Bristol

IEC - 61131 - 3 - The First Universal Process Control Language

Bruce Morris, Presenter

Introduction

IEC-61131-3, from Emerson Process Management, is the first international standard for process control software. By using IEC-61131-3, a programmer can develop a control algorithm for a particular brand of controller, and import that same program to another brand with minimum modifications, primarily to process input/output subsystems.

Description of the Fundamental Concepts of IEC-1131

The basic principle of IEC-61131-3 is that a programmer can develop a control algorithm (referred to as a "Project") using any combination of five control languages; Instruction List, Structured Text, Ladder Diagram, Function Block Diagram, and Sequential Function Chart. The control algorithm can include reusable entities referred to as "program organization units (POUs)" which include Functions, Function Blocks, and Programs. These POUs are reusable within a program and can be stored in user-declared libraries for import into other control programs.

The IEC-61131-3 Standard includes a library of pre-programmed functions and function blocks. Any controller that is IEC compliant supports these as a "firmware" library, that is, the code for these is pre-written into a prom or flash ram on the device.

Additionally, manufacturers can supply libraries of their own functions. Typically, these would also be firmware libraries. An important consideration here is that, if a firmware library is used, the device that receives the program must support that library.

Users can also develop their own libraries, which can include calls to the IEC standard library and any applicable manufacturers' libraries.

All user-declared POUs, regardless of type, can be written in any of the five languages. Under some circumstances, a POU can have a combination of languages. A function block program, for example, can incorporate ladder diagram logic in it.

The general construct of a control algorithm includes the use of "tasks", each of which can have one or more Program POUs. A task can be assigned a cyclic rate, can be event driven, or be triggered by specific system functions, such as startup.

The Five Languages

Instruction List (IL) -
The Assembler-style Language

Instruction List is most popular for relatively simple, yet frequently used, algorithms. Assembler language is relatively tedious to program, but is supposedly faster to execute. The following is the code used to calculate the absolute value of the difference between two variables named IN1 and IN2 (comments are contained between the (* and *) symbols:

```
LD IN1  (* Load IN1 into the calculations register *)

SUB IN2  (* Subtract IN2 from that register, storing the result into that register *)

ABS  (* Perform the Absolute Value function on that register *)

ST  Error_Calc  (* Store the value in the register into the variable called Error_Calc *)
```

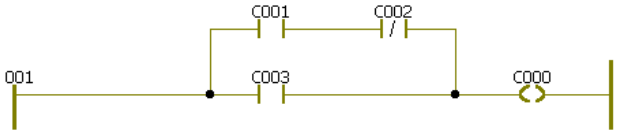
IEC - 61131- 3 - The First Universal Process Control Language

Structured Text (ST) - The High-Level Language

Structured Text is a Pascal-like language that generally allows greater flexibility, and less tedium, in writing control algorithms. It has operators to allow logical branching (IF), multiple branching (CASE), and looping (FOR, WHILE, REPEAT). Typically, a programmer would create his own algorithms as Functions or Function Blocks in Structured Text and use them as callable procedures in any of the five languages. Using Structured text, the code above is written as: **Error_Calc := ABS(IN1-IN2);**

Ladder Diagram (LD) - The Electrical Technicians' Language

Ladder Diagram is probably the most popular language for situations that involve relay logic with AND and OR gates. This allows graphical representation of logic in a form easily understood by electrical technicians and engineers alike. A brief example would be:

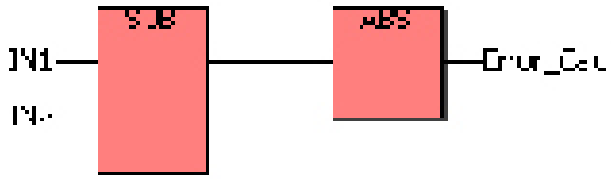


If this algorithm were to be written in Structured Text, it would be:
C000 := (C001 & (NOT C002)) OR C003;

Function Block Diagram (FBD) - The Graphical Language

The Function Block Diagram Language allows control algorithms to be developed graphically by inserting the program units called Functions and Function Blocks into a control program. These blocks can be called from a library of functions specified by the IEC standard, or can be called from manufacturer-supplied or user-created libraries. These function blocks can be written in any of the five languages, including the Function Block Diagram language again. Inputs and outputs between the blocks are wired graphically using a mouse.

