PLC Programming Guidelines

Dan Kandray

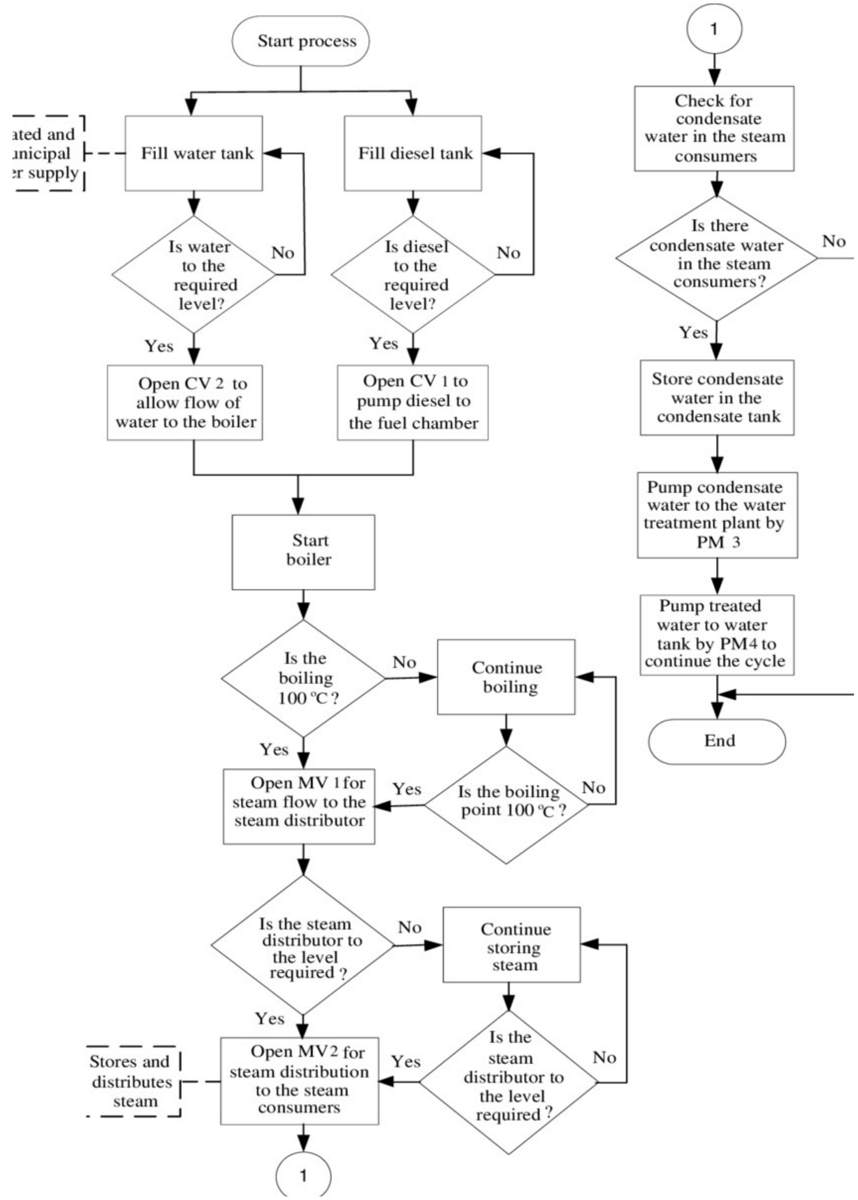
January 30, 2020

PLC Programming Guidelines

I. Develop and Document the Project Plan
II. Create a State Logic Diagram
III. Organize Project Structure (Code) into Tasks, Programs and Routines
IV. Develop Code for Reuse
V. Standardize Naming Conventions
VI. Develop Routine Logic Using State Logic Programming Methods
VII. Simulate the ladder logic program of instructions to verify logic continuity
VIII. Download and verify the program on the actual machine, workstation, or system being controlled.

