Tag Naming Conventions and Data Structures for Industrial PLCs

Gary E. Choquette
Optimized Technical Solutions, LLC
March 24, 2015

Introduction

Industrial programmable logic controllers (PLC[1]) were first introduced in the late 1960's. The initial programming languages were designed to mimic the relay based logic that they were replacing. The first controllers only provided discrete control (on/off) and used the concept of contacts and coils with software library modules that performed the function of drums and timers. The controllers were programmed using ladder logic with coils and contact numbers assigned to represent both real (via input/output module interfaces) and virtual logic states. Programmers soon insisted that the development tools needed to be enhanced to add text descriptions to the coil/contact names and the ability to add comments to the code as it took too much time for the developer to manually track data registers using only the data register name (e.g., C204).

As PLCs evolved, they soon added analog values and the associated math functions (add, subtract, multiply, etc.). Math functions were performed using blocks in the ladder logic. Data registers for numeric values were typically referenced by their memory location (e.g., V1012). As with discrete values, the programmer relied on assigning text descriptions to each memory location and cross reference reports to track where/how the memory locations were used. Processing power and available memory also increased over time.

PLCs evolved to include text based coding. These were commonly used to write special function routines that were more complicated mathematically and were intended for multiple use within the application. An example is a routine used to calculate the compressibility of natural gas. This was typically much more efficient than writing the equivalent code using ladder logic functions.

A significant challenge associated with early PLCs and their associated software development platforms was that each platform was proprietary. The software developed for one platform was not easily portable to other platforms. Standards committees were created to address this and other issues. The first PLC related standards committee (NEMA Programmable Controllers Committee) was initiated in the 1970's. Other standards committees were also created around the world (GRAFCET, DIN, IEC, etc.). By the 1990's, the various efforts of those committees cumulated into the development of IEC 1131-3. In 1996, IEC 1131-3 was renamed to IEC 61131-3. [1]

[1] There are other conventions used for digital logic controllers including programmable automation controllers (PAC). In modern control systems, the line that distinguishes the difference between PLCs and PACs is very wide and very gray. For the purposes of this document, PLC generically refers to any digital logic controller capable of performing the functions and data structures described.

IEC 61131-3 has resolved many of the programming limitations and portability of earlier PLC software development applications. While there are some significant differences the adopters of IEC 61131-3, the core syntax and data structures are largely the same between products making the portability of software between software development environments much easier. In addition, IEC 61131-3 adopted additional programming features that allows for better task management and execution optimization. Improvements were also made to improve encapsulation, code reusability, and data register (aka tags[2]) organization and management. IEC 61131-3 vastly improves the readability, reusability, and portability by adopting some of the concepts from modern program object orientated languages.

Adopting object oriented programming (OOP) concepts has the potential advantages of:

- Improved software development productivity through code reuse. Code reuse is achieved through modularity and extensibility,
- Improved software maintainability,
- Lower development costs, and
- Higher quality software.

For the purposes of this document, the recommended practices of data structures outlined below assume that the software development environment is reasonably compliant with IEC 61131-3.

Discussion

There are numerous aspects of IEC 61131-3 that can be used to enhance code readability, reliability, robustness, and reuse.

Tag Naming Conventions

How variables are named in a PLC can greatly simplify the software development and maintenance. Users have gone from specifying individual memory addresses to having the ability to provide descriptive names. IEC 61131-3 compliant development applications go beyond this to include scoped variables, aliases, and data structures; all of which will be discussed in more detail below.

As a general principal, tag names should contain enough information to help the developer/reader of the code to understand types of information contained therein and how it fits into the larger control architecture. For example, including process and instrumentation diagram (P&ID[3]) codes to the associated tag in a PLC allows for the conceptual tie of an analog value directly to its corresponding physical location in the process. The data associated with a variable that ties to other information will be referred to as metadata in this paper. Other types of metadata include data types (e.g., real, integer, discrete), data flow (e.g., input, output), usage (e.g. input/output, temporary variable, index counter), scope (e.g., local, global), process parameter (e.g., pressure, flow, temperature), and location usage (e.g., engine, compressor, turbocharger).