The following is the calculation shown in Instruction List and Structured Text above, as a Function Block Diagram:

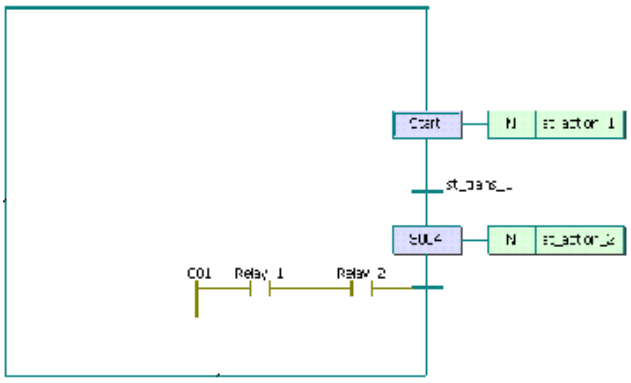


Sequential Function Chart (SFC) - The "Everything" Language

Sequential Function charts allow complex algorithms to be executed using a series of "steps" and "transitions". In the example below, the box called "Start" is a step upon which the program stops until the statement "st_trans_1" is driven

TRUE. This could be done by anything within the program. The box called "st_action_1" represents a calculation, called an "Action Block", which will be continuously executed while on that step. The program then stops on the next step, "S004", until the ladder diagram shown is driven to TRUE. While on that step, the Action Block "st_action_2" will continuously execute.

Complex algorithms can be developed using multiple branching techniques., and several actions can be linked to a step. Also actions can be directed to continue running, run once, or terminate, instead of running continuously.



Program Organization Units (POUs)

Functions

Functions are pre-programmed calculations that accept numerous inputs, but return only one output. The Function must be declared as a variable type, can be created in any of the five languages, and can be used in any of the five languages. It is always referred to by its created name. The standard library of IEC-61131-3 consists mostly of Functions.

A Function, when used, does not consume additional memory. It is simply a procedure call, which uses an existing equation.

The following is an example of the Function shown on the previous page, in a Structured Text equation:

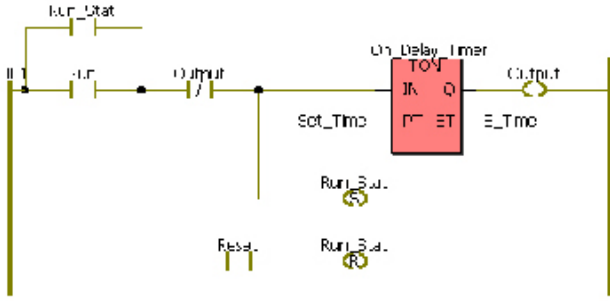
Error :=Error_Calc (IN1 ,IN2);

Function Blocks

Function Blocks are pre-programmed calculations that accept numerous inputs, and can return several outputs. The Function Block can be created in any of the five languages, and can be used in any of the five languages. Any use of a Function Block is referred to as an “instance” of that block. Each instance must be given a name that is unique to the POU in which the block resides. What distinguishes a Function Block from a Function is that each instance contains a unique set of values that are retained with every execution of the instance.

The following Function Block code was created using a combination of a standard IEC Function Block called “TON” (notice it has been given the instance name “On_Delay_Timer”), and Ladder Diagram code. Its purpose is to generate a Boolean pulse (the coil “Output”) at regular intervals when started by a Boolean variable (the relay “Run”) going high, and it can be stopped by another Boolean variable (the relay “Reset”) going high:

If shown as a Function Block, it looks like this:



If shown as Structured Text, it looks like this:

```
Restarting_Timer_1(
Set_Time := Time_Var,
Run := Start,
Reset := Stop);
Pulse := Restarting_Timer_1.Output;
Elapsed_Time := Restarting_Timer_1.E_Time;
```

Programs

Programs are simply POUs created in any of the languages, which can incorporate unique code, or can include any Functions or Function Blocks, created locally to a Project, or referenced from external Libraries. A Program is the only POU type that can be inserted into a running Task.

Tasks

System

Tasks are the devices that execute Program POUs. System tasks are triggered to execute once on specific events within the running program. These include cold and warm program starts, floating point errors, and stack overflows.

Cyclic

Cyclic tasks run at programmer-declared intervals. IEC allows multiple tasks, and these can be assigned priorities from 0 to 31 (0 being highest).

Variable Declarations

Variable Types

All variables used within a project must be declared, either locally to a POU or globally to the project. Regardless of the type of POU or Language used, all variables must be declared.

IEC-61131-3 allows a full range of variable types, including integer (INT,SINT,DINT), logical (BOOL), and floating point (REAL), byte (BYTE, WORD, DWORD), and time period (TIME).

