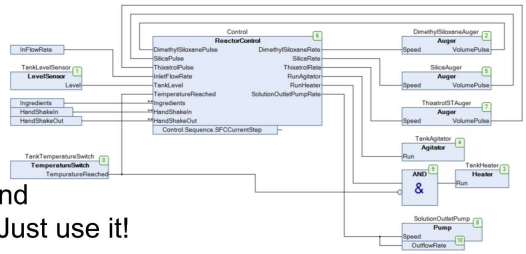
Object-Oriented Programming (OOP) began to be widely used by computer scientists in the 1990s but has been slow to be adopted into the Industrial Controls world due to its complexity and the lack of a supporting graphical language environment. OOIP provides a great way to get started with OOP by providing most of the productivity of OOP while maintaining the simplicity and reliability of traditional PLC programming.

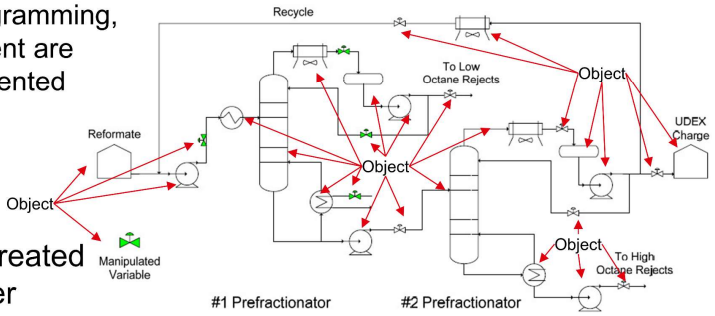
And with CODESYS, you are not locked into either solution as CODESYS fully supports both the OOIP subset as well as complete OOP. We began discussing some of these OOIP subset in the POUs chapter and will complete the remainder in this chapter. The full OOP suite will be discussed in the Advanced Topics chapter.

Visualization, Alarming, I/O, OPC UA, and Persistence also have their versions of OOIP which are discussed in their respective chapters later in this book.

Why OOIP?

- What is the advantage of OOIP?
  - Encourages and simplifies code reuse
  - Encapsulates the functionality inside the object
    - There is no need to know or understand the implementation inside an object. Just use it!
  - The resulting program is much easier to understand, modify, and maintain
  - OOIP is a perfect approach for industrial controls programming, as plants and equipment are themselves Object Oriented
- Objects are typically created by the Library Engineer





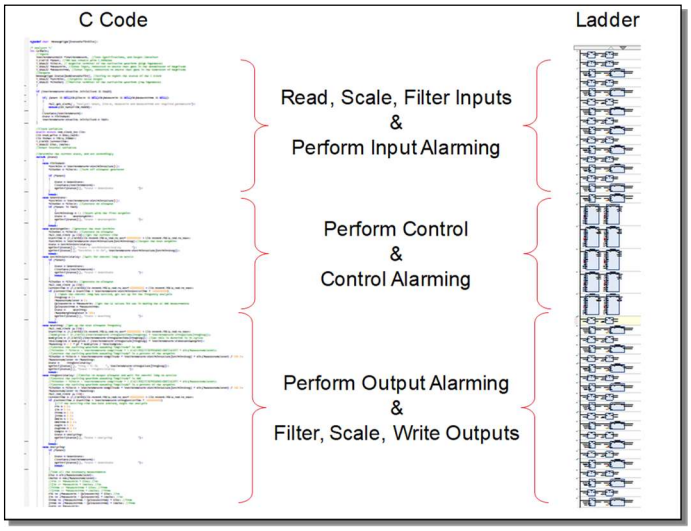
This Photo by Unknown Author is licensed under CC BY-ND

Object-Oriented Industrial Programming (OOIP) simplifies the design of your plant or equipment and makes it much easier and more likely to reuse control objects on future designs. Your plant or equipment is made of objects (motors, actuators, sensors, etc.); the control for your plant or equipment should be too!

Just as a motor is a completely self-contained object which does not require assembly or modification, the control for that motor should be a self-contained object which does not require assembly or modification. In software terms, this is known as Encapsulation. Everything that is required to control a motor is encapsulated inside the motor control block. Just drop it into the design and it works. Just as the plant or equipment designer doesn't need to be a motor design expert to specify and install a motor, the control designer doesn't need to be a motor control expert to configure and use the motor control block.

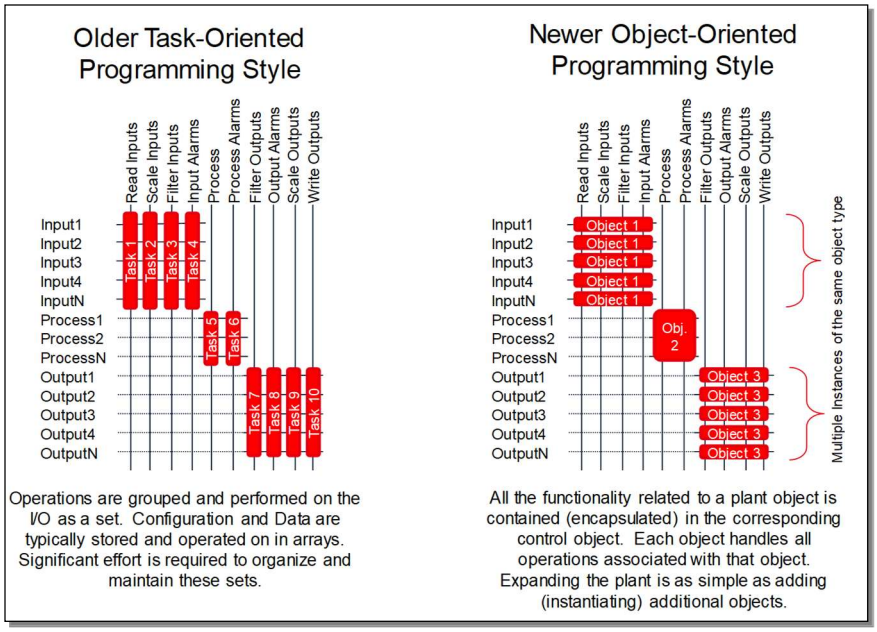
Reusing control blocks in your control code should be just as easy as reusing physical equipment in your plant. Physical equipment objects are specified, purchased, and installed – Control objects are placed, wired, and configured. The thought process is the same, and both can, in fact, be done in parallel. In the future, equipment manufacturers may well supply the OOIP control blocks for their equipment along with the physical equipment.

Evolution from Flat Programming



In the early days of industrial automation, programming was flat. We read the inputs, scaled the inputs, generated alarming on the inputs, performed the control algorithms to generate outputs, performed alarming on the outputs, scaled the outputs, and wrote the outputs using memory mapped I/O. Later when Functions became available, we consolidated some of the duplicate code, but the process was still essentially flat.

Task-Oriented vs Object-Oriented



When control software began to accommodate multiple tasks, industrial programmers adopted a centralized task-oriented approach. This approach divided the operations up into separate tasks and then a sequence of centralized processes performed each separate operation on the tags in the program. The first task would read all the inputs, the next task would scale all the inputs, the next would perform alarming on the scaled points, and so on.

A central characteristic of the task-based approach is a number of lists and processes which must be maintained. For instance, in a task-based approach, the global variable list is added first, then functionality is bolted on in multiple layers growing out from that (such as scaling, alarming, filtering, logging, retaining, Visualization, etc., etc.). Any modification or addition to the basic functionality requires modification or updating to all these layers. Great effort and attention to detail is required to avoid missing a step in that process and introducing latent defects (which, of course, will not show up until the worst-possible moment).

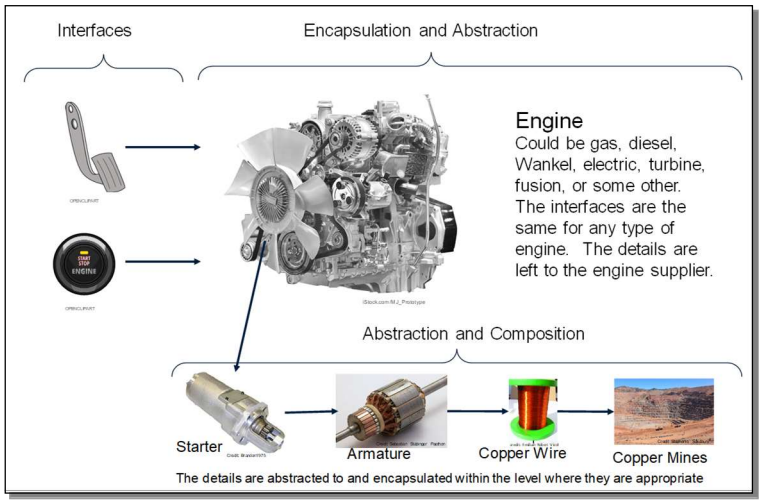
This centralized task-oriented approach was a big advancement over the flat approach, but it suffered from the need to modify each list or task when new functionality was added to the program. In addition, task-oriented programming often made it difficult to see the flow of information and to understand the cause-and-effect relationships in the control code. These drawbacks made programming more difficult to design and more complicated for plant technicians to maintain.

OOIP turns the task-oriented process on its side as shown in this graphic. Instead of the functionality being spread out amongst many tasks, the functionality is contained inside “Objects”. A single object performs everything that is associated with an input (reading, scaling, filtering, alarming, persistence, etc.), and that single object is reused for each input. To accommodate another input, simply add and configure another input object. Same with output objects (such as motors and valves).

In OOIP, since all the control is encapsulated inside the object, all that is necessary to add additional functionality is to add another self-contained block. No separate lists, processes, global variables to update and maintain (or *forget* to update).

The difference between task-based control and object-based control can be compared to different forms of governments. Task-based control is analogous to a strong centralized government where new functionality must register with the Federal Bureau of Scaling, and the Federal Bureau of Alarms, and such. Object-based is analogous to a decentralized government where new functionality is self-supporting and can largely take care of itself.

Since industrial control plants consist of objects (such as: motors, conveyors, valves, and sensors), Object-Oriented Programming is a natural choice for industrial controls – perhaps even more than the computer science programming for which OOP was originally created! It may have been more natural for OOP to have been invented in the PLC world and then spread to the IT world, instead of the other way around.

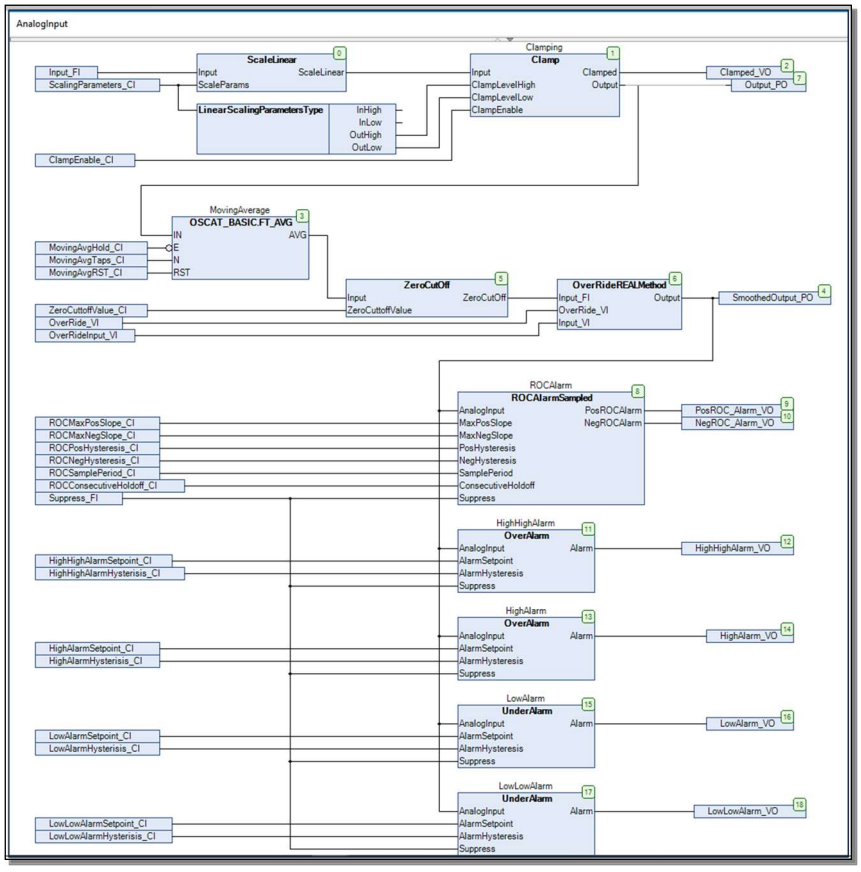


Encapsulation, Abstraction, and Interfaces

The object behind Object-Oriented Industrial Computing (OOIP) is to Encapsulate all the complexity into an object, and to Abstract the complexity into hierarchical levels where that complexity is necessary.

Encapsulation allows objects to be created which contain all the functionality and data necessary to control its matching plant object. The user does not need to know or understand the underlying implementation … they just use it! A good analogy is a car engine. The engine encapsulates pistons, valves, bearings, and a multitude of other objects and complex functionality. The driver doesn't need to know how an engine works – they only need to understand and interact with its interfaces: the start button and the accelerator pedal.