[2] For the purposes of this document, the term 'tag' refers to any named PLC memory location.
[3] For the purposes of this document, process and instrumentation diagrams and process flow diagrams are treated as the same.

There are many different methods that can be used to include metadata. Some of them have been incorporated into standards such as ANSI/ISA 5.1 Instrumentation Symbols and Identification. [2] The primary adoption of this standard is on P&IDs. This standard includes definitions of device types (e.g., valves, switches, control valves, etc.) with corresponding codes for each type (V, S, CV respectively). This was an attempt to incorporate metadata into the tag name. The standard also defines standard conventions for alarm limits (e.g., the function modifiers H and HH for high limit and high-high limits respectively). Identification letters were also defined based on the type of parameter being measured (e.g., the letter I is used for current measurements, F for flow, J for power, P for pressure, etc.) This standard is still commonly used to define instrument names on P&IDs. The use of the underscore character was used to separate blocks of data. For example, PT_103 would be used for pressure transmitter number 103. In the PLC application, the tag could match that same name but usually it is expanded to provide additional descriptive information (e.g., PT_103_SUC). There may also be other parameters associated with that point such as a low-low alarm (e.g., PT_103_SUC_LL). In most cases, the transmitter number means little to the programmer except that the point can be tied back to a location on a drawing. Rather, the data function (SUC for suction in the example above) has more meaning. Another disadvantage with using specific transmitter identifiers in the code is that it is unlikely that the transmitter number is used consistently for all like facilities; thus tag names have to be rewritten to apply the code to sister facilities. For those situations, a more generic name is desirable (e.g., PT_SUC).

As a general rule, including metadata into the tag name aids in the software development and maintenance process thereby making code development more robust (i.e., fewer bugs), flexible, and easier to maintain. Prior to IEC 61131-3, all of the metadata had to be tied into the tag name. This led to either very long tag names or the use of shorter tag names with the loss of some of the metadata. Usually, tag naming conventions favored shorter names to reduce the time required to type in information and reduce memory requirements.

Ideally, tag names used in a PLC should:

- Identify the source/use of the data,
- Identify the process being assessed (if applicable),
- Identify the location in the process (if applicable), and
- Be sufficiently generic such that the application does not have to be rewritten to reuse the code in other applications.

Tag names should also be sufficiently descriptive such that the developer can logically ascertain the parameter by looking at just the tag name. In the example above, P represents pressure, T represents transmitter, and SUC represents suction. This may not be the best example of a tag name as P could represent other parameters as well (such as power). It is usually redundant or irrelevant that a transmitter is involved so it is recommended that T for transmitter be dropped in tag name usage. Also, it is unclear from a tag name like PT_SUC what is the actual process fluid being measured. To avoid ambiguity while minimizing the overall tag length, it is recommended that the tag name be made up of abbreviations of 3-5 characters. To the extent it is relevant, the tag name should imbed: the process information (gas, lubricating oil, coolant, water, waste water, refined product, etc.), the parameter being measured (pressure, temperature, flow, etc.), and the location relative to the object (inlet, interstage, outlet, etc.). Each section should be separated by an underscore ('_') character. Project specific information should be stored in associated fields within a user defined datatype (more information below). As an example, a better name for PT_SUC would be GAS_PRS_SUC. Note that not all three sections will be required for all tags. For some parameters, a state should be identified in the tag name (e.g., open or closed). The key is to design a system, document it thoroughly, and implement it consistently throughout an application.

As a best practice, the approach outlined in this paper adopts principles from that standard but also takes advantage of some of the abilities and advantages of the data structure options offered as part of IEC 61131-3. Specific features of IEC 61131-3 that aid software development include:

- User defined datatypes,
- Arrays, and
- Datatype nesting.