Additionally, user-declared variable types can be created that are “structures”, or combinations of several variable types. Also, arrays can be created that are combinations of structures. An example is:

```
TYPE

    Scada_Record  :     STRUCT
    Time_Stamp    :     DWORD;
    F101_Avg      :     REAL;
    F101_Max      :     REAL;
    F101_Min      :     REAL;
    F102_Avg      :     REAL;
    F102_Max      :     REAL;
    F102_Min      :     REAL;
                        END_STRUCT;
    Scada_Array   :     ARRAY [1..60]
                        OF Scada_Record;

    Totals_Record :     STRUCT
    Time_Stamp    :     DWORD;
    F101_Tot      :     REAL;
    F102_Tot      :     REAL;
                        END_STRUCT;

    Totals_Array  :     ARRAY [1..144]
                        OF Totals_Record;

END_TYPE
```

Local (to POU)

When POUs are created, variables used within them may be declared as “local” to that POU; that is. The variable’s name can be used in other POUs with no conflict within the project. An example:

```
VAR
    Pulse_3_Min : BOOL;
    Temp_Scada_Record :  Scada_Record;
    Scada_Record_Conf :  Scada_Record;
    Scada_Conf_1  : DWORD;
    Scada_Conf_2  : REAL;
    Scada_Conf_3  : REAL;
    Scada_Conf_4  : REAL;
    Scada_Conf_5  : REAL;
    Scada_Conf_6  : REAL;
    Scada_Conf_7  : REAL;

END_VAR
```

Notice the “Scada_Record” variables that are variables of the user-declared type in the previous example.

Input/Output

Input/Output variables are a special case for local variables. When a Function or Function Block is created, it must have input and output terminals. These terminals are the Input/Output variables. An example:

```
VAR_INPUT
    Input : REAL;
    Reset : BOOL := FALSE;
END_VAR
VAR_OUTPUT
    Output_Current : REAL;
    Output_Previous : REAL := 1.0e+30;

END_VAR
```

Notice that Input and Output variables can be assigned initial values. The Input named “Reset” above has been assigned a default value of FALSE, which will be used if the “Reset” terminal is left unwired. The Output named “Output_Previous” has also been given an initial value, which will be used on initial execution of the Function Block.

IEC - 61131- 3 - The First Universal Process Control Language

Global

When variables are to be linked to I/O points, or are to be used in several POU's in a project, they must be declared "Globally". An example:

```
VAR_GLOBAL
    F101_SETPOINT_001      :      REAL;
    F101_SETPOINT_002      :      REAL;
    F101_SETPOINT_003      :      REAL;
    F101_SETPOINT_004      :      REAL;
    F101_SETPOINT_005      :      REAL;
    F101_SETPOINT_006      :      REAL;
    F101_SETPOINT_007      :      REAL;
    F101_SETPOINT_008      :      REAL;
    Trend_Data_1   :  Scada_Array;
    Trend_Data_2   :  Scada_Array;
    Totals_Data    :  Totals_Array;

END_VAR
```

This declaration would occur in the "System Resource" section of the project.

External

Once variables have been declared globally, they can then be used in any POU by being re-declared as "External".

An example:

```
VAR_EXTERNAL
    F101_SETPOINT_001      :      REAL;
    F101_SETPOINT_002      :      REAL;
    F101_SETPOINT_003      :      REAL;
    F101_SETPOINT_004      :      REAL;
    F101_SETPOINT_005      :      REAL;
    F101_SETPOINT_006      :      REAL;
    F101_SETPOINT_007      :      REAL;
    F101_SETPOINT_008      :      REAL;
    Trend_Data_1   :  Scada_Array;
    Trend_Data_2   :  Scada_Array;
    Totals_Data    :  Totals_Array;

END_VAR
```

Bringing It All Together

IEC-61131-3 is a powerful, flexible, and adaptable standard that includes something for every programming taste and style. Users can create programs in any combination of the five languages, and can develop code as POUs that are re-usable within a project, and can be stored as libraries for use in other projects. The manufacturers who support this standard are free to use their own on-line interface software, and future developments will allow more powerful communications capabilities between controllers of different manufacturers.

This declaration would occur in the "System Resource" section of the project.

Emerson Process Management
Remote Automation Solutions

Watertown, CT 06795 USA T 1 (860) 945-2200
Mississauga, ON 06795 Canada T 1 (905) 362-0880
Worcester WR3 8YB UK T 44 (1) 905-856950

Website: www.EmersonProcess.com/Remote

EMERSON™
Process Management