Abstraction is where detail is grouped by level in a hierarchy so that the programmer only needs to deal with the relevant level of complexity at any one level of the design. Composition is where Objects instantiate other Objects to build and logically partition large hierarchical systems. **Interfaces** provide a standardized means of interacting with the next level in the hierarchy. In the Mustang analogy from the previous chapter, the Mustang has an engine, which has a starter, which has an armature, which has copper wire, which is mined and refined at certain locations around the world as shown in this graphic. Abstraction allows you to leave the nested complexity of the engine and the mining of its copper to others where that level of detail is appropriate for their level in the hierarchy. You only need to know the Interfaces to engine – the ignition switch and the gas pedal.

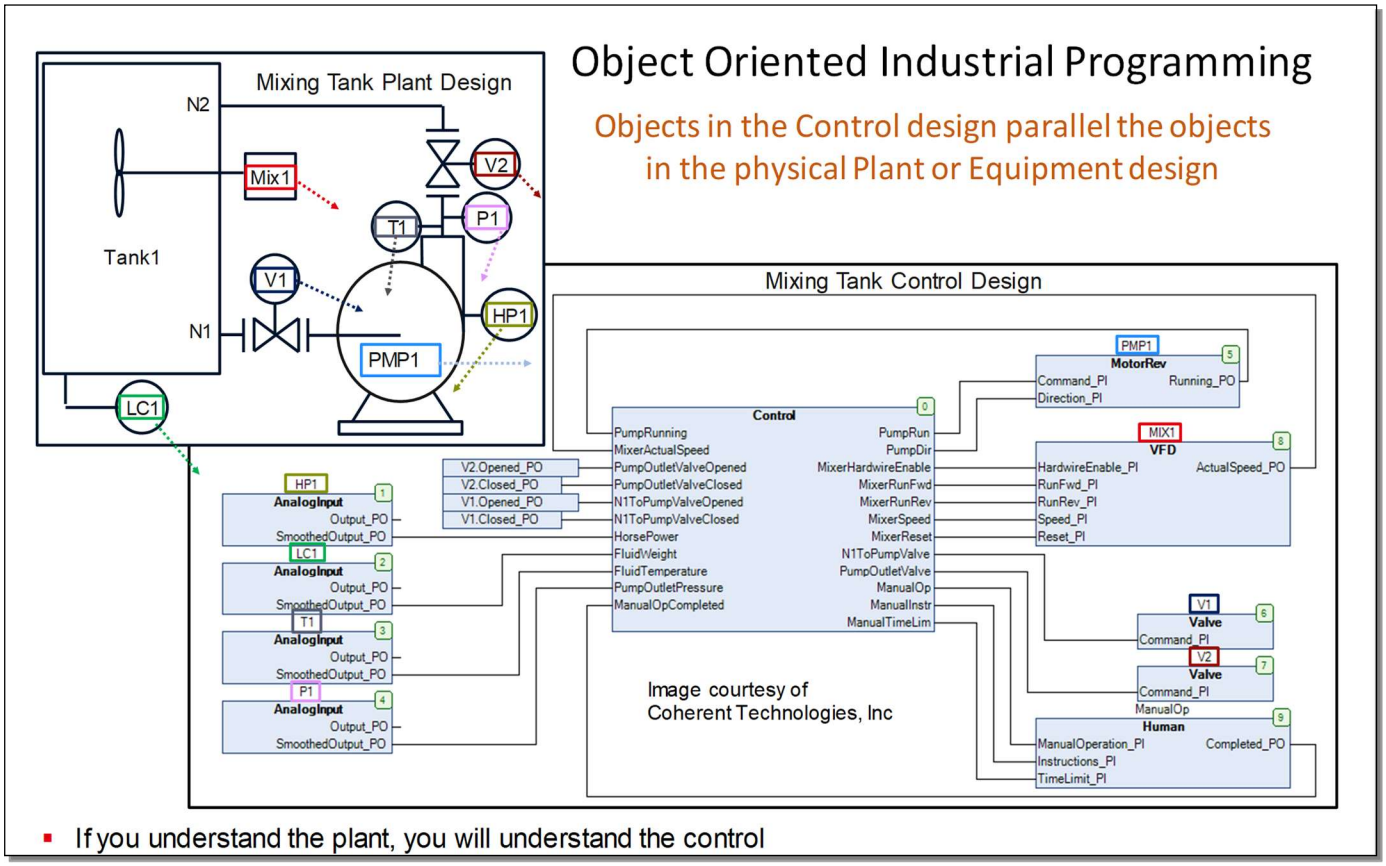


Sample Analog Input Object

This graphic shows an example of Analog Input object (courtesy of Coherent Technologies). This block encapsulates all the complexity of an Analog Input including scaling, clamping, filtering, override, rate-of-change alarming, and high/low alarming. The programmer is only concerned with the configuration of the block (the inputs on the left ending in _CI) and connecting the outputs (Output_PO and SmoothedOutput_PO). The programmer doesn't need to understand or be concerned with the underlying complexity. Just drop it in and use it … just like the engine in a car.

An example of this Analog Input object in actual use appears in the lower left side of the design on the next page.

Analog Input Object used in Plant Design



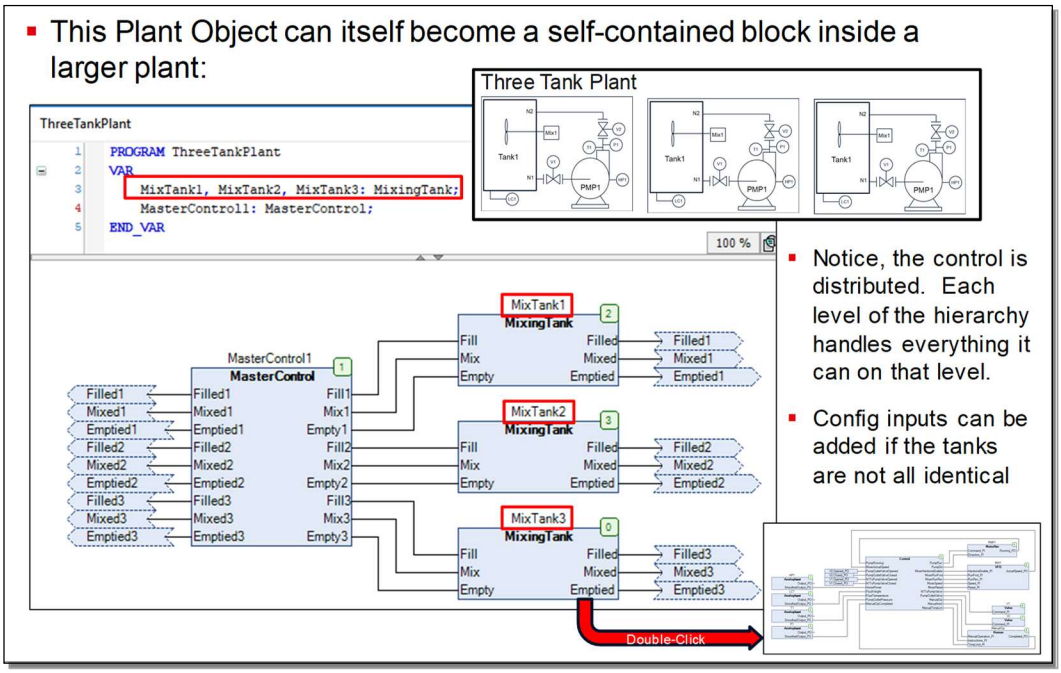
Control Objects can be designed to correspond to the objects in the plant in such a way that the control program begins to look similar to the physical plant design as shown in this graphic. Notice the one-to-one correspondence between the physical objects and the Control Objects.

The plant has four physical sensors for level, horsepower, temperature, and pressure – the control has four AnalogInput objects matching those four sensors. The plant has two motors and two valves – the control has two motor objects (MotorRev and VFD) and two Valve objects (and a Human object, which is explained later).

Due to this one-to-one correspondence, the plant design tool and the control design tool might one day be one-in-the-same. And suppliers of equipment might also provide the control block for that piece of equipment. It will become just as easy to configure and use an equipment control block as it is to specify and purchase a physical piece of equipment.

Notice this design has a “Human” object (lower-right corner). In this case, the plant had a human operation (dumping a bag of chemical into the mixer). In the control design, the human plant object is treated just like any other plant object. In this case, when the recipe calls for the chemical to be added, the control toggles the ManualOperation_PI input, and the Human control object signals the operator through the HMI. When the human is done, he/she responds to the HMI, which triggers the “Completed_PO” output which signals the control code to continue to the next step.

Object of Objects



Again, the object of OOIP is to build completely self-contained and self-reliant objects which can be used without any additional programming (such as adding its variables to a global variable list, or adding its alarms to an alarm manager, or adding its persistent variables to the Persistence Manager, etc.). Add the block into the design, connect its Program input and Program output pins, configure its parameters, and use it.

The beauty of this approach of encapsulating all the functionality in an object, is that the object can then itself be used as a self-contained and self-reliant building block. This is illustrated in this graphic where the mixer Program in the previous page is converted into a reusable object and then used to make a three-mixer plant. If the individual mixing tanks are not identical, then configuration inputs are added to modify the behavior of instance to accommodate those differences.

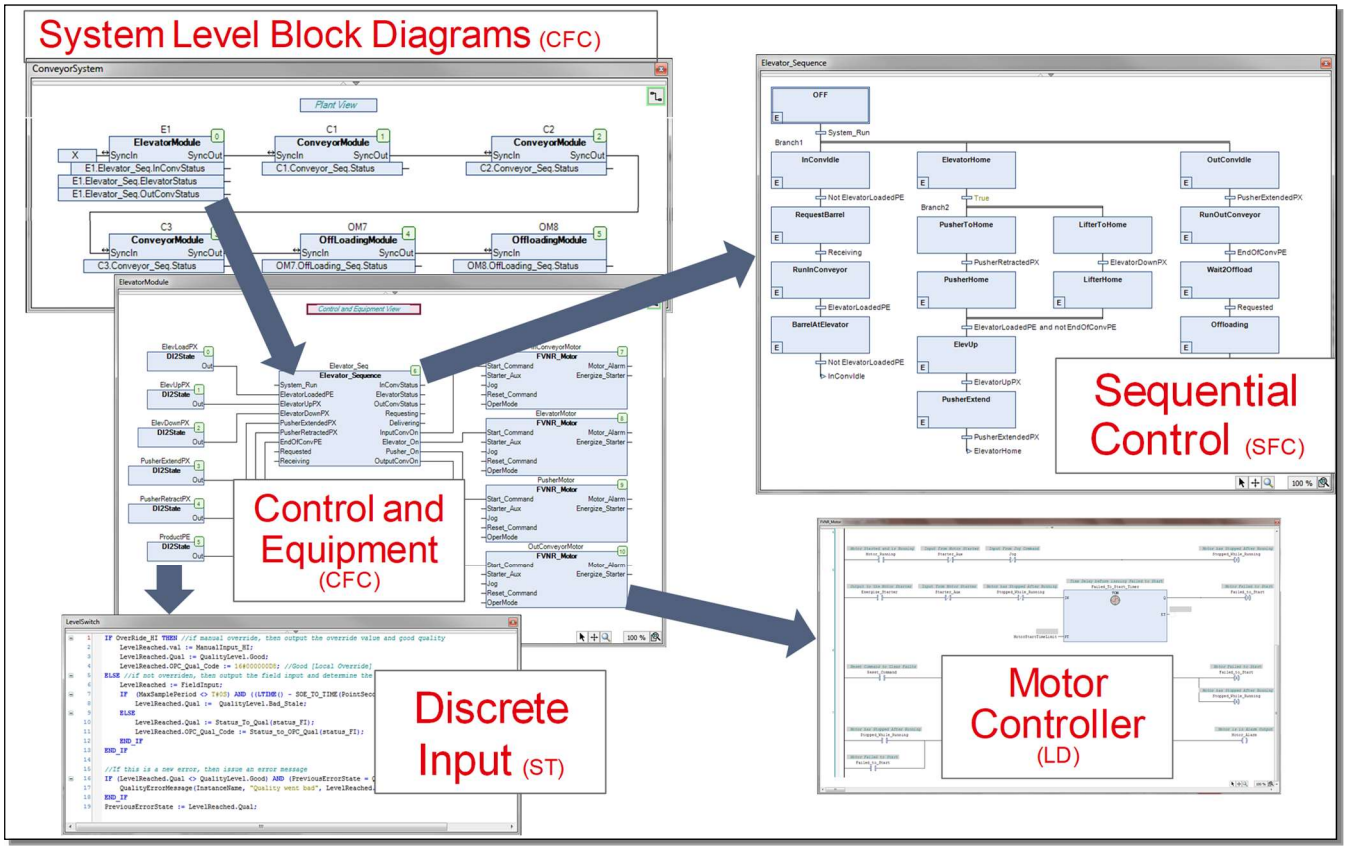
This is somewhat analogous to building a high-rise building. The single mixing tank is the first floor. It represents a firm foundation onto which to build the next floor – the three-mixer plant. This then could become the firm foundation on which to build the next level of the plant. In this way a plant of any complexity can be built just like a high-rise of any number of floors can be built as long as the floors below are properly designed. Since each object is self-contained and self-reliant, the complexity remains constant as the Program grows, instead of growing exponentially as with traditional PLC programming.

Let's use a motor controller as another example. Since the motor controller is totally self-contained, we drop it into the design, wire up its “Run” Program input to whatever control tells the motor to run, configure its parameters (including mapping its physical I/O), and we are done. Since the motor controller is totally self-contained and self-reliant, it handles all its own alarming, restarting, and such. In many cases, the level that uses the motor controller doesn’t even need to know if the motor actually started. That level just tells the motor to Run and that’s it. Everything is handled internally.

