Snowy Technical Standards		
SHL-ELE-143	***Guidelines for Coding PLCs***	
Subject Matter Expert *Terry Foxcroft* *Manager Asset Technology*		Version Date: 9 July 2019 Revision: *E*

1. Executive Summary

Snowy Hydro Limited (SHL) maintains PLC based systems at its power stations and auxiliary sites for monitoring and control of the plant. The PLC based systems are predominantly based on Allen Bradley (A-B) equipment, including ControlLogix, CompactLogix and MicroLogix PLC platforms.

This document describes the guidelines to be followed when designing and developing PLC programs and modifications for SHL sites.

These guidelines are to be followed to ensure that all software developed follows a consistent set of standards with the aim to:

- Ensure consistency
- Remove ambiguity
- Simplify operation
- Simplify maintenance
- Simplify training
- Simplify future modifications
- Increase safety

2. Scope

This document describes in general terms the PLC control hardware and network configuration, structure of PLC applications, and software programming standards.

Guidelines specific to SHL projects are also detailed covering naming and documentation, standard data structures, and standard software modules for PLC systems.

The standards described in this document have been developed based on SHL's main PLC hardware platform – A-B ControlLogix series PLCs. Where other platforms are required to be used (e.g. A-B MicroLogix PLCs), the intent of guidelines are to be followed unless specifically documented otherwise.

Additional detail is included for PLC interfaces to plant HMIs and remote SCADA interfaces, although the HMIs and SCADA interfaces themselves are outside the scope of this document.

Standard logic and configuration for smart electronic overload relays and their use in motor control is also included, along with PLC interface standards.

3. Definitions

Abbreviations and acronyms used in this project are described below:

A-B	Allen-Bradley
BOOTP	Bootstrap Protocol
CIP	Common Industrial Protocol
CLX	ControlLogix
COS	Change Of State
DNP3	Distributed Network Protocol Version 3
GSV	Get System Variable
HART	Highway Addressable Remote Transmitter protocol
HMI	Human Machine Interface
IEC	International Electro-technical Commission
I/O	Input / Output
NUT	Network Update Time
PID	Proportional Integral Derivative controller
PLC	Programmable Logic Controller
RIO	Universal Remote Input / Output Network
RPI	Requested Packet Interval
RTS	Real Time Sampling period
SCADA	Supervisory Control And Data Acquisition
SHL	Snowy Hydro Limited
SSV	Set System Variable
UDT	User Defined (data) Type

A glossary and list of relevant definitions is provided below:

Alarm	Warning to operator of dangerous or potentially dangerous situation, or loss of production. The control system may take remedial action.
Alarm Acknowledgement	Confirmation that an operator has become aware of an alarm.
Backplane	A circuit board with connections or sockets that provides a standardized method of transferring signals between plug-in circuit cards.
Consumed	Receiving Data (Consuming) from another controller
Device	An operating element such as a relay, contactor, circuit breaker, or switch used to perform a given function in the operation of electrical equipment.
Interlock	A condition that prevents the device or equipment from being in an energised state.
Local Control	Control from pushbuttons and switches that are located at the equipment or within sight of the equipment. For a power station, the controls that are typically located on the unit control panel or turbine gauge panel
Logic	Predetermined sequence of operating of control devices.
Manual Control	Control in which the system or main device, whether direct or power-aided in operation, is directly controller by an operator.
Permissive	A condition that initially prevents a device from actuating. However once actuated, the permissive is no longer enforced until the device is de-energised again.
Produced	Transmitting Data (Producing) to other controller(s).

Sequential Control	A mode of control in which control actions are executed consecutively.
User Interface	A functional system used specifically to interface the computer-based control system to the operator, maintenance personnel, engineer, etc.

4. Technical Requirements

4.1. General Design Considerations

The reference manual "Logix 5000 Controllers Design Considerations" provides guidelines for optimising system design. It includes:

- Guidelines that should be followed
- Programming practices that can improve system performance
- Issues to consider when making design decisions
- System information that can affect system performance

The recommendations in this manual should be followed. Any deviation from these recommendations should be noted within this document, together with the basis for the deviation.

4.2. Hardware

4.2.1. Equipment Specification

SHL maintain a list of preferred ControlLogix PLC hardware in order to increase standardisation, minimise spares holdings and reduce training and maintenance requirements.

When specifying PLC hardware for new SHL PLC applications, hardware provided should be selected from SHL's preferred suppliers list.

SHL's current preferred suppliers list is included in SHL-ELE-156 General Low Voltage Electrical Requirements, Annexure Q.

4.2.2. Spares Requirements

New processors must have a maximum of 50% of the available memory used calculated based on installed capacity at time of final approval by SHL.

30% memory should remain spare for future uses. If modifications to an existing processor result in spare memory capacity falling below 30%, SHL should be informed.

New processors and communications modules must have a minimum of 30% spare communications connections on installed capacity. The 30% value is to be based on the installed capacity of the final project approval by SHL.

If modifications to an existing processor or communications module result in spare connections falling below 30%, SHL should be informed.

Refer to project specification requirements, and general low voltage electrical requirements SHL-ELE-156, for requirements for the provision of spares for I/O capacity and cabinet space.

4.2.3. Installation

All physical design and installation is to be performed in line with the manufacturer's recommendations, and with SHL's general low voltage electrical requirements (SHL-ELE-156, and annexures).

4.2.4. Chassis Layout

SHL arranges the layout of ControlLogix chassis in order to allow standardisation of software configuration, improve maintainability, and allow for future expansion.

When arranging a new chassis or modifying an existing chassis for a SHL PLC application, the following guidelines should be considered when designing the location of modules to be placed in racks.

PLC modules are to be arranged in groups, in the following order:

- Controllers
- Communications modules (EtherNet/IP first, followed by any others)
- Digital input modules
- Analogue input modules
- Digital output modules
- Analogue output modules

Where spare slots are required to meet project spares requirements for future expansion, these spares are to allocated at the end of each of the above groups so that the above module grouping can be maintained when adding modules in future.

Examples are provided in the following sections:

4.2.4.1. Main Unit PLC Chassis with Governor PLC

SHL's main unit control PLC rack typically includes both a unit control PLC and governor PLC in the same rack.

The Main Chassis is a seventeen slot. An example layout is provided below.

Unit PLC and communications modules are included from slot 0.

I/O modules for the Unit PLC are included in secondary I/O chassis.

Governor PLC and I/O modules are included from slot 11.

Example of Main Unit PLC Chassis layout is as follows (refer Figure 3.1):

- Slot 0: Unit PLC Processor
- Slot 1: Spare
- Slot 2: Unit PLC EtherNet/IP Communications Module
- Slot 3: Unit PLC EtherNet/IP Communications Module
- Slot 4: Unit PLC DNP Communications Module
- Slot 5: Unit PLC DNP Communications Module
- Slot 6: Spare
- Slot 7: Spare
- Slot 8: Spare
- Slot 9: Spare
- Slot 10: Spare
- Slot 11: Governor PLC Processor
- Slot 12: Governor PLC Digital Input Module
- Slot 13: Governor PLC Configurable Flow Meter Module
- Slot 14: Governor PLC Analogue Input Module
- Slot 15: Governor PLC Digital Output Module
- Slot 16: Governor PLC Analogue Output Module

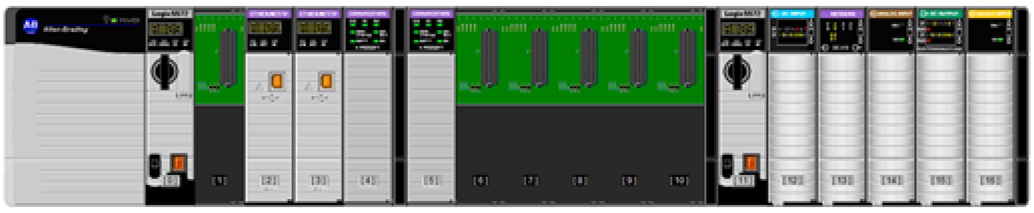


Figure 3.1 - Main Chassis Layout

4.2.4.2. I/O or Secondary Chassis Layout

The I/O or Secondary Chassis is at least a ten slot. An example layout is provided below.

EtherNet/IP module placed in slot 0 to provide link to the main chassis.

Additional communications modules to be installed immediately after EtherNet/IP module, if required.

Digital input modules to be installed after communications modules, followed by spare slot(s) for extension.

Analog input modules to follow on from digital input modules, followed by spare slot(s) for extension.

Digital output modules to follow on from analog input modules.

Analog output modules to be located after the digital output modules.

Example of I/O or secondary chassis layout is as follows (refer Figure 3.2):

- Slot 0: EtherNet/IP Communications Module
- Slot 1: Digital Input Module
- Slot 2: Digital Input Module
- Slot 3: Digital Input Module
- Slot 4: Spare
- Slot 5: Analog Input Module
- Slot 6: Spare
- Slot 7: Digital Output Module
- Slot 8: Analog Output Module
- Slot 9: Spare

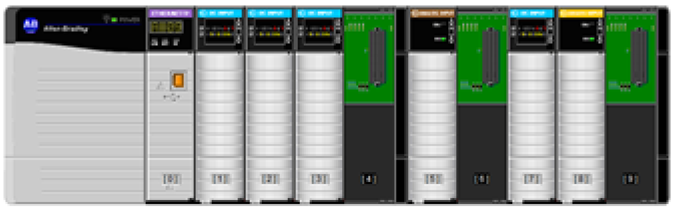


Figure 3.2 - I/O or Secondary Chassis Layout

4.3. Network Design and Communications

4.3.1. EtherNet/IP Networks

EtherNet/IP shall be used for the unit control, device and instrumentation network, and station common network.

The following systems should be installed on the unit control, device and instrumentation network:

- Unit controller
- Unit controller remote I/O chassis
- Remote I/O (Point-IO)
- Governor controller (if in separate chassis to unit controller)
- Excitation controller
- Unit HMIs – operating and turbine floor
- Power monitor
- Electronic overload relays (E300)
- Temperature and condition monitoring monitoring modules (e.g. Dynamix 1444)
- Stack (Tower) lights
- Other sensors (proximity, speed, flow, pressure, etc.)
- Mobile or temporary HMI systems (Operator Interface)

The following systems should be installed on the station common network:

- Main transformer controller
- Station controller
- Switching station controller
- Guard gate controller
- RTUs and data concentrators

Fixed, unique IP addresses shall be assigned for each Ethernet connected device. IP address formatting will be advised by SHL's Asset Technology or SCADA Network Administrator.

BOOTP and DHCP configuration shall be disabled.

4.3.2. DNP3 Networks

DNP3 protocol should be used for the following purposes:

- Providing interfaces between unit/station controllers and SHL's remote SCADA system (using either native protocol support, in-rack DNP modules, or standalone RTUs/Data Concentrators)
- Providing interfaces to between controllers and devices with a timestamped DNP3 interface, such as protection relays
- Providing interfaces to external third parties

DNP3 over Ethernet is required wherever possible. Notable exceptions to this are third party connections where the other party requires a Serial connection.

Timestamp data applied at the source should be preserved when mapping DNP3 event data to SCADA interfaces.

Fixed, unique IP addresses shall be assigned for each Ethernet connected device. Unique DNP3 addresses shall be assigned for each DNP3 connected device. IP and DNP3 address formatting will be advised by SHL's SCADA Network Administrator.

4.3.3. Produced/Consumed Data

ControlLogix and CompactLogix controllers support the ability to produce (broadcast) and consume (receive) system-shared tags.

For two controllers to share produced or consumed tags, both controllers must be attached to the same control network (such as an EtherNet/IP network). Produced and consumed tags cannot be configured over two bridged networks.

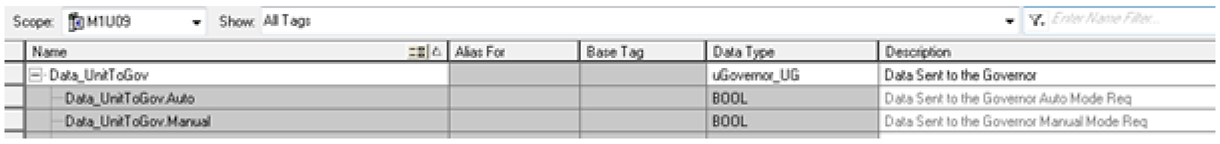
ControlLogix and CompactLogix controllers can produce and consume tags over EtherNet/IP networks.

4.3.3.1. Creating Produced/Consumed Tags

Tags created as produced or consumed should be configured as a controller scoped UDT instance containing all data required to be read by or read from the remote controller.

The UDT created for the data transfer should be identical in each controller's project file.

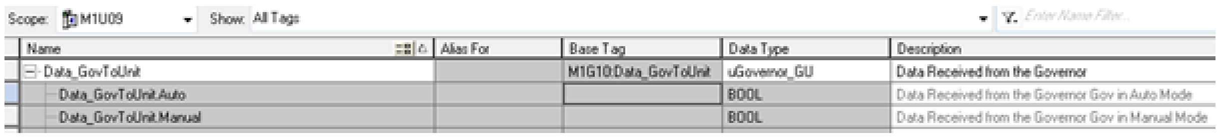
Example: For a produced tag, create a UDT called **uGovernor_UG**, and then a controller tag instance of the UDT called **Data_UnitToGov**


Scope: M1U09 Show: All Tags

Name	Alias For	Base Tag	Data Type	Description
Data_UnitToGov			uGovernor_UG	Data Sent to the Governor
Data_UnitToGov.Auto			BOOL	Data Sent to the Governor Auto Mode Req
Data_UnitToGov.Manual			BOOL	Data Sent to the Governor Manual Mode Req

Figure 4.1 - Example of a Produced UDT tag

Example: For a consumed tag, create a UDT called **uGovernor_GU**, and then a controller tag instance of the UDT called **Data_GovToUnit**



Scope: M1U09 Show: All Tags

Name	Alias For	Base Tag	Data Type	Description
Data_GovToUnit		M1G10:Data_GovToUnit	uGovernor_GU	Data Received from the Governor
Data_GovToUnit.Auto			BOOL	Data Received from the Governor Gov in Auto Mode
Data_GovToUnit.Manual			BOOL	Data Received from the Governor Gov in Manual Mode

Figure 4.2 - Example of a Consumed UDT tag

The connection between the producer and consumer should be monitored to verify the data is current and not stale.

4.3.3.2. Managing Produced/Consumed Tags

UDTs created for produced and consumed data should follow the same rules and limitations for UDT creation described in section 4.7.2.

To reduce network traffic, the size of produced and consumed tags should be minimised. The use of produced and consumed tags for high-speed, deterministic data such as interlocks should also be minimised.

Ensure the number of consumers configured for a produced tag matches the actual number of controllers that will consume the tag. If the number is set higher than the actual number of controllers, connections may be used up unnecessarily.

If possible, limit the produced data between two controllers to a single UDT packet. This will ensure that only a single controller connection is required. If there are multiple produced and consumed connections between two controllers and one connection fails, all the produced and consumed connections will fail.

4.3.3.3. Produced/Consumed Tag Status

The status of Produced/Consumed tags are to be monitored by the PLC program (for sequence control decisions as well as alarming). If possible, the built-in method of inserting a CONNECTION_STATUS tag into the first slot of the transferred UDT shall be used. If not possible (due to firmware limitations or other reasons) then a GSV function or heartbeat is to be used.

4.3.4. EtherNet/IP Messaging

The ControlLogix and MicroLogix platforms provides a mechanism to establish and manage EtherNet/IP connections to other controllers and devices using a MSG instruction in logic.

All messaging to other PLC's should have a watchdog timer associated with them to make sure a message isn't taking too long to execute. Messages shouldn't take longer than 1 second to complete (.DN). If the watchdog timer times out, the (.TO) bit within the message control word should be set. This will remove the message from the queue and immediately set the error bit (.ER). Waiting for the error bit (.ER) to assert could otherwise take up to 40 seconds. This could cause the message queue to fill and impact the performance of other messages within the PLC.

For ControlLogix systems, inter-controller communications for interlocking and process control should be done using a produced/consumed UDT with spare registers for future use.

4.4. Software Naming Conventions and Commenting

4.4.1. Project File Name

The PLC software project file name should be representative of its primary control function and should include the machine or system name that it is controlling and the date it was last altered. Using the date as part of the file name assists in version control and revision history tracking. Having the date backwards (year, month, day) also ensures a list of project files viewed from Windows Explorer (in alphabetical order) will always list the latest project last.

The following Syntax should be used when naming the PLC project:

Syntax:	***aaaaa_yyyymmdda***	
Where:	***aaaaa*:** Station number, system type, and system number	
	***yyyy*:**	Year
	***mm*:**	Month
	***dd*:**	Day
	***a*:**	Minor rev letter (a to z) for multiple revisions within the day.

Note: Name shall start with an Underscore or Alpha character to meet IEC 1131 standards.

Example: **M1U09_20171115** – Project File for Controller Murray 1, Unit 09. Program Revised on 15/11/2017.

4.4.2. Controller Name

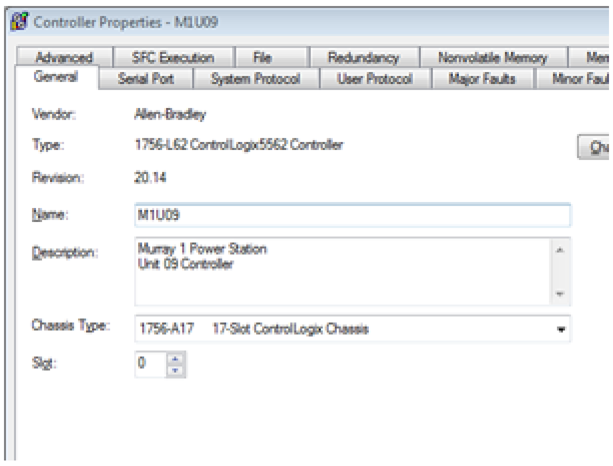
The controller name is not the same as the PLC project name by default, but it is advised that it be made the same but without the date suffix.

The following Syntax should be used when naming the controller:

Syntax:	***aaaaa***
Where:	***aaaaa*:** Station number, system type, and system number

Note: Name shall start with an Alpha or Underscore character to meet IEC 1131 standards.

Refer to section 4.4.5.2 for additional detail on the station number, system type, and system number.


Controller Properties - M1U09

Advanced | SFC Execution | File | Redundancy | Nonvolatile Memory | Men
General | Serial Port | System Protocol | User Protocol | Major Faults | Minor Faul

Vendor: Allen-Bradley
Type: 1756-L62 ControlLogix5562 Controller
Revision: 20.14
Name: M1U09
Description: Murray 1 Power Station
Unit 09 Controller
Chassis Type: 1756-A17 17-Slot ControlLogix Chassis
Slot: 0

Figure 5.1 - Controller Name Example

4.4.3. Communication Modules and Adapters

ControlLogix configuration allows communications modules and adapters to be provided with unique names. Communication modules and adapters should be named in a way so that the path to the module from the controller can be traced. This means specifying the local slot number of the module followed by the network type, the node address, the destination slot number, the location and finally the type of I/O.

The following syntax should be adopted when naming communication modules or adapters:

Syntax: ***Laa_bbcc_ddd***

Where:	**L**	Abbreviation for local slot
	aa:	Local slot number **00-31**
	bb:	Network type:
		DP=DNP 3.0
		EN=Ethernet
		MB=Modbus
		RM=Remote I/O
	cc:	Node or rack address of communications module
		For: Network identifier – for multiple parallel/redundant networks
	ddd*:	Cubicle name where the module is located

* Omitted if the module resides within the processor chassis

Example: **L02_ENA** – For Network ‘A’ EtherNet/IP module in slot 02

4.4.4. I/O Modules

ControlLogix I/O modules should be named similar to the communication and adapter modules.

The following syntax should be used when naming I/O modules:

Syntax:	**L*aa_bbcc_ddd_See_ff***	
Where:	**L**	Abbreviation for local slot
	aa:	Local slot number 00-31
	bb*:	Network type:
		EN = Ethernet
		RM = Remote I/O
	cc*:	Node or rack address of communications module
		For: Network identifier – for multiple parallel/redundant networks
	ddd*:	Cubicle that module is located in
	S	Abbreviation for slot
	ee*:	Slot number that module is located in
	ff:	Type of I/O module or device
		DI = Digital Input Module
		DO = Digital Output Module
		AI = Analogue Input Module
		AO = Analogue Output Module
		PV = PanelView

PM	= Power Monitor
E300	= E300 Electronic Overload Relay
CFM	= Configurable Flow Meter
ETAP	= EtherNet/IP Tap

* Omitted if module in local processor chassis

Example: **L02_ENA_GATE_S05_DI** – For digital input module in slot 5 of Network ‘A’ Gatehouse rack.

L02_EN_88GQa_E300 – For EtherNet/IP networked E300 for plant item ‘88GQa’

4.4.5. ControlLogix Tag Naming and Commenting

ControlLogix programs are created in RSLogix 5000 and use IEC 61131-3 complaint, symbolic data addressing. Tags are created by assigning a name and defining the data type. This provides self-documented logic and eliminates the need to memorise the controller’s memory layout.

Controller tag naming should generally be based upon the name of the equipment found in the P&ID drawings so that there is a one-to-one correspondence between the physical equipment and the controller tags within the program.

4.4.5.1. General Tag Naming Considerations

When naming new tags in a ControlLogix application, use mixed case and underscore characters. Although tags are not case sensitive (upper case A is the same as lower case a), mixed case is easier to read.

Example: **CoolingWater** can be easier to read than **coolingwater**

Underscores should be used in tag names if there is a digit inside the tag name or where a tag name is partly made up of capitals.

Example: **PP_88GQa** is easier to read then **PP88GQa** or **PID_Loops** is easier to read then **PIDLoops**.

RSLogix 5000 software displays tags of the same scope in alphabetical order. To make it easier to monitor related tags, use similar starting characters for tags from related plant areas or equipment.

Example: Consider using **Tank_North** and **Tank_South** rather than **North_Tank** and **South_Tank**

RSLogix 5000 software uses a simple sort to alphabetise tag names in the Tag Editor and Data Monitor. This means if you have Tag1, Tag2, Tag11, and Tag12, the software displays them in order as Tag1, Tag11, Tag12, and then Tag2. To keep them in numerical order, name them using a leading zero Tag01, Tag02, Tag11, and Tag12.

All tags should have a shorter length than the maximum tag length of 36 characters.

4.4.5.2. I/O Base Tag Naming

All I/O to the ControlLogix system should be identifiable by a tag name that is derived from site abbreviations and apparatus identification. Most equipment I/O will be represented and named accordingly on P&ID drawings.

The following syntax should be used when naming tags representing plant equipment. Please note that some of the different coloured sections of the tag name may be omitted depending on its use and location.