User Defined Datatypes

User defined datatypes allow multiple parameters to be lumped into a single parameter. As an example, metadata associated with and of interest to an analog input point include:

- Process fluid (gas, oil, coolant…)
- Process parameter (e.g., flow, pressure…)
- Engineering Units (e.g., "Hg, Psig, Psia…)
- P&ID identifier (e.g. PT_103)
- Value
- High-high alarm limit
- High alarm limit
- Low alarm limit
- Low-low alarm limit
- Alarm configuration flags
- Broken wire alarm
- Over range alarm
- Under range alarm
- High-high alarm
- High alarm
- Low alarm
- Low-low alarm

Prior to IEC 61131-3, all of this information would have been listed as individual tags; the only way to link them being duplicating the first part of the tag (e.g, PT_103_PAR, PT_103_EU, PT_103_ID, PT_103_VAL, etc.). Often times, values were hardcoded rather than creating tags due to the overhead of defining tags; this often required compiling and downloading a new program to change hardcoded values.

Under IEC 611311-3, the user is allowed to create custom data types (aka user defined datatypes or UDTs). UDTs allow the encapsulation of related data into a single tag. Using the example, a UDT for an analog point might look like:

```
(*Analog data structure*)

ST_Analog           :       STRUCT
        Process     :       STRING [10];   (*Process fluid description *)
        EU          :       STRING [10];   (*Engineering units*)
        PID_ID      :       STRING [15];   (*P&ID tag*)
        ENG_VAL     :       REAL;          (*Value in engineering units*)
        STD_VAL     :       REAL;          (*Value in standard engineering units*)
        ENG_FAC     :       REAL           (*Conversion factor from standard units to EU*)
        HH_VAL      :       REAL;          (*High-high alarm set-point*)
        H_VAL       :       REAL;          (*High alarm set-point*)
        L_VAL:      :       REAL;          (*High alarm set-point*)
        LL_VAL      :       REAL;          (*High alarm set-point*)
        CFG_ALM     :       INT;           (*Alarm configuration bits. Bits 0-3 HH to LL*)
        ALM_BWIRE   :       BOOL;          (*Broken wire alarm*)
        ALM_OVR     :       BOOL;          (*Over range alarm*)
        ALM_UNDR    :       BOOL;          (*Under range alarm*)
        ALM_HH      :       BOOL           (*High-high alarm*)
        ALM_H       :       BOOL           (*High alarm*)
        ALM_L       :       BOOL           (*Low alarm*)
        ALM_LL      :       BOOL           (*Low-low alarm*)
                            END_STRUCT;
```

With modern PLCs and the associated development software, memory is not typically a constraint. In addition, development software applications often utilize autocomplete typing which simplifies the code development when using the longer tag names created through the use of UDTs. In most cases, the tradeoff to longer tag names is more than compensated by data encapsulation and enhanced readability. The user need only define a single tag of the user defined datatype. For example:

```
SUC_PRES : ST_AnalogInput;
```

would define a parameter that contains all of the components of a ST_AnalogInput dataset. Parameters within the data structure are accessed through syntax where the child parameter is separated from the parent by a period. For example, to set the PID_ID, the following syntax would be used in structured text:

```
SUC_PRES.PID_ID := “PT_103”;
```

The power of data structures is that all of the related information is linked together. For example, the parent variable can be passed to subroutines as a single parameter and any of the related data can be accessed in that routine. Using this example, all of the alarm limits could be set by a reusable subroutine function that is only passes the parent parameter (in this example SUC_PRES).

Arrays

Where multiple versions of a data type are needed, arrays of data can be created. For example, a PLC may have 16 analog inputs on a given card. An array of type ST_AnalogInput could be created for each input on the card:

```
        RK01_SLOT03          :          ST_AnalogInput[16];
```
[4]

One of the downsides of arrays is that most software development platforms do not allow dynamic array allocations. This requires the programmer to allocate the array large enough to accommodate the largest installation. This consumes excess memory for some applications. It also requires a configuration parameter to tell the application how many items in the array are actually used.