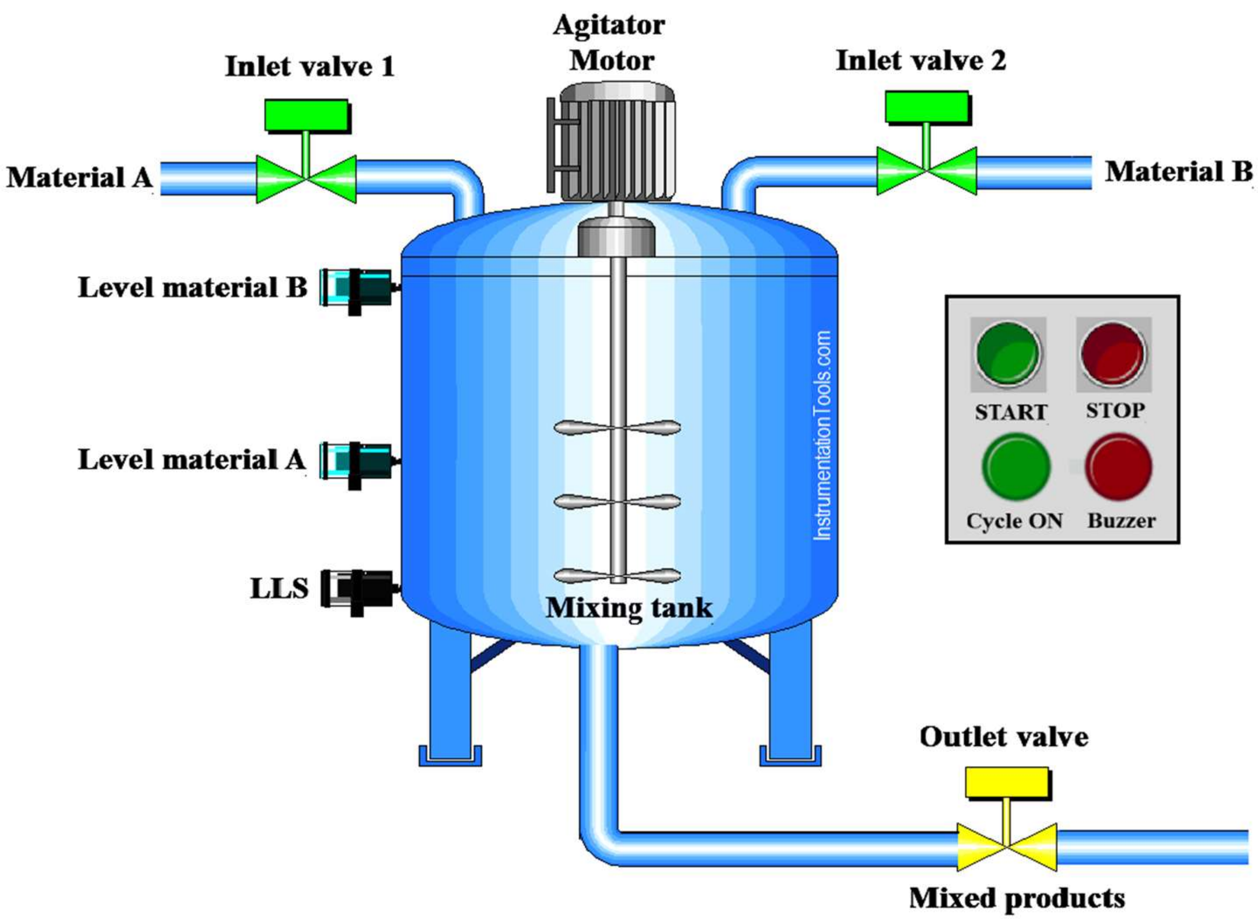
I. Develop and Document the Project Plan

1. From project narrative, create a clear flowchart that details machine functions.
2. Develop a firm understanding of the process
3. Simplify the process as much as possible