That is, unless the next level up needs to know if the motor is responding, as would be the case if the system had a redundant motor. However, in that scenario, the additional functionality would be abstracted away into an additional layer of hierarchy. A new RedundantMotor Function Block would be created which instantiates two or more Motor blocks along with the control logic to start a redundant motor if the current motor fails. Thus, the level that instantiates the RedundantMotor doesn’t need to know or care what is happening inside the RedundantMotor level. It just tells the RedundantMotor to run, and it is confident that the RedundantMotor will do what is necessary to keep a motor running. The RedundantMotor is totally self-contained and totally self-reliant.

Are you beginning to see the beauty of Object-Oriented Industrial Programming?

Example of Object-Oriented Plant Hierarchy



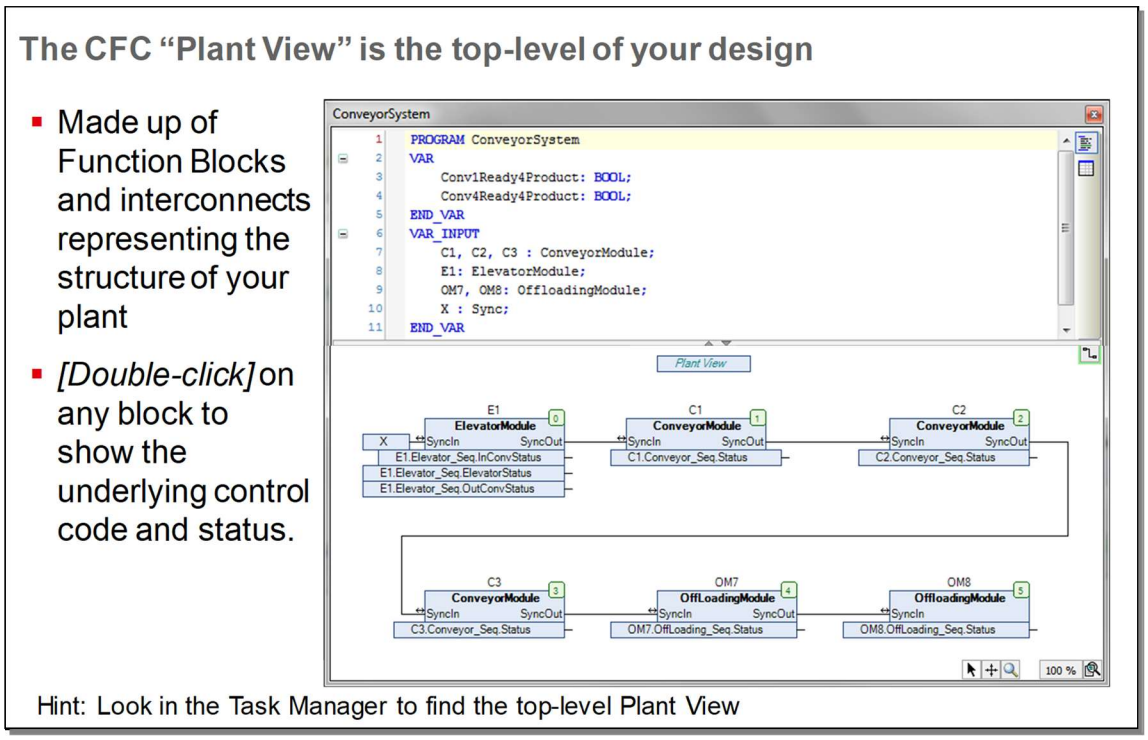
This graphic shows another example of an Object-Oriented Industrial Design. The foundation of this high-rise building is a Discrete Input module (proximity switches), and Motor Controllers. A control block is added to complete this Elevator module which is the foundation for the next level of the high-rise, which consists of this Elevator module plus three Conveyor modules and two Offloader modules. This container handler system could then itself be used as a foundation for a bigger plant, and so on and so forth. Again, since everything is self-contained and self-reliant, designs do not get bogged down with greater and greater complexity as the design grows. The complexity is addressed at each level, so there is no limit to the number of levels that can be built.

Again, with OOIP just drop in the objects, wire them up, configure them, and be done. There is no secondary work required such as adding the object's tags to an Alarm Manager, or a Persistence Manager, or global variable lists, or scaling lists, or filtering lists. Just drop it in and use it. Need another one? Just drop in a second one and use it. Need 100? Just drop them in and use them. Getting the idea yet?

This approach was not necessary when PLC programs were small (a split-level ranch). But, as the power of PLCs grows and the application for PLCs become more complex and demanding, OOIP provides a way to manage that complexity to allow unlimited growth (a 200-story skyscraper).

Each of the levels of this design are explained in more detail on the next several pages.

The Plant View

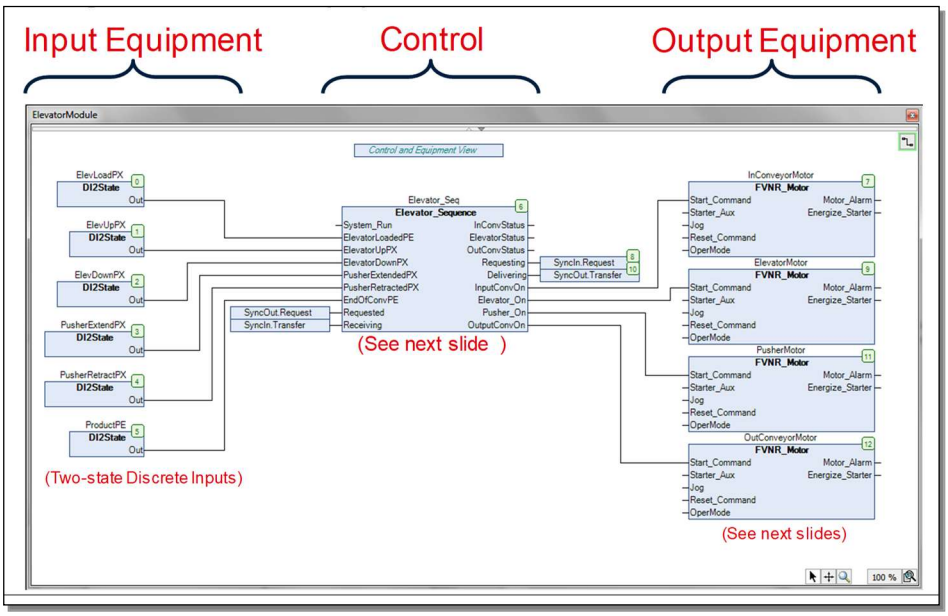


This graphic details the top-level of the material handling system described on the previous page. This is a block-diagram of the plant. (Or this could be a block diagram of a section of the plant, since each level can stand on its own – there is no way to know (except in CODESYS the top-level is a Program and all other levels are Function Blocks.))

This level of the plant consists of an Elevator, three Conveyors, and two Offloaders. Each of these are completely self-contained, self-reliant objects. Notice that the CFC toolbox "Input" items are used as a way of showing the internal status of these modules at this level (these are not part of the module; they are CFC Inputs that are butted up against the module's box). Notice these use relative path names to reach down in and grab variables further down in the hierarchy. Also note that these objects are declared as VAR_INPUT. Declaring these instances in this way allows the simulator to access to the internal variables of the instance that would otherwise not be accessible from this level (another alternative is provided later in this chapter).

Because this is CFC, the underlying Function Block can be opened by double-clicking on the box. When online, the underlying instance can also be opened by double-clicking on the box. This will open the editor for that particular instance showing the values of the variables for that particular instance.

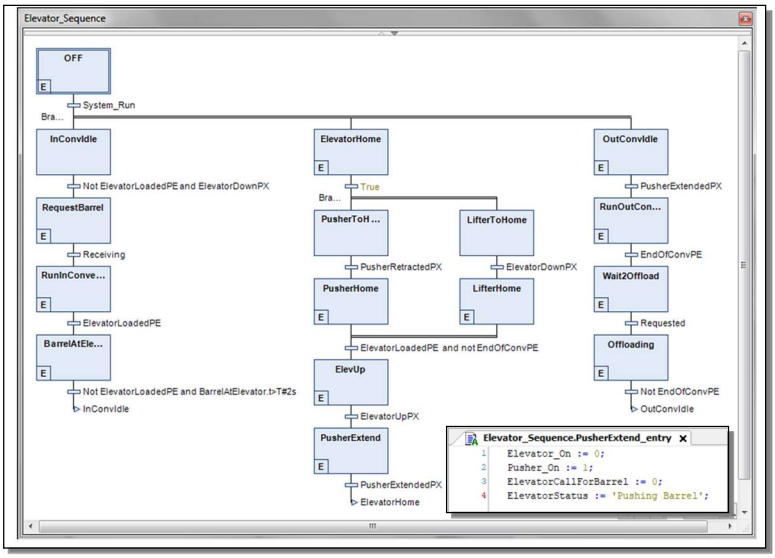
Notice that physical I/O has yet to be mentioned in this book. In fact, it won't be mentioned for several more chapters. In traditional PLC programming, the effort begins with the I/O tag list and proceeds from there. In OOIP programming, the effort begins by determining and assembling the required functionality. Mapping the physical I/O into that functionality is one of the last steps. In that way, when similar functionality is required in a different plant or piece of equipment, it's a simple matter to update the mapping with the new I/O. No rewriting the global variable list and all the task-based services on which it relies. Reusability is maximized.



Control and Equipment View (C&E)

Double-clicking on the ElevatorModule block opens the editor for the next level of the design – the Elevator's Control and Equipment Diagram (C&E). A C&E consists of Input objects on the left (proximity sensors, in this case), Output objects on the right (motor controllers, in this case), and a Control Block in the middle.

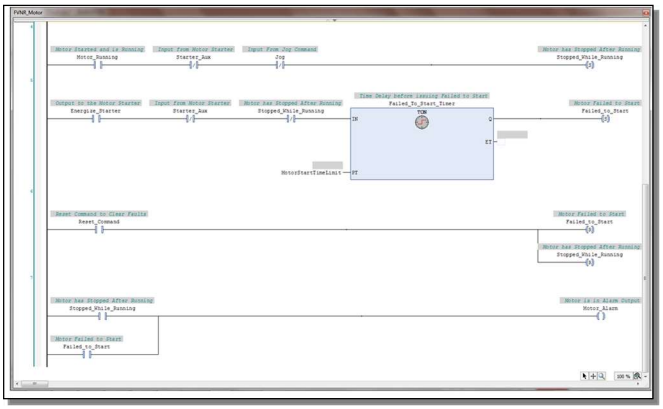
Notice that the outputs of the input objects drive the inputs to the control block (the ElevLoadPX.OUT output drives the Elevator_Seq.ElevatorLoadedPE input). And similarly, the control block outputs drive the inputs to the output objects (the Elevator_Seq.Pusher_On output drives the PusherMotor.Start_Command input).



Each of these objects are totally self-contained. Physical I/O is mapped to these Input and Output objects as the very last step in the design process as shown later in this chapter.

Control View

Double-clicking on the Control block opens the SFC editor showing how the Elevator module is controlled. Since this is a discrete sequential process, SFC is the best language for this job. The outputs of the proximity switches drive the Transitions, and the assignments in the Entry Actions drive the motor controller inputs.

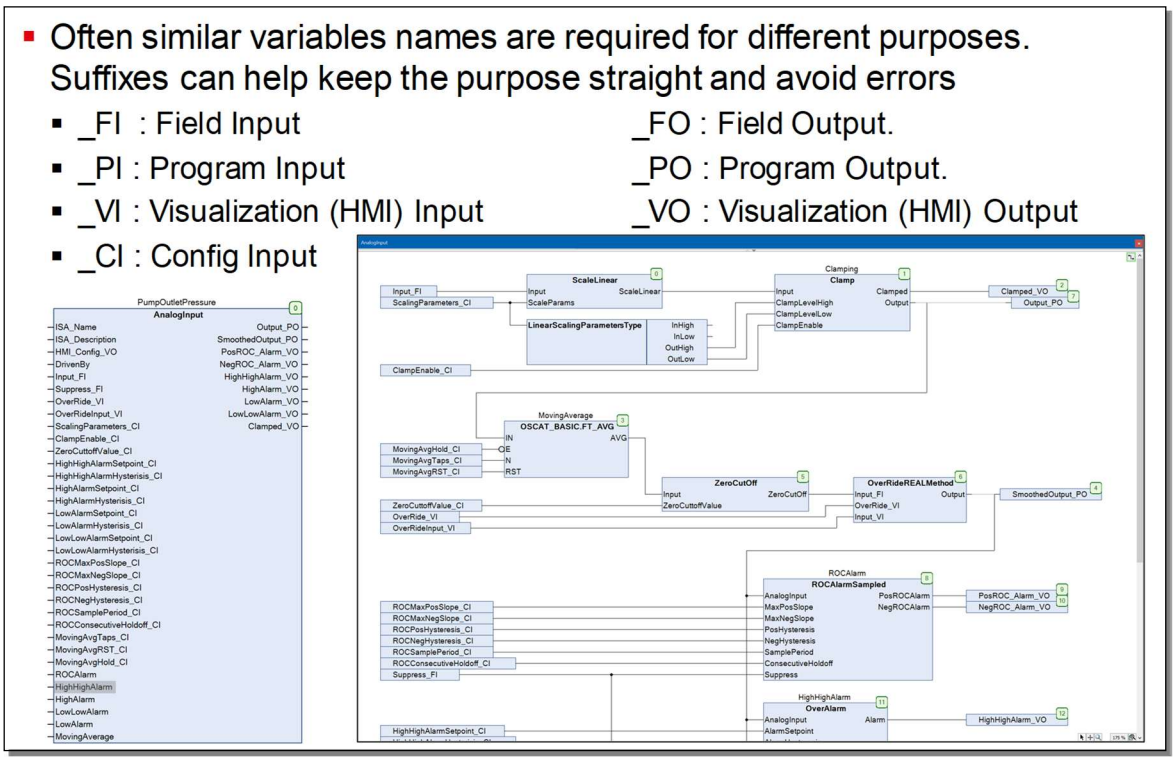


Motor View

Going back to the C&E and double-clicking on an FVNR_Motor shows the design of the motor controller. Since this is completely discrete logic consisting of timers and relays, it makes sense to build this in LD. (However, this does have seal-in contacts, so it is actually a State Machine and perhaps would be better built in SFC.)