Syntax: ***sssstuu_ddd_ppp.._ii***

Where: **[N1] ssss**: The station number (eg T3 or M1)

[N1] t: The system type within the station, defined as;

U	Unit
G	Governor
T	Transformer
C	Station Common System
S	Switching Station

[N1] uu: Is the system number (eg Unit 07 or Transformer T13)

[N2] _: The underscore character

ddd..: Is the Snowy Hydro device number (see section 4.2.1).

As a minimum every tag representing equipment should contain this identifier.

ppp: Is the letter code that identifies the instrument function. (See section 4.2.3)

[N3] ii: Is the I/O type identifier

DI = Digital Input

DO = Digital Output

AI = Analogue Input

AO = Analogue Output

RTD = Resistance Temperature Device Input

[N1] The station number, system type, and system number prefixes are used only for tags produced by other PLC controllers. These are normally excluded so that common tags and code can be used in every similar unit at a station

[N2] Underscore is used to avoid use of numeric character at start of core of tag (not allowed under IEC1131)

[N3] I/O type identifier suffix is only used for tags that are aliased to real I/O addresses.

All process digital inputs such as level, flow, pressure and temperature etc. should be debounced and buffered in the “Digital_Inputs” program. Other inputs may need to be buffered simply to prevent the state of the input changing asynchronously during program execution.

Any buffered (and perhaps debounced) digital input tags should have an equivalent tag name but with the “**d**” debounce identifier.

Example: **_6GQa_ZSC_dDI** is the buffered/debounced tag of **_6GQa_ZSC_DI**.

All analogue inputs will be processed in the “Analog_Inputs” program. Where ControlLogix modules are used, scaling will be done locally within the module. All alarms however are processed within the PLC logic as this provides greater flexibility in filtering and debouncing alarms.

Analogue inputs are buffered into a UDT structure of type ‘AnalogIn’ as they are more complex containing instrument and alarming information.

Example: **_63GQe_LIT_AI** is buffered in engineering units into analogue tag structure **_63GQe_LIT.EU.PV**.

Output tags do not need buffering as they can be allowed to change asynchronously to the program scan.

4.4.5.3. Alias Tag Naming

An alias tag in the ControlLogix platform lets you create one tag that represents another tag. Characteristics of alias tags include:

- Both tags share the same value as defined by the base tag
- When the value of a base tag changes, all references (aliases) to the base tag reflect the change
- Aliasing a device base tag to an I/O point of a module provides the capability to force the I/O directly from the base tag.

When assigning aliases, avoid:

- Nesting aliases (you cannot have an alias of an alias)
- Using multiple aliases to the same tag

Typically, aliases are used in an application when a device base tag is aliased to an I/O point of a module. For example, an input tag **_63GQb_LSL_DI** is aliased to the input address '**Local:1:I.Data.1**' of the ControlLogix input module. The tag address '**Local:1:I.Data.1**' is not very descriptive as it only tells you what point the device is physically wired to. However, aliasing the device name **_63GQb_LSL_DI** to the address gives you both description and a wiring point. It also allows the base tag (LSL_63GQb_DI) to be forced directly during commissioning or troubleshooting

4.4.5.4. Devise Tag (UDT) Naming

User defined types (UDT) in the ControlLogix platform allow for the combination of multiple tags of varying data types into a single data structure template that represents an instrument, equipment, or functional element. These types can then be instantiated and assigned a single tag name.

These tags represent the actual device itself and not the I/O point. The name adopted for these tags will be equivalent to the name shown on the P&ID diagrams. In some cases devices such as analogue inputs will have both a UDT and a real analogue input with the same name. In this case the UDT will be named as shown on the P&ID diagram and the real input will have the _AI suffix appended to it.

Example: **_63GQe_LIT_AI** is the analogue input tag (level reading) for governor receiver hydraulic oil level and _63GQe_LIT is the name given to the UDT tag representing the level transmitter.

Example: **_6GQa_K1_DO** is the output tag (contactor output) for governor oil pump A and _88GQa is the name given to the UDT tag representing the pump.

Some UDTs will have members defined that represent digital inputs or outputs. These tags are prefixed with the letter "i" and "o" respectively. The digital input device base tags are mapped into the input members at the start of every standard routine. The digital output device base tags are mapped out of the UDT output members at the end of every standard routine.

More information on UDTs can be found in Section 4.7.3.

4.4.5.5. Remote Data Transfer Tags

When data is transferred between controllers it must be done so using blocks of registers to optimise communication packets and reduce the amount of connections used. The following syntax should be used for naming tags used for all remote data block transfers between controllers:

Syntax: **Data_*aaaaa*_*bbbbb*** or **Data_*aaaaa*To*bbbbb***

Where: **aaaaa**: Controller name or description where the data is produced.

To: The word 'To'

bbbbb: Controller name or description where the data is consumed. This portion of the name may be omitted if data is consumed by multiple controllers.

The usage may differ depending on whether the logic is intended to be standardised. In this case, references to specific unit numbers may be eliminated to allow the logic to be re-used.

Example: **Data_M1U04_M1G04** is the data structure produced from the Unit 4 ControlLogix processor inside Murray 1 power station, and consumed by the Governor ControlLogix processor of unit 4, inside Murray 1 power station.

Data_UnitToGov is the data structure produced from the unit processor, and consumed by the Governor processor. Note this tag name is unit and station non-specific and can be re-used across multiple similar units.

Individual tags may then be copied out of the data block received from another controller. However the individual tags representing data from other controllers must have the source prefix in front of the tag name.

Example: **M1U04_6GQa_ZSC** is the tag received from Murray 1, Unit 4 PLC controller to say the governor oil pump A contactor is closed.

4.4.5.6. Tag Description

Every tag that is used must have a description.

By using a tag description convention, the system will be easier and faster to design, understand and maintain. Each tag generally describes database points associated with an item of equipment. Each item of equipment is generally associated with a subsystem. For example; Governor Oil Pump A is associated with the governor subsystem

If a tag is representing a device or a piece of plant equipment, then the tag description should include the following characteristic information:

System:	Where the device is located. E.g. Unit, Group Transformer
Sub-System:	The sub-system the device belongs to. E.g. Cooling Water
Equipment:	The type of device. E.g. Pump or Valve
Attribute:	The attribute or particular parameter that is associated with the actual tag. E.g. Input, Output, UDT member description.

Examples	**System**	**Subsystem**	**Equipment**	**Attribute**
Motor	Unit	Governor	Oil Pump A	Running Input
Valve	Unit	Cooling Water	Suction Valve	Open Input
Instrument	Transformer	Oil	Flowmeter	Flowrate Input

The description format should follow closely what is allowable in the RSLogix5000 tag description editor. A tag description in RSLogix5000 cannot be longer than 128 characters. By default RSLogix5000 is configured to display up to 20 characters per line in a tag description. This means there should be up to of 6 lines of description text with 20 characters in each line, and only 8 characters in the seventh line.

Where possible, each of the above description types should start on a new line. Descriptions for UDTs need only have a "." (full-stop) character on the attribute line so that the UDT member description is automatically transferred onto a new line.

For tags representing a state or steps in a sequence, each step must have a unique description of what the step is generally doing. These descriptions must match descriptions on any associated state diagrams.

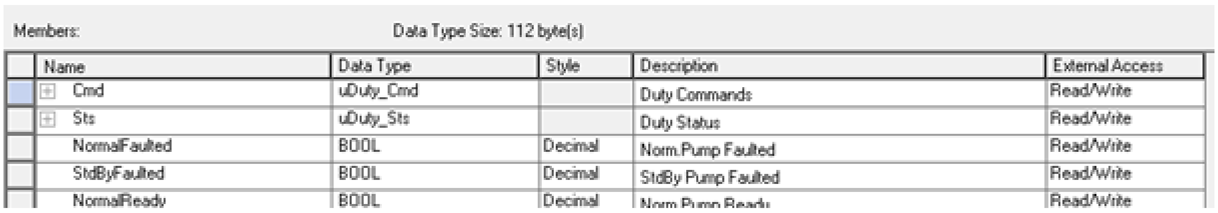
4.4.5.7. UDT Descriptions

With custom user defined data types (UDT), the software concatenates the tag's root description with the data type's member description. This creates a very specific description, saves development time, and improves the resulting documentation.

When creating new UDTs, the member tags should always have short meaningful descriptions of the member functionality (no more than 19 characters if possible), so that they are neatly merged with the base tag description.

Consequently new tags of UDT type should have a base description that only describes the equipment or process the base tag represents. The member description will then be automatically appended to the end of the base tag description to refine the purpose of the singular tag. A full stop "." on a new line should always be added to the end of the UDT base tag description to force the member description to start on a new line.

Example: The UDT **uDuty** has descriptions added to each individual element in the UDT definition. When a tag of this type is created, **_88GQ_Duty**, a description of that plant is added to the top level tag only, **Governor Oil Pumps A&B Duty**. Sub-elements of the created tag automatically append the top level tag to each element description from the UDT definition – **Governor Oils Pump A&B Duty . Duty Commands.**


Members: Data Type Size: 112 byte(s)

Name	Data Type	Style	Description	External Access
⊞ Cmd	uDuty_Cmd		Duty Commands	Read/Write
⊞ Sts	uDuty_Sts		Duty Status	Read/Write
NormalFaulted	BOOL	Decimal	Norm.Pump Faulted	Read/Write
StdByFaulted	BOOL	Decimal	StdBy Pump Faulted	Read/Write
NormalReady	BOOL	Decimal	Norm.Pump Ready	Read/Write

Figure 5.2 - Example of UDT and Tag Descriptions

4.4.6. MicroLogix Symbols and Descriptions

The MicroLogix platform uses a series of data files of different data types to contain a block of addressable data for use in PLC logic. Each data file can include an optional name and description. Each addressable element of a data file can include an optional symbol and description.

4.4.6.1. Data Files

Data files used in the PLC application shall be configured with a name representing the function of the data within the file. The name is displayed as a suffix to the data file type and number in the project explorer and provides an easy mechanism to find the location of data related to different functional areas.

4.4.6.2. Symbols

Data file address symbols provide a method to add a unique tag name to a data file address, up to 20 characters long.

Symbols should be used to identify hardwired and communications I/O with a field referenceable tag similar to that used for aliases in section 4.4.5.3.

4.4.6.3. Descriptions

Descriptions provide a method to add a comment to a data file address, up to 120 characters long over 5 lines.

4.4.7. Routine Comments

At the start of all ladder routines, a rung with a NOP instruction shall be created with a rung comment for the purpose of documenting the routine. For Function Block Diagrams, a text box shall be used to document the routine. The routine comment shall contain the following information:

- Routine name:
- Created by:
- Date the routine was created:
- Routine revision number:
- Last Person to change the routine:
- Date of the last routine change:
- Document change no. or ref no. for last change:

4.4.8. Ladding Rung Comments

All rungs should be given descriptions that clearly define their purpose. Rungs that represent steps in a sequence should clearly describe the purpose of the step. All the possible transitions from the step should also be listed explaining the exact conditions that initiate the transfer.

4.4.9. Task, Program and Routine Naming Conventions

ControlLogix Projects are broken down into Tasks, Programs, and Routines (refer to Section 4.6 for more details). The following naming conventions should be followed when laying out the ControlLogix Project structure.

4.4.9.1. Tasks

<u>Continuous Task</u>

There can be only one continuous task per ControlLogix controller. A recommended name would be “MainTask”.

<u>Periodic Task</u>

ControlLogix can have up to 31 periodic tasks configured. ControlLogix Periodic task names should include the function of the task, and the rate at which it is called.

Syntax: **aaaaaaaa_bbb**ms

Where:	**aaaaaaaa:**	Periodic Task Description
	bbb:	Rate at which the task is executed in ms

Example: A periodic task dedicated to process PID loops every 100ms.

PID_Loops_100ms

4.4.9.2. Programs

All programs should be named according to the sub-system that they control. Any standard programs defined for a site should be named the same in all applications. Some suggested standard programs to use in each project are:

● General	For general house-keeping and inter-processor communications
● Interface_	Contains logic and mapping for each communications interface
● Digital_Inputs	For buffering and de-bouncing all digital inputs
● Analog_Inputs	For processing analogue inputs and their alarming
● Unit_Protection	Contains upper level logic for unit protection
● Unit_Control	Contains upper level logic for unit control
● CoolingSystems	Contains upper level logic for the cooling systems
● MIV	Contains upper level logic for the main inlet valve
● Governor	Contains upper level logic for governor

● Excitation	Contains upper level logic for excitation system
● AutoSync	Contains upper level logic for the auto synchronisation system
● Gen_CB	Contains upper level logic for the generator circuit breaker
● Turbine	Contains upper level logic for the turbine
● MotorDOL	Contains all the standard routines for DOL motors
● MotorE300_DS	Contains all the standard routines for dual supply motors with E300
● MotorE300_SS	Contains all the standard routines for single supply motors with E300
● Valves_SC	Contains all the standard routines for single coil valves
● Valves_DC	Contains all the standard routines for dual coil valves

4.4.9.3. Routines

Routines should be given a name representative of the logic they execute. Every program will contain one Main Routine in which all other sub-routines in that program are called from via a jump to subroutine (JSR) instruction. The main routine will simply be called "**MainRoutine**".

All programs should have a routine named "**AlarmSum**". This routine is for mapping any program generated HMI alarms to the alarm summary array. Any alarm inhibits to the remote SCADA system are also performed here.

Routines for sequence logic should be named according to an abbreviated sequence description. The sequence routine name will be prefixed with the letters "SQ" to keep them grouped together and identifiable. State machine logic will be prefixed with the letters "SM".

Example: **SM_Unit** Is the routine name for the unit state machine.

SQ06_Unit Start Is the routine name for the unit start control sequence.

The names of all general sub-routines should contain a prefix that will allow the routines to be displayed in order of how they are called from the main program. The following syntax should be used for naming general sub-routines:

Syntax: ***raa_bbb…***

Where: ***aa:*** Routine order sequence **01-99**

bbb… : Sub-system or function the routine is controlling

Example: **r01_AlarmReset** Is the routine name for any alarm reset logic.

Standard routines for devices within standard device programs should be named identically to the devices' UDT tag name.

Example: **_88GQa** Is the routine name for the "Governor Oil Pump A" standard routine.

4.5. Module Configuration

Modules and devices should be configured in the PLC applications I/O configuration as per the following sections.

4.5.1. Specifying an RPI for I/O Modules

When adding I/O modules to a controller project, you specify a "Requested Packet Interval" (RPI) rate. Depending on the controller platform, you can select an RPI rate per module (ControlLogix) or an RPI rate per controller (CompactLogix and FlexLogix).

4.5.1.1. Recommended RPI Rate

Specify a Requested Packet Interval (RPI) at 50% of the interval actually required. Setting the RPI much faster than what the application needs wastes resources, such as network and CPU processing time.

For example, if information is needed every 80 msec, set the RPI at 40 msec. The data is asynchronous to the controller scan, so data is sampled twice as often (but no faster) than needed to make sure the most current data is available.

4.5.2. Local Discrete I/O Modules

4.5.2.1. Communication Format

In most cases for local modules, the communication format can be left in the default format that is Input Data. This establishes one connection per module.

4.5.2.2. Electronic Keying

As a minimum, the electronic keying selection should be set to “Compatible Module”. This ensures the following criteria are met:

- Module type, catalogue number, and major revision must match
- The minor revision of the physical module must be equal to or greater than the one specified in the software

The electronic keying selection should be set to “Exact Match” for any applications that require the firmware major and minor revisions to be exact (e.g. SIL2 systems).

4.5.2.3. Requested Packet Interval (RPI)

The RPI for digital I/O in the “local” chassis can be left at the default 10ms RPI.

4.5.2.4. I/O Configuration

If high speed is required from a particular input point, it can be configured for Change of State (COS). The default setting will send data immediately upon detecting a change of input state. This should be configured on all input points where high-speed update is required or where the update required needs to be faster than the RPI rate.

This is only applicable to Input Modules.

4.5.3. Local Analogue I/O Modules

4.5.3.1. Communication Format

In most cases for local modules, the communication format can be left in the default format that is “Float Data”.

4.5.3.2. Requested Packet Interval (RPI)

Analogue input modules should be set at a rate demanded by the process control. For instance, if an analogue input is used by a PID instruction that executes in a periodic task running every 100ms then a RPI of 40ms to 100ms should be adequate. Analogue output modules can typically be set to a faster RPI of 20ms.

However, data to I/O in “remote” chassis should be set to the highest value as possible to reduce the amount of network traffic.

4.5.3.3. I/O Configuration

The Analogue Module configuration can vary greatly between applications. There are many factors that can dictate the module’s configuration such as speed, scaling, signal type, etc…Therefore the recommendations outlined below are general in nature and may not meet certain application requirements.

4.5.3.4. Real Time Sampling (RTS)

The Real Time Sample (RTS) period should be set no less than the RPI for the module. The recommendation is to set the RTS value to the same value as the RPI for the module. This will ensure data is sampled at the same rate as the data being sent to the controller.

4.5.3.5. Engineering Units

If "Float Data" is selected as the communication format then engineering units become an option for scaling the analogue data. Configuring engineering units within the module for the analogue signals is recommended for ease of understanding of the data being sent to the controller. Scaling locally within the ControlLogix analogue module also relieves the processor from executing the scaling equations within the PLC program.

4.5.3.6. Alarm Configuration

Process alarm values are expressed in engineering units and not analogue signal values. Setting alarms directly in the analogue input module can be achieved in many modules, however this is not the case in the 1756-IF16 when selected as a 16 point analogue input card. Generating alarms from the PLC logic rather than within the analogue input module provides greater flexibility for filtering alarms and changing set points from the HMI.

4.5.3.7. Electronic Keying

As a minimum, the electronic keying selection should be set to "Compatible Module". This ensures the following criteria are met.

- Module type, catalogue number, and major revision must match
- The minor revision of the physical module must be equal to or greater than the one specified in the software

Any applications that require the firmware major and minor revisions to be exact (e.g. SIL2 systems) the electronic keying selection should be set to "Exact Match".

4.5.4. Remote I/O Modules

The communication format determines whether the controller connects to the I/O module via a direct or a rack-optimized connection. The communication format also determines the type and quantity of information that the module will provide or use.

4.5.4.1. Direct Connection

Each module passes its data to/from the controller individually. Communication modules bridge data across networks.

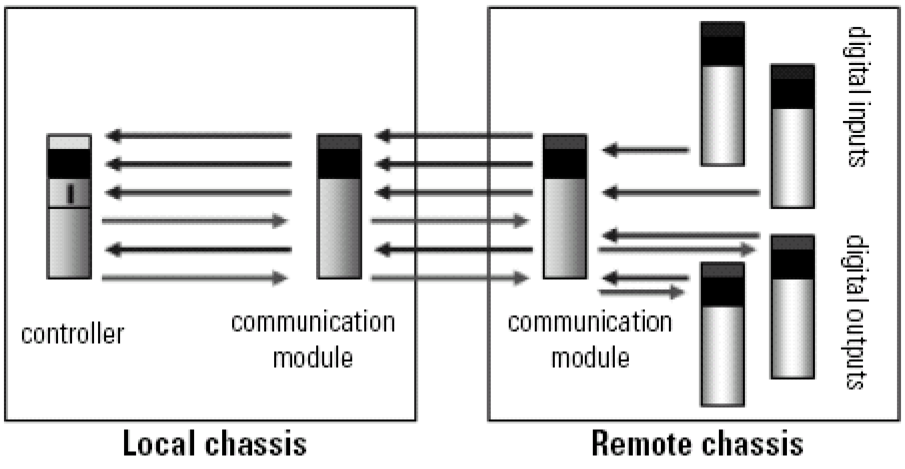


Figure 6.1 - Direct Connections to Remote I/O

This is the only method supported in the local chassis. Remote analogue modules only support this method as well. This method requires additional connections and network resources however each module can determine its own rate (RPI).

4.5.4.2. Rack Optimised Connection

The communications module in a remote chassis consolidates data from multiple modules into a single packet and transmits that packet as a single connection to the controller.

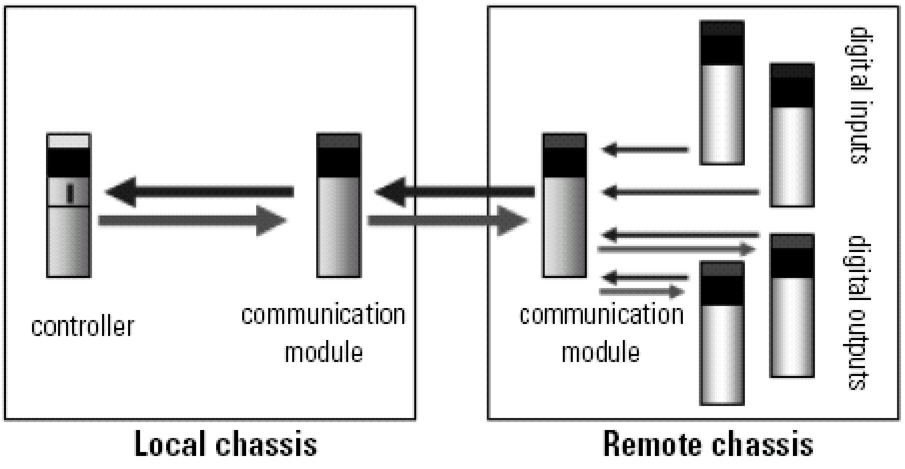


Figure 6.2 - Rack Optimised Connection to Remote I/O

One connection can service a full chassis of digital modules which reduces network resources and loading. However:

- All modules are sent at the same rate
- Unused slots are still communicated
- Still need a direct connection for analogue and diagnostic data
- Limited to remote chassis
- I/O data presented as arrays with alias tags for each module

4.5.4.3. Connection Selection

In most cases if there are more digital modules then there are analogue modules in a remote rack: then select the rack-optimised connection.

In some cases, all direct connections work best. For a small number of digital modules in a large chassis, it might be better to use direct connections because transferring the full chassis information might require more system bandwidth than direct connections to a few modules.

4.5.4.4. Requested Packet Interval (RPI)

The RPI for I/O in “remote” chassis can initially be left at the default 10ms RPI. If communication of a high volume of I/O data over the network causes problems, the RPI should be reviewed and increased as required to reduce over utilisation of the network.

4.5.5. HART I/O Modules

HART protocol relies on digital communication signal superimposed on top of a standard 4-20mA analogue signal, to allow additional command, configuration and monitoring of the connected instrument.

SHL’s standard is to use Highway Addressable Remote Transmitter (HART) protocol to communicate with field instrumentation where possible.

The following features of HART protocol should be integrated with the PLC application where supported by the HART module and connected instruments:

- Reading standard mA signal
- Reading HART primary, secondary, third and fourth values where available
- Reading HART device status and alarm information
- Storing and managing the transmitter configuration in the PLC (where possible)

4.5.6. EtherNet/IP Network Devices

SHL has standardised on EtherNet/IP as the primary communication protocol for data transfer between controllers, HMIs, instrumentation and associated equipment on the plant local area networks.