I. Develop and Document the Project Plan

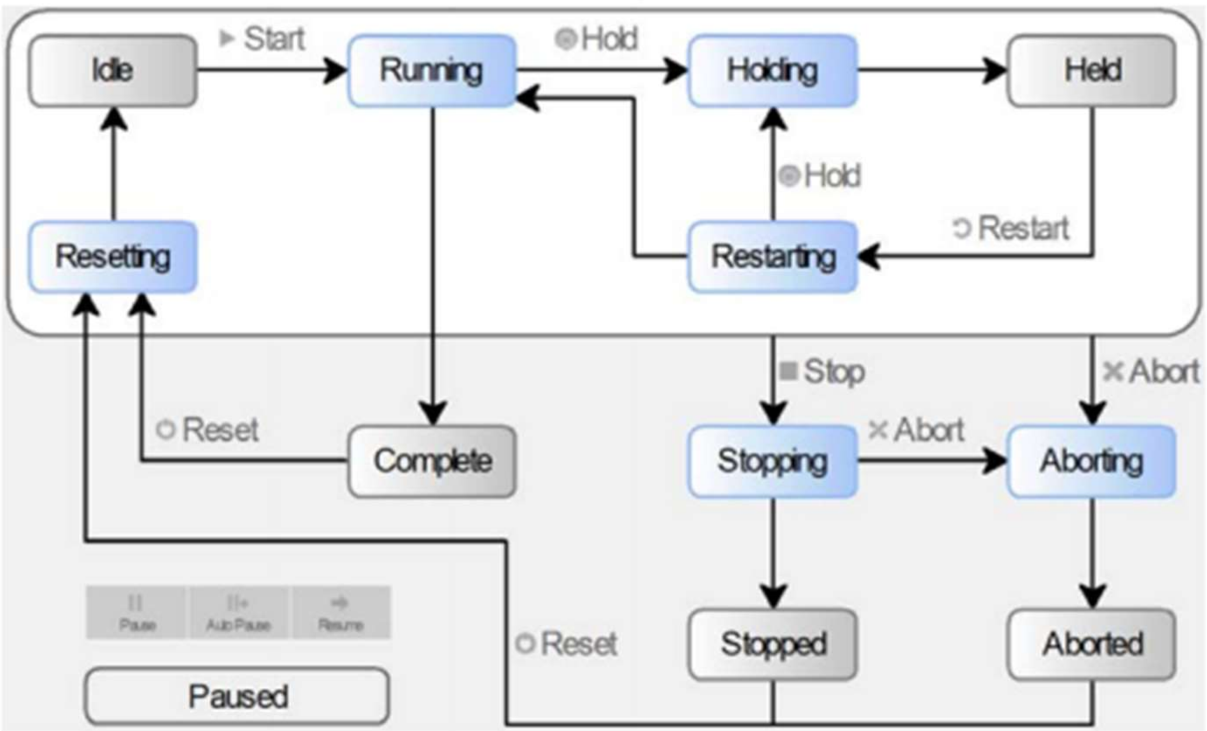
4. List all anticipated physical inputs, outputs, parameters and alarms



II. Create a State Logic Diagram

Swagelok

1. Identify the machine states and document the transitions (state selectors) on a State Logic Diagram
    a) Each state is identifiable by the unique condition of the outputs