The physical I/O will be mapped to Starter_Aux and Energize_Starter (its coil is on a lower rung that isn't visible in this screen shot). These will be mapped as described in the I/O chapter. (Unfortunately, this design was created without using the naming convention described on the next page. It would have been much clearer if those had been named Starter_Aux_FI and Energize_Starter_FO.)

Variable Naming Convention



In OOIP (and programming in general), Function Block VAR_INPUTS and VAR_OUTPUTs are connected to several different types of sources and destinations. Sometimes it become confusing exactly which I/O is intended for which source or destination. A naming convention is extremely helpful to address this issue. This author prefers this naming convention for VAR_INPUTs and VAR_OUTPUTs:

- ***._FI** and ***._FO**: for physical inputs from field devices and outputs to field devices.
- ***_PI** and ***_PO:** these are the interconnections within the program. _POs from one block typically connect to the *_PIs of the next, and vice versa.
- ***_VI** and ***_VO**: primarily intended for connections to the Visualization or HMI.
- ***_OpcI** and ***_OpcO**: Data to and from Client and Server connections via OPC UA
- ***_CI**: Configuration Inputs for adjusting the behavior of the object to meet the requirements of the application. These are usually initialized with one of the techniques described in the POUs chapter.

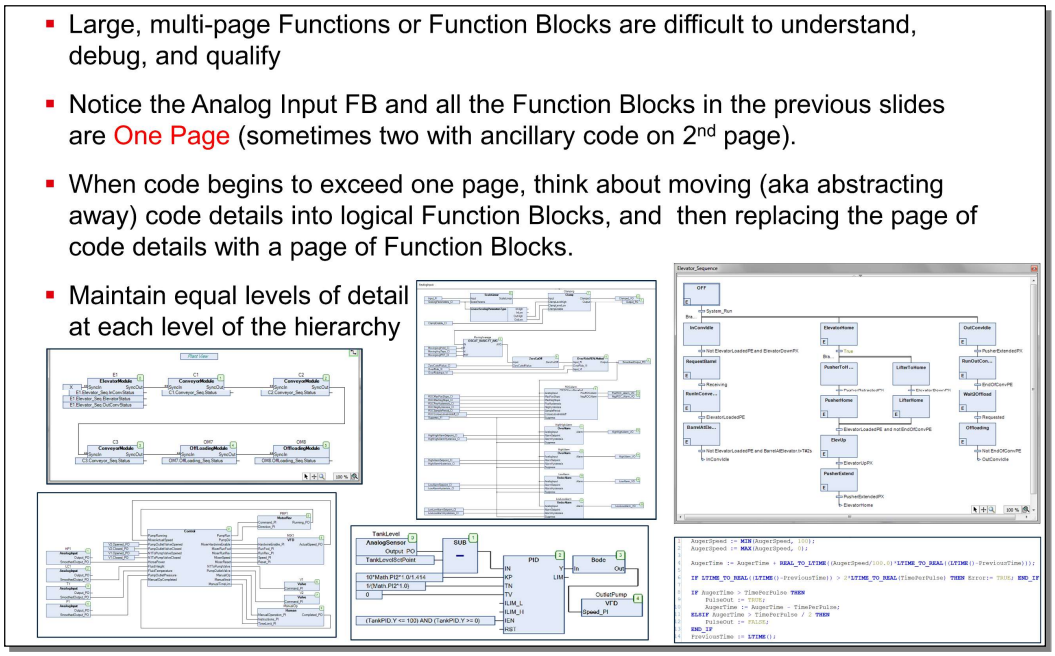
Where * represents the normal descriptive variable name.

Note: Some users prefer to place the naming convention in front of the variable so when IntelliSense or Input Assist display variable names in alphabetical order, the variables are ordered by intended type of source or destination.

Using this or a similar naming convention makes it much easier to find the correct variable and much less likely to use the wrong variable. For instance, when mapping the Analog Input object to the physical analog sensor in this example, one would look for a variable ending in _FI (Input_FI in this case). Or, when looking for a tag for the Visualization to indicate that an input is out of range, one would look for a variable ending in VO (Clamped_VO in this case). Or, when looking to connect the analog output to the next programming object that needs that signal, look for a variable ending in _PO (Output_PO or SmoothedOutput_PO in this case).

That having been said, there are exceptions to every rule. For instance, perhaps there is a reason the program needs to know if the input was out of range, so it would be necessary to use Clamped_VO as a program output. Exceptions can and will occur, but this covers the vast majority of cases and clears up the vast majority of possible errors and confusion.

Hierarchical vs Flat Design – Best Practice



This author finds it far better to build a large design from a hierarchical arrangement of small and manageable blocks, than to build one large flat design. A well-designed hierarchical design consisting of logically partitioned blocks is much easier to understand, debug, and validate than one large flat design. This author typically limits the size of a building block to one page (or two pages with the ancillary code relegated to page 2).

As we discussed in the Abstraction graphic earlier, try to partition your design into Function Blocks that have the level of detail that is appropriate for that position in the hierarchy. Just like the engine, the top-level is made up of the major components (alternator, starter, block, etc.). The next level down would be the pistons, cam shaft, etc. Continuing down the pistons would be the rings. Partition your control designs in exactly the same way. The engineer taking over your code will greatly appreciate it (as will you in two years if your memory is as bad as this author's). At every level, think what is important to the future user of this code to understand at this level of the design. If it has more detail than is necessary at that level, move the functionality into a Function Block with a descriptive name, and replace the code with that Function Block instance.

As a point of illustration as to why the concept of small building block and consistent level-of-detail is important, one only needs to open a typical Ladder Logic design. Most LD design consists of dozens or hundreds of pages of flat code, intermingling detailed design with high-level calls to other Function Blocks. This violates both the one-page building-block rule and the separation-of-detail rule. It is much better to partition the design into multiple layers, with the higher levels showing the overall functionality and interrelationships (the Block Diagram), and lower levels showing the detailed implementation.

Other areas that could be argued violate the separation-of-detail rule are Hungarian notation and namespace prefixes. Typically, the data type of a variable is only important when it is being declared. Some would argue that prefixing the variable with its data type introduces unnecessary clutter to the code which makes the code more difficult to read and understand. That detail is better left to a tooltip. Likewise, the library from which an object is obtained is again usually only important when the object is being chosen. Prefixing the object with its library of origin usually represents unnecessary clutter which makes the code difficult to read and understand. IEC 61131-3 and CODESYS desperately need a way to open namespaces for each POU so the library of origin can be handled in the declaration area where that level of detail is important, not in every line of code where an element of the library is used.

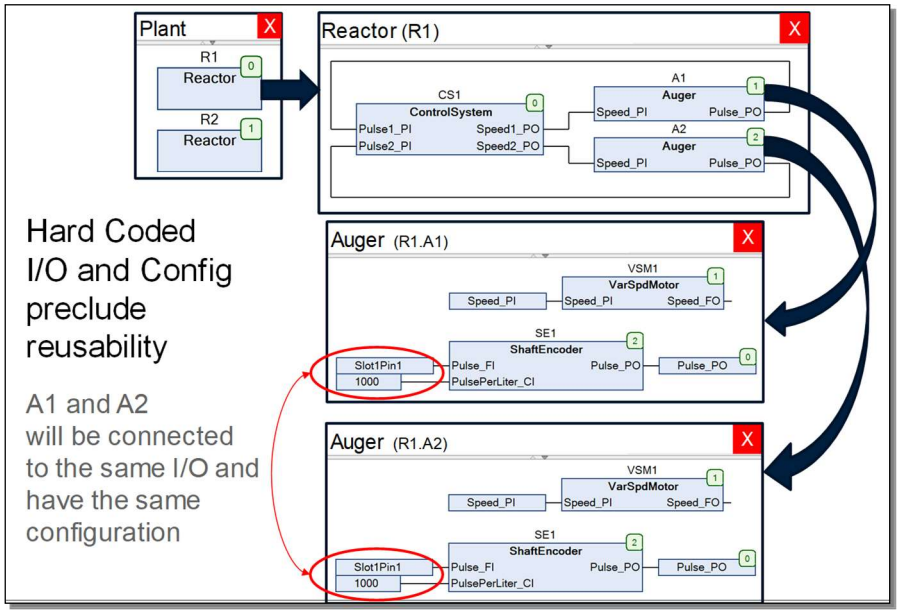
This concept will be revisited several times over the next few chapters (along with additional suggestions on how CODESYS could enhance the IDE to help their customers use these best practices).

IO and Configuration Mapping in OOIP

- It is impossible to *"hard code"* global variables onto objects that are re-used.
- Object re-use is a central tenet of structured object-oriented programming.
- For these reasons, it is necessary to map your I/O directly to your object instances using full path names. The traditional flat/global dating back to the 1970s is not compatible with Object-Oriented Programming techniques.
- Likewise, it is necessary to use similar techniques to write unique configuration parameters to individual instances of objects.
- See the next page for an example …

To fully realize the reusability benefit of OOIP, I/O mapping, and Parameters cannot be hard-coded into the instantiation of any object as they are in older programming techniques. This will be illustrated on the next several pages.

Hard-Coded Configuration and I/O Mapping



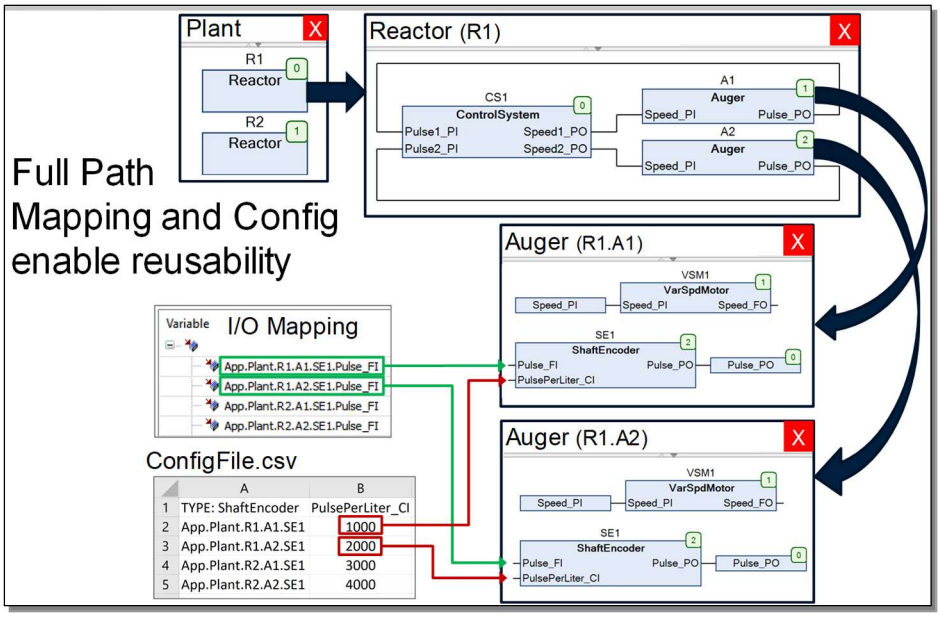
This figure shows how Abstraction, Composition, and Interfaces can be used to build a hierarchical process plant. At the top level, the Plant Program instantiates two Reactor objects, each of which have abstracted away the complexity of two Auger objects which themselves instantiate Motor and Shaft Encoder objects. The Shaft Encoder and Motor objects encapsulate all the functionality required to receive pulses from the shaft encoder and control the motor.

Thanks to Abstraction, our only concern at any one level of the hierarchy are the interfaces to the next level. For instance, the Variable Speed Motor in the Auger has an interface to set the speed of the motor. At the Auger level, we have no need to know or deal with any of the underlying complexity of the motor, such as determining if the motor is responding or generating alarms. The Variable Speed Motor is self-contained and self-reliant and takes care of its own error conditions and alarming.

Unfortunately, the traditional technique of hard-coding I/O and configurations is fundamentally incompatible with OOIP. For instance, in this example we have a Plant which has two Reactors, each of which have two Augers, all four of which have Shaft-Encoder interfaces which need a physical discrete input for the pulses from the physical shaft encoder (Pulse_FI) and four configurations for the amount of material that is moved per pulse of the shaft-encoder (PulsePerLiter_CI). If the physical input of the Auger were hard-coded to a global variable, all four augurs would be hard-coded to that single input. The Auger is not reusable. Same problem with the configuration. If that is hard-coded, then the Augur could only be used with augers of the same diameter, pitch, and encoder. The only solution would be to make four copies of the Auger, hard code the I/O global variable and the configuration into each of the four copies, make two copies of the Reactor and hard code two each of the Augers into each copy, and hard-code those two Reactor variations into the Plant. This is a laborious process, duplicate code is rampant, nothing is reusable, and there are huge opportunities to introduce errors in the code.