The use of arrays is especially beneficial when performing mathematical processes on the data. An example is averaging all of the values within the array like in the following structured text example:[5]

```
        AvgSum := 0.0
        IF NumIOPoints >= 1 AND NumIOPoints >= 10 THEN
                ERR_FLG.9 := 0;  (*Valid number of points for averaging*)
                FOR AryIndex := 0 TO NumIOPoints – 1 DO
                        AvgSum := AvgSum + RK01_SLOT03[AryIndex].STD_VAL;
                END_FOR;
                IOAverage := AvgSum / NumIOPoints;
        ELSE;
                ERR_FLG.9 := 1;  (*Invalid number of points for averaging*)
                IOAverage := 0.0;
        END_IF;
```

Many platforms support multidimensional arrays such as:

```
        INP_ANALOGS          :          ST_AnalogInput[4][16];
```

This would encapsulate all of the analog inputs from four 16 channel cards into a single variable. Note that in doing so, some metadata is lost (the rack/slot information in this case) because we have to use a more generic tag name (i.e., INP_ANALOGS). This issue can be resolved using data nesting.

Datatype Nesting

Datatype nesting is simply using UDTs in UDTs. For example, the UDT used as an example above could potentially be better structured as:

```
(*Analog alarm configuration data structure*)

ST_AnalogInfo :         STRUCT
        Process         :       STRING [10];    (*Process fluid description *)
        EU              :       STRING [10];    (*Engineering units*)
        PID_ID          :       STRING [15];    (*P&ID tag*)
                                END_STRUCT;
```

And

```
(*Analog alarm setpoint configuration data structure*)

ST_AnalogAlmCfg         :       STRUCT
```

[4] Array definition can vary by PLC manufacture. Some arrays are zero based (as in the case of Allen-Bradley) others can define the allowable range of indices (e.g., Array [1..4] as in the case of Phoenix Contact).

[5] The example also demonstrates good programming techniques for trapping to avoid potential faults (in this case, array index out of bounds) and tying the status of the error check to a bit flag.

```
          HH_VAL      :       REAL;         (*High-high alarm set-point*)
          H_VAL       :       REAL;         (*High alarm set-point*)
          L_VAL       :       REAL;         (*High alarm set-point*)
          LL_VAL      :       REAL;         (*High alarm set-point*)
          CFG_ALM     :       INT;          (*Alarm configuration bits. Bits 0-3 HH to LL*)
                              END_STRUCT;
```

And

```
(*Analog alarms data structure*)

ST_AnalogAlm          :       STRUCT
          ALM_BWIRE   :       BOOL;         (*Broken wire alarm*)
          ALM_OVR     :       BOOL;         (*Over range alarm*)
          ALM_UNDR    :       BOOL;         (*Under range alarm*)
          ALM_HH      :       BOOL          (*High-high alarm*)
          ALM_H       :       BOOL          (*High alarm*)
          ALM_L       :       BOOL          (*Low alarm*)
          ALM_LL      :       BOOL          (*Low-low alarm*)
                              END_STRUCT;
```

And

```
 (*Analog data structure*)

ST_Analog             :       STRUCT
          ENG_VAL     :       REAL;                  (*Value in engineering units*)
          STD_VAL     :       REAL;         (*Value in standard engineering units*)
          ENG_FAC     :       REAL          (*Conversion factor from standard units to EU*)
          INFO        :       ST_AnalogInfo;         (*Associated information*)
          CFG_ALM     :       ST_AInputAlmCfg;       (*Alarm configuration parameters*)
          ALM         :       AnalogAlm;             (*Alarm status*)
                              END_STRUCT;
```

This allows the little used information (e.g., alarm configuration) to be less visible in the data structure. It also allows for the reuse of the alarm configuration data structure in different data structures (e.g., a data structure for set points). Data nesting allows for pseudo object creation, class inheritance, and the reuse of code and display interfaces. This reuse not only reduces development time but it also tends to creates more robust code both in term of reliability and functionality because more time can be invested in the code development and refinement. The investment value is returned as the code is applied in subsequent projects with very little development and testing time needed. It also provides implicit metadata (e.g, all items below GAS_PRS_SUC.CFG_ALM are configuration parameters associated with alarms and all items below GAS_PRS_SUC.ALM are the actual alarm status). The data is filtered by category using this type of data structure.