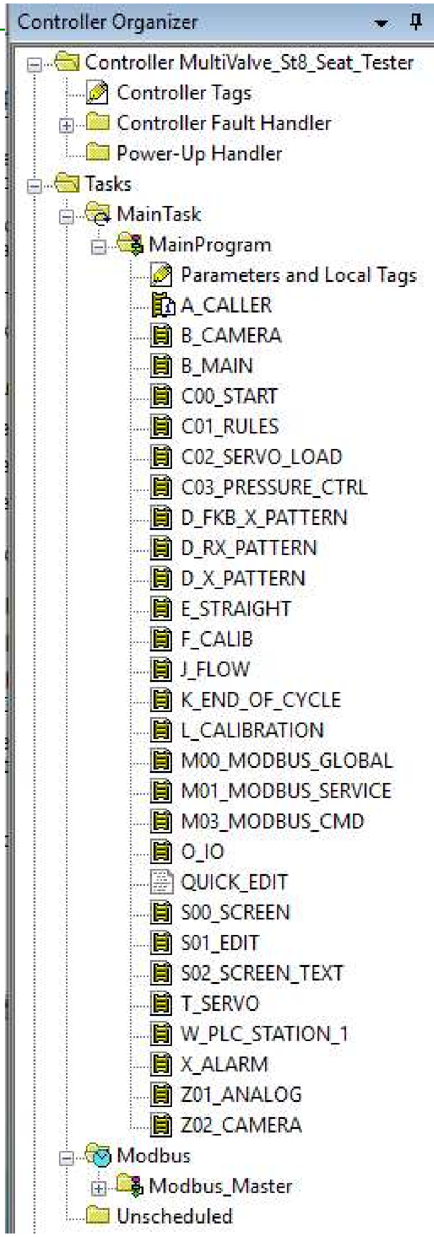


III. Organize Project Structure (Code) into Tasks, Programs and Routines

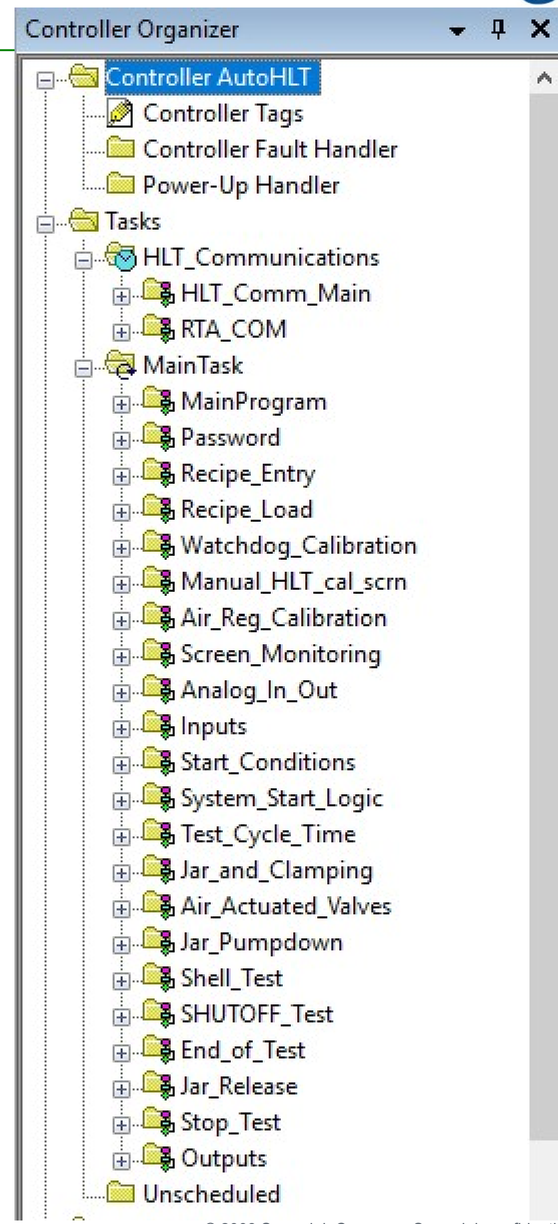
Swagelok

1. Tasks
   a) Most code should reside in a continuous task
   b) Use periodic tasks for slower processes or when time based operations is critical
   c) Use event tasks for operations that require synchronization to a specific event
2. Programs
   a) Separate distinguishable equipment or equipment functions into isolated programs
   b) Control the execution order of the programs from Task (properties) Program Scheduler
   c) Centralize Outputs into one program
   d) Isolate reusable code and/or different programmers
3. Routines
   a) Use ladder logic language in the routine and modularize code into subroutines.
   b) Always place reset conditions in a branch preceding the set condition