Earlier, we drew the analogy that the older task-based approach is like a centralized government, where the newer object-based approach is a society where individuals take care of themselves. Now, you might be thinking: "Even the most ardent libertarian agrees there is a need for some level of government." and "Not all 1964 Ford Mustangs are identical – they have different features and options – how are these handled in OOIP?" and "How does global I/O memory work with OOIP?" These are all very good questions. The first issue is addressed with Central Services, the second with Configuration Parameters, and the third with Full-Path I/O Mapping as described below.

Object-Oriented Configuration and I/O Mapping



CODESYS addresses the I/O mapping with a remarkable feature they designed into the I/O mapping tool. In traditional programming, a global variable would be placed in the I/O map, and then that global variable would be hard-coded into the program where it is used. With OOIP, instead of mapping to a global variable, the I/O is mapped with a full path name directly to the instance and variable that connects to the physical I/O. A full path name is the dot-separated combination of the Program name, followed by all the intervening instance names, and ending with the variable name.

In this example, the first discrete I/O point is mapped to the PULSE_FI input in ShaftEncoder SE1, in Auger A1, in Reactor R1, in the Plant Program, in the App Application. So, its full path name is App.Plant.R1.A1.SE1.Pulse_FI. The second discrete input is connected to the second auger, so its full path name is App.Plant.R1.**A2**.SE1.Pulse_FI. The third and fourth discrete inputs are mapped to the same augers except in the second reactor (**R2**). Because of this I/O mapping technique, the Auger is now reusable. Notice, the naming convention (*_FI) to make it easier to determine where to connect the physical I/O.

Unfortunately, CODESYS doesn't offer something similar for setting configurations. Fortunately, ControlSphere has created the ControlSphere Parameter and Persistence library (CPP) to address this issue. The CPP library automatically creates a spreadsheet which is organized by FB type and has a row for each FB instance and a column for each configuration input for that FB. The CPP library creates this CSV file containing the default values for the configuration inputs. These values may be changed and the CSV file read back in to apply those changes.

In this example, each Shaft Encoder needs to know the number of pulses per liter of material. The CPP library writes a CSV file which is grouped by Function Block type (ShaftEncoder), a header-line showing the names of all the configurable inputs to the ShaftEncoder (PulsePerLiter_CI), a row containing a full path name to each Shaft Encoder instance (App.Plant.R1.A1.SE1, etc.), and the default values for each configurable input in each instance of the ShaftEncoder. The user can then change the default values to that which is appropriate for each instance (1000, 2000, 3000, and 4000 in this example). Then, the CSV file can then be read by the controller and the new values

will be copied in their appropriate variables. This read operation can be performed automatically on startup, or can be commanded by the control or HMI code.

If the Application includes Visualization which allows the operator to set configurations, that Visualization page would change the configuration variable values and then write the CSV file so those changes become permanent (will survive a power-cycle, equipment replacement, or even movement to a new plant or piece of equipment).

The Advanced chapter will show how additional OOP techniques are used to implement CPP library.

Sample of an Actual Configuration/Persistence CSV File

	A	B	C	D	E	F	G	H	I	J	
48	TYPE:CalcCylOilLoadsFB	ISA_Name_CI	BoreInch	RodInch	SPARE	GasRatio	InitialGas	MaxOperF	CalibratedPositionFromF		
49	BOATCYLINDERS.GENOASTAYSAIL.CALCCYLOILLOADS	Genoa Stay Sail	1.75	0.5	SPARE	10	5	500	250		
50	BOATCYLINDERS.HEADSTAY.CALCCYLOILLOADS	Head Stay	3.125	0.875	SPARE	10	1.00E-06	500	125		
51	BOATCYLINDERS.JIBCUNNINGHAM.CALCCYLOILLOADS	Jib Cunningham	2.188	0.625	SPARE	10	5	500	200		
52	BOATCYLINDERS.JIBINOUT.CALCCYLOILLOADS	Jib In Out	1.75	0.5	SPARE	10	5	500	200		
53	BOATCYLINDERS.JIBUPDOWN.CALCCYLOILLOADS	Jib Up Down	1.75	0.5	SPARE	10	5	500	250		
54	BOATCYLINDERS.LOWERDEFLECTOR.CALCCYLOILLOADS	Lower Deflector	1.75	0.5	SPARE	10	5	500	275		
55	BOATCYLINDERS.MAINCUNNINGHAM.CALCCYLOILLOADS	Main Cunningham	1.125	0.375	SPARE	10	5	500	200		
56	BOATCYLINDERS.OUTHAUL.CALCCYLOILLOADS	Out Haul	1.5	0.438	SPARE	10	5	500	125		
57	BOATCYLINDERS.SPARE.CALCCYLOILLOADS	Spare	1.75	0.5	SPARE	10	5	500	250		
58	BOATCYLINDERS.UPPERDEFLECTOR.CALCCYLOILLOADS	Upper Deflector	2.75	0.813	SPARE	10	5	500	275		
59	BOATCYLINDERS.VANG.CALCCYLOILLOADS	Vang	2.375	1.25	SPARE	10	5	500	232.5		
60											
61	TYPE:HydraulicPumpControlFB	Title	EngineMinRPM	EngineRPI	EngineMa	PartyMod	RPMtoLPM	PowerShi	PowerShi	MaxPowe	Min
62	BOATHYDRAULICPUMPCONTROL	Hydraulic Pump Co	500	300	2900	8	10	1800	T#3s	55	
63											
64	TYPE:KeelCylinderSystem	Title	DownPosition	UpPositio	PositionT	DownPres	UpPressu	UpStopDe	DnStopDe	StuckTime	OilL
65	BOATCYLINDERS.KEEL	Keel	85.70625	1.04375	5	50	275	10000	3000	30000	
66											
67	TYPE:KiteRetrieverSystem	Title	Disable	Pretensio	Pretensio	MinFlowC	RaceMaxF	CruiseMa	RaceMaxF	CruiseMa	Min
68	BOATWINCHES.KITERETRIEVER	Kite Retriever	FALSE	425	125	350	700	700	700	700	
69											
70	TYPE:SADECylinderSystemFB	Title	Disable	DoubleCli	DoubleCli	OilUserCl	ExtendPo	RetractPo	CylParams_CI.Stroke		
71	BOATCYLINDERS.GENOASTAYSAIL	Genoa Stay Sail	FALSE	1	T#400ms	2	0	1000	500		
72	BOATCYLINDERS.HEADSTAY	Head Stay	FALSE	1	T#400ms	2	0	1000	250		
73	BOATCYLINDERS.JIBCUNNINGHAM	Jib Cunningham	FALSE	1	T#400ms	2	0	1000	400		
74	BOATCYLINDERS.JIBINOUT	Jib In Out	FALSE	1	T#400ms	2	0	1000	400		
75	BOATCYLINDERS.JIBUPDOWN	Jib Up Down	FALSE	1	T#400ms	2	0	1000	500		

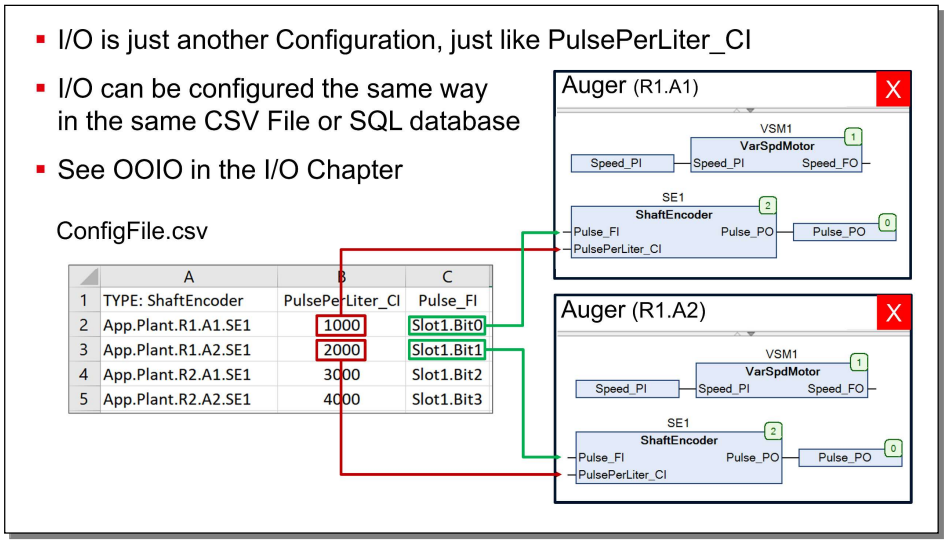
Example courtesy of Marine Hydraulics Consultancy, Poulsboro, WA

Here is an example of a real configuration file compliments of Marine Hydraulics. Notice the CSV file is grouped by FB type (CalcCylOilLoads, PumpControlFB, KiteRetrieverSystem, SADECylinderSystemsFB, etc.). Notice within each group there is a row for each instance of that FB. Also, notice there is a header row at the beginning of each section containing the names of each of the configuration inputs, with the values of each configuration parameter for each instance in cells below the header.

Marine Hydraulics has a mixture of configurations that are fixed for each boat, and some configurations which can be changed by the operator through a Visualization. The programmer codes the boat-specific configurations into the CSV file which is read on startup. The Visualization writes the configuration file after it makes any changes to the operator-configurations so that those changes become permanent.

This shows the Power of Object-Oriented Industrial Programming. Declare instance of your reusable Function Blocks, create a Configuration CSV file with all the default values, update those values as appropriate, read the file back into the project, and start your plant or equipment.

I/O Mapping as a Configuration



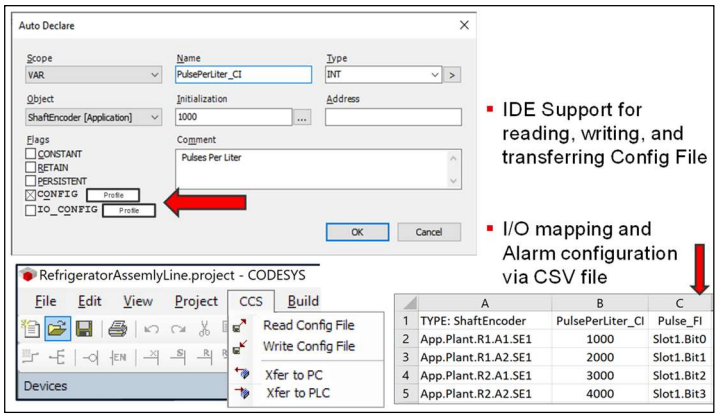
Taking this concept one step further – if one thinks about it, the I/O mapping is just another configuration. It is something of interest to each instance and should be configured along with all the other configuration inputs for that instance. It's just another column in the configuration spreadsheet (maybe two columns if the I/O itself is configurable, like as an Input or an Output, or Analog or Discrete, etc.). All that information is unique to each instance and should be part of that instance's configuration.

In this scenario, an I/O module would be a configurable reusable Function Block just like every other object we've talked about in this chapter. It would have its own configuration for items like its name, CAN address, and such. It would offer up its name and its I/O points to an I/O Mapping Central Service which would then coordinate with each equipment instance to make the connections specified in the equipment's CSV configuration.

Alternatively, the I/O mapping could be specified in the CSV configuration for the I/O module FB instance.

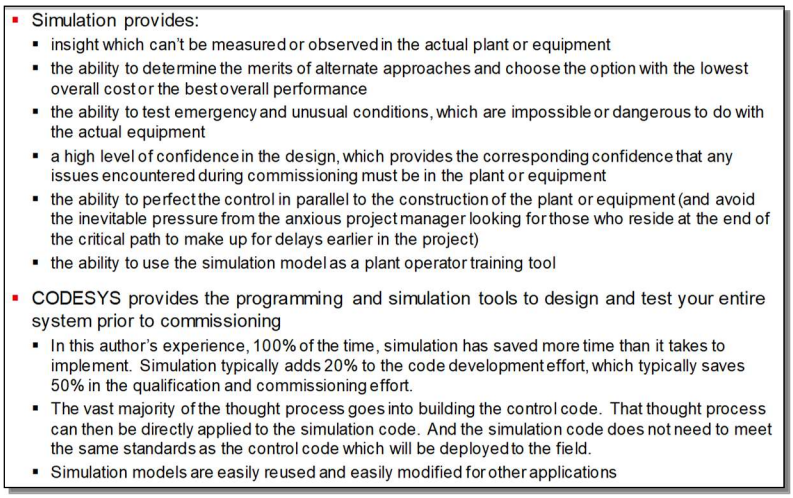
In either scenario, the CODESYS I/O wouldn't be used at all. See the Object-Oriented I/O (OOIO) section of the I/O chapter for more information on this technique.

Future Enhancement



In the future, IDE enhancements could be made to integrate the features of the Parameter and Persistence library into the IDE. This could include additional flags that designate a variable as a Parameter/Retain Configuration or an I/O Configuration. Automation could be included to replace the Visualization used to read and write the configuration files, and automation could be added to transfer the configuration file back and forth between the PC (for editing) and the PLC (for transferring the configuration variable values to the runtime). It could also open the file directly in Excel.