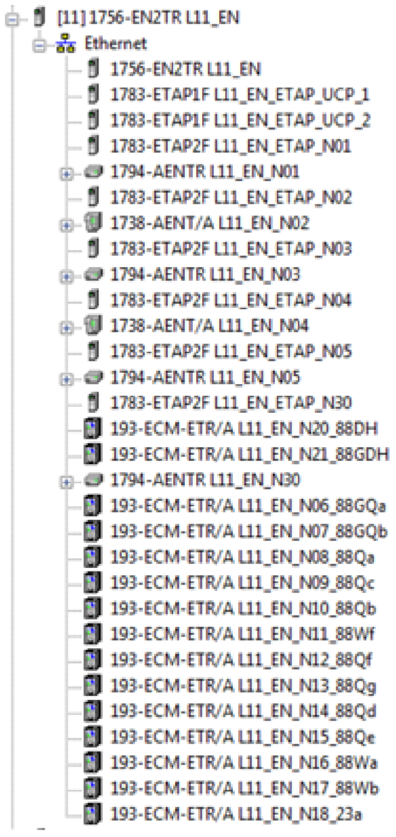
Where field I/O, instrumentation, and associated equipment (such as drives or electronic relays) are required to be controlled and monitored by the PLC, an EtherNet/IP connection should be configured and established in the PLC where supported by the networked device.

Where possible, the primary method of establishing and maintaining connection to a networked device should be by adding the device to the ControlLogix PLC's I/O Configuration tree using a native module type or add-on-profile.

Using native and add-on modules for the configuration of networked devices ensures that:

- Standardised, device-specific tag structures are created to represent the device interface
- Configuration parameters for the device can be stored in the PLC configuration
- Fault finding and maintenance is simplified by improving visibility of networked devices, and concentrating device configurations and interfaces in the PLC project

The status of networked device configured in the PLC's I/O Configuration tree should be monitored and alarmed in PLC software using GSV instructions.



```
[11] 1756-EN2TR L11_EN
  Ethernet
    1756-EN2TR L11_EN
    1783-ETAP1F L11_EN_ETAP_UCP_1
    1783-ETAP1F L11_EN_ETAP_UCP_2
    1783-ETAP2F L11_EN_ETAP_N01
    1794-AENTR L11_EN_N01
    1783-ETAP2F L11_EN_ETAP_N02
    1738-AENT/A L11_EN_N02
    1783-ETAP2F L11_EN_ETAP_N03
    1794-AENTR L11_EN_N03
    1783-ETAP2F L11_EN_ETAP_N04
    1738-AENT/A L11_EN_N04
    1783-ETAP2F L11_EN_ETAP_N05
    1794-AENTR L11_EN_N05
    1783-ETAP2F L11_EN_ETAP_N30
    193-ECM-ETR/A L11_EN_N20_88DH
    193-ECM-ETR/A L11_EN_N21_88GDH
    1794-AENTR L11_EN_N30
    193-ECM-ETR/A L11_EN_N06_88GQa
    193-ECM-ETR/A L11_EN_N07_88GQb
    193-ECM-ETR/A L11_EN_N08_88Qa
    193-ECM-ETR/A L11_EN_N09_88Qc
    193-ECM-ETR/A L11_EN_N10_88Qb
    193-ECM-ETR/A L11_EN_N11_88Wf
    193-ECM-ETR/A L11_EN_N12_88Qf
    193-ECM-ETR/A L11_EN_N13_88Qg
    193-ECM-ETR/A L11_EN_N14_88Qd
    193-ECM-ETR/A L11_EN_N15_88Qe
    193-ECM-ETR/A L11_EN_N16_88Wa
    193-ECM-ETR/A L11_EN_N17_88Wb
    193-ECM-ETR/A L11_EN_N18_23a
```

Figure 6.3 - Example of E300 overload relays integrated with PLC I/O Configuration

Communications between ControlLogix controllers on the same EtherNet/IP network may be established using Produced / Consumed Data (refer section 4.3.2).

Where communications to networked devices cannot be integrated with the PLC's I/O Configuration tree, MSG instructions may be configured in logic to establish and maintain the connection (refer section 4.3.4).

4.5.7. Module Health Monitoring

In ControlLogix PLC applications, local and network module health should be monitored in logic using GSV instructions to detect any module status other than normal/healthy.

Alarming should be implemented if the unhealthy status persists for longer than a short period of time.

Fault capture code should also be implemented to detect changes in the module status, and trap and log fault codes into a local PLC first-in-first-out buffer array for monitoring by maintenance personnel.

Each module is required to maintain a separate buffer of at least twenty records. A new record should be created on each change in a module's fault code. The fault code should be stored in the buffer with the date and time of the change in state.

4.6. PLC Application Structure

SHL's guidelines for coding PLCs results in PLC applications that use similar programs, routines and standard modular code as much as possible. This provides quicker and easier development, increases reliability, is intuitively easier to follow and maintain, and future extensions can be performed with the least amount of effort.

A ControlLogix application is made up of one main, continuous task and multiple optional periodic and event tasks. Each task may contain up to 100 programs that execute in the order in which they are listed in the controller organiser.

Each program has one main routine, and as many sub-routines as memory permits. Sub-routines are called programmatically from within the main routine for each program.

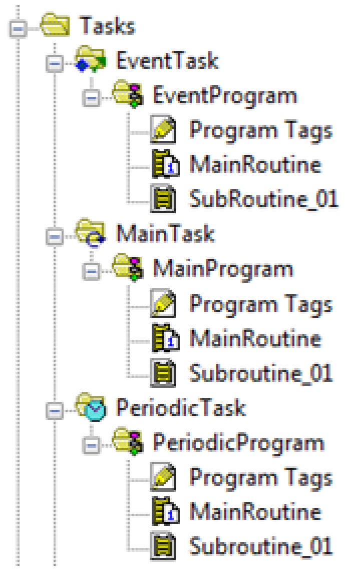


Figure 7.1 - Example PLC Application Structure

4.6.1. Tasks

Tasks are created to group together major functions that have similar logic and execution timing requirements.

Tasks can be configured as either continuous, periodic or event:

Continuous task – runs in the background. Any CPU time not allocated to other operations or tasks is used to execute the continuous task. There can only be one continuous task.

The continuous task should be used for majority of the logic in SHL PLC applications.

Periodic tasks – performs a function at a specific time interval. Whenever the time for the periodic task expires, the periodic task:

- Interrupts any lower priority tasks
- executes once
- returns control to where the previous task left off

Periodic tasks should only be used for logic requiring execution at precise pre-defined time intervals. This is typically logic executing time-based functions such as PID loops, totalisers, etc.

Event tasks – performs a function only when a specific event (trigger) occurs. Whenever the trigger for the event task occurs, the event task:

- interrupts any lower priority tasks
- executes one time
- returns control to where the previous task left off

Event tasks are typically used for operations that require synchronisation to a specific event.

Event tasks are not commonly used in SHL PLC applications.

4.6.1.1. Task Usage Considerations

Limit the number of tasks used as much as possible. If there are too many tasks then:

- The continuous task may take too long to complete.
- Other tasks may experience overlaps. If a task is interrupted too frequently or for too long, it may not complete its execution before it is triggered again.
- Controller communications might be slower.
- If the application is designed for data collection (e.g. for a HMI), try to avoid multiple tasks. Switching between multiple tasks limits communication bandwidth.

If the application has an excessive amount of communications (such as message instructions or RSLinx communications), consider using a periodic task rather than a continuous task. This avoids the overhead associated with task switching, which can improve performance.

At the end of a task, the controller performs overhead operations (output processing) for the output modules in the system. This output processing may affect the update of the I/O modules in the system. Output processing for a specific task can be turned off, which reduces the elapsed time of that task. For example, a task created for totalising should have the automatic output processing disabled to reduce the task overhead.

4.6.2. Programs

In a ControlLogix application, programs provide functional separation and execution order control.

Listed below is a typical example of the separation of functional requirements into tasks and programs based on high-level order of execution requirements. This example should be followed where possible for SHL PLC applications.

Continuous Task

- General house-keeping functions
- Communication logic (Read/Write Data) – MODBUS, DNP3
- Input data buffering and de-bouncing
- Sequence and control logic (plant functions)
- Standard device logic (setting/resetting device outputs)
- Other output mapping (status and output data mapping to the remote SCADA servers)

Periodic Tasks

- PID loop instructions
- Totalisers
- Priority based logic – e.g. Governor control

The application should be structured so that house-keeping and input processing is done first, followed by sequence and control logic, and finally the output functions such as device control and output mapping. The device control programs are responsible for controlling the outputs to actuate/de-actuate the devices, and typically are implemented in standard equipment routines.

The following diagram depicts the information above.

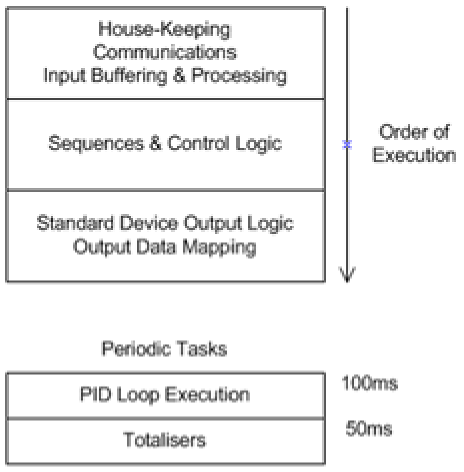


Figure 7.2 - Typical functional layout of a PLC application

In each of the categories above, designers should further divide major equipment pieces, plant cells or functions into isolated programs. This elegantly segregates the code into independent modules that can be developed by individual programmers.

Once the plant is divided and the functions defined, the programmers must go through the process of creating the programs and ordering their execution intuitively according to the process flow. There are some programs that will be standard and used in every application and these must also be included as well.

Each program will have a defined main routine that will always execute when the program starts executing. It will also have one or more subroutines containing the majority of the logic for the program. The main routine should only contain jump-to-subroutine instructions (JSR) to call the subroutines.

Each program should also have a routine created for alarm mapping for HMI and SCADA applications. This helps to arrange alarms into functional groups. It also helps in locating particular alarm logic in the future.

Details of standard programs used in SHL PLC applications can be found in section 4.9.

4.6.3. Routines

Routines are created within programs and contain the logic for execution by the controller.

ControlLogix processors can be programmed in a mixture of any four languages. Each routine within a program however can only contain logic in one programming language.

Details of standard routines used in SHL PLC applications can be found in section 4.9.

The four languages offered for programming routines are:

- Ladder Logic
- Function Block Diagram
- Sequential Function Chart (SFC)
- Structured Text

4.6.3.1. Ladder Logic

Ladder logic should be used for majority of the programming.

It is the language most understood by maintenance personnel who may have to interpret code in order to troubleshoot the machine or process. Ladder logic is best suited for programs requiring the following functionality:

- Continuous or parallel execution of multiple operations
- Boolean or bit-based operations
- Complex logical operations
- Message and communication processing
- Machine interlocking

4.6.3.2. Function Block Diagram

Function block diagrams should typically be only used for the following functionality:

- PID loops and continuous process control
- Analogue scaling and alarming
- Totalising

4.6.3.3. Sequential Function Chart

Sequential function charts are useful for high-level management of multiple operations, repetitive sequence of operations, batch processes and state machine operations. They provide a diagrammatic view of the sequence within the controller that allows the user to easily identify what step the sequence is at and the required conditions to transition to other steps.

The various states of a unit may easily be translated into a sequential function chart. Other sequential operations may also be best controlled by a sequential function chart.

A current drawback to sequential function charts is that accepting some online edits to a chart results in the edited function chart to reset back to the initial step. Complex function charts where there are a lot of possible transitions between steps are also not displayed neatly within the Logix editor.

For the above reasons, SHL's standard is that sequence logic should be developed in Ladder Logic.

4.6.3.4. Structured Text

Structured text is less widely understood and more difficult to troubleshoot than other alternatives, and for these reasons is rarely used. Its use should be limited to:

- Transitions of a sequential function chart
- Complex mathematical operations
- Specialised array or table loop processing
- ASCII string handling
- Protocol processing

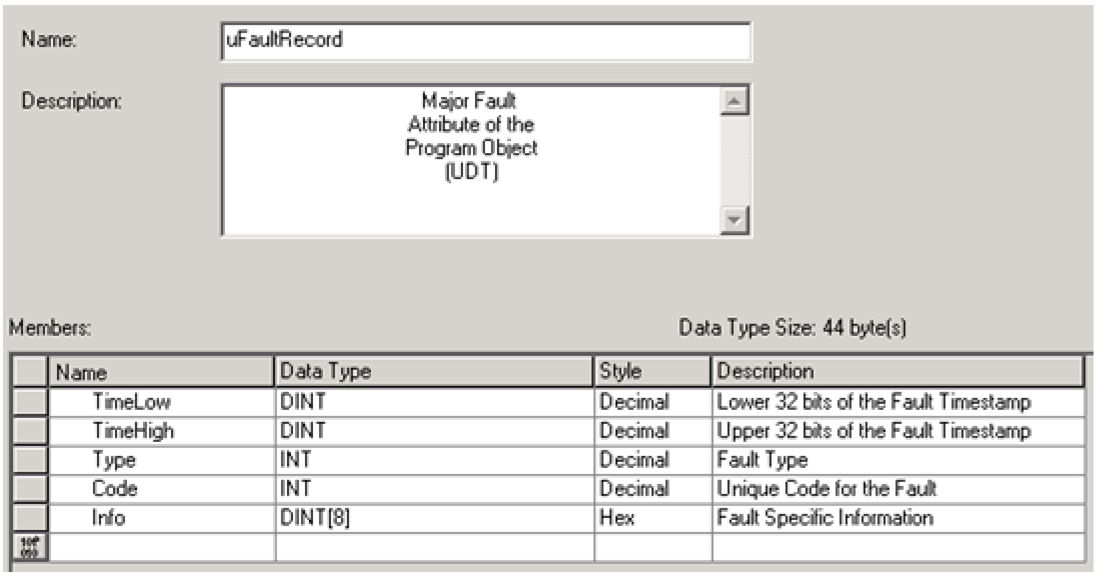
4.6.4. Fault Handler Routines

Controller faults are generated by illegal commands in the programming (e.g. indirect pointers exceeding array dimensions) or by hardware failures within the ControlLogix system. In the case of major controller faults, both the process and the program scan are stopped.

Depending on the application, it may be necessary that the entire system does not shutdown in the case of certain major faults. In these cases, a fault routine should be used to clear a specific fault and let part of the system continue to operate.

As a minimum fault handling should be provided at the controller level fault handler. Although individual fault handler routines may also be included in each program.

To support capturing and clearing of major faults, a UDT should be created with the relevant fault information fields as per the example below.



Name: uFaultRecord

Description: Major Fault Attribute of the Program Object (UDT)

Members: Data Type Size: 44 byte(s)

Name	Data Type	Style	Description
TimeLow	DINT	Decimal	Lower 32 bits of the Fault Timestamp
TimeHigh	DINT	Decimal	Upper 32 bits of the Fault Timestamp
Type	INT	Decimal	Fault Type
Code	INT	Decimal	Unique Code for the Fault
Info	DINT[8]	Hex	Fault Specific Information

Figure 7.2 - Example Fault Record UDT

A GSV instruction is used to retrieve and store the major fault code.

An SSV instruction is used to clear the fault if required.

4.6.5. Power Up Handler Routine

The power-up handler is an optional controller level task that executes when the controller powers up in run or remote run mode.

The power-up handler should be used when either of the following is required after power is lost and then restored:

- Prevent the controller from returning to run mode - the power-up handler will produce a major fault, type 1, code 1, and the controller will enter the faulted mode.
- Execute specific fault routine / actions and then resume normal execution of the logic.

4.6.6. MicroLogix Application Structure

MicroLogix PLCs programmed in RSLogix 500 contain a single main program file, and multiple configurable subroutine program files.

For small applications, the entire logic may be programmed directly into the main program file. For larger or more complex applications, the logic shall be segregated into different subroutine program files that are then called programmatically from the main file using JSR instructions.

Functionality should be segregated similar to ControlLogix functionality as described in section 4.6.2.

Program files may only be configured in ladder logic.

4.7. Signal Addressing

4.7.1. Addressing Data in ControlLogix Controllers

ControlLogix controllers support IEC 61131-3 atomic data types, such as BOOL, SINT, INT, DINT, and REAL. The controllers also support compound data types, such as arrays, predefined structures (such as counters and timers) and user-defined structures.

In SHL ControlLogix PLC applications, User Defined Types (UDTs) are used heavily to combine related signals (e.g. for a pump) into a single data structure. The majority of tags used for internal PLC handling should be stored in a UDT specific to the functional plant area being controlled or monitored.

Mapping is used in input sections of code to copy hardwired and communication input tags into UDT members before they are used, and in output sections of code to copy UDT member signals to hardwired and communication output tags after they've been processed or actioned.

Additional detail can be found in the following sections.

4.7.1.1. Data Types

DINT data type should be used whenever possible. The ControlLogix controllers natively perform DINT (32 bit) and REAL (32 bit) math operations. DINT data types use less memory and execute faster than other data types.

The following atomic types are used most commonly:

- DINT for most numeric values and array indexes
- REAL for manipulating floating-point, analogue values
- BOOL for binary signals
- SINT (8 bit) and INT (16 bit) for user-defined structures or when communicating with external devices that do not support DINT values

When mapping BOOL alarm and status values over communications to a HMI, they should be grouped into DINT arrays to make best use of controller memory, and to easily facilitate status comparisons via the FBC and DDT instructions.

4.7.1.2. Arrays

A tag created as an array defines a contiguous block of memory to store a specific data type (either atomic or structure) as a table of values. Note that:

- Tags support arrays in one, two, or three dimensions.
- User-defined structures can contain a single-dimension array as a member of the structure.

Arrays may be used although their use should be limited to a single dimension. Reasons are:

- Better support by native file instructions
- Fully supported in user-defined structures and arrays
- Smallest impact (execution time and memory) for indexed references
- Can create new arrays when programming online

Indexed array addressing should be avoided. Indexed array addressing executes slower as all tag references must be calculated at run time. If array indexing must be used then make sure DINT tags are used as the array indexer. DINT tags execute the fastest. SINT, INT, and REAL tags require conversion code that can add additional scan time to an operation.

4.7.1.3. Tag Scope

Data scope defines where tags can be accessed. When a tag is created, its scope is assigned as either controller or program. Controller-scoped tags are accessible by all programs and are considered to be 'global'. Program-scoped tags are only accessible by the code within a specific program.

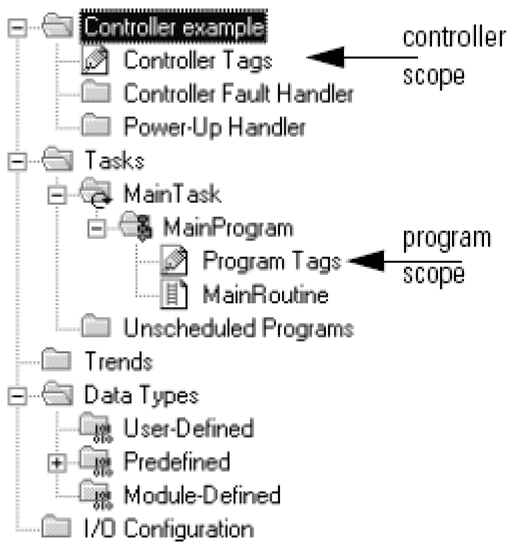


Figure 8.1 - Tag scoping within an application

Program scoped tags can have the benefit of providing isolation between programs, preventing tag name collisions and improve the ability to reuse code. Logic developed by multiple programmers can also be integrated together without having tag collisions (tags with identical names).

All equipment and instrument UDT tags and I/O tags should be controller scoped. This way any program within the ControlLogix application can access these tags for status and control.

All tags used by external interfaces (such as SCADA or HMI tags) should also be controller scoped. This way tag naming is greatly simplified when using 'Direct Tag Referencing' from the HMI system as the program name does have to be included.

Any constants used by the application should be controller scoped so that any program has access to these constants. It is good practice to name constants and other miscellaneous controller scoped tags, starting with the letter 'g' (global) e.g. "gAlarmExtendDelay". This is so that the tags are recognised as being controller (globally) scoped. It also groups all these tags together when viewing them in the tag editor.

Any tags used only once or only within a single program (miscellaneous timers, bits etc.) should be made program scoped. Doing this will prevent them from clogging up the controller tag list and keep them neatly tied in within the program.

In summary, any tag that has any connection with the outside world or is a constant parameter or setting used by multiple programs should be controller scoped. All other tags used only within one program should be program scoped.

4.7.2. User-Defined Types (Custom Data Types) in ControlLogix Controllers

User-defined tag structures allow the combination of multiple data types into a single structure. These types can then be instantiated and assigned a single tag name, with individual members of the structure also able to be individually addressed using dot notation (e.g. **UDT_Tag.Element_01**).

I/O data should normally be stored in a UDT instance and referenced in logic using the UDT instance member instead of the input tag directly. If UDTs include members representing I/O data, logic should be used to copy the data to and from the corresponding I/O tags.

UDT member names that represent real I/O should be prefixed with lowercase 'i' for inputs and 'o' for outputs.

4.7.2.1. Controller Memory Utilisation

A ControlLogix controller aligns every data type along an 8-bit boundary for SINTs, a 16-bit boundary for INTS, or a 32-bit boundary for DINTs and REALs. BOOLs also align on 8-bit boundaries, but if they are placed adjacent to each other in a user-defined structure, they are mapped so that they share the same byte.

To ensure efficient use of memory within a structure, members of the same data type should be grouped together. When you use the BOOL, SINT, or INT data types, place members that use the same data type in sequence:

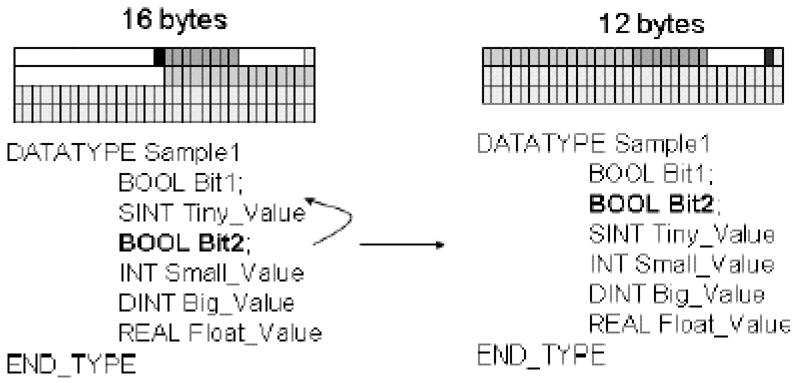


Figure 8.2 - Grouping different data within a UDT

When creating UDTs, individual members should be grouped together and placed in the following order:

- Other UDTs
- BOOLs
- SINTs
- INTs
- DINTs
- REALs
- STRINGs
- Other built-in predefined structures (TIMERs, COUNTERs, etc.)