III. Organize Project Structure (Code) into Tasks, Programs and Routines



VS

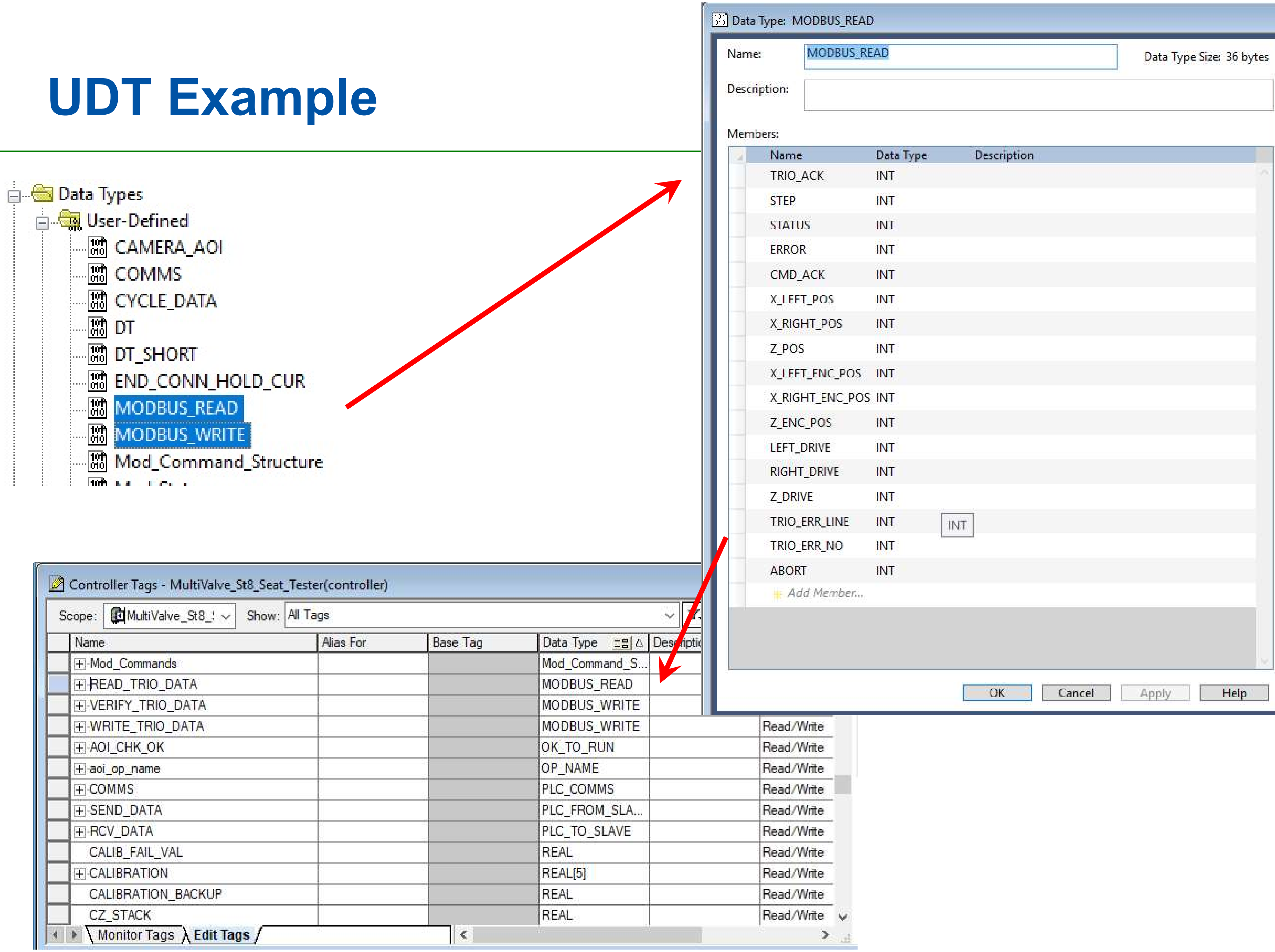


IV. Develop Code for Reuse

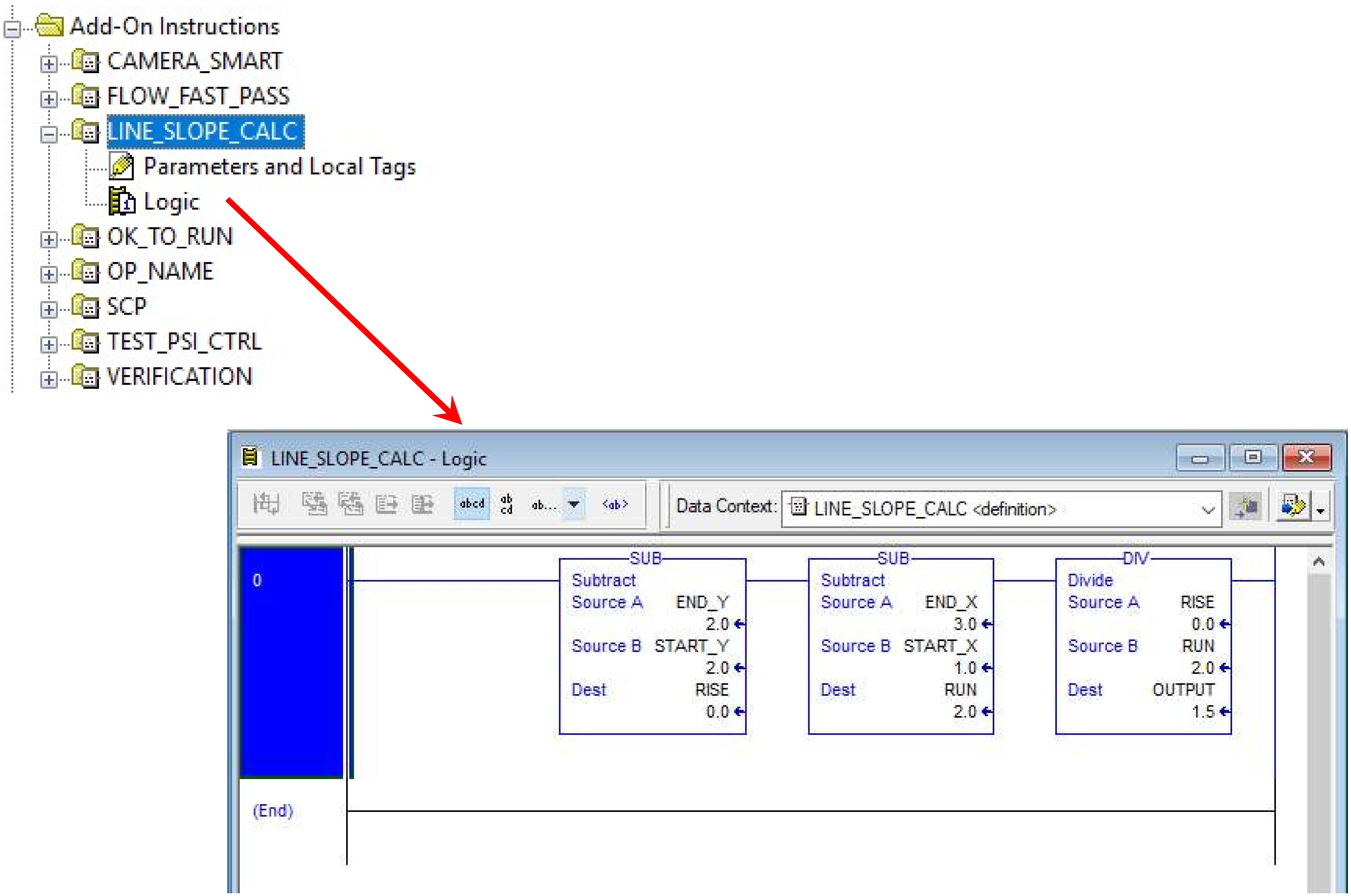
Swagelok

1. Use user-defined data types (UDTs) to group data
   a) A UDT lets you organize or group data logically, so that all of the data associated with a device (such as a pressure transmitter or variable-frequency drive) can be grouped.
   b) The tag names that you assign self-document the structure Programs
2. Use Add-On Instructions to create standardized modules of code for reuse across a project.
   a) Used to encapsulate a specific or focused operation or function
3. Use subroutines to reuse code within a program
   1. Can pass UDTs
   2. Can only be called from with the program they reside

UDT Example


Data Types
User-Defined
CAMERA_AOI
COMMS
CYCLE_DATA
DT
DT_SHORT
END_CONN_HOLD_CUR
MODBUS_READ
MODBUS_WRITE
Mod_Command_Structure
Data Type: MODBUS_READ