This programming construct is typically underutilized. It is one of the more powerful concepts of IEC 61131-3 as will be demonstrated later.

Bit Flags

Bit flags are a way to store multiple bits of Boolean information into a single variable. Individual bits of an integer (or double integer) variable are used to represent a state. In the ST_AnalogAlmCfg example, CFG_FLGS is intended to be a bit flag indicating which of the four alarms will be configured. There are two ways to approach configuring bit flags. In many development tools, individual bits can be set. For example:[6]

```
SUC_PRES.CFG_ALM.CFG_FLGS.0 := 1;
```

would set the '0' bit to true which would be defined as enabling the high-high alarm check. If that were the only alarm to be enabled, SUC_PRES.CFG_ALM.CFG_FLGS would have a value of 1. The second approach is to directly set the variable to the desired value. For example, if only the high-high and low-low (bit 3) alarms are to be enabled, the variable can be set to the numerical equivalent of those bits (2 ^ 3 + 2^0 = 9):

```
SUC_PRES.CFG_ALM.CFG_FLGS := 9;
```

The power of bit flags is in the ability to efficiently store bitwise configuration information and perform Boolean logic on a group of Booleans. For example, a bit flag could be used to track error calculation checks in a subroutine where bits 0 through 5 are for critical errors; where each bit representing a different specific logic check (e.g., the numerator is zero in a divide calculation so an error flag bit is set and the calculated value is set to a default value) and the other bits are for non-critical errors. Using this example, checking for a critical error can be performed by:

```
IF (Err_Flag AND 63) > 0 THEN….
```
[7]

Other logical operators can also be used including NOT, OR, and XOR allowing a wide variety of logic combination checks.

Tag Scoping

Another advantage of IEC 61131-3 is that tags can be created at the PLC level (i.e., global), within individual task groups, or within add-on instructions (discussed in more detail below). This is advantageous in that it can declutter the global memory locations such that only parameters that are needed at a global code access level are stored there. This can prevent errors with commonly used names. As an example, creating an integer variable named Index in the global tag area can be dangerous as there might be more than one routine in the application that uses Index as a tag name to loop through arrays. If the value for Index is changed in one routine, it may lead to unexpected results in another routine that also uses that tag. Rather, it is better if each task group has its own definition of Index defined locally which avoids this issue.

[6] The methods shown here are those used by Allen-Bradley. This may be different for other development software systems.

[7] The value 63 being the decimal equivalent of bits 0 through 5 set in binary (00111111). Note that other values could be used. For example if bits 1, 3, 4, and 5 are only of interest, the associated binary value would be 00111010 which is 58 decimal.

Using Tags in Human-Machine Interfaces

An advantage of using data structures is that it can simplify the configuration of human-machine interfaces (HMI). For example, a faceplate (template) can be created to configure alarms and a single UDT (or a child branch of a UDT that is also a UDT) can be passed to the faceplate. The faceplate knows, by the datatype of the UDT, the expected sub parameters that are available and can therefore be pre-linked to input/output fields on the faceplate. This saves a significant amount of time in mapping tags in the HMI, especially in the case where there are multiple uses for the same function (e.g., configuring alarms for each analog input).

Add-on Instructions

While add-on instructions[8] (AOIs) are not specifically related to naming conventions or data structures, a brief discussion is warranted here to outline how they help support pseudo object oriented programming in a PLC. AOIs can be thought of as a standalone subroutine. An AOI could be written for a data structure and collectively they can perform many of the functions of a software object. For example, an AOI can be written to perform the calculations necessary for a reciprocating compressor and another to perform the calculations for a centrifugal compressor. Each AOI handles the unique calculations specific to its type of compressor.