When modifying an existing UDT, the above order should also be followed by inserting new members adjacent to their associated group.

4.7.2.2. Communication Size Limitation

The size of user-defined structures should be limited if they are to be communicated outside the controller. Produced and consumed tags are limited to 500 bytes over the backplane and 480 bytes if over a network. RSLinx can optimize user-defined structures that are less than 480 bytes.

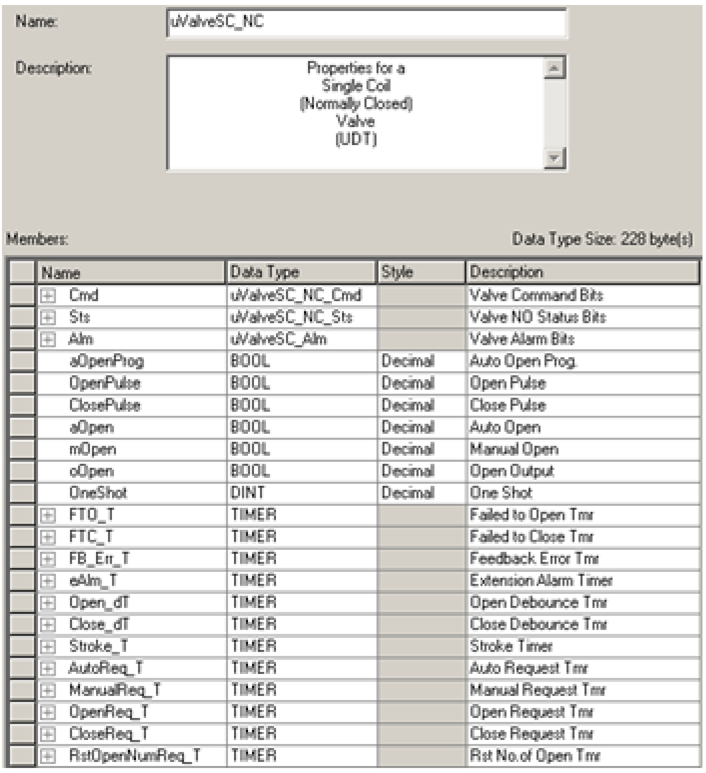
4.7.3. User Defined Types (UDT) Design

4.7.3.1. Creating Objects

User-defined structures let you combine multiple data types into one structure. This is an enormous benefit when encapsulating data for one object into one neat data structure. This means plant objects such as pumps or valves may have a customised data structure created that incorporates all the properties, configurable parameters, alarms, timers, counters etc. ever needed for the object.

It keeps the data for any one object nicely bound together in one place. Monitoring objects then becomes easier and programming is also simplified by not having to remember all the different tag names.

Standard data structures often replicate standard routines. All device and instrument UDT structures should contain all the tags required by the equivalent standard device routine.


Name: uValveSC_NC

Description: Properties for a Single Coil (Normally Closed) Valve (UDT)

Members: Data Type Size: 228 byte(s)

Name	Data Type	Style	Description
⊞ Cmd	uValveSC_NC_Cmd		Valve Command Bits
⊞ Sts	uValveSC_NC_Sts		Valve NO Status Bits
⊞ Alm	uValveSC_Alm		Valve Alarm Bits
aOpenProg	BOOL	Decimal	Auto Open Prog.
OpenPulse	BOOL	Decimal	Open Pulse
ClosePulse	BOOL	Decimal	Close Pulse
aOpen	BOOL	Decimal	Auto Open
mOpen	BOOL	Decimal	Manual Open
oOpen	BOOL	Decimal	Open Output
OneShot	DINT	Decimal	One Shot
⊞ FTO_T	TIMER		Failed to Open Tmr
⊞ FTC_T	TIMER		Failed to Close Tmr
⊞ FB_Err_T	TIMER		Feedback Error Tmr
⊞ eAlm_T	TIMER		Extension Alarm Timer
⊞ Open_dT	TIMER		Open Debounce Tmr
⊞ Close_dT	TIMER		Close Debounce Tmr
⊞ Stroke_T	TIMER		Stroke Timer
⊞ AutoReq_T	TIMER		Auto Request Tmr
⊞ ManualReq_T	TIMER		Manual Request Tmr
⊞ OpenReq_T	TIMER		Open Request Tmr
⊞ CloseReq_T	TIMER		Close Request Tmr
⊞ RstOpenNumReq_T	TIMER		Rst No.of Open Tmr

Figure 8.3 - Typical User Defined Type (UDT) for a normally closed single coil valve

Structures created for SHL include:

<u>Standard Device Structures</u>

- Analogue Input Structure
- DOL Motor (Direct-On-Line)
- Single Supply Motor with E300
- Dual Supply Motor with E300
- Standard Single Coil Valve
- Standard Dual Coil Valve
- Switch
- MIV (Main Inlet Valve)

<u>Other Structures</u>

- Sequences
- Duty (Normal/Standby pump duty structure)
- Unit
- Governor
- Generator circuit breaker
- AVR
- Auto synchroniser
- Generator heaters
- Input de-bouncing
- Clock
- PLC diagnostics
- Module diagnostics

4.7.3.2. Designing UDT to Suit HMI Pop-ups

It is important when designing UDT tags that the HMI is taken into consideration. Some OPC data servers provide UDT data transfer optimisation. This means a SCADA system referencing data from a UDT structure will read the whole structure as one chunk even if it only wants one bit from the UDT. This can be a benefit if all the data in the UDT is required by the HMI and a pitfall if only small amounts of data within the UDT is actually required.

To get the best out of the communications optimisation and also make the UDT structure neat and tidy, it is best to group data within the UDT into the following four groups:

- Command Bits - Commands received from the local panel HMI
- Remote Command Bits - Commands received from the remote SCADA system
- Status Bits - Information that will be displayed on a HMI pop-up window representing the object
- Misc Tags – Miscellaneous tags used by the PLC logic or standard routines only.

The following diagram shows how a typical tag structure might look like:

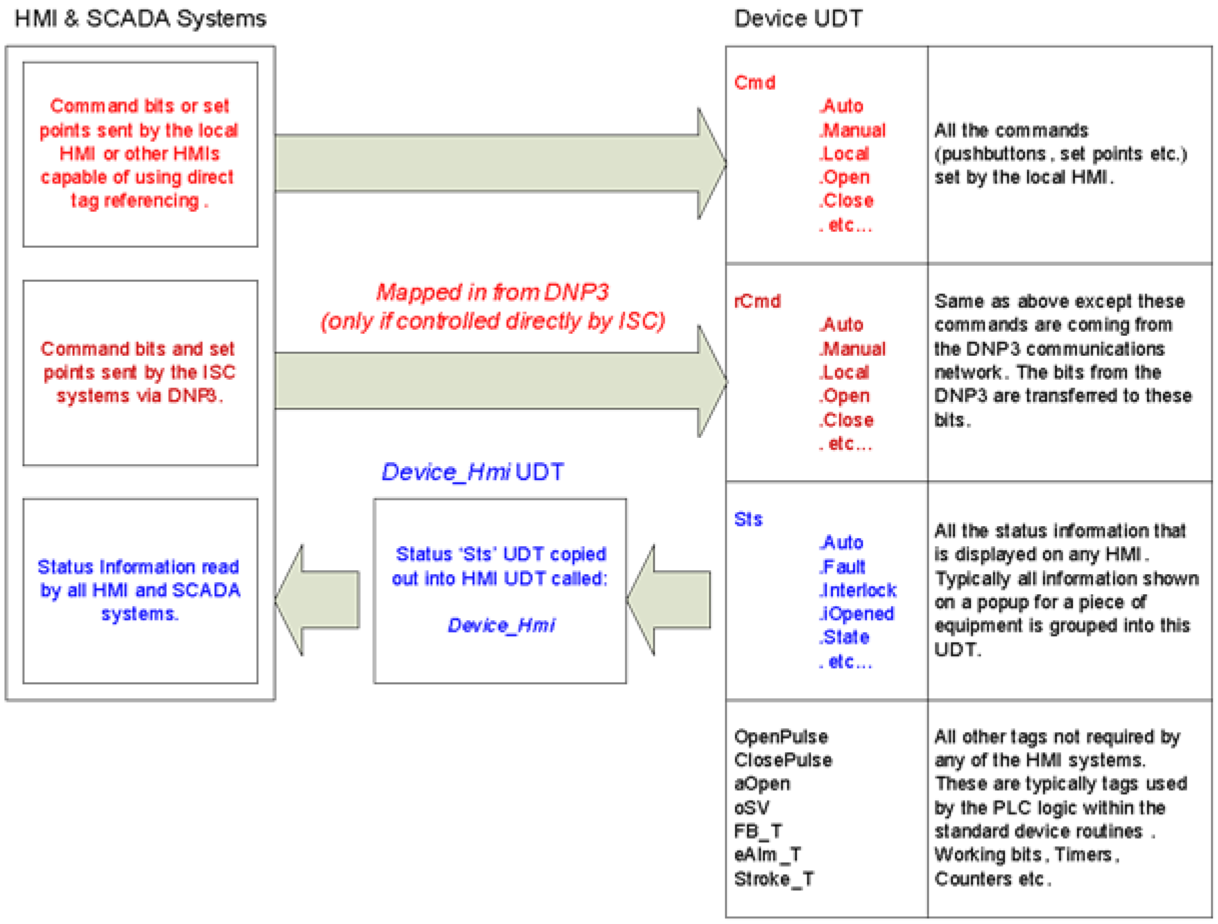

HMI & SCADA Systems
Device UDT
Command bits or set points sent by the local HMI or other HMIs capable of using direct tag referencing.
Command bits and set points sent by the ISC systems via DNP3.
Status Information read by all HMI and SCADA systems.
Mapped in from DNP3 (only if controlled directly by ISC)
Device_Hmi UDT
Status 'Sts' UDT copied out into HMI UDT called: Device_Hmi
Cmd .Auto .Manual .Local .Open .Close . etc...
All the commands (pushbuttons, set points etc.) set by the local HMI.
rCmd .Auto .Manual .Local .Open .Close . etc...
Same as above except these commands are coming from the DNP3 communications network. The bits from the DNP3 are transferred to these bits.
Sts .Auto .Fault .Interlock .iOpened .State . etc...
All the status information that is displayed on any HMI. Typically all information shown on a popup for a piece of equipment is grouped into this UDT.
OpenPulse ClosePulse aOpen oSV FB_T eAlm_T Stroke_T
All other tags not required by any of the HMI systems. These are typically tags used by the PLC logic within the standard device routines. Working bits, Timers, Counters etc.

Figure 8.4 - Typical structure of a User Defined Tag (UDT) with HMI interfacing

From the diagrams it can be seen that the command bits written into the UDT by the HMI, are grouped together into structures called “Cmd” (direct commands) and “rCmd” (remote commands) within the object structure. The “rCmd” group of commands are for the SCADA systems using DNP3 or other protocols not native to the ControlLogix platform. All status bits and other information read by the HMI are grouped into another structure called “Sts” (Status).

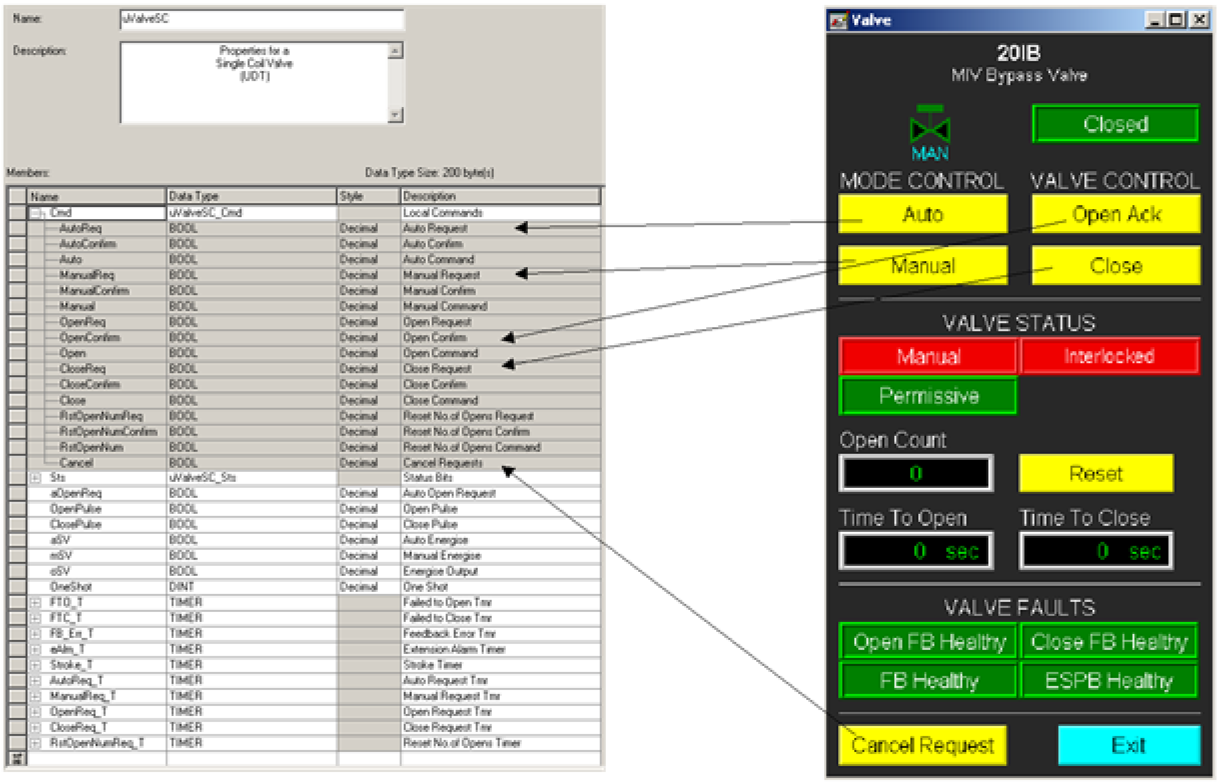
If an OPC data server is used in a SCADA system that supports UDT optimisation, the “Sts” structure within the device (object) UDT can easily be copied out into a separate HMI tag (usually named “*Device_Hmi*”) that the HMI can read optimally, without reading all the other peripheral tags such as timers, counters etc. Hence the “Sts” structure should be designed to include all the contents of the corresponding HMI device pop-up.

Example: A command tag (written into by the HMI) might look like: **_20GQ.Cmd.Open**

A status tag (read by the HMI) might look like: **_20GQ.Sts.Auto**

If UDT optimisation is supported then the tag preferably read by the HMI will look like: **_20GQ_Hmi.Auto**

Standard user defined type structures are detailed in Appendix A – Standard User Defined Types.

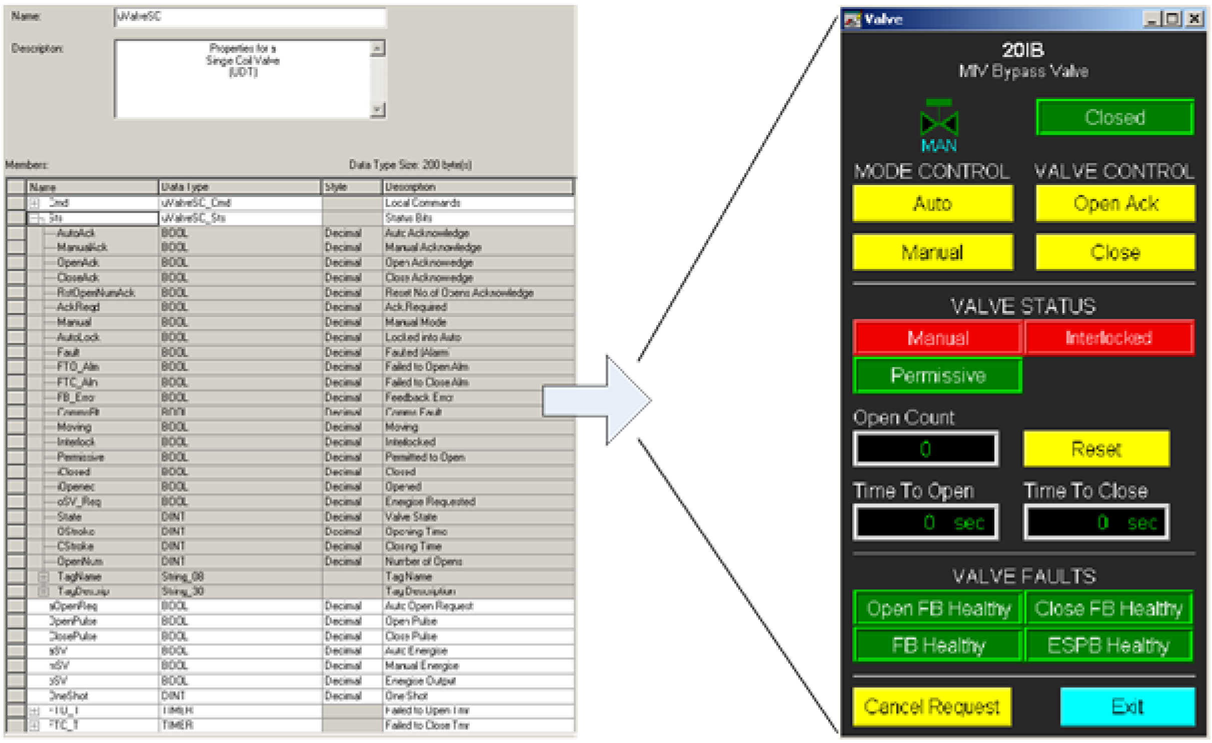

Name: uValveSC

Description: Properties for a Single Coil Valve (UDT)

Members: Data Type Size: 200 byte(s)

Name	Data Type	Style	Description
Cmd	uValveSC_Cmd		Local Commands
AutoReq	BOOL	Decimal	Auto Request
AutoConfirm	BOOL	Decimal	Auto Confirm
Auto	BOOL	Decimal	Auto Command
ManualReq	BOOL	Decimal	Manual Request
ManualConfirm	BOOL	Decimal	Manual Confirm
Manual	BOOL	Decimal	Manual Command
OpenReq	BOOL	Decimal	Open Request
OpenConfirm	BOOL	Decimal	Open Confirm
Open	BOOL	Decimal	Open Command
CloseReq	BOOL	Decimal	Close Request
CloseConfirm	BOOL	Decimal	Close Confirm
Close	BOOL	Decimal	Close Command
RstOpenNumReq	BOOL	Decimal	Reset No.of Opens Request
RstOpenNumConfirm	BOOL	Decimal	Reset No.of Opens Confirm
RstOpenNum	BOOL	Decimal	Reset No.of Opens Command
Cancel	BOOL	Decimal	Cancel Requests
Sts	uValveSC_Sts		Status Bits
aOpenReq	BOOL	Decimal	Auto Open Request
OpenPulse	BOOL	Decimal	Open Pulse
ClosePulse	BOOL	Decimal	Close Pulse
aSV	BOOL	Decimal	Auto Energise
mSV	BOOL	Decimal	Manual Energise
oSV	BOOL	Decimal	Energise Output
OneShot	DINT	Decimal	One Shot
FTO_T	TIMER		Failed to Open Tmr
FTC_T	TIMER		Failed to Close Tmr
FB_Err_T	TIMER		Feedback Error Tmr
eAlm_T	TIMER		Extension Alarm Timer
Stroke_T	TIMER		Stroke Timer
AutoReq_T	TIMER		Auto Request Tmr
ManualReq_T	TIMER		Manual Request Tmr
OpenReq_T	TIMER		Open Request Tmr
CloseReq_T	TIMER		Close Request Tmr
RstOpenNumReq_T	TIMER		Reset No.of Opens Timer

Figure 8.5 - A popup setting command bits within the "Cmd" UDT


Name: uValveSC

Description: Properties for a Single Coil Valve (UDT)

Members: Data Type Size: 200 byte(s)

Name	Data Type	Style	Description
Cmd	uValveSC_Cmd		Local Commands
Sts	uValveSC_Sts		Status Bits
AutoAck	BOOL	Decimal	Auto Acknowledge
ManualAck	BOOL	Decimal	Manual Acknowledge
OpenAck	BOOL	Decimal	Open Acknowledge
CloseAck	BOOL	Decimal	Close Acknowledge
RstOpenNumAck	BOOL	Decimal	Reset No.of Opens Acknowledge
AckReqd	BOOL	Decimal	Ack Required
Manual	BOOL	Decimal	Manual Mode
AutoLock	BOOL	Decimal	Locked into Auto
Fault	BOOL	Decimal	Faulted /Alarm
FTO_Alm	BOOL	Decimal	Failed to Open Alm
FTC_Alm	BOOL	Decimal	Failed to Close Alm
FB_Erro	BOOL	Decimal	Feedback Error
CommsFlt	BOOL	Decimal	Comms Fault
Moving	BOOL	Decimal	Moving
Interlock	BOOL	Decimal	Interlocked
Permissive	BOOL	Decimal	Permitted to Open
iClosed	BOOL	Decimal	Closed
iOpened	BOOL	Decimal	Opened
oSV_Req	BOOL	Decimal	Energise Requested
State	DINT	Decimal	Valve State
OStroke	DINT	Decimal	Opening Time
CStroke	DINT	Decimal	Closing Time
OpenNum	DINT	Decimal	Number of Opens
TagName	String_08		TagName
TagDescrip	String_30		TagDescription
aOpenReq	BOOL	Decimal	Auto Open Request
OpenPulse	BOOL	Decimal	Open Pulse
ClosePulse	BOOL	Decimal	Close Pulse
aSV	BOOL	Decimal	Auto Energise
mSV	BOOL	Decimal	Manual Energise
oSV	BOOL	Decimal	Energise Output
OneShot	DINT	Decimal	One Shot
FTO_T	TIMER		Failed to Open Tmr
FTC_T	TIMER		Failed to Close Tmr

Figure 8.6 - A popup reading status bits from the "Sts" UDT

4.7.4. Addressing Data in MicroLogix Controllers

MicroLogix programs are created in RSLogix 500 and use Data Files to represent memory areas preallocated for storage of tag data. Data files represent a group of addressable elements of a like type (e.g. Binary, Integer, Float, Timer, etc.). Logic is developed using address references to individual data file members (e.g. B3:0/9) as opposed to a tag name.

4.8. General Guidelines

4.8.1. System Overhead Percentage

The system overhead time-slice specifies the percentage of continuous task execution time that is devoted to communication and background functions. System overhead functions include:

- Communicating with programming and HMI devices (such as RSLogix 5000 software)
- Responding to messages
- Sending messages
- Serial port message and instruction processing

The controller performs system overhead functions for up to 1ms at a time. If the controller completes the overhead functions in less than 1ms, it resumes the continuous task. The following chart compares a continuous and periodic task.