Name: MODBUS_READ
Data Type Size: 36 bytes
Description:
Members:
Name | Data Type | Description
TRIO_ACK INT
STEP INT
STATUS INT
ERROR INT
CMD_ACK INT
X_LEFT_POS INT
X_RIGHT_POS INT
Z_POS INT
X_LEFT_ENC_POS INT
X_RIGHT_ENC_POS INT
Z_ENC_POS INT
LEFT_DRIVE INT
RIGHT_DRIVE INT
Z_DRIVE INT
TRIO_ERR_LINE INT
TRIO_ERR_NO INT
ABORT INT
Add Member...
OK
Cancel
Apply
Help
Controller Tags - MultiValve_St8_Seat_Tester(controller)
Scope: MultiValve_St8_!
Show: All Tags
Name | Alias For | Base Tag | Data Type | Description
Mod_Commands Mod_Command_S...
READ_TRIO_DATA MODBUS_READ
VERIFY_TRIO_DATA MODBUS_WRITE
WRITE_TRIO_DATA MODBUS_WRITE Read/Write
AOI_CHK_OK OK_TO_RUN Read/Write
aoi_op_name OP_NAME Read/Write
COMMS PLC_COMMS Read/Write
SEND_DATA PLC_FROM_SLA... Read/Write
RCV_DATA PLC_TO_SLAVE Read/Write
CALIB_FAIL_VAL REAL Read/Write
CALIBRATION REAL[5] Read/Write
CALIBRATION_BACKUP REAL Read/Write
CZ_STACK REAL Read/Write
Monitor Tags
Edit Tags