Note that functionality similar to the Central Configuration Services could be used to provide both a Central Alarm Service and a Central I/O Service. In addition to simplifying the I/O mapping process, the Central I/O service would provide the ability to reconfigure I/O in an Online Change (which is currently not possible and a significant issue using CODESYS in applications which cannot be shut down for modification). These are examples of functionality which is handled as an external manual post process in traditional PLC programming that could instead be built into self-contained and self-reliant Function Blocks.



Plant-Level Simulation

To simulate, or not to simulate: that is the question. 'Tis nobler (or at least more efficient) to spend the time to create simulation models and test the design prior to deployment, or to spend the time testing the design during deployment?

While I can't speak for Shakespeare, I can say that in my 40-year career 'tis always more efficient to perfect the design prior to deployment. I've successfully applied simulation to a wide variety of applications including many different types of industrial controls systems, electronic PCB circuit design, and FPGA design. In my experience, simulation typically pays for itself many times over due to:

- the insight simulation models provide which can't possibly be measured or observed in the actual plant or equipment,
- the ability to quickly determine the merits of alternate approaches and choose the option with the lowest overall cost or the best overall performance,
- the ability to test emergency and unusual conditions which are impossible or dangerous to do with the real equipment,
- the high level of confidence in the design which provides the corresponding confidence that any issues encountered during commissioning must be in the plant or equipment,
- the ability to perfect the control in parallel to the construction of the plant or equipment (and avoid the inevitable pressure from the anxious project manager looking for those of us who reside at the end of the critical path to make up for delays earlier in the project).

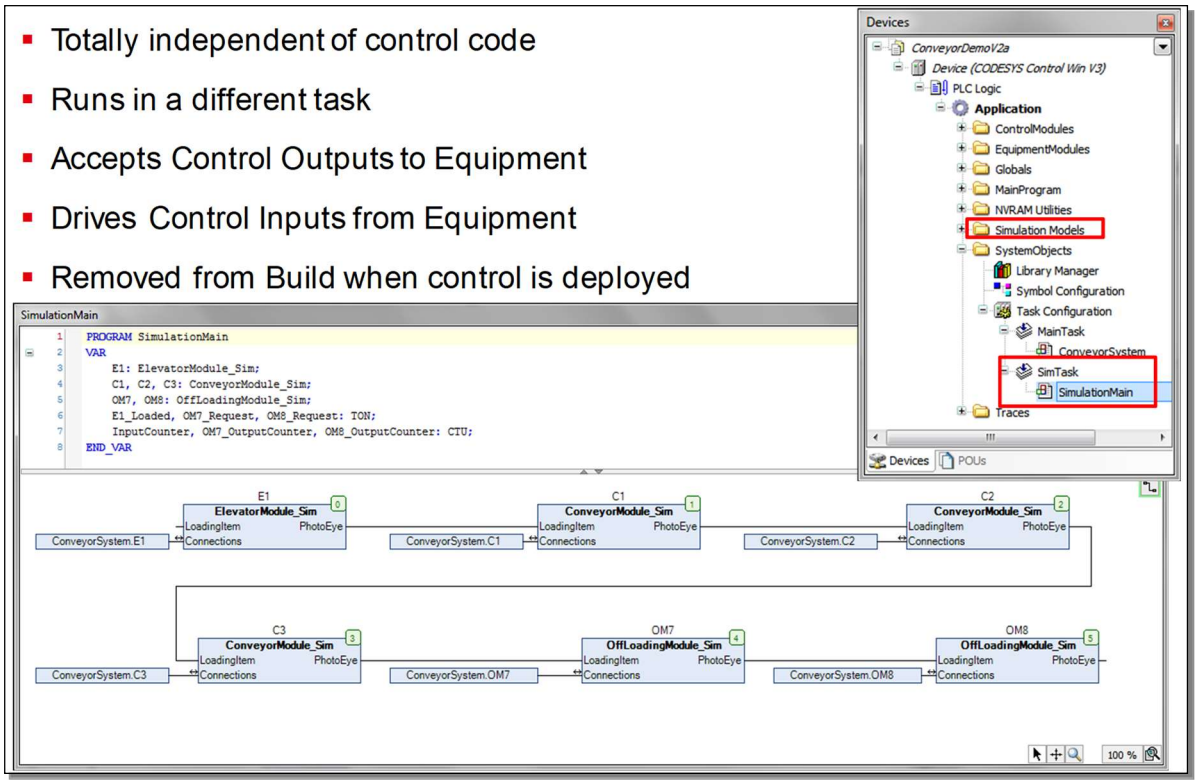
This return on investment becomes even greater with modern development and simulation environments which include Object-Oriented Industrial Programming (OOIP) tools to accelerate development, and advanced debugging features which accelerate the time-to-insight.

The characteristics of a good Industrial Controls (IC) simulation environment look very similar to that of a good IC development environment:

- Versatile and powerful programming languages
- Full featured language editors
- Full suite of debugging tools including:
  - Code and data breakpoints
  - Single-stepping, step-in, step-out, etc.
  - Live Mode (to show instantaneous variable values, not just end of cycle values)
  - Write and Force variables, and move the execution point
  - Virtual digital oscilloscope which samples at the controller cycle time
- Built-in HMI for creating test control panels
- A complete controller runtime which runs as a service on the development computer
- Support for Object-Oriented Industrial Programming (OOIP)

Fortunately, the CODESYS IDE and the CODESYS ControlWin soft PLC offer all these features and is just as good as a simulation/verification environment as it is a development environment. With such an environment, creating the simulation code is as easy as creating the original code. The next few pages will show how this is accomplished using the CODESYS IDE.

Simulation Code



The simulation code often mirrors the control code. Here is the simulation code for the container handling system described earlier in this chapter. It contains the same blocks as the control code, except this is using the simulation version of each system block (notice the *_Sim in the name). (This diagram was actually a copy of the system-level diagram with the names changed.)

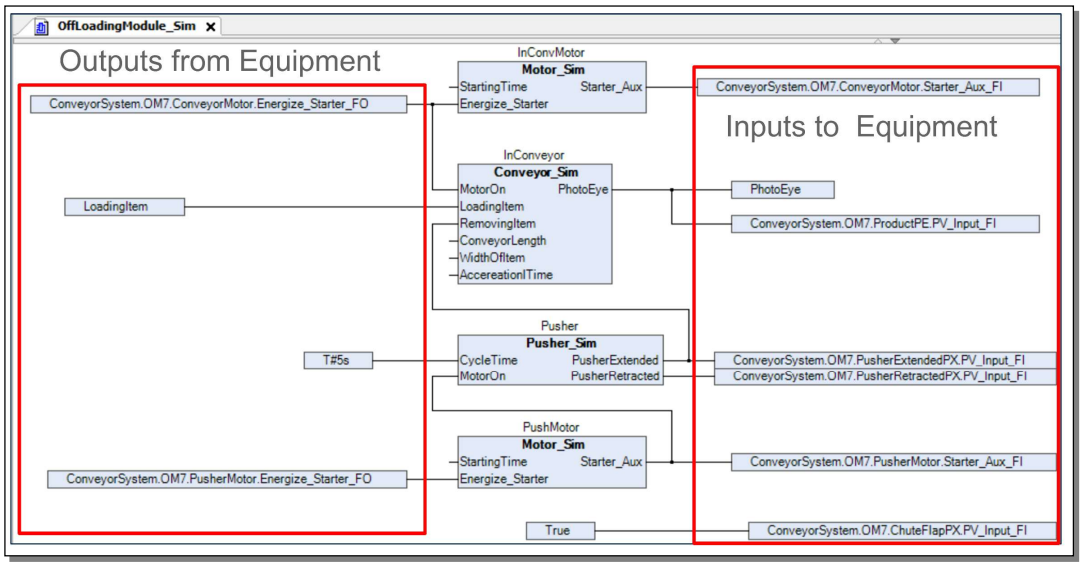
Notice that this author prefers the simulation code to be totally separate from the control code. It's stored in its own separate folder and it has its own separate Task. The Task is usually set to run much faster than control code, because the simulator is usually simulating something that is running in real-time. It's advisable to select a cycle time that is not an even sub-multiple of the control cycle time, so there is no chance some sort of synchronizing or aliasing could conceal a control issue. (For instance, selecting 3ms simulation for a 20ms control cycle.)

With the control code as a separate entity in its own folder, it's a simple matter to exclude or include the folder in the build to add or remove the simulation code. This way, when the simulation is not being used, it is completely out of the way, it takes no controller resources, nor could it possibly ever interfere with the control code.

Some engineers prefer to place the simulation code in the control code and have a global variable to enable and disable the simulator. This often reduces or eliminates the I/O mapping step as described on the next page. But it does use controller resources after the control code has been released to the field, and there is always the chance the code could cause issues. There is no right answer with respect to the decision of where to place the simulation code.

As was mentioned in the debugging chapter, one issue that occurs with simulation is what to do with the configured physical I/O when simulating. This isn't a problem when using Simulation built into the IDE because Simulation ignores any configured I/O. But in many cases Simulation has limitations which force the use of ControlWin (which does not ignore I/O). Some hardware vendors provide a version of ControlWin which accepts and ignores the I/O from that vendor. That is the best solution when it is offered. Without that, the only way to use ControlWin to simulate is to delete the I/O and change the device to ControlWin, then change the device back to the actual device and paste the I/O back into the tree to run on the actual hardware with physical I/O.

Simulation Model



Pushing into the OffloadingModule_SIM shows that the simulator for this module consists of two Motor simulators, a Conveyor simulator, and a Pusher simulator. Notice that the inputs to the simulators are the full path names to the physical outputs of the control code and vis versa for the simulator outputs. These are the same full path names that you will use when mapping the real I/O to the control code.

The Motor_SIM accepts the *Energize_Starter_FO control code output and after Starting_Time has expired it sets the *Starter_Aux_FI control code input to true. Starting_Time can be changed by the simulator test sequence code to test the control code response when the motor starts slower than expected or fails to start. These are a few rungs of LD. The Conveyor_Sim records the time when the LoadingItem transitions to true, then knowing the length and speed of the conveyor, activates the *PV_Input_FI input when each item reaches the end of the conveyor. This requires a dozen or so lines of ST code. The Pusher_SIM cycles between activating the PusherExtendedPX and PusherRetractedPX PV_INPUT_FI inputs to simulate an arm on the end of a rotating cam with proximity switches on the two extremes. Another few lines of code.

In total, the entire Offloader simulator is a couple dozen or so lines of LD and ST code – a very small price to pay for the confidence that the design is complete and correct before it is deployed to the field. This code can be viewed in in the StartingLabs.project, Lab_11_PowerOfObjectOrientedProgramming, DiscreteProcess application.

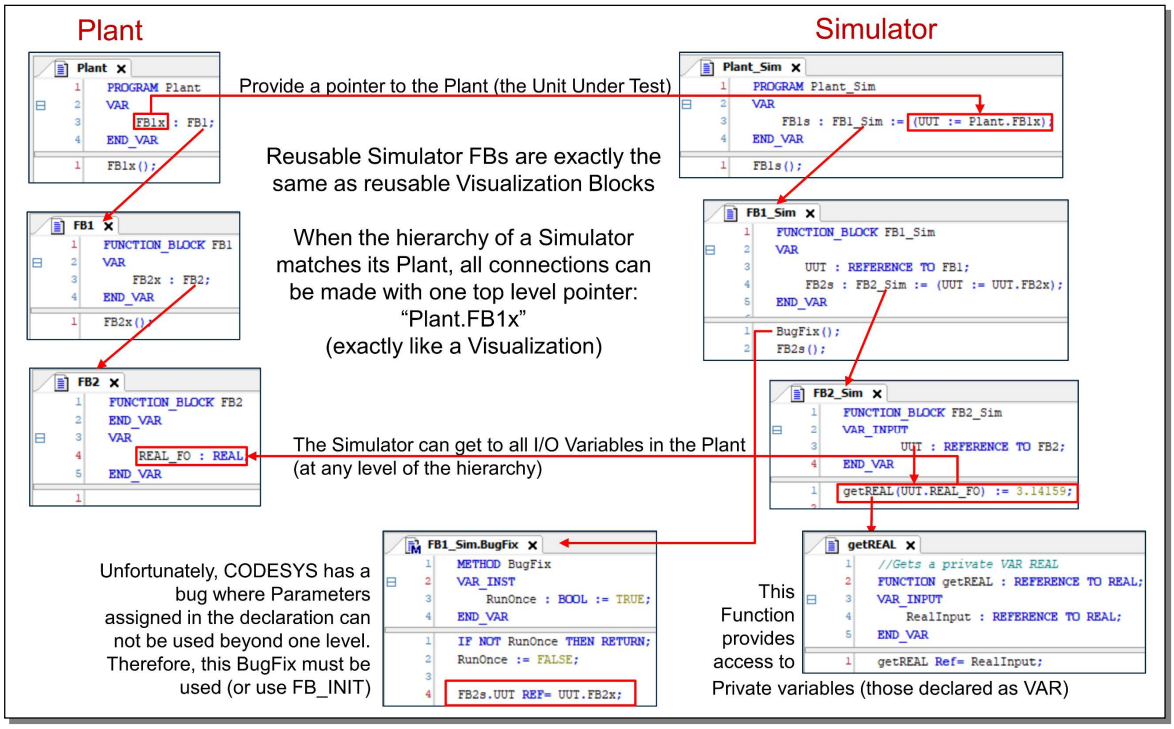
Note: To promote good coding techniques, IEC61131-3 does not allow user-access to private variables (i.e., variables declared as VAR, or variables declared as VAR_INPUTs and VAR_OUTPUT inside a Function Block whose instance is declared as a VAR). This is not an issue for I/O variables because the CODESYS I/O mapping tool provides super-user access to private variables (Note 2). Unfortunately, your user-created simulation code does not share that special access. You have two options to provide your simulation code with the same access as the CODESYS I/O mapping tool:

1. Declare all your I/O variables as VAR_INPUT or VAR_OUTPUT, and declare all instances of FBs between the top level program and the FB where those variables appear as VAR_INPUT.
2. Use the Reference technique as show in the getReal() function on the next page (Note 1).