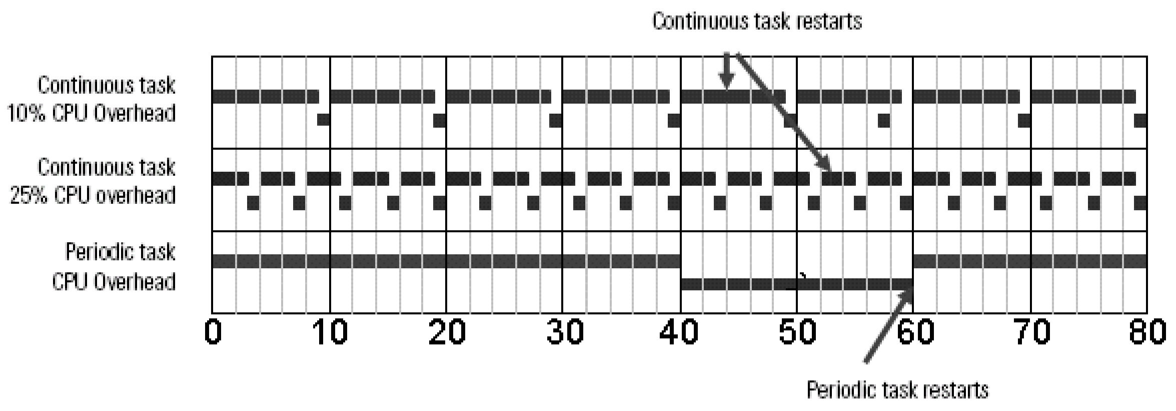


Figure 9.1 Comparison between a continuous and periodic task for communication processing

Increasing the system overhead time-slice percentage decreases execution time for the continuous task while it increases communications performance.

Increasing the system overhead time-slice percentage also increases the amount of time it takes to execute a continuous task - increasing the overall scan time.

4.8.1.1. Setting the Overhead Time Slice Percentage

When creating a new PLC application the time-slice should be set to 50% to begin with and then lowered by 5% at a time until both the continuous task scan time and HMI responsiveness is acceptable.

When significant modifications to a PLC application are required, the time-slice may need to be adjusted to maintain an acceptable continuous task scan time and HMI responsiveness.

4.8.2. Repetitive Routines

ControlLogix controllers allow for the development and execution of standard routines using a number of different methods. Refer to 'Programming Methods' in section 2 of 'Logix5000 Controllers Design Considerations'.

The three methods described are:

- Inline duplication
- Indexed Routine
- Buffered Routine

SHL standard is to use inline duplication due to the ease of maintenance and speed of execution. Inline duplication involves writing multiple copies of the code with different tag references.

Buffered routines may be used for simple, well-defined modules such analogue input processing.

4.8.3. HMI Command Acknowledgement

HMI command requests by a user to the controller should first set a request bit. If the request is permitted by the PLC logic an acknowledgement should then be displayed on the HMI. This acknowledgement should then be separately confirmed by the operator via a second button press. Only once the command has been accepted and acknowledged by the user should the controller execute the command. The controller should then reset the command request bit automatically.

Logic should also be implemented in the controller to time out HMI requests after a short period of time. HMI requests should also be cancellable by a user by a separate button on the HMI interface.

4.8.4. Alarms

All alarm bits for SCADA system alarm summaries shall be grouped together within arrays of DINT tags. For the sake of neatness, a separate alarm array could be created for each device type.

Alarm bits are to be set "true" when the alarm condition exists.

Analogue high and low limit alarms are to include a dead band and filtering functions to minimise unnecessary alarms.

Alarms should be configured for:

- Communication failures
- Watch dog timers
- Low battery
- Motor or drive status discrepancy (e.g. should be running, but is stopped)
- Valve status discrepancy (e.g. should be open, but is not open)
- Analogue high and high-high limits
- Analogue low and low-low limits
- Process inputs and conditions nominated as alarms

4.8.5. Signal Conventions

Use positive logic. Inputs, bits, outputs and HMI commands should be logically "true" (energised) to cause the action. The only exceptions are some digital inputs that protect the plant equipment should be made "fail-safe" in the field. These inputs should be logically "false" (de-energised) to cause an action such as an alarm, process stoppage or shut down.

Variation of an analogue value from low to high should correspond to the transition from closed to open for a modulating valve, and low speed to high speed for a variable speed motor.

Examples:

<u>Digital Inputs:</u>

● LSL=1	when level is low
● LSH=1	when level is high
● ZSC=1	when valve is closed
● ZSO=1	when valve is opened
● FS=1	when adequate flow is not detected
● PSH=1	when pressure is high
● Stop Button=1	when pressed
● Start Button=1	when pressed

Digital Outputs:

● 1 = Action	(eg. Run, Open)
● 0 = No Action	(eg. Stop, Close)

Analogue Inputs 4-20mA:

Always provide minimum and maximum engineering scaling range equating to 4 and 20mA respectively.

● 4mA	= minimum engineering range
● 20mA	= maximum engineering range
● Less than 3mA	= Under-range fault
● Greater than 20.5mA	= Over-range fault

Analogue Outputs 4-20mA:

Always provide minimum and maximum engineering scaling range equating to 4 and 20mA respectively.

● 4mA	= minimum engineering range
● 20mA	= maximum engineering range

Variation of an analogue value from 'low to high' should correspond to 'closed to opened' for a modulating valve and 'low speed to high speed' for a variable speed motor.

Flow transmitter pulses:

Provide value and units of volume or mass indicated by each pulse. e.g. 10 litre / pulse

Configure to be less than the maximum frequency capable by the digital input card (say 100Hz for 1756-IB16)

4.8.6. Analogue Scaling and Engineering Units

If possible, data should be scaled to engineering units in the I/O module. This allows best use of the producer/consumer model and avoids duplication of scaling.

Use the same units for the same variable type where possible. All engineering units shall be based on the metric system and the table below.

Engineering units and ranges to be used are as follows:

Variable	Engineering Units	Units Abbreviation	Comments
Current	kiloamps	kA	Only where unit full rating is greater than 1 kA. Else use amps.
Flow	litres/second	l/s	For cooling water flow, oil flow etc.
Flow (water)	cubic metres per second	cumecs	For water through a generator or from a dam
Frequency	hertz	Hz	
Head	metres water	m	Water head is always in metres
Length	metres or millimetres	m or mm	Millimetres used only for small values i.e. less than 0.5 metres. Centimetres are never to be used.
Level	metres RL	metres	Metres with respect to Snowy sea level datum (slightly different to Australian standard datum).

(Water)			
Level (Not water)	metres or millimetres	m or mm	Millimetres used only for small values i.e. less than 0.5 metres. Centimetres are never to be used.
Power (Reactive)	Megavolt Amps Reactive	MVAr	Small loads (less than 1 MVAr) to use kVAr.
Power (Real)	Megawatt	MW	Small motors (less than 1 MW) will use kW.
Pressure	kiloPascals megaPascals metres H_2O	kPa MPa m H_2O	High pressures (such as Tumut 1 governor – 16 MPa) will use MPa. Almost all items in kPa.
Speed	revolutions per minute	rpm	
Temperature	degrees Celsius	°C	
Volts	kiloVolts	kV	Only for systems greater than 1 kV. Other systems use volts.

4.8.7. Patterns and Systems

Names, numbers, addresses, files, structures etc. should be allocated in an easy to recognise pattern or system. Equipment names shown in P&ID diagrams should be used as the basis for naming tags and routines in logic associated with the equipment. This enhances readability and understanding by others. In addition, it introduces a level of information redundancy allowing errors to be more easily detected

4.8.8. Arrangement

Where possible, logic should be intelligently grouped such that related items are located near to each other. For example, alarm generation logic for a piece of equipment should be kept close to the control logic for that equipment.

4.8.9. Program Initialisation

Pointers and variables should be initiated before their use.

Initialise timer presets, if practically possible, especially for timers used by sequences.

Global constant values, sequence timers and limits should be initialised in a common constants routine at the beginning of the logic.

Limits for analogue values should be initialised in analogue processing routines.

Individual timers for standard drive and valve modules should be initialised in the logic related to the drive or valve.

Any latched bits that form part of sequence control, should be unlatched during processor power-up

4.8.10. Output Logic

A specific device Output shall only be written to once throughout the entire coded logic, i.e. multiple usage of the same output is not acceptable.

4.8.11. Logic Complexity

Logic should be easy to read and understand. Do not complicate logic just to save a few rungs.

Instructions such as JMP and MCR should be avoided where possible as they make programs difficult to follow.

Instructions such as latches should be used sparingly since a condition may exist whereby the latch is not unlatched when it needs to be. Using a soft latch is preferred and achieves the same functionality.

If using latched/unlatched bits, try to keep them in the same rung or at least the neighbouring rung. The unlatch coil should follow the latch coil, to reset the coil in the event both conditions are active at the same time.

High level instructions, although powerful, are difficult to "watch work", but they may be the "best-tool-for-this-job". These instructions, when employed in portions of a program likely to be used often for troubleshooting, must be clearly documented. Some "dummy" diagnostic rungs using the ‘NOP’ instruction may be added to enhance clarity.

Indirect and indexed addressing will be used with discretion as they can make programs difficult to follow and consume more processor bandwidth.

4.8.12. Performance

The performance requirements of the process and network communications should be considered as part of the design.

Performance constraints and bottlenecks should also be identified and addressed to achieve the required performance.

4.8.13. Software Reliability

The development of electronic safety systems has led to the production of standards that provide guidelines on writing reliable software. Relevant guidelines are included in Appendix B – Standard Device Routines.

4.9. Standard Programs

Standard programs should be used for configuration and control of similar equipment, such as motors and valves. This provides the following advantages:

- Easier understanding
- Faster development
- Faster commissioning
- Easier fault finding and maintenance

All equipment should be placed into the corresponding program to make it easier for the programmer or maintainer to locate the logic in future. If a valve has problems, the logic for it can easily be found within the valve program as long as the programmer knows the valve name.

Each standard program should also have an alarm summary routine (AlarmSum) that is responsible for mapping any alarms generated within the standard program to the HMI alarm array.

Standard programs within a project to be configured for SHL include:

General	- ControlLogix house-keeping
Digital_Inputs	- Digital Input de-bouncing and buffering
Analog_Inputs	- Scaling*, alarm generation and filtering
MotorDOL	- Standard control routines for DOL motors
MotorE300_SS	- Standard control routines for single supply motors with E300
MotorE300_DS	- Standard control routines for dual supply motors with E300
Valves_SC	- Standard control routines for valves with single coil solenoids
Valves_DC	- Standard control routines for valves with dual coil solenoids
Switches	- Standard control routines for electrical switchgear

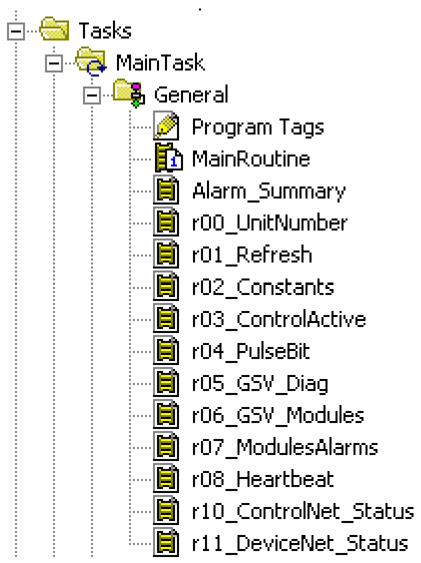
*Only required for non-ControlLogix (1756 family) analogue I/O modules.

Other standard routines may be identified and used during design and development if appropriate.

4.9.1. General Program

This program should be located within the continuous task of the ControlLogix application and is responsible for executing any miscellaneous housekeeping tasks for the controller. Generally any GSV instructions for monitoring hardware status, processor heartbeats, inter-processor tag mapping and messaging, parameter definitions (Constants) and hardware alarming are executed in this program.

The following figure shows typical routines that might be used in the General Program. The following sections provide detail of typical functionality that may be included in each of the routines.



```
Tasks
  MainTask
    General
      Program Tags
      MainRoutine
      Alarm_Summary
      r00_UnitNumber
      r01_Refresh
      r02_Constants
      r03_ControlActive
      r04_PulseBit
      r05_GSV_Diag
      r06_GSV_Modules
      r07_ModulesAlarms
      r08_Heartbeat
      r10_ControlNet_Status
      r11_DeviceNet_Status
```

Figure 10.1 - Routines found in the 'General' program

4.9.1.1. Main Routine

This routine calls all other sub-routines in that program via a jump-to-subroutine (JSR) instruction.

4.9.1.2. Alarm Summary Routine

This routine maps any processor or hardware related alarms to the HMI alarm array

4.9.1.3. Unit Number

This routine is responsible for identifying what plant the PLC belongs to. It is important for applications where there are identical plants and the same program is used for each plant. For instance all the units within the same power station will have the same program loaded into each PLC but the PLC will need to identify exactly which unit it’s controlling so that it can message to the correct group transformer etc. Identification must be done by a combination of:

- Hardwiring three digital inputs into the PLC’s local digital input card, and
- Checking and verifying the Ethernet module IP address.

4.9.1.4. Refresh Routine

This routine generates a refresh pulse every so often so that timer preset values and other set points may be refreshed to their required values. This is only there to increase the bandwidth of the processor by relieving it from loading the same constant values every PLC scan.

4.9.1.5. Constants Routine

The use of tag constants throughout the PLC program simplifies the configuration and maintenance of common parameters used throughout the application. A constant value used many times in a PLC program can then be easily changed from one point of the application.

4.9.1.6. Control Active Routine

This routine is responsible for setting the control active bit. When the PLC first starts up it checks the current status of the plant before it assumes control. It allows the PLC control sequences time to align themselves with the plant.

4.9.1.7. Pulse Bit Routine

This routine creates a bit that pulses on for one second and off for one second. This pulse bit “gPulseOn” is useful for flashing lights etc.

4.9.1.8. GSV Diagnostics Routine

This routine executes any GSV commands to obtain status information from the PLC processor. Items such as major and minor fault codes, battery alarms, wall clock, scan times, key switch mode etc. are extracted from here.

4.9.1.9. GSV Modules Routine

This routine executes any GSV commands related to the module status information.

4.9.1.10. Module Alarms Routine

All module related alarms are generated here.

4.9.1.11. Heartbeat Routine

This routine generates a heartbeat for other PLCs or HMI to monitor

4.9.1.12. Timestamp Routine

This routine creates pulses every 10 minutes for standard motor start count logic. It also creates pulses for other time periods throughout the day required by other logic.

4.9.1.13. Communications Status Routines

These routines check for any error on connected communications networks (e.g. EtherNet/IP). This may vary depending on the communication network, but should include the following where possible:

- Module status
- Port or channel status
- Ring status
- Network node statuses for each connected node

4.9.1.14. Set PLC Clock Routine

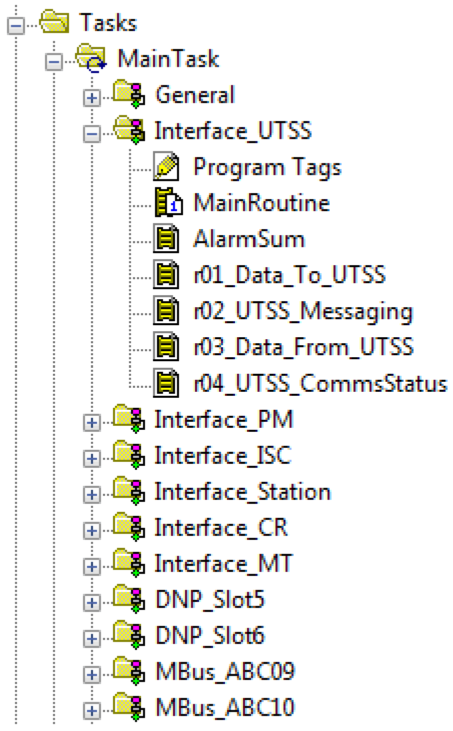
This routine synchronises the PLC's real time clock with a time provided by a IEEE1588 PTP network time clock or DNP3 SCADA master (if no local time source available).

4.9.1.15. Clock Stamp Sub-Routine

This sub-routine creates a date and time string for logging of alarms and events based on the current PLC clock time. This sub-routine may be called by other logic when a timestamp string is required.

4.9.2. Interface Programs

These programs located within the continuous task of the ControlLogix application are responsible for managing connections to communications devices, and mapping and transferring information between various interfaces. Each interface should be separated into its own program.



```
Tasks
  MainTask
    General
    Interface_UTSS
      Program Tags
      MainRoutine
      AlarmSum
      r01_Data_To_UTSS
      r02_UTSS_Messaging
      r03_Data_From_UTSS
      r04_UTSS_CommsStatus
    Interface_PM
    Interface_ISC
    Interface_Station
    Interface_CR
    Interface_MT
    DNP_Slot5
    DNP_Slot6
    MBus_ABC09
    MBus_ABC10
```

Figure 10.2 - Interface programs and example routines

The routines included in each interface program will vary depending on the particular communications protocol, but will typically including the following:

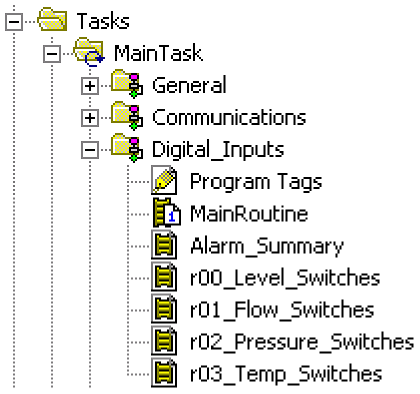
- Mapping and preparation of data to be sent to the remote device(s)
- Establishing and managing the connection to the remote device(s)
- Mapping of data received from the remote device(s)
- Monitoring the communications status of the link and device(s)

4.9.3. Digital Inputs Program

This program located within the continuous task of the ControlLogix application is responsible for de-bouncing and buffering any digital inputs before they are used throughout the rest of the program. De-bouncing is generally used for process inputs that generate alarms. Buffering is only required for inputs that are not part of a greater structure such as motors or valves. Flow, pressure and level inputs are generally the ones that require buffering and de-bouncing.

Buffering is implemented because inputs are transferred into the PLC asynchronously to the execution of the logic. Hence an input can change state in the middle of a program scan which might not be desirable in some instances.

The inputs are processed in groups according to functionality. Each routine represents a type of process input e.g. level switches. Grouping logic into routines should make it easier to locate the input and make the logic tidier.



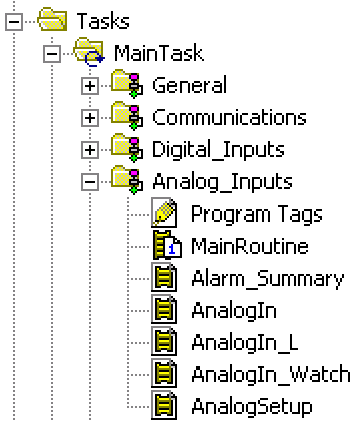
```
Tasks
  MainTask
    General
    Communications
    Digital_Inputs
      Program Tags
      MainRoutine
      Alarm_Summary
      r00_Level_Switches
      r01_Flow_Switches
      r02_Pressure_Switches
      r03_Temp_Switches
```

Figure 10.3 - Typical routines found within the 'Digital_Inputs' program

4.9.4. Analogue Inputs Program

This program located within the continuous task of the ControlLogix application is responsible for configuring and mapping the analogue tag structures and generating and filtering all analogue related alarms.

A buffered routine is used to process the analogue alarm logic. This means that there is only one routine to maintain. The actual analogue UDT is passed into the routine via a buffer analogue UDT (SBR). The routine then executes using the buffer UDT and passes the results back to the original analogue UDT when complete (RET). (See section "Programming Methods" in reference manual "Logix5000 Controllers Design Considerations" for an example).



```
Tasks
  MainTask
    General
    Communications
    Digital_Inputs
    Analog_Inputs
      Program Tags
      MainRoutine
      Alarm_Summary
      AnalogIn
      AnalogIn_L
      AnalogIn_Watch
      AnalogSetup
```

Figure 10.4 - Typical routines found within the 'Analog_Inputs' program

4.9.4.1. Alarm Summary Routine

This routine maps the generated analogue alarms to the HMI analogue alarm array.

4.9.4.2. Analogue Setup Routine

This routine configures the analogue input structure with the following parameters:

- Module fault mapping
- Minimum and maximum engineering units mapping
- Alarm dead-band value
- Whether LL and HH alarm limits are used
- Whether L and LL alarm limits are used
- Whether H and HH alarm limits are used
- Whether rate of change (ROC) alarms are used
- Analogue alarm debounce and extend (hold) timer values
- Under-range, and over-range alarm thresholds

4.9.4.3. AnalogIn Routine

This routine is the standard routine for generating analogue alarms according to the set points changeable from the HMI. The alarms generated in this routine are not latched. Hence when the process variable (PV) moves outside the alarm range, the corresponding alarm is deactivated after a configurable alarm extend time.

This is a buffered routine that is called for each analogue input requiring processing.

If scaling is not performed by the module, then it may be calculated in this routine.

4.9.4.4. AnalogIn_L Routine

This routine is the standard routine for generating analogue alarms exactly as described above. The only difference is that it latches on the alarms. The alarms can only be reset by the corresponding reset button. It is typically used only in situations where alarms are inhibited by running equipment that is also interlocked by the same alarm.

This is a buffered routine that is called for each analogue input requiring processing where the alarms need to be latched on.

Example: A low flow alarm is inhibited when the pump is not running. When pump starts running for a set period, the low flow alarm logic is enabled. However the pump may have an interlock to stop running if a low flow alarm is detected and hence stop. This causes re-inhibiting the low flow alarm making it difficult for the operator to see what caused the pump to stop.

4.9.4.5. AnalogIn_Watch Routine

This routine is identical to the AnalogIn routine. It is only there for the programmers benefit to troubleshoot analogue alarming. Because the buffering method is used to process all the analogues through the same standard routine, it is difficult to monitor what is happening as the same routine is executed multiple times per scan. This watch routine allows the programmer to divert only the one analogue point of interest to this routine to monitor the logic. When the programmer is finished with watching the analogue, it should be diverted back to the normal routine.

4.9.5. Control Logic and Sequence Programs

These programs contain the control and sequence logic that controls the actions of various devices and equipment within the plant.

The plant in total is broken down into functional areas, tasks or equipment groups. Each of these functional groups will have an individual program that will contain the actual sequence logic specific for that group.

Additional general information on sequences can be found in section 4.11.

4.9.6. Motor DOL Program

This program located within the continuous task of the ControlLogix application is responsible for executing all the standard motor logic for DOL (Direct-On-Line) motors. Each routine within the program represents one DOL motor. The routines are named identically to the actual names of the motors drawn in the P&ID drawings.