Add-On Instruction Example



V. Standardize Naming Conventions

1. Controller
   - Area/Unit +Type (Abreviation)
     - Example: Mixing_CPX
       - Note abbreviation is for CompactLogix PLC
2. Controller Project
   - Controller name, the letter C, 1-digit major revision number, underscore, 2-digit minor revision number
     - Example: Mixing_CPX_C2_07.ACD

V. Standardize Naming Conventions

Swagelok

3. I/O Module

- Controller name, underscore, abbreviation of rack location (L=local, R=remote), underscore, the letter S, 2-digit slot number, underscore, abbreviation of function
  - Analog input: AI
  - Analog output: AO
  - Discrete input: DI
  - Discrete output: DO
- Example:

- Mixer123 Controller, Local chassis, Slot 4, Analog Output - Module Name:
  - M123_CPX_L00_S04_AO
- Mixer123 Controller, Remote chassis #2, Slot 5,Discrete Input - Module Name:
  - M123_CPX _R02_S05_DI

V. Standardize Naming Conventions

4. Tags
   - The tag name should be meaningful to future application users
   - Utilize a prefix with the abbreviation of the type of tag
     - Input tag: I_*tag name*
     - Output tag: O_*tag name*
     - Machine State: Sta_*tagname*
     - Parameter: Par_*tagname* (Variables that are received from an external source that can be internal or external to the program)
     - Set point: Set_*tagname (*Variables received from an operator or HMI and are not part of an external source)
     - Value: Val_*tagname* (Designates a value that might not be the primary output of the structure)
     - Report: Rpt_*tagname* (Designates a value that is typically used for reporting.)
     - Examples:

I_GRN_PB	O_GRN_LT
Sta_Idle	Par_TargetFillLevel
Set_TankHILevel	Val_midpoint
Rpt_Tank1Temp	

V. Standardize Naming Conventions

Swagelok

5. UDT

- A UDT lets you organize or group data logically, so that all of the data associated with a device (such as a pressure transmitter or variable-frequency drive) can be grouped.
  - You can mix data types, such as real or floating point values, counters, timers, arrays, Booleans, and other UDTs, within one UDT.
  - You can copy a UDT from one project to another, and even from one Logix controller type to another.
  - A UDT is self-documenting based on the tag names you assign, and provides a logical representation of parts or subsystems.
- Format: UDT_*Function or purpose of the UDT*
- Examples:
  - Inventory tracking tag - UDT_InventoryTracking
  - Clean in place system - UDT_CIP

V. Standardize Naming Conventions

6. Add-On Instructions

   - An Add-On Instruction encapsulates commonly used functions or device controls.
     - It is not intended for use as a high-level hierarchical design tool.
   - Once an Add-On Instruction is defined in a project, it behaves similarly to the built in instructions that are already available in the programming software.
   - The Add-On Instruction appears on the instruction toolbar and in the instruction browser

VI. Develop Routine Logic Using State Logic Programming Methods

- State Programming
  - Ladder logic program is based on the different states or modes of operation of the system being controlled
  - Process of viewing process or machine operation in terms of states as defined by the outputs, and transitions as defined by the inputs.