AOIs are by nature designed to be reusable components. If properly designed, updates to an instance of an AOI can be implemented relatively easily.

Passing Tags to Subroutines

When calling an AOI, data can be passed as individual parameters, a branch of a UDT, or as an entire UDT. Note any branch of a UDT can be passed, either standard variables (i.e., REAL. DINT, etc.) or nested UDTs (e.g., CFG_ALM). The ability to pass any branch of a UDT is a very powerful tool which allows the creation of pseudo objects in IEC 61311-3 compatible development environments.

Pseudo Objects

The concept of software objects is a mainstay of modern programming languages for desktop and web applications. Object oriented programming is well suited for code reuse and data/metadata encapsulation. The basic principles of object oriented programming are structuring data structures and associated code around a physical object. Software objects contain:

- Properties – constructs that describe the physical and virtual attributes of an object (length, width, height, color, etc.)
- Methods – calculations that are performed that are unique to object. For example, for a 'cube' object, a method could be used to calculate volume knowing the objects properties of length, width, and height.
- Events – triggers that are initiated within the object and can be used to trigger actions outside of the object. Alarms are a type of event in this context.

Other important aspects of object oriented programming are:

[8] Allen-Bradley uses the term Add-on Instruction but this is not a standardized term under IEC 61131-3 and the term used by other PLC providers can vary.

- Inheritance – the ability of an object to adopt the properties of another object and add additional (or override existing) properties, methods, and/or events. For example, we can have a ‘Shape’ object that includes properties of Location, Orientation, Color, and a Volume method. A ‘Cube’ object can inherent all of the features of the ‘Shape’ object and add new properties for Length, Width, and Height. It would have to overwrite the method to calculate the shapes volume as the methods to calculate volume for a cube is unique to that type of shape. A ‘Sphere’ object could also be inherited from the ‘Shape’ object but would have to add a property of Diameter and would also have to overwrite the method to calculate volume.
- Polymorphism – is the ability of the program to treat different objects inherited from the same class similarly. In a true object oriented environment, ‘Cube’ and ‘Sphere’ objects can be stored generically as ‘Shapes’ and the Volume can be calculated for each of those objects without knowing specifically if the shape is a cube or a sphere.

IEC 61131-3 does not allow for true object oriented programming. However its constructs do allow implementation of aspects of object oriented programming which will be explained in more detail below.

Engineering Units

As a best practice, all engineering units within an application should be the same. For example, all pressures in psig, all times in seconds. This avoids the potential for calculation errors associated with unit conversions and helps assure code reusability. The HMI can be configured to dynamically convert the displayed parameters into user preferred engineering units. Alternatively, both user values and standard values can be used in a UDT with the developer responsible for performing the unit conversion where the UDT is populated or via an AOI.

Bringing it All Together

As a best practice, data structures should be built to match physical objects. Data within those structures should be grouped by logical usage (e.g., input/output, set-points, configuration information, and calculated data). To the extent that multiple versions of the same data is used, the use of arrays should be considered. The data structure should be written for the largest possible implementation with configuration parameters to match the actual implementation.

As a worked example, consider a compressor unit made up of a reciprocating engine driving a reciprocating gas compressor. In most cases, the engine directly drives the compressor but there are some instances where a gearbox or belt drive system is employed to change the speed between the driver and the compressor. Objects in this system include:

- Engine
    - Turbocharger(s)
    - Aftercooler
    - Wastegate
    - Power cylinders
    - Pistons
    - Connecting rods
    - Crankshaft

    - Oil pump
    - Water pump
    - Flywheel
    - Ignition system
- Gearbox
- Compressor
    - Crankshaft
    - Cylinders
        - Pockets/unloaders
    - Unit valves
        - Suction
        - Discharge
        - Bypass
    - Piping
    - Aftercooler