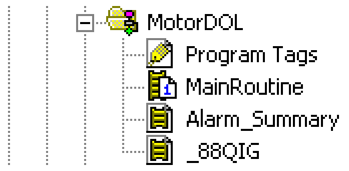


Figure 10.5 Typical DOL motor routines found in the 'MortorDOL' program

The "Alarm_Summary" routine is used to map the motor fault bits to the HMI alarm summary array. Only one fault bit is used per motor. The HMI system can then display a more specific description by opening the popup related to the motor.

4.9.7. Single Supply Motor with E300 Program

This program located within the continuous task of the ControlLogix application is responsible for executing all the standard motor logic for single supply motors with E300. The motors grouped into this program can only be supplied from a single power source. They also have intelligent motor overload relay attached to them (E300) providing motor protection, diagnostics and local control.

Separate routines for duty control (where appropriate) of A/B motor pairs are also found within this standard program.

EtherNet/IP is used to communicate to the E300 overload relays.

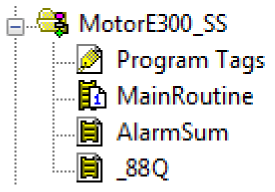


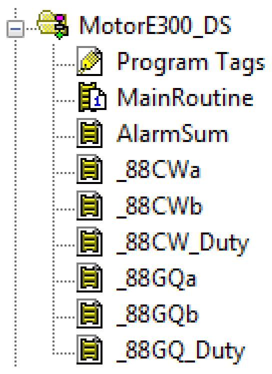
Figure 10.6 Typical single supply motor routines found in the 'MotorE300_SS' program

4.9.8. Dual Supply Motor with E300 Program

This program located within the continuous task of the ControlLogix application is responsible for executing all the standard motor logic for dual supply motors with E300. The motors grouped into this program can be supplied from either one of two power sources. They also have intelligent motor E300 overload relays attached to them providing motor protection, diagnostics and local control.

Separate routines for duty control (where appropriate) are also found within this standard program.

Ethernet/IP is used to communicate to the E300 overload relays.

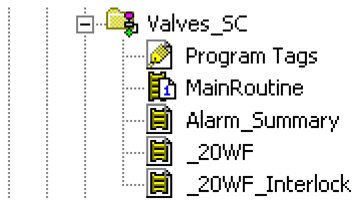


```
MotorE300_DS
    Program Tags
    MainRoutine
    AlarmSum
    _88CWa
    _88CWb
    _88CW_Duty
    _88GQa
    _88GQb
    _88GQ_Duty
```

Figure 10.7 Typical dual supply motor routines found in the 'MotorE300_DS' program

4.9.9. Single Coil Valves Program

This program located within the continuous task of the ControlLogix application is responsible for executing all the standard valve logic for single coil valves. Each routine within the program represents one valve. The routines are named identically to the actual names of the valves drawn in the P&ID drawings.

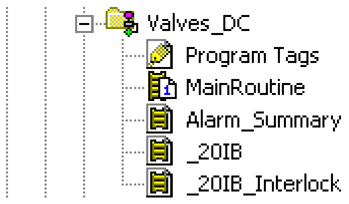


```
Valves_SC
    Program Tags
    MainRoutine
    Alarm_Summary
    _20WF
    _20WF_Interlock
```

Figure 10.8 Typical valve routines found in the 'Valves_SC' program

4.9.10. Dual Coil Valves Program

This program located within the continuous task of the ControlLogix application is responsible for executing all the standard valve logic for dual coil valves. Each routine within the program represents one valve. The routines are named identically to the actual names of the valves drawn in the P&ID drawings.



```
Valves_DC
    Program Tags
    MainRoutine
    Alarm_Summary
    _20IB
    _20IB_Interlock
```

Figure 10.9 Typical valve routines found in the 'Valves_DC' program

4.10. Standard Instrument and Device Routines

Using a standard methodology to develop control system programs provides many benefits for the programmer and the end user. An integral part of this methodology is to make use of standard modular code that controls all the features and modes of an instrument or piece of plant equipment. For example, a pump can be described as an object that is required by the plant to perform a function that is critical for the plant to operate continuously and perform the overall task of making product.

The pump object would be represented by a standard piece of code that could be configurable to suit the environment it is in and perform the necessary tasks required by the system as a whole. The standard code within that object would be the same for all objects of the same type however there would be some defined parameters and settings that could be changed and methods that can be used by an external sequence to make that object unique in its control. The standard logic found within each of the different objects would be thoroughly tested and proven for its reliability and functionality. After this it can be re-used over and over without the need for any further changes.

This section provides specific details for each of the objects that will be used in the Unit Control System.

4.10.1. Reasons for Standard Code

Once standard code is written for all the defined objects within a plant, it can be re-used many times for any object of the same type. It saves time writing bulk of the equipment’s logic, thus providing more time for concentrating on developing control sequences or more difficult or unusual aspects of the plant.

A standardised approach has other benefits besides saving time writing initial logic. It also makes it easier to train others, like the end user and other programmers, on how the program works. A generic write-up such as this explaining the logic can be used as a training aid. Other programmers will find it easier to lend a hand and be more proficient when they can acclaim themselves to the equipment faster. The training of new employees is faster and you can convey how you want the programs written in a consistent manner.

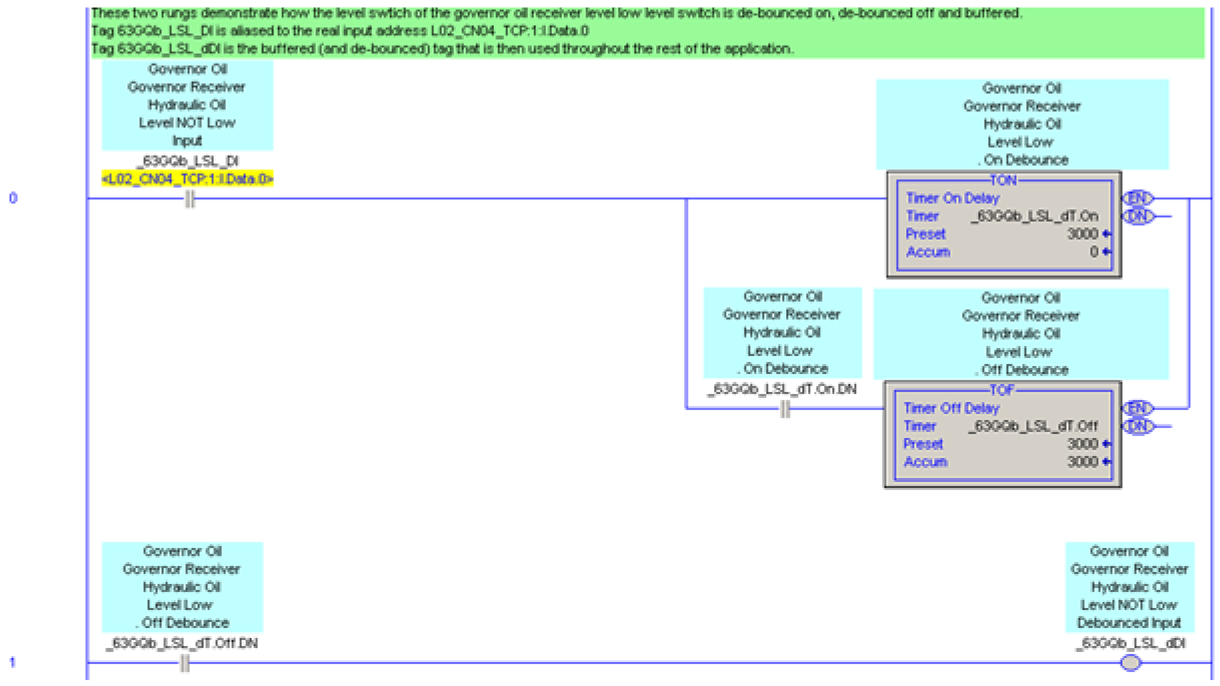
The reasons for using standard logic can be summaries as follows:

- Proven and tested logic that is safe to operate
- Control sequences can be created independently from the standard control modules
- Minimises commissioning time
- Minimises downtime when fault finding
- Makes programs easy to change
- Makes programs easy to create

The following sections describe the typical standardised routines used in SHL PLC applications.

4.10.2. Digital Inputs

This logic debounces each digital input using a debounce timer. If the input is debounced and still set then the input may be optionally held on for an extended period using an extension (hold) timer. This is highly recommended for those inputs that are used for alarming or trips. Holding alarm inputs on, assists in the HMI and SCADA systems to catch fleeting alarms.



These two rungs demonstrate how the level swtich of the governor oil receiver level low level switch is de-bounced on, de-bounced off and buffered.
Tag 63GGb_LSL_DI is aliased to the real input address L02_CN04_TCP:1:I.Data.0
Tag 63GGb_LSL_dDI is the buffered (and de-bounced) tag that is then used throughout the rest of the application.

0
Governor Oil Governor Receiver Hydraulic Oil Level NOT Low Input
_63GGb_LSL_DI
<L02_CN04_TCP:1:I.Data.0>
Governor Oil Governor Receiver Hydraulic Oil Level Low . On Debounce
TON
Timer On Delay
Timer _63GGb_LSL_dT.On
Preset 3000
Accum 0
EN
DN
Governor Oil Governor Receiver Hydraulic Oil Level Low . On Debounce
_63GGb_LSL_dT.On.DN
Governor Oil Governor Receiver Hydraulic Oil Level Low . Off Debounce
TOF
Timer Off Delay
Timer _63GGb_LSL_dT.Off
Preset 3000
Accum 3000
EN
DN
1
Governor Oil Governor Receiver Hydraulic Oil Level Low . Off Debounce
_63GGb_LSL_dT.Off.DN
Governor Oil Governor Receiver Hydraulic Oil Level NOT Low Debounced Input
_63GGb_LSL_dDI

Figure 11.1 How an input can be debounced on/off and buffered

4.10.3. Analogue Inputs

4.10.3.1. Analogue Input Configuration

Analogue inputs are configured from a “Setup” routine located within the “Analog_Inputs” program. In this routine the following configuration settings are mapped into the analogue input UDT for each analogue input signal:

- Minimum engineering value
- Maximum engineering value
- Alarm dead-band
- Enable both high-high and low-low alarms
- Enable low and low-low alarms
- Enable high and high-high alarms
- Configure alarm debounce times
- Configure alarm extend (hold) times to catch any fleeting alarms
- Configure under-range set point
- Configure over-range set point

4.10.3.2. Analogue Input Processing

After they are configured, Analogue Inputs are processed in a separate “Process” routine in the “Analog_Inputs” program. The main purpose of this logic is to process analogue alarm conditions.

Analogue inputs are prepared for processing by mapping the current field data value from the controller input tags to the analogue input UDTs. Communications and module health status are also mapped into the UDT. The UDT is then passed to a standard, buffered sub-routine where the following occurs:

HMI new alarm set point verification

- Under-range and over-range alarm processing
- Rate of change alarm processing
- Low and low-low alarm processing
- High and high-high alarm processing
- Hardware (bad-quality) alarm generation (for alarm summary)
- Process variable alarm generation (for alarm summary)

Where possible, scaling to engineering units should be done within the analogue input module.

Recent example projects containing standard analogue input logic routines are available on request.

4.10.4. Standard DOL Motor Logic

The main purpose for the standard DOL motor logic is to process motor alarm conditions, HMI mode changes (auto, manual and local), auto and manual control, and generate motor status. DOL motors are only supplied by the CSB power source.

The standard DOL motor routine is typically made up of three sections:

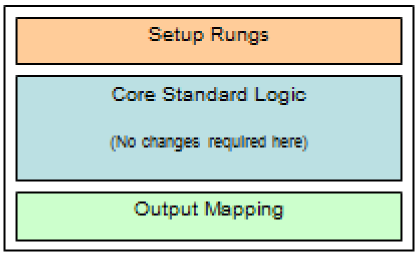


Figure 11.2 Structure of a typical standard motor routine

Standard DOL motor logic should be replicated using inline duplication. The entire routine should be copied and named with the equipment name, search-and-replace used to update the base equipment UDT tag, and setup rungs customised for the new equipment.

4.10.4.1. DOL Motor Setup Rungs

The first section contains rungs dedicated to configuring the operation and behaviour of the motor. The following items must be set to configure the DOL motor:

- Map all inputs related to the motor into the motor UDT (contactor feedback, isolator, etc.)
- Enter all emergency stop conditions that should stop the motor
- Enter all interlock conditions that prevent the motor from running
- Enter all supply healthy conditions that are required to run the motor
- Enter all permissive conditions that permit the motor to start
- Enter all conditions that should force the motor into auto mode
- Enter all auto run conditions that make the motor run automatically when the motor is in auto mode
- Enter all conditions that can reset any motor faults
- Configure all timer presets used within the standard motor logic.

4.10.4.2. DOL Motor Core Standard Logic

The second section is the actual standard motor logic that should not be changed. This standard logic includes the following functions:

- Generate “Failed-to-Start” and “Failed-to-Stop” alarms
- Generate a “Fault” bit (for alarm summary)
- Logic to change modes (Auto, Manual and Local) from the HMI
- Motor restart inhibit logic based on the number of allowable starts per hour
- Bump-less transfer of motor state from auto to manual to local and back to manual
- Manual start and stop commands from the HMI
- Motor running time totaliser
- Motor number of starts accumulator
- Reset alarms from the HMI
- Motor status (for HMI multi-state indicator)

4.10.4.3. DOL Motor Output Mapping

The last rung is typically assigned for mapping any outputs from the motor UDT to the real output tags. The following output tag is mapped out of the UDT:

- Run motor from the CSB supply

Recent example projects containing standard DOL motor logic routines are available on request.

4.10.5. Standard Single Supply Motor with E300 Logic

The structure of the single supply E300 motor routines are basically the same as the DOL motor routines except that the motors have some extra parameters, diagnostics, and outputs provided by the E300.

The standard single supply E300 motor routine is typically made up of three sections.

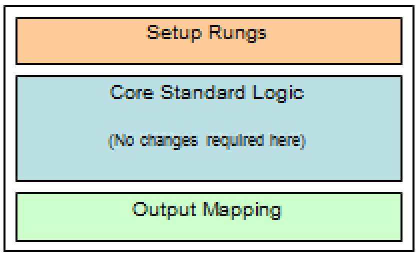


Figure 11.3 Structure of a typical standard single supply ES+ motor routine

Standard E300 motor logic should be replicated using inline duplication. The entire routine should be copied and named with the equipment name, search-and-replace used to update the base equipment UDT tag, and setup rungs customised for the new equipment.

4.10.5.1. Single Supply E300 Motor Setup Rungs

The first section contains rungs dedicated to configuring the operation and behaviour of the motor. The following items must be set to configure the single supply motor with E300:

Map all inputs related to the motor into the motor UDT (contactor feedback, isolator, E300 parameter diagnostics, etc.)

- Enter all emergency stop conditions that should stop the motor
- Enter all interlock conditions that prevent the motor from running
- Enter all supply healthy conditions that are required to run the motor
- Enter all permissive conditions that permit the motor to start
- Enter all conditions that should force the motor into auto mode
- Enter all auto run conditions that make the motor run automatically when the motor is in auto mode
- Enter all conditions that can reset any motor faults or trips
- Configure all timer presets used within the standard motor logic.

4.10.5.2. Single Supply E300 Motor Setup Rungs

The second section is the actual standard motor logic that should not be changed. This standard logic includes the following functions:

- Determine if motor is running based on contactor feedback and detection of current (amps)
- Generate “Failed-to-Start” and “Failed-to-Stop” alarms
- Generate “Warning” and “Fault” bits (for alarm summary)
- Logic to change modes (Auto, Manual and Local) from the HMI
- Motor restart inhibit logic based on the number of allowable starts
- Bump-less transfer of motor state from auto to manual to local and back to manual
- Manual start and stop commands from the HMI
- Motor running time totaliser
- Motor number of starts accumulator
- Alarm and trip reset logic
- Motor status (for HMI multi-state indicator)

4.10.5.3. Single Supply E300 Motor Output Mapping

The last rung is typically assigned for mapping any outputs from the motor UDT to the real output tags. The following output tags are mapped out of the UDT:

- Run motor from the CSB supply
- Reset E300 trip
- Enable local control of E300 based on HMI mode selection
- Enable or disable E300 setting to run on detection of comms fault

Recent example projects containing standard E300 single supply motor logic routines are available on request.

4.10.6. Standard Dual Supply E300 Motor Logic

The structure of the dual supply E300 motor routines are basically the same as the single supply motor routines except that the motors may be run from either the CSB supply or the USB supply. Some extra parameters, diagnostics, and outputs are also provided by the E300.

The standard dual supply E300 motor routine is typically made up of three sections.

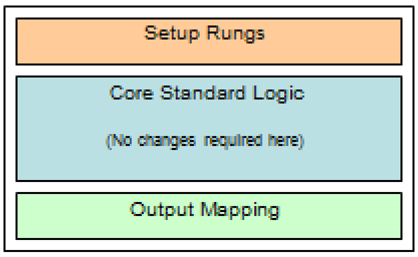


Figure 11.4 Structure of a typical standard dual supply E300 motor routine.

Standard E300 motor logic should be replicated using inline duplication. The entire routine should be copied and named with the equipment name, search-and-replace used to update the base equipment UDT tag, and setup rungs customised for the new equipment.

4.10.6.1. Dual Supply E300 Motor Setup Rungs

The first section contains rungs dedicated to configuring the operation and behaviour of the motor. The following items must be set to configure the dual supply E300 motor:

Map all inputs related to the motor into the motor UDT (contactor feedback, isolator, E300 diagnostic parameters, etc.)

- Enter all emergency stop conditions that should stop the motor
- Enter all interlock conditions that prevent the motor from running
- Enter all supply healthy conditions that are required to run the motor
- Enter all permissive conditions that permit the motor to start
- Enter all conditions that should force the motor into auto mode
- Enter all auto run conditions that make the motor run automatically when the motor is in auto mode
- Enter all conditions that can reset any motor faults or trips
- Configure all timer presets used within the standard motor logic.

4.10.6.2. Dual Supply E300 Motor Core Standard Logic

The second section is the actual standard motor logic that should not be changed. This standard logic includes the following functions:

- Determine if motor is running based on contactor feedback and detection of current (amps)
- Generate “Failed-to-Start” and “Failed-to-Stop” alarms
- Generate “Warning” and “Fault” bits (for alarm summary)
- Logic to change modes (Auto, Manual and Local) from the HMI
- Motor restart inhibit logic based on the number of allowable starts
- Determine which power source to use (CSB or USB)
- Bump-less transfer of motor state from auto to manual to local and back to manual
- Manual start and stop commands from the HMI (CSB and USB)
- Motor running time totaliser
- Motor number of starts accumulator
- Alarm and trip reset logic
- Motor status (for HMI multi-state indicator)

4.10.6.3. Dual Supply E300 Motor Output Mapping

The last rung is typically assigned for mapping any outputs from the motor UDT to the real output tags. The following output tags are mapped out of the UDT:

- Run motor from the CSB supply
- Run motor from the USB supply
- Reset E300 trip
- Enable local control of E300 based on HMI mode selection
- Enable or disable E300 setting to run on detection of comms fault

Recent example projects containing standard E300 dual supply motor logic routines are available on request.

4.10.7. Standard Single Coil Valve Logic

The main purpose for the standard single coil valve logic is to process valve alarm conditions, HMI mode changes (auto and manual), auto and manual control, and generate valve status.

The standard single coil valve routine is also typically made up of three sections.

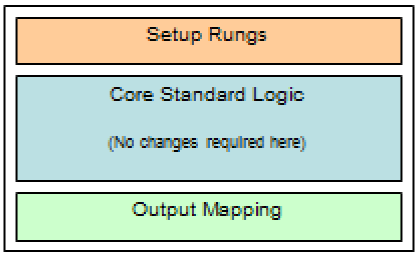


Figure 11.5 Structure of a typical standard single coil valve routine

Standard valve logic should be replicated using inline duplication. The entire routine should be copied and named with the equipment name, search-and-replace used to update the base equipment UDT tag, and setup rungs customised for the new equipment.

4.10.7.1. Single Coil Valve Setup Rungs

The first section contains rungs dedicated to configuring the operation and behaviour of the valve. The following items must be set to configure the valve:

- Map all inputs related to the valve into the valve UDT (position feedback, supply OK, etc.)
- Enter all interlock conditions that prevent the valve from operating
- Enter all supply healthy conditions that are required to operate the valve
- Enter all permissive conditions that permit the valve to operate
- Enter all conditions that should force the valve into auto mode
- Enter all auto run conditions that make the valve operate automatically when the valve is in auto mode
- Enter all conditions that can reset any valve faults
- Configure all timer presets used within the standard valve logic.

4.10.7.2. Single Coil Valve Core Standard Logic

The second section is the actual standard valve logic that should not be changed. This standard logic includes the following functions:

- Generate "Failed-to-Open" and "Failed-to-Close" alarms
- Generate a "Fault" bit (for alarm summary)
- Logic to change modes (Auto and Manual) from the HMI
- Bump-less transfer of valve state from auto to manual
- Manual open and close commands from the HMI
- Measure the transition times (time to open, time to close)
- Number of valve operations accumulator
- Reset alarms from the HMI
- Valve status (for HMI multi-state indicator)

4.10.7.3. Valve Output Mapping

The last rung is typically assigned for mapping any outputs from the valve UDT to the real output tags. The following output tag is mapped out of the UDT:

- Energise valve solenoid

Recent example projects containing standard valve logic routines are available on request.

4.10.8. Standard Dual Coil Valve Logic

The main purpose for the standard dual coil valve logic is to process valve alarm conditions, HMI mode changes (auto and manual), auto and manual control, and generate valve status.

The standard dual coil valve routine is also typically made up of three sections.

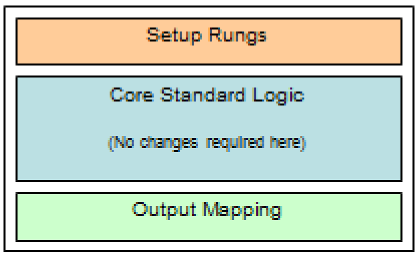


Figure 11.6 Structure of a typical standard dual coil valve routine

Standard valve logic should be replicated using inline duplication. The entire routine should be copied and named with the equipment name, search-and-replace used to update the base equipment UDT tag, and setup rungs customised for the new equipment.