Note 1: If CODESYS supported the ability to dereference the results of the ADR() function, (i.e., "(ADR(VarToReach))^"), this would be a simpler alternative to the Reference technique. Unfortunately, CODESYS does not currently support this syntax.

Note 2: Other tools in CODESYS also have super-user access, including: Visualization, Recipes, Trace, etc.

Reusable Simulation Objects



In the Visualization chapter, you will learn you can make a Visualization Block that mates to a Function Block. You will also learn that when the hierarchy of your control Function Blocks matches the hierarchy of your Visualization, you can make all the connections between the two with one pointer. The same can be done with Simulation models. If the Simulation model hierarchy matches the control code hierarchy, you can make all the I/O connections with one pointer (Plant.FB1x, in this example).

The only minor difference is Visualizations use VAR_IN_OUT, where simulations use References. Both are pointers, and the subtle differences are covered in the Advanced Topics chapter.

So, in this example, the Plant has an FB1, and the FB1 has an FB2. The simulation model for FB is called FB1_Sim, and FB1_SIM has a "REFERENCE TO FB1" (called UUT). To connect the Simulation model to the Plant model, just provide the address Plant.FB1x to the FB1_Sim. This assignment could be made in the declaration of the instance of FB1_SIM as shown in the graphic, or "UUT : REFERENCE TO FB1" could be declared as a VAR_INPUT and then assigned when the instance is called (in Plant_Sim line 1, in this example). The advantage to doing the assignment in the instance declaration, is that it is done during compile time and contributes no runtime CPU load.

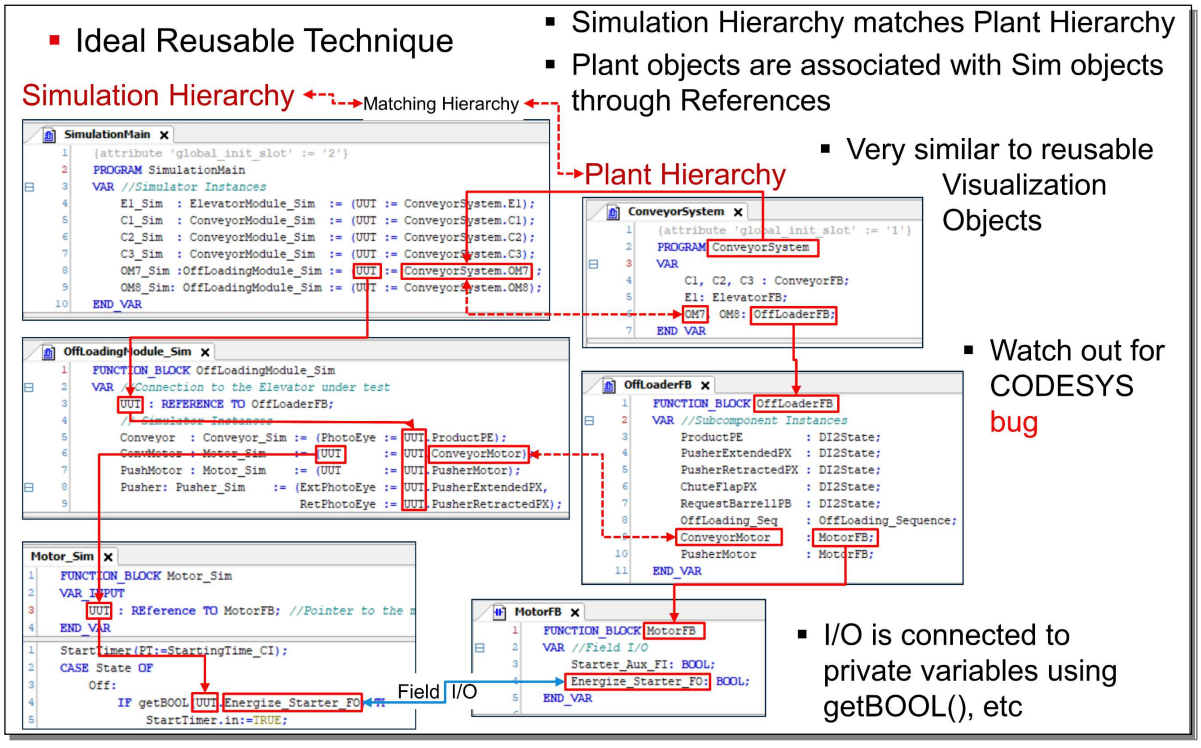
Unfortunately, CODESYS has a bug that does not allow initialization parameters to be passed down more than one level in the hierarchy. The graphic shows a workaround until that bug is fixed (BugFix). Another alternative is to use Constructors (FB_INIT) to pass the pointers (initialization technique #8 as described in the POU chapter).

In the control code, it is preferable to declare field I/O as private variables (VAR) so the pins do not clutter your block diagrams, and they are not accidentally assigned in the control code. Unfortunately, since the simulator is user-level code, it does not have the same super-user ability to connect to private variables as does the I/O mapping in the CODESYS I/O tree. To address that issue, the graphic shows an example of getX Functions (i.e., getReal) that provide user-level code with super-user access to private variables. You must create a version of this Function for every data type of I/O you have in your control design. This provides the user-level simulation code with the same super-user access to variables as the CODESYS I/O mapping tool. In control code, the fact that private variables are private for a very good reason, so this technique to work around that rule should only be used in simulation code – not in control code.

Notice, that since the control and simulation hierarchies match, the I/O can be assigned using the local pointer for that level of the hierarchy (the "UUT" in "UUT.REAL_FO," in this example). There is no need to use full path names for every assignment (which would make the simulation model non-reusable). To keep this example simple, it might not be the best demonstration of the value of reusable simulation objects. Instead, think about how this would look if there were hundreds of FB1s and FB2s distributed throughout the hierarchy of the plant. Having reusable simulation objects which can be tied to matching plant objects can save a huge amount of duplicate code and hard-coded path names.

Note that in this case "UUT : REFERENCE TO FB1" is declared as VAR, where "UUT : REFERENCE TO FB2" is declared as VAR_INPUT. The reference to FB1 can be declared as VAR because the CODESYS bug does not come into play for the first level of the hierarchy. The reference to FB2 must be declared VAR_INPUT for BugFix to work. Sorry about that. The other alternative is to place the "UUT := UUT.FB2x" in the call to FB2s on line 2, for example "FB2s(UUT := UUT.FB2x);". When using that technique, the UUT could be declared as a VAR_IN_OUT (which is usually preferable to VAR REFERENCE when possible). This technique is shown on the Simulation Code graphic, earlier in this chapter.

Examples of the GetReal and BugFix can be found in the StartingLabs.project, Lab_11_PowerOfObjectOrientedProgramming, DiscreteProcess application.

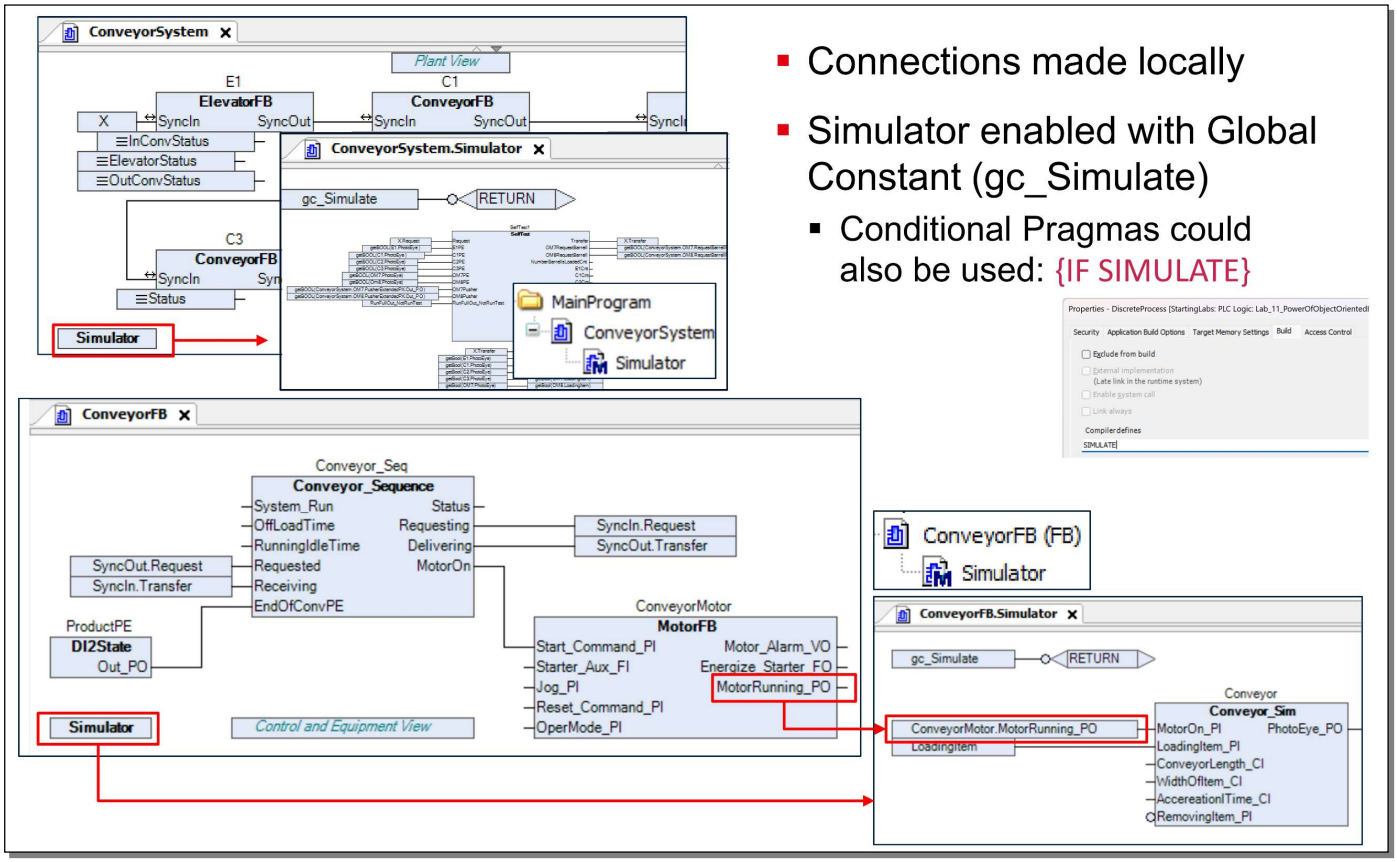


Here is the technique on the previous graphic applied to the Conveyor System. Notice the amount of reuse, and also notice because the hierarchies match, hundreds of connections between the Plant and the Simulation model can be made with one set of top-level connections.

Be advised that this technique runs into issues with the CODESYS bug described in the previous graphic, and would need the BugFix methods added to work around that bug. Other alternatives would be to make the assignment to UUT in the call to the FBs, instead of the declaration of the FBs, or use FB_INIT.

This example can be found in StartingLabs.project, Lab_11_PowerOfObjectOrientedProgramming, DiscreteProcess application.

Simulation Models Embedded in Control Code



Of course, the entire issue of connecting the simulator to the control code can be avoided by putting the simulation code in the control code. However, this benefit in making connections has drawbacks in the flexibility of the simulation code and the ability to fine-tune the simulation code to best match the actual environment (for instance, since the simulation code will run at the same rate as the control code, it will be difficult to exactly match the group delay to that in the real plant). It is also impossible to guarantee that the idle embedded code will not take memory space on the deployed hardware, nor interfere with the production code.

With embedded simulation, the simulation models are attached directly to the control object it is simulating, and the I/O between the control and simulation model is made with local/permanent connections. The simulator model needs to be called at the end of the control module and when not needed the simulation code is disabled with a global variable as shown in the graphic. If the design is implemented in ST, conditional pragmas can be used to exclude the simulation calls and code from the build (see the Miscellaneous chapter). The conditional pragma could be defined in the Application's Properties (Build tab), or a command-line argument when CODESYS is invoked.

This technique is demonstrated in StartingLabs.project, Lab_11_PowerOfObjectOrientedProgramming, DiscreteProcessEmbeddedSimModels application.

Another alternative for simulation I/O mapping is to use Object-Oriented I/O (OOIO) to configure the simulator I/O in CSV files where it could be possible to use the same CSV files to configure the actual I/O as the simulator I/O. OOIO is explained in more detail in the I/O chapter.