Some of these objects are not of interest (e.g., pistons, connecting rods) because there is no input/output or control functions associated with those objects. For those objects that are of interest, it is recommended to have the following groups of data:

- CFG – configuration data
- IO – input/output data
- ALM – alarm states
- CTL – control set-points
- CALC – calculated data

Using this as a basis, a valve object would have the following data structure:

```
VALVE
    .   ERR_FLG        :       DINT;          (*Error flag for the object*)
    .   CFG
        .   ALM_FLG        :       DINT;          (*Alarm configuration flags*)
        .   VALV_TYP       :       DINT;          (*Valve type code*)
        .   TIME_OPN_LMT       :       REAL;          (*Maximum open travel time*)
        .   TIME_CLS_LMT       :       REAL;          (*Maximum close travel time*)
        .   FLW_COEF           :       REAL[11][3];   (*Valve flow coefficient lookup table*)
        .   POS_PID            :       PID_CFG;       (*PID configuration and tuning*)
    .   IO
        .   INP
            .   OPN_LMT    :       BOOL;          (*On the open limit*)
            .   CLS_LMT    :       BOOL;          (*On the close limit*)
            .   OPN_PCT    :       ST_Analog;     (*Percent open*)
            .   PRES_UPS   :       ST_Analog;     (*Upstream pressure*)
            .   PRES_DWN   :       ST_Analog;     (*Downstream pressure*)
        .   OUT
```

```
                .   OPN_SOL      :      BOOL;         (*Energize the open solenoid*)
                .   CLS_LMT      :      BOOL;         (*Energize the close solenoid*)
                .   OPN_PCT      :      ST_Analog;    (*
    .   CTL
            .   OPN     :       BOOL;         (*Initiate open*)
            .   CLS     :       BOOL;         (*Initiate close*)
            .   STOP    :       BOOL;         (*Stop the valve at its current position*)
            .   SPT_OPN         REAL;         (*Valve position setpoint*)
    .   ALM
            .   RESET           :       BOOL;         (*Reset all valve alarms*)
            .   OPN_FAIL        :       BOOL;         (*Open time > TIME_OPN_LMT*)
            .   CLS_FAIL        :       BOOL;         (*Close time > TIME_CLS_LMT*)
            .   OPN_SLW         :       BOOL;         (*Open time > 85% of TIME_OPN_LMT*)
            .   CLS_SLW         :       BOOL;         (*Close time > 85% of TIME_CLS_LMT*)
    .   CALC
            .   FLOW_EST        :       REAL;         (*Estimated flow through the valve*)
            .   OPN_TTM         :       REAL;         (*Calculated open travel time*)
            .   CLS_TTM         :       REAL;         (*Calculated close travel time*)
```

Note that this data structure is sufficiently generic to accommodate both discrete valves (solenoid to open/close and analog position valves). The data could also have been structured as two different data structure types eliminating the data that is unnecessary for the specific type (e.g., OPN_PCT and POS_PID are not needed for valves that only operate with solenoids. Having a single data structure is more versatile as all of the code for all types of valves can be contained in a single AOI but does use unnecessary memory. To the extent that multiple data structures are used, it is recommended that common tags between the structures (e.g., the estimated flow through the valve) use the same tag name in both data structures. This can simplify the usage as in the case of displaying information on an HMI; the tag for flow rate is the same regardless if it is a solenoid or an analog valve). More examples will be provided below on the advantage/disadvantage of similar but different data structures.