4.10.8.1. Dual Coil Valve Setup Rungs

The first section contains rungs dedicated to configuring the operation and behaviour of the valve. The following items must be set to configure the valve:

- Map all inputs related to the valve into the valve UDT (position feedback, supply OK, etc.)
- Enter all interlock conditions that prevent the valve from operating
- Enter all supply healthy conditions that are required to operate the valve
- Enter all open permissive conditions that permit the valve to open
- Enter all close permissive conditions that permit the valve to close
- Enter all conditions that should force the valve into auto mode
- Enter all auto run conditions that make the valve open automatically when the valve is in auto mode
- Enter all auto run conditions that make the valve close automatically when the valve is in auto mode
- Enter all conditions that can reset any valve faults
- Configure all timer presets used within the standard valve logic.

4.10.8.2. Dual Coil Valve Core Standard Logic

The second section is the actual standard valve logic that should not be changed. This standard logic includes the following functions:

- Generate “Failed-to-Open” and “Failed-to-Close” alarms
- Generate a “Fault” bit (for alarm summary)
- Logic to change modes (Auto and Manual) from the HMI
- Bump-less transfer of valve state from auto to manual
- Manual open and close commands from the HMI
- Measure the transition times (time to open, time to close)
- Number of valve operations accumulator
- Reset alarms from the HMI
- Valve status (for HMI multi-state indicator)

4.10.8.3. Valve Output Mapping

The last rung is typically assigned for mapping any outputs from the valve UDT to the real output tags. The following output tag is mapped out of the UDT:

- Energise valve open solenoid
- Energise valve close solenoid

Recent example projects containing standard valve logic routines are available on request.

4.11. Sequences

There are some control aspects of the unit that are sequential in nature. For instance the Main Inlet Valve must open first before the governor stop valve is opened or the AVR is started. This sequential process is best controlled by sequence logic that activates and de-activates equipment based on the step in the sequence. Sequences provide high level control of the overall unit function, sub-systems and devices.

Examples of sequences used in SHL PLC applications are:

- Overall Unit Control State Machine – defines and controls the state of the unit plant for reference by other sub-systems
- Unit Initialisation sub-sequence – used to synchronise the main unit control state machine with the physical state of the plant

- Unit Start Sequence – a set of steps and transitions to control a normal unit startup
- Unit Stop Sequence – a set of steps and transitions to control a shutdown of the unit
- Excitation Dynamic Braking – manages the operation of the dynamic braking system
- Excitation Mode Change – manages the interface with the excitation system to change the unit excitation
- Mode Change to Synchronous Condenser (and vice-versa) – manages the change in unit state from generator to Synchronous Condenser and back.

Sequences should be created in separate routines in the program most relevant to their function. Sequences monitor signals from various plant areas and issue controls that should then be used by the standard device routines for the actual control of the equipment.

Refer to section 4.4.9.3 for naming conventions that should be used when creating sequences.

SHL's standard is to use Ladder Logic for sequences.

4.11.1. Sequence Modelling

The following sections describe how to design a sequence, produce documentation that explains exactly how the sequence will work, and convert that into logic.

4.11.1.1. Reasons for Modelling Sequences

In order to produce quality software, solely the application of programming languages and tools does not suffice. In manufacture and construction of products it is essential that preliminary models or blueprints, describing functional and physical features of the product, are built. In software development and production a similar engineering approach should also be used. Software engineering is a technical branch which provides methods and activities for modelling and construction of quality software. Software engineering activities make use of modelling languages and graphical modelling tools.

Relay ladder logic is often written in a non-structured way that makes it hard to analyse and troubleshoot. But this can be remedied via a simple, structured approach using state-chart diagrams. A state-chart diagram is a pictorial representation, that is, a special type of flow chart, of a sequential control process that shows all the possible paths the process can take and the Boolean conditions necessary to go from one state to another.

4.11.1.2. Modelling State / Step Sequences using Function Charts

In process control software, dynamic view of the system is the most important. In order to model the behaviour, function chart diagrams will be used. Function charts are used to describe in clear steps how the program is to operate the plant and equipment. The IEC standard 60848 Ed.2 describes preparation of function charts for control systems.

The steps in the chart are referred to directly in the program for ease of cross reference and documentation.

The code generated is called the state logic or state machine. Its purpose is to resolve all the current physical inputs and operator inputs into a single step number for each sequence.

This is achieved by generating an expression for each step which can be represented as follows:

```
If (Step.1 is Active AND Transition A is True) Then
    Activate Step.2 AND De-Activate Step.1
Else If
End If
```

The first step in this approach to structured programming is to prepare a verbal description of what you what to do. From this description, a state-chart diagram is made by determining the unique state through which the process will go. Any time there is a change in output conditions, a new state needs to be shown. The third step is to connect these states with lines and arrows to indicate the desired directions of change. Finally, add the logical conditions, written in Boolean form, which will cause a change from one state to the other.

4.11.1.3. State-Chart Definitions

A state-chart diagram represents a state machine. By documenting all the transitions, a state-chart diagram shows the sequence of states an object goes through during its life.

A function chart comprises of the following items:

Steps

A state represents a condition of an object in which a defined set of rules, policies and actions are executed. Each state is given a name that is unique within the state machine. States are also numbered. The number of the state can be arbitrary but to be able to follow a sequence the numbering should be consecutive according the usual steps taken to produce the required outcome. All sequences start at step zero. Within a sequence all step numbers are unique. The figure below shows how the step numbers and names are written inside the state box.

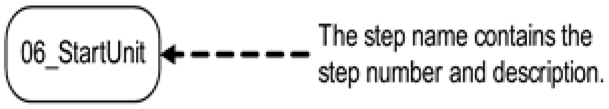


Figure 12.1 A Step symbol. The step name contains the step number and description.

While it's important to choose good, meaningful state names, the name of the state is just an arbitrary label. The real meaning of the state is in the behaviour associated with the state – the actions executed upon entry to the state.

Transitions

Transitions describe what particular events or conditions must happen before the sequence is able to transition to another step. There may be multiple transitions out of a step only if they are caused by different events. There may also be multiple transitions into one step, each caused by the same event or by different events. A transition without a marker and logic is unconditional. A step with an unconditional outgoing transition will only be active for only one program scan.

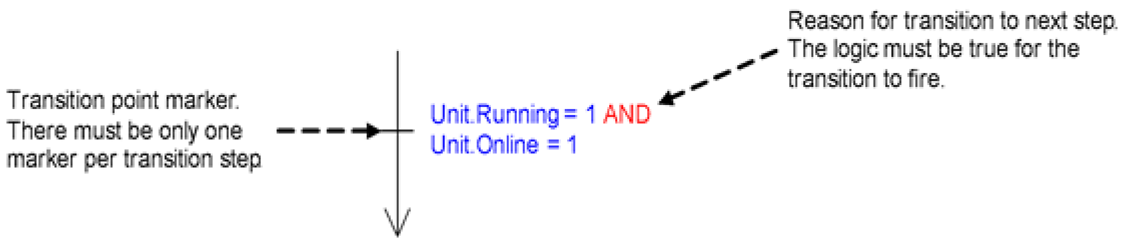
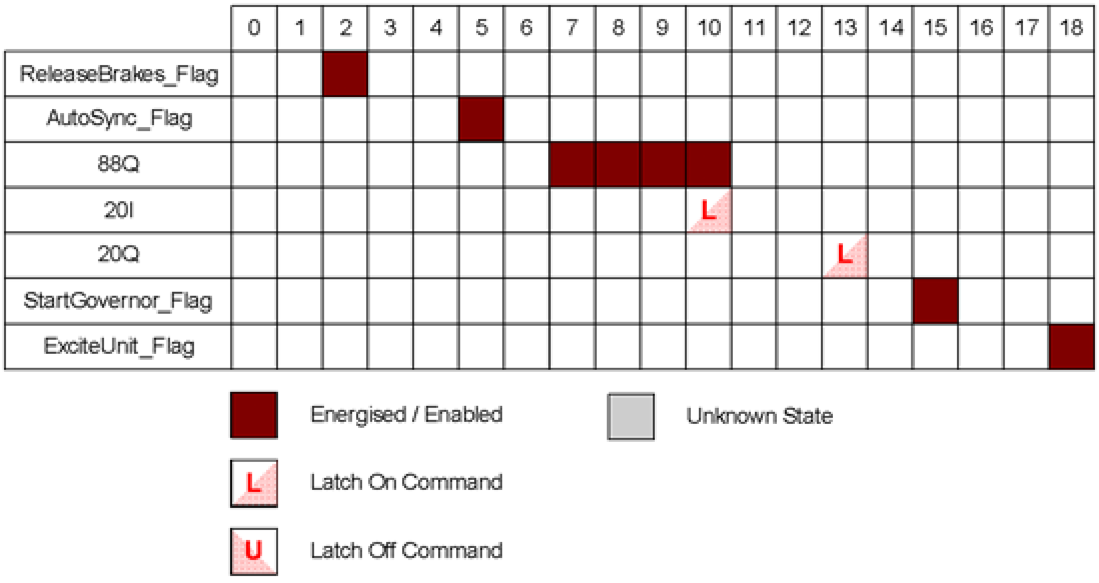

Reason for transition to next step. The logic must be true for the transition to fire.
Transition point marker. There must be only one marker per transition step
Unit.Running = 1 AND
Unit.Online = 1

Figure 12.2 A transition symbol. Logic is shown to illustrate what causes the transition.

4.11.1.4. Bingo Charts

Bingo charts are an effective way to illustrate what devices or flags are controlled by the sequence and the steps they're actioned upon. It is basically a matrix with all the device/flags listed on the left hand side and all the steps listed along the top row. The boxes intersecting the device row with the step column identifies what the device is doing.

The figure below shows an example of a bingo chart.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
ReleaseBrakes_Flag			🟥																
AutoSync_Flag						🟥													
88Q								🟥	🟥	🟥	🟥								
20I											L								
20Q														L					
StartGovernor_Flag																🟥			
ExciteUnit_Flag																			🟥

🟥	Energised / Enabled	⬜	Unknown State
L	Latch On Command		
U	Latch Off Command		

Figure 12.3 A bingo chart identifying all the devices controlled by the sequence

4.11.1.5. Converting State-Charts to code

Converting a state charts into code is a relatively simple process. The following example is typical of a function chart step – transition – step sequence:

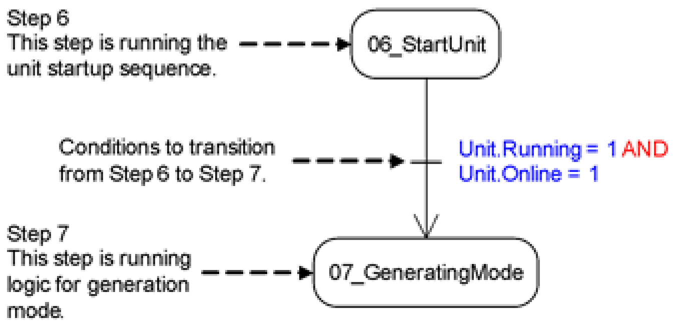
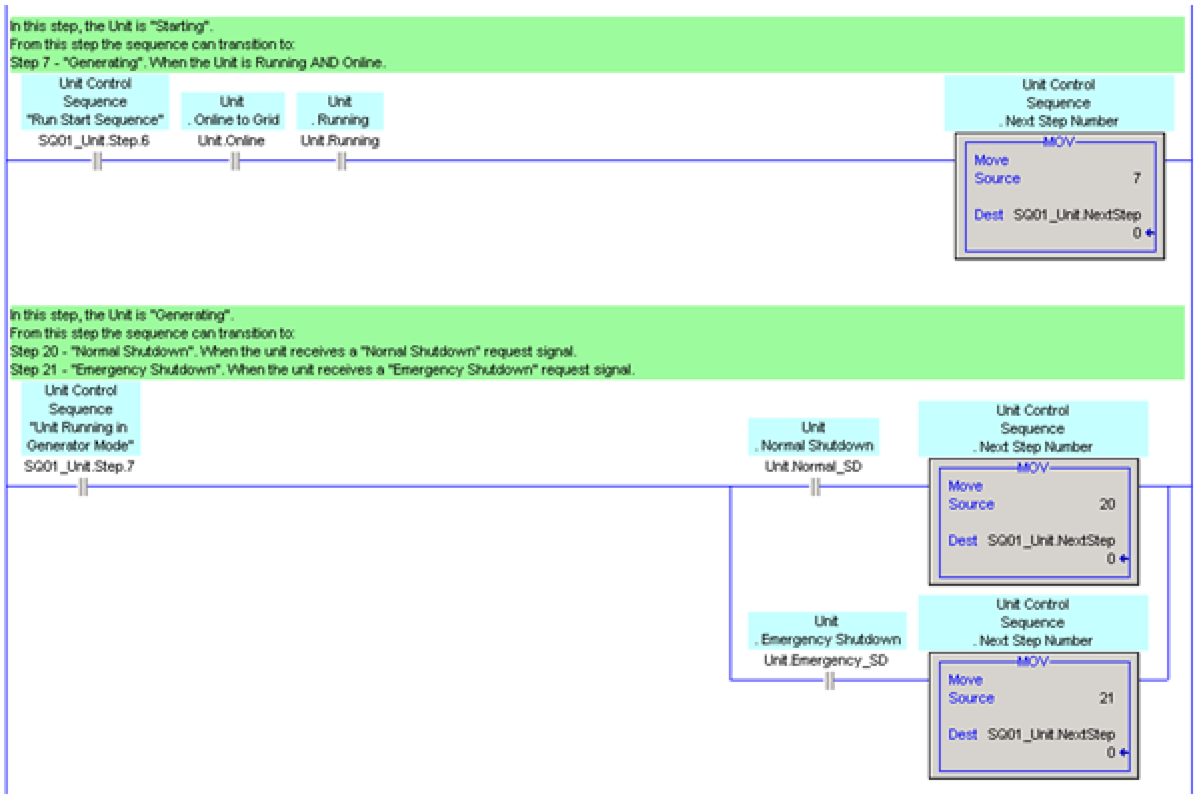

Step 6
This step is running the unit startup sequence.
06_StartUnit
Conditions to transition from Step 6 to Step 7.
Unit.Running = 1 AND
Unit.Online = 1
Step 7
This step is running logic for generation mode.
07_GeneratingMode

Figure 12.4 A typical sequence chart showing a transition from one step to another

The ladder translation of the above section of sequence is as follows:


In this step, the Unit is "Starting".
From this step the sequence can transition to:
Step 7 - "Generating". When the Unit is Running AND Online.

In this step, the Unit is "Generating".
From this step the sequence can transition to:
Step 20 - "Normal Shutdown". When the unit receives a "Nornal Shutdown" request signal.
Step 21 - "Emergency Shutdown". When the unit receives a "Emergency Shutdown" request signal.

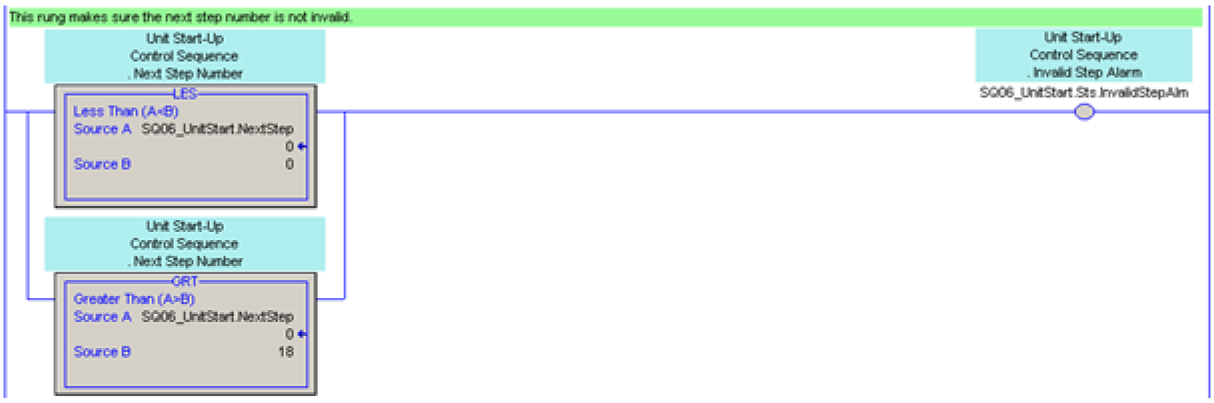
Figure 12.5 Translation of the state-chart diagram to ladder logic.

There is always one rung per step with all transition logic from that step to other steps on that same rung. Having all the possible transitions from that one step located on the same rung reduces confusion over where that step can transition to.

The move to any shutdown states is typically always the last transition examined so that this move takes precedence over any other valid moves. This ensures the shutdown states always take priority.

There are additional housekeeping rungs required at the end of each section of state machine logic to convert the 'NextStep' number (as an integer) to the actual step bit which is then used in the state machine logic and device/equipment auto run conditions.

The first of the two rungs checks whether the 'NextStep' number is a valid step. When using steps bits this number must be any integer of value from 0 to 31. An invalid step number would raise an alarm and prevent the next step number from being converted into a step bit.


This rung makes sure the next step number is not invalid.

Unit Start-Up Control Sequence .Next Step Number
LES
Less Than (A<B)
Source A SQ06_UnitStart.NextStep
0
Source B 0
Unit Start-Up Control Sequence .Next Step Number
GRT
Greater Than (A>B)
Source A SQ06_UnitStart.NextStep
0
Source B 18
Unit Start-Up Control Sequence .Invalid Step Alarm
SQ06_UnitStart.Sts.InvalidStepAlm

Figure 12.6 Checking if the 'NextStep' is a valid number

The second of the two housekeeping rungs converts the 'NextStep' number into a step bit. This is simply done by clearing all the step bits before using indirect addressing to set the appropriate step bit. For example, setting bit six of the DINT 'Step' tag enables step six of the sequence. Only ever one step can be active in any one time.

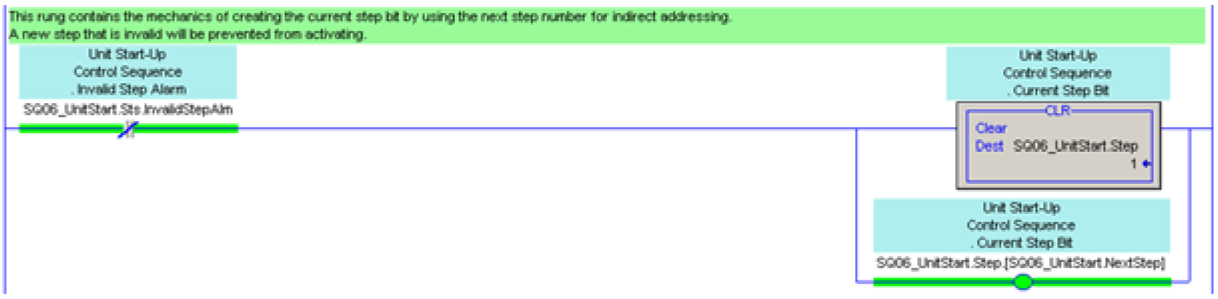
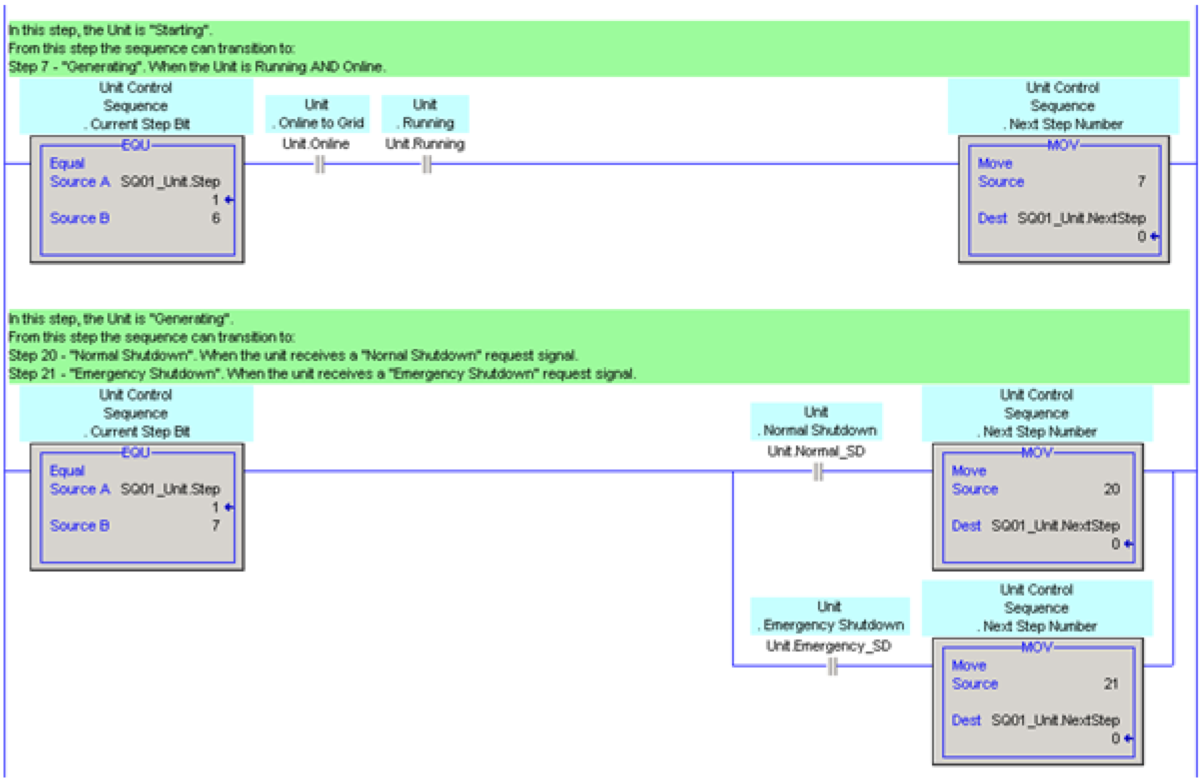

This rung contains the mechanics of creating the current step bit by using the next step number for indirect addressing.
A new step that is invalid will be prevented from activating.

Figure 12.7 Generating a new step bit using indirect addressing

Step bits are generally used for control rather than a step integer (number) as they are easier to read and execute faster when using an XIC instruction instead of an EQU instruction. The limitation of using step bits is that there are only 32 bits available within a DINT. Meaning only 32 steps can be programmed for one sequence. Sequences with more than 32 steps will have to use the 'Step' as an integer rather than bits. The figure below shows what the sequence logic would look like when using a step integer.


In this step, the Unit is "Starting".
From this step the sequence can transition to:
Step 7 - "Generating". When the Unit is Running AND Online.

In this step, the Unit is "Generating".
From this step the sequence can transition to:
Step 20 - "Normal Shutdown". When the unit receives a "Normal Shutdown" request signal.
Step 21 - "Emergency Shutdown". When the unit receives a "Emergency Shutdown" request signal.

Figure 12.8 Equivalent sequence ladder logic using step integers instead of step bits.

The following figure shows the logic to translate the 'NextStep' integer to the current step integer.

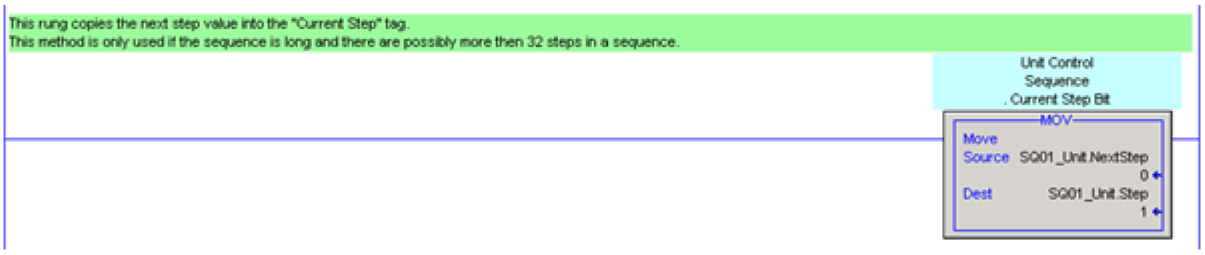

This rung copies the next step value into the "Current Step" tag.
This method is only used if the sequence is long and there are possibly more then 32 steps in a sequence.

Figure 12.9 Generating a new step number (integer) instead of generating a step bit.

4.12. Intelligent Motor Starters (E300)

A-B E300 Electronic Overload Relays are used for motor protection and control of most 415V and 230V DOL drives. Protection is provided by monitoring the electrical supply to the motor. Control is provided by switching one or more contactors to supply the motor.

4.12.1. DeviceLogix Firmware

DeviceLogix is a function that has been added to a number of Rockwell Automation devices to control outputs and manage status information locally within the device. For the E300 Electronic Overload Relays, an RSLogix Add-On-Profile provides the ability to configure the E300 parameters and DeviceLogix from within RSLogix 5000 software. The DeviceLogix editor is a software tool that provides a graphical interface for configuring ladder logic to provide local control within DeviceLogix-capable devices.

The DeviceLogix device consists of:

- a specific number of inputs and/or outputs
- local logic that determines its behavior

DeviceLogix should be configured for each E300 as per the project functional design specification requirements.

4.12.2. SHL Standard Functionality

SHL have developed a standardised, custom DeviceLogix program for DOL drives that may be used for all E300s, provided it meets the requirements of the project functional design specification.

An overview of the functionality is provided below.

The E300s utilise DeviceLogix firmware which allows basic ladder logic to run inside the E300 independently of the controlling PLC.

4.12.2.1. Dual Supply Capability

In some cases the motors will have the ability to start from either the common services board (CSB) or the unit services board (USB). SHL's standard motor control circuit design, and standard logic, allows the E300 to interface to either one or two contactors to provide electrical supply from either the CSB or USB.

For dual motor systems (A and B), one motor will be assigned as the normal duty motor and the other as standby. Normal duty motors will generally always run off the USB supply when the unit is excited and the standby motors will run off the CSB supply.

All other single supply motors will run off the CSB.

4.12.2.2. Communications Interface

E300's are interfaced to a ControlLogix controller using EtherNet/IP. All configuration and logic for each E300 is stored in the E300 module in the ControlLogix application I/O configuration.

A ControlLogix PLC with configured E300s will scan the EtherNet/IP configuration for E300s with matching IP address, and when found, will automatically establish communications and download the configuration stored in the PLC.

4.12.2.3. Handling of Communications Failure

The E300s are configured to maintain motor operation even if communications to the unit controller is lost. In such cases the motor must maintain its last state and have the ability to be controlled locally.

There are also some motors that will be configured to automatically start running if communications is lost and others to automatically stop if running after 30 seconds has elapsed.

5. References

5.1. Documents referenced herein include the following:

1756-RM094I-EN-P	Logix 5000 Controllers Design Considerations
SHL-ELE-156	SHL General low voltage electrical requirements, and annexures
IEC standard 60848	GRAFCET Specification Language for Sequential Function Charts
IEC standard 61131-3	Programmable controllers – Part 3: Programming languages
IEC standard 61508-3	Functional Safety – Part 3: Software requirements

5.2. Related Documents

The following documents were reviewed during preparation of this Standard:

SHL-ELE-143 Rev D	Guidelines for Coding PLCs
PJ015-SWG00L001	Control Software Standard
PJ015-SWG00G001	Project Dictionary

APPENDIX A - STANDARD USER DEFINED TYPES

The following sections display examples of typical, standard UDTs used for various equipment.

Recent example projects containing up-to-date standards for UDTs are available on request.

A1. uAnalogIn UDT

This UDT is for analogue input instruments.

Tag Name	Type	Display	Description
Cmd	**uAnalog_Cmd**		**Analogue HMI Commands**
ResetFault	*BOOL*	*Decimal*	*Reset Fault*
HH_Enter	*BOOL*	*Decimal*	*New High-High SP*
H_Enter	*BOOL*	*Decimal*	*New High SP*
L_Enter	*BOOL*	*Decimal*	*New Low SP*
LL_Enter	*BOOL*	*Decimal*	*New Low-Low SP*
Cmd.SP	***uAnalog_SP***		***New Analogue Alarm Set Points***
Trip	*REAL*	*Float*	*Trip Setpoint in Eng.Units*
OL	*REAL*	*Float*	*Overload Setpoint in Eng.Units*
HH	*REAL*	*Float*	*High-High Setpoint in Eng.Units*
H	*REAL*	*Float*	*High Setpoint in Eng.Units*
L	*REAL*	*Float*	*Low Setpoint in Eng.Units*
LL	*REAL*	*Float*	*Low-Low Setpoint in Eng.Units*
Alm	**uAnalogAlm**		**Analogue Alarms**
Fault	*BOOL*	*Decimal*	*Combined Fault*
Trip	*BOOL*	*Decimal*	*Tripped Alarm*
OL	*BOOL*	*Decimal*	*Overload Alarm*
HH	*BOOL*	*Decimal*	*High-High Alarm*
H	*BOOL*	*Decimal*	*High Alarm*
L	*BOOL*	*Decimal*	*Low Alarm*
LL	*BOOL*	*Decimal*	*Low-Low Alarm*
Hardware	*BOOL*	*Decimal*	*Hardware Fault*
UR	*BOOL*	*Decimal*	*Under-Range Alarm*
OR	*BOOL*	*Decimal*	*Over-Range Alarm*
ChanFlt	*BOOL*	*Decimal*	*Channel Fault*
ModuleFlt	*BOOL*	*Decimal*	*Module Fault*
PowerLoss	*BOOL*	*Decimal*	*Loss of Power*
State	*DINT*	*Decimal*	*Alarm State*
EU	**uAnalogEU**		**Scaled Input Eng.Units**
PV	*REAL*	*Float*	*Scaled Input Eng.Units*
Min	*REAL*	*Float*	*Min. Value Eng. Units*
Max	*REAL*	*Float*	*Max. Value Eng. Units*
SP	**uAnalogSP**		**Analogue Setpoints**
Trip	*REAL*	*Float*	*Trip Setpoint in Eng.Units*
OL	*REAL*	*Float*	*Overload Setpoint in Eng.Units*
HH	*REAL*	*Float*	*High-High Setpoint in Eng.Units*
H	*REAL*	*Float*	*High Setpoint in Eng.Units*
L	*REAL*	*Float*	*Low Setpoint in Eng.Units*
LL	*REAL*	*Float*	*Low-Low Setpoint in Eng.Units*
Cfg	**uAnalogCfg**		**Alarm Configuration**
Inhib_H	*BOOL*	*Decimal*	*Inhibit High Alarms*
Inhib_L	*BOOL*	*Decimal*	*Inhibit Low Alarms*
Trip_En	*BOOL*	*Decimal*	*Using Trip Function*
OL_En	*BOOL*	*Decimal*	*Using Overload*
HH_En	*BOOL*	*Decimal*	*Using HH Alarm*
H_En	*BOOL*	*Decimal*	*Using H Alarm*

L_En	*BOOL*	*Decimal*	*Using L Alarm*
LL_En	*BOOL*	*Decimal*	*Using LL Alarm*
LH_Ctrl_En	*BOOL*	*Decimal*	*L&H Used for Ctrl*
Info	**uAnalog_Info**		**Analogue Info**
TagName	*String_12*	*Decimal*	*Tag Name*
TagDescrip	*String_32*	*Decimal*	*Tag Description*
Units	*String_04*	*Decimal*	*Units*
HighLimit	*String_14*	*Decimal*	*High Limit Text*
LowLimit	*String_14*	*Decimal*	*Low Limit Text*
dTmr	**uAnalogTmrs**		**Alarm Debounce Timers**
OL	*TIMER*		*Overload Alarm Timer*
HH	*TIMER*		*High-High Alarm Timer*
H	*TIMER*		*High Alarm Timer*
L	*TIMER*		*Low Alarm Timer*
LL	*TIMER*		*Low-Low Alarm Timer*
eTmr	**uAnalogTmrs**		**Alarm Extension Timers**
HH	*TIMER*		*High-High Alarm Timer*
H	*TIMER*		*High Alarm Timer*
L	*TIMER*		*Low Alarm Timer*
LL	*TIMER*		*Low-Low Alarm Timer*
UR_SP	REAL	Float	Under Range Set Point
OR_SP	REAL	Float	Over Range Set Point
DeadB	REAL	Float	Alarm Deadband

PV+ Write
Remote SCADA Write
HMI Read
PV+ Read/Write

A2. Single Supply Motor Starter with Overload UDT (uMotorE300_SS)

This UDT is for single supply motors with E300 overload relays.

Tag Name	Type	Display	Description
Cmd	**uMotorE300_SS_Cmd**		**HMI Commands**
AutoReq	*BOOL*	*Decimal*	*Auto Request*
AutoConfirm	*BOOL*	*Decimal*	*Auto Confirm*
Auto	*BOOL*	*Decimal*	*Auto Command*
ManualReq	*BOOL*	*Decimal*	*Manual Request*
ManualConfirm	*BOOL*	*Decimal*	*Manual Confirm*
Manual	*BOOL*	*Decimal*	*Manual Command*
LocalReq	*BOOL*	*Decimal*	*Local Request*
LocalConfirm	*BOOL*	*Decimal*	*Local Confirm*
Local	*BOOL*	*Decimal*	*Local Command*
Start_Req	*BOOL*	*Decimal*	*Start Request*
Start_Confirm	*BOOL*	*Decimal*	*Start Confirm*
Start	*BOOL*	*Decimal*	*Start Command*
StopReq	*BOOL*	*Decimal*	*Stop Request*

StopConfirm	*BOOL*	*Decimal*	*Stop Confirm*
Stop	*BOOL*	*Decimal*	*Stop Command*
RstRunHrsReq	*BOOL*	*Decimal*	*Reset Run Hrs Req*
RstRunHrsConfirm	*BOOL*	*Decimal*	*Reset Run Hrs Cfm*
RstRunHrs	*BOOL*	*Decimal*	*Reset Run Hrs Cmd*
RstStartNumReq	*BOOL*	*Decimal*	*Reset No.of Starts Request*
RstStartNumConfirm	*BOOL*	*Decimal*	*Reset No.of Starts Confirm*
RstStartNum	*BOOL*	*Decimal*	*Reset No.of Starts Command*
ResetFault	*BOOL*	*Decimal*	*Reset Fault*
Cancel	*BOOL*	*Decimal*	*Cancel Requests*
StartRead	*BOOL*	*Decimal*	*Start Para Read*
StopRead	*BOOL*	*Decimal*	*Stop Para Read*
NOS_AlmSP	*DINT*	*Decimal*	*New NOS Alm SP*
Sts	**uMotorE300_SS_Sts**		**General Status**
ManualAck	*BOOL*	*Decimal*	*Manual Acknowledge*
LocalAck	*BOOL*	*Decimal*	*Local Acknowledge*
Start_Ack	*BOOL*	*Decimal*	*Start Ack*
StopAck	*BOOL*	*Decimal*	*Stop Ack*
RstRunHrsAck	*BOOL*	*Decimal*	*Reset Run Hrs Ack*
RstStartNumAck	*BOOL*	*Decimal*	*Reset No.of Starts Acknowledge*
AckReqd	*BOOL*	*Decimal*	*Ack.Required*
AutoLock	*BOOL*	*Decimal*	*Locked into Auto*
Manual	*BOOL*	*Decimal*	*Manual Mode*
Local	*BOOL*	*Decimal*	*Local Mode*
Duty	*BOOL*	*Decimal*	*Normal Duty*
Fault	*BOOL*	*Decimal*	*Fault (Alarm Sum)*
Warning	*BOOL*	*Decimal*	*Warning (Alm Sum)*
DeviceConfig	*BOOL*	*Decimal*	*Device Config Warn*
Interlock	*BOOL*	*Decimal*	*Interlocked*
Permissive	*BOOL*	*Decimal*	*Permitted to Run*
CSB_OK	*BOOL*	*Decimal*	*CSB Supply OK*
aReady	*BOOL*	*Decimal*	*Ready to aRun*
mReady	*BOOL*	*Decimal*	*Ready to mRun*
Running	*BOOL*	*Decimal*	*Running*
RunReq	*BOOL*	*Decimal*	*Run Required*
iIsolated	*BOOL*	*Decimal*	*Isolator Open*
iESPB_OK	*BOOL*	*Decimal*	*ESPB Healthy*
State	*DINT*	*Decimal*	*Drive State*
RunHrs	*DINT*	*Decimal*	*Hours Run*

StopHrs	*DINT*	*Decimal*	*Stopped Hours*
StopDays	*DINT*	*Decimal*	*Stopped Days*
StartNum	*DINT*	*Decimal*	*Number of Starts*
NOS	*DINT*	*Decimal*	*No.of Starts pHr*
NOS_AlmSP	*DINT*	*Decimal*	*No.of Starts Alm SP*
TimeToTrip	*INT*	*Decimal*	*Time to Trip*
TimeToRst	*INT*	*Decimal*	*Time to Reset*
TagName	*String_08*		*Tag Name*
TagDescrip	*String_32*		*Tag Description*
Alm	**uMotorE300_SS_Alm**		**Alarm Bits**
Tripped	*BOOL*	*Decimal*	*E300 Tripped*
ESPB	*BOOL*	*Decimal*	*Emergency Stop*
TestTrip	*BOOL*	*Decimal*	*Test Trip*
Overload	*BOOL*	*Decimal*	*Overload Trip*
PhaseLoss	*BOOL*	*Decimal*	*Phase Loss Trip*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbal.Trip*
NV_MemFlt	*BOOL*	*Decimal*	*Non-Vol.Memory Flt*
HW_Flt	*BOOL*	*Decimal*	*Hardware Fault*
Warning	*BOOL*	*Decimal*	*E300 Warning*
wOverload	*BOOL*	*Decimal*	*Overload Warning*
wCurrentImbal	*BOOL*	*Decimal*	*Current Imbalance Warning*
FTR	*BOOL*	*Decimal*	*Failed to Run Alm*
FTS	*BOOL*	*Decimal*	*Failed to Stop Alm*
NOS_H	*BOOL*	*Decimal*	*No.of Starts High*
CommsFlt	*BOOL*	*Decimal*	*Comms Fault*
E300	**uMotorE300_Para**		**E300 Parameters**
TimeToTrip	*INT*	*Decimal*	*Time to Trip*
TimeToRst	*INT*	*Decimal*	*Time to Reset*
L1_Current	*REAL*	*Float*	*L1 Current (A)*
L2_Current	*REAL*	*Float*	*L2 Current (A)*
L3_Current	*REAL*	*Float*	*L3 Current (A)*
AveCurrent	*REAL*	*Float*	*Average Current*
L1_FLA	*REAL*	*Float*	*L1 FLA (%)*
L2_FLA	*REAL*	*Float*	*L2 FLA (%)*
L3_FLA	*REAL*	*Float*	*L3 FLA (%)*
Ave_FLA	*REAL*	*Float*	*Average FLA (%)*
Current_Imbal	*REAL*	*Float*	*Current Imbalance*
ThermUtil	*REAL*	*Float*	*% Thermal Utilised*
Status	**uMotorE300_Status**		**Device Status**

Tripped	*BOOL*	*Decimal*	*Tripped*
Warning	*BOOL*	*Decimal*	*Trip Warning*
OutputA	*BOOL*	*Decimal*	*Output A On*
OutputB	*BOOL*	*Decimal*	*Output B On*
Input1	*BOOL*	*Decimal*	*Input 1 On*
Input2	*BOOL*	*Decimal*	*Input 2 On*
Input3	*BOOL*	*Decimal*	*Input 3 On*
Input4	*BOOL*	*Decimal*	*Input 4 On*
MotorCurrent	*BOOL*	*Decimal*	*Motor Current*
GndFltCurrent	*BOOL*	*Decimal*	*Ground Flt Current*
Spare10	*BOOL*	*Decimal*	*Spare 10*
Spare11	*BOOL*	*Decimal*	*Spare 11*
Spare12	*BOOL*	*Decimal*	*Spare 12*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
Warn	**uMotorE300_Warn**		**Trip Warnings**
Spare0	*BOOL*	*Decimal*	*Spare 0*
Overload	*BOOL*	*Decimal*	*Overload*
Spare2	*BOOL*	*Decimal*	*Spare 2*
GroundFlt	*BOOL*	*Decimal*	*Ground Fault*
Spare4	*BOOL*	*Decimal*	*Spare 4*
Jam	*BOOL*	*Decimal*	*Jam*
Underload	*BOOL*	*Decimal*	*Underload*
PTC	*BOOL*	*Decimal*	*PTC*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbalance*
CommFlt	*BOOL*	*Decimal*	*Comms Fault*
CommIdle	*BOOL*	*Decimal*	*Comms Idle*
Spare11	*BOOL*	*Decimal*	*Spare 11*
DeviceConfig	*BOOL*	*Decimal*	*Device Config*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
Trip	**uMotorE300_Trips**		**Trip Status**
TestTrip	*BOOL*	*Decimal*	*Test Trip*
Overload	*BOOL*	*Decimal*	*Overload*
PhaseLoss	*BOOL*	*Decimal*	*Phase Loss*
GroundFlt	*BOOL*	*Decimal*	*Ground Fault*
Stall	*BOOL*	*Decimal*	*Stall*

Jam	*BOOL*	*Decimal*	*Jam*
Underload	*BOOL*	*Decimal*	*Underload*
PTC	*BOOL*	*Decimal*	*PTC*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbalance*
CommFlt	*BOOL*	*Decimal*	*Comms Fault*
CommIdle	*BOOL*	*Decimal*	*Comms Idle*
NV_MemFlt	*BOOL*	*Decimal*	*Non-Vol.Memory Flt*
HW_Flt	*BOOL*	*Decimal*	*Hardware Fault*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
AutoLast	BOOL	Decimal	Auto Previously
ManualLast	BOOL	Decimal	Manual Previously
ResetFltProg	BOOL	Decimal	PLC Fault Reset
ResetFlt	BOOL	Decimal	Reset Fault
ControlActive	BOOL	Decimal	Control Active
CurrentFlow	BOOL	Decimal	Current Detected
aRunProg	BOOL	Decimal	Auto Run Prog.
StartPulse	BOOL	Decimal	Start Pulse
StopPulse	BOOL	Decimal	Stop Pulse
aRun	BOOL	Decimal	Auto Run
mRun	BOOL	Decimal	Manual Run
iLocalStart	BOOL	Decimal	Local Start
iLocalStop	BOOL	Decimal	Local Stop
oStart	BOOL	Decimal	Start Output
oStop	BOOL	Decimal	Stop Output
oResetFlt	BOOL	Decimal	Fault Reset
oLocal	BOOL	Decimal	Local Mode Output
oSOCF	BOOL	Decimal	Stop On Comms Flt
E300_Bits	INT	Decimal	E300 Status Bits
NOS_Array	DINT[6]	Decimal	No.of Starts Array
NOS_Ctrl	CONTROL		No.of Starts Ctrl
OneShot	DINT	Decimal	One Shot
FTR_T	TIMER		Failed to Run Tmr
FTS_T	TIMER		Failed to Stop Tmr
LocalTripRst_T	TIMER		Local Trip Reset Timer
TripRst_T	TIMER		Trip Reset Timer
Running_T	TIMER		Running Timer
Stopped_T	TIMER		Stopped Timer

RunOn_T	TIMER		Motor Run On Timer
ControlOn_T	TIMER		Control Active Tmr
AutoReq_T	TIMER		Auto Request Tmr
ManualReq_T	TIMER		Manual Request Tmr
LocalReq_T	TIMER		Local Request Tmr
Start_Req_T	TIMER		Start Request Tmr
StopReq_T	TIMER		Stop Request Tmr
RstRunHrsReq_T	TIMER		Reset Run Hrs Request Timer
RstStartNumReq_T	TIMER		Reset No.of Starts Request Timer

PV+ Write
Remote SCADA Write
HMI Read

A3. Dual Supply Motor Starter with Overload UDT (uMotorE300_DS)

This UDT is for dual supply motors with E300 overload relays.

Tag Name	Type	Display	Description
Cmd	**uMotorE3_DS_Cmd**		**Hmi Commands**
AutoReq	*BOOL*	*Decimal*	*Auto Request*
AutoConfirm	*BOOL*	*Decimal*	*Auto Confirm*
Auto	*BOOL*	*Decimal*	*Auto Command*
ManualReq	*BOOL*	*Decimal*	*Manual Request*
ManualConfirm	*BOOL*	*Decimal*	*Manual Confirm*
Manual	*BOOL*	*Decimal*	*Manual Command*
LocalReq	*BOOL*	*Decimal*	*Local Request*
LocalConfirm	*BOOL*	*Decimal*	*Local Confirm*
Local	*BOOL*	*Decimal*	*Local Command*
StartCSB_Req	*BOOL*	*Decimal*	*Start CSB Request*
StartCSB_Confirm	*BOOL*	*Decimal*	*Start CSB Confirm*
StartCSB	*BOOL*	*Decimal*	*Start CSB Command*
StartUSB_Req	*BOOL*	*Decimal*	*Start USB Request*
StartUSB_Confirm	*BOOL*	*Decimal*	*Start USB Confirm*
StartUSB	*BOOL*	*Decimal*	*Start USB Command*
StopReq	*BOOL*	*Decimal*	*Stop Request*
StopConfirm	*BOOL*	*Decimal*	*Stop Confirm*
Stop	*BOOL*	*Decimal*	*Stop Command*
ForceCSB_Req	*BOOL*	*Decimal*	*Force CSB Request*
ForceCSB_Confirm	*BOOL*	*Decimal*	*Force CSB Confirm*
ForceCSB	*BOOL*	*Decimal*	*Force CSB Command*
RstRunHrsReq	*BOOL*	*Decimal*	*Reset Run Hrs Req*
RstRunHrsConfirm	*BOOL*	*Decimal*	*Reset Run Hrs Cfm*
RstRunHrs