Other Simulation Options

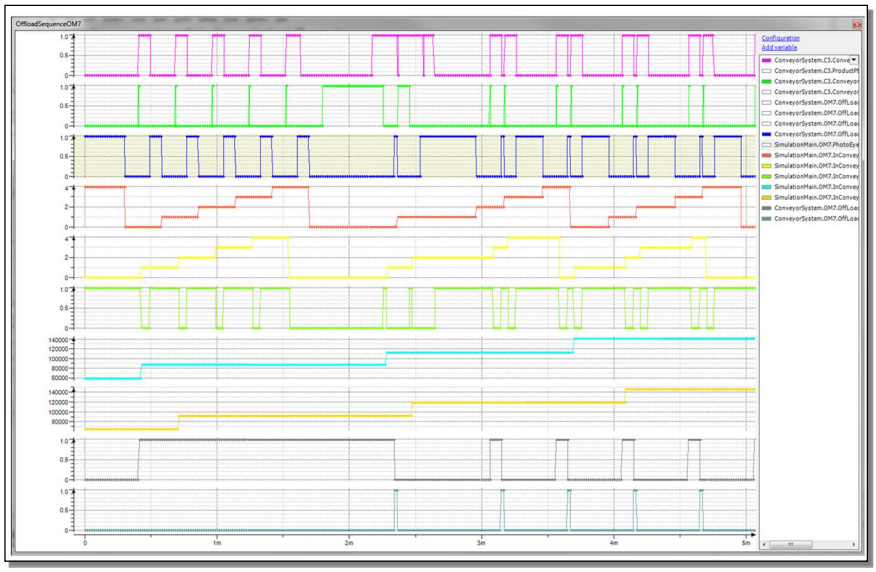
- CODESYS SysShm (System Shared Memory) Library
  - Use shared memory to provide communications between Control and Simulation running on the same hardware
- OPC-UA Symbol Editor
  - Expose the Control I/O points to OPC-UA and connect to a simulator which has an OPC-UA client
- Future CODESYS Virtual Commissioning interface (expected 2025)
  - Provides standard interface to simulators like PLCSIM© or Emulate3D©
- Future CODESYS Simulation Interface (expected 2025)
  - Uses OPC-UA pub/sub and Information Model to provide interface to free-standing simulators

In addition to the simulation options shown earlier, the CODESYS SysShm Shared Memory library can also be used to connect the control I/O points to the simulator I/O points. To use Shared Memory, both applications need to be running on the same hardware and the OS needs to support Shared Memory. Code would need to be added to map the I/O to the shared memory block on both sides. This technique is used in the OOIO demo in the I/O chapter.

Another alternative is to expose all the I/O points on the Control side to OPC UA, and use a simulator with an OPC UA Client to connect to these points. Object-Oriented OPC UA as described in the Peer to Peer chapter would be helpful with this alternative.

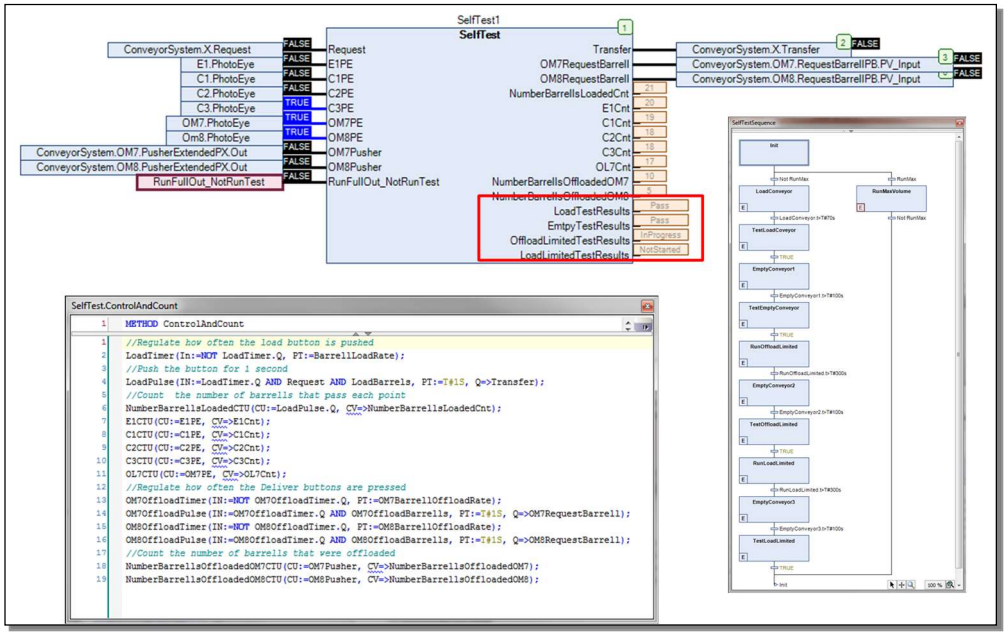
CODESYS is also working on a Virtual Commissioning interface that will provide a link between CODESYS control and 3rd party simulator such as PLCSIM© by Siemens© or Emulate3D© by Rockwell©. They are also working on a Simulation Interface based on the OPC UA Information Model and Pub/Sub that will provide a standard interface between CODESYS Control and other free-standing simulators (such as another CODESYS Application running plant simulation code). This interface will minimize the changes to the project required to switch between simulation and actual plant/equipment I/O.

Simulation Results



A Trace can be used to monitor the results of the simulation. And/or the simulation models themselves can monitor for the expected behavior in a fully automated test system. In this case, the traces show the handshaking between each piece of equipment, the outputs of the proximity sensors, the inputs to the motors, and the status of each conveyor simulator.

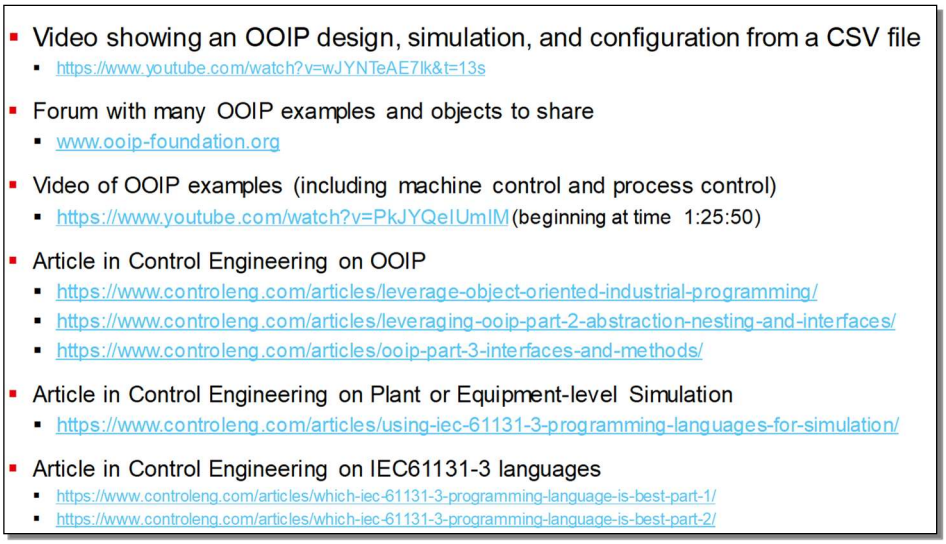
Fully Automated Test



The simulation can be extended to include a full self-test of all the corner-cases and unusual conditions of the control code. This is extremely helpful for future developers to test their modifications and verify that none of the existing operations were disturbed by the modifications.

In this example, when RunFullOut_NotRunTest is FALSE, the system enters the self-test mode. In this test, the sequence on the right is executed which consists of multiple pairs of steps which first execute a test operation followed by the test evaluation. As each test pair is completed, the results are outputted to the status signals on the lower-right side of the SelfTest block.

Other OOIP Resources



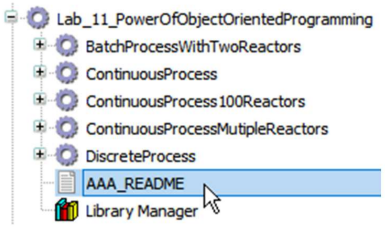
Tool vendors are beginning to make the benefits of OOIP available to Controls Engineers. To leverage those benefits, Controls Engineers need only master three key OOP concepts: Encapsulation, Abstraction, and Composition. With that knowledge, controls engineers can encapsulate the functionality of physical objects into matching control objects, and then instantiate those objects to create a control design which mirrors the plant or machine design. Not only does OOIP make the design easy to build, it makes the design easy to troubleshoot for plant technicians and easy to maintain for future controls engineers. Just as the best of other general software advancements have been adopted into the world of industrial controls, Object-Oriented Industrial Programming is following that same pattern. OOIP is clearly the future of Controls Engineering.

How do you know if your control system supports OOIP? Look for these capabilities:

- A means to create self-contained control objects which correspond to matching plant objects and carry out all the functionality required for that plant object such as alarming, auditing, physical I/O, HMI I/O, scaling, control, etc.
- A graphical editor allowing an unlimited number of instances of objects to be declared, instances of objects to be interconnected in arbitrary fashions, and objects to instantiate other objects into a hierarchy of arbitrary depth and complexity. During runtime, the editor should allow for simple navigation of the hierarchy such as double-clicking on an instance of an object to descend into the project hierarchy and to navigate back.
- The ability to debug individual instances of objects during runtime, including: setting breakpoints within individual instances, single-stepping into individual instances, and viewing/changing the private variables of an instance of an object.
- A means for instances of the same objects to be differentiated by assigning unique values to the instance's configuration inputs anywhere the instance may be in the project hierarchy. Preferably, these configuration values are sourced from a CSV/Excel file, SQL database, or via OPC UA. There must also be a way to search on the values of these configuration variables during runtime (for instance, to search on an ISA tag name configuration).
- The ability to map physical I/O to any variable in any instance anywhere in the project hierarchy (including mapping a physical input point to multiple instances). Composite I/O such as from a fieldbus device must be able to be mapped to individual variables, or to one or more data structure variables anywhere in the project hierarchy. The tool must provide a way to trace the path of a signal from its input, through the logic, and to the outputs it drives (likewise in reverse from the physical output back through the logic to the physical inputs which influence that output).
- The capability to build hierarchical HMI objects which match the hierarchical control objects and the ability to interconnect the two objects (and their potentially thousands of underlying interconnections) via the top-level object's instance name.
- The ability to print a "flattened" version of the hierarchical design showing the interconnections between the object instances and the unique configuration values on each instance.
- The ability to implement Inheritance, Methods, Polymorphism, and Interfaces can be helpful.
- An active user community and lively forum where open-source Plant Objects and advice can be freely shared.

LAB: Object-Oriented Industrial Programming Examples

- Open Lab_xx_PowerOfObjectOrientedProgramming
  - Read the AAA_Readme file



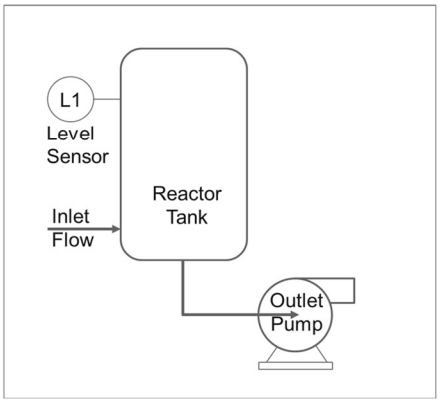
  - Login and run the simulations for the Discrete, Batch, and Continuous process demos (these demos use Recipes which uses File I/O, so they will not function properly in the Simulate mode; make sure you are logged out from any other application)
    - Explore and understand how the designs work
      - Note: Find the top level “Plant” in the Task. Double-click from there to explore the hierarchy. View the Control blocks while running
    - Imagine yourself as a plant technician trouble-shooting the plant using the Plant View and Control & Equipment Views

You can also view this video beginning at 1:44:00: https://www.youtube.com/watch?v=wJYNTeAE7lk (or search YouTube for Bedrock CODESYS IDE Demo).

LAB: Extra Credits

- Go back to the Hello World project you created in topic “01 Intro to CODESYS” and move your simulation line of code into a separate simulator:
  - Change your program to a FB and call it Reactor
    - Declare the "input" variable as an input (VAR_INPUT) and the "output" variable as an output (VAR_OUTPUT)
  - Create a new program called Plant
    - Instantiate your Reactor in it (name it R1)
    - Hint: when you instantiate your original program into your new program, you will need to declare it as a VAR_INPUT to be able to be accessed via a full path name. Or, use the pointer trick shown in this chapter (i.e., getREAL)
  - Create a new program called “Simulator”:
    - move the “Input:=Output” simulation line from your original program to the Simulator program
    - Add the necessary full-path names to your input and output to connect to your plant:
      - Plant.R1.Input := Plant. R1.Output;
    - Add a new task for the simulator program
- Change Simulator to a FB and connect with the technique described earlier in “Reusable Simulation Objects”

- Duplicate the tank-controller demo from this article: https://www.controleng.com/articles/using-iec-61131-3-programming-languages-for-simulation/
  - Go to www.ooip-foundation.org and download the Analog Sensor example from the "Posting Guidelines" post
  - Create a VFD that uses a scaler to convert 0 to 100 (percent) to 4-20mA
  - Create a VFD_Simulator that uses a scaler to convert 4-20mA to 0 to 100
  - Build the control block
    Simulator, and
    Test Bench
  - Test



- Think of a new feature and implement the new feature in any of the Discrete, Batch, or Continuous process demos
- Add a flow sensor, control valve, and PID to control the flow rate of the glycerin. Add simulation models for the sensor and valve
- In the Continuous Demo, run the loop analyzer (see README). Change the tank level loop bandwidth to 2.5Hz and observe the results