Using our compressor unit as an example, a skeleton data structure[9] would look like:

```
UNIT    :       ST_RCOMPUNIT
    .   ERR_FLG         :       DINT;         (*Error flag for the object*)
    .   CFG
            .   ALM_FLG         :       DINT;         (*Alarm configuration flags*)
            .   ….other
    .   IO
            .   INP
                    .   AMB_PRES     :      ST_Analog;          (*Ambient pressure *)
                    .   ….other
    .   CTL
            .   SPT_PWR         :       REAL;         (*Power set-point*)
            .   …other
    .   ALM
            .   RESET           :       BOOL;         (*Reset all valve alarms*)
```

[9] To display the full data structure would be too verbose for this paper.

```
        .   …other
.   CALC
        .   OVR_EFF     :       REAL            (*Overall efficiency*)
        .   REL_EFF     :       REAL            (*Relative efficiency*)
        .   …other
.   DRV     :       ST_RENG;            (*Reciprocating engine*)
        .   ERR_FLG     :       DINT;           (*Error flag for the object*)
        .   CFG
                .   …
        .   IO
                .   …
        .   CTL
                .   …
        .   ALM
                .   …
        .   CALC
                .   …
        .   BANK[2]     :       ST_RENGBK           (*Reciprocating engine bank*)
                .   IO
                        •   AIR_TEMP_IN         :       ST_ANALOG;
                        •   AIR_PRES_IN         :       ST_ANALOG;
                .   ALM
                        •   …
                .   CALC
                        •   AIR_FLOW_RATE       :       REAL;
                        •   AIR_MASS_RATE       ;       REAL;
                .   …
                .   TURBO           :       ST_TURBO    (*Turbocharger/blower*)
                        •   IO
                                .   ….
                        •   …
                .   PWR_CYL[10]         :       ST_PWRCYL (*Power cylinder*)
                        •   …
.   GRBX            :       ST_GEARBOX          (*Gearbox*)
.   COMP            :       ST_RCOMP    (*Reciprocating compressor*)
        .   ERR_FLG     :       DINT;           (*Error flag for the object*)
        .   CFG
                .   …
        .   IO
                .   …
        .   CTL
                .   …
        .   ALM
                .   …
        .   CALC
                .   GAS_PWR     :       REAL;           (*Gas power for all services*)
                .   GAS_FLW     :       REAL;           (*Gas flow rate*)
                .   …other
.    SRV[4] :        ST_RCOMPSRV       (*Reciprocating compressor service/stage*)
         .   …
         .   CYL[6]        :        ST_RCOMPCYL       (*Reciprocating cylinder*)
                 •   …
```

With this data structure, an AOI can be written to perform individual control/logic at the object level. An example would be an AOI used to calculate compressor cylinder clearance, volumetric efficiency, adiabatic efficiency, normalized adiabatic efficiency, flow rate, rod load, etc. The AOI calculations are called by looping through the active cylinders for each compression service (or stage). Those results are then summed up at the compressor level via an AOI written for the reciprocating gas compressor. Similar logic is performed for the driver.

The advantage of this modular approach is:

- The data structures logically match physical or virtual objects,
- The needed for an object level AOI should largely be self-contained in the UDT, and
- Different parent blocks can be generated using different child blocks.

To better illustrate the last point, it is common to have an electric motor driving a reciprocating compressor. In that case, the AOIs and UDTs generated for the reciprocating gas compressor and gearbox can be reused without modification. New AOIs and UDTs would be needed for the electric motor and the electric drive reciprocating compressor unit. In this case, it is unrealistic to create a generic unit and driver AOIs and UDTs as the data and calculations associated with the different types of drivers is significantly different.

To the extent that there are common parameters between the different types of units/drivers, the tag names should be duplicated (e.g., UNIT.CALC.REL_EFF) such that all compressor units appear polymorphic at the HMI level.

Conclusion

Spending the energy to develop a comprehensive tag naming convention can save significant time and energy when developing code that will be applied to many like objects. The savings is achieved through code reuse, associated configuration and display templates, and HMI configuration. Even larger benefits can be achieved if a standard tag conventions and data structures are adopted at an industry level.

This document outlines best practices that can be used to develop data structures and the associated code to facilitate reliable, robust, and reusable code for PLC control systems.

Bibliography

[1] *61131-3 Programmable controllers - Part 3: Programming languages*, IEC/ANSI, 2013.

[2] *5.1 Instrumentation Symbols and Identification,* ANSI/ISA, 2009.