States of a Process

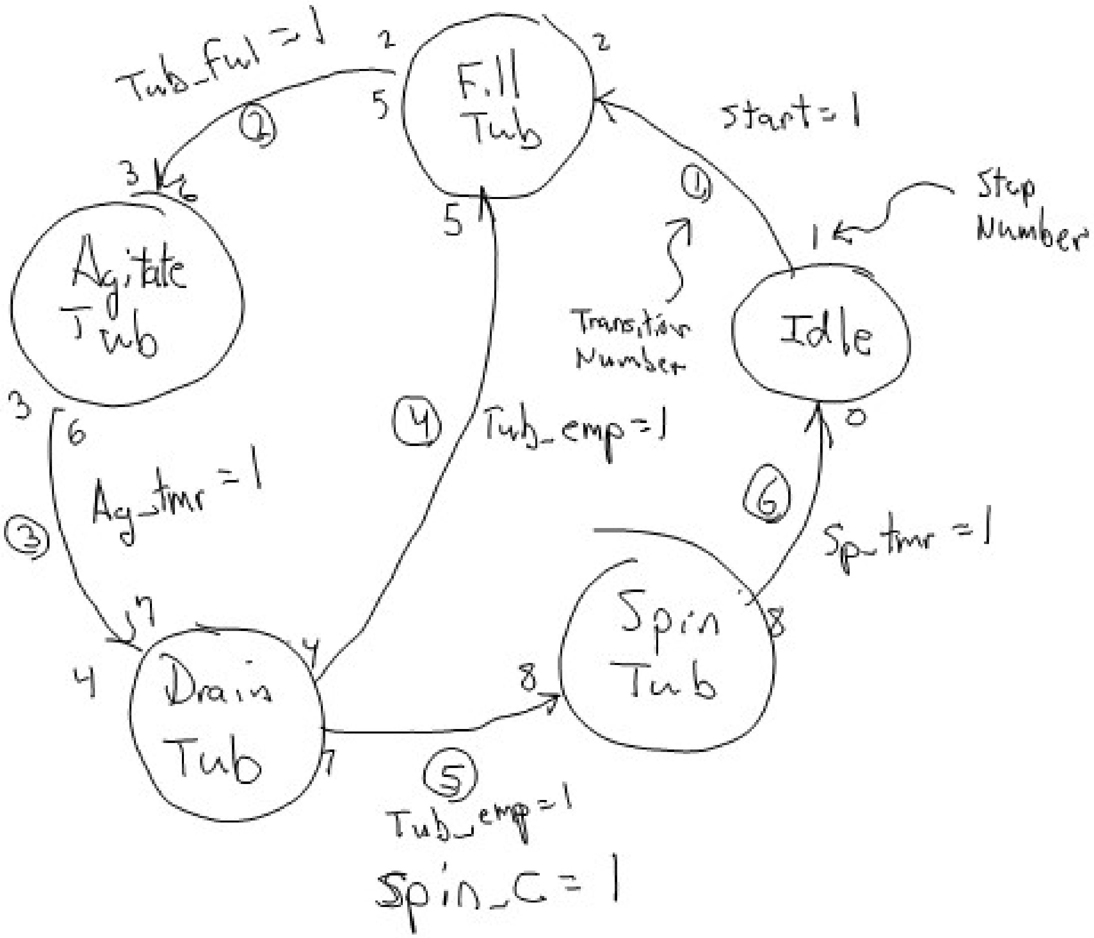
- When a PLC controlled machine is performing its intended function(s), the status of its outputs will define its mode, or *state*, of operation.
- States
  - Modes of operation where the machine is performing an identifiable activity that has to be initiated and then stopped.

State Transitions

- Input status facilitates the *transition* from one state to another.
- States
  - Defined by the outputs
- Transitions
  - Defined by condition (Inputs/Timers/Counters)

State Diagrams

- Graphically displays the various states and corresponding transitions between those states of the machine or system being controlled.



State Logic Programming Steps

1. *Identify and document the system states on a state diagram*
2. *Identify and document the system transitions on a state diagram*
3. *Create the state table*

State Logic Programming Steps

4. *Write the program state ladder logic per the state table*

   a. Define each *active state* as an Examine_ON input condition with a unique “internal” address.

   b. Create a rung for each output.

   c. Use the appropriate state Examine_ON input condition to activate the appropriate output.

      - If more than one state activates an output they are to be ORed together on that rung.

State Logic Programming Steps

5. *Write the program transition ladder logic per the state diagram*

   a. Transition lines into the state bubble of the state diagram are Examine_ON (or same as transition state) input conditions.

   b. Transition lines out of the state bubble of the state diagram are Examine_OFF (or opposite of transition state) input conditions that are ANDed with the lines into the bubble.

   c. Multiple transition lines into the state bubble should be ORed.

   d. Multiple transition lines out of the state bubble (with the same transition Input address) that lead to another state may require a counter or timer to differentiate the logic paths.

State Logic Programming Steps

6. *Add process interrupt logic to the program*
   - *Nested States*
     - *States that can only exist if another (higher state) exists*
       - *Example – Hydraulic cylinders only work if hydraulics are turned on!*

State Table

State	Outputs				
	Red_Lt	Green_Lt	Yellow_Lt	Blue_Lt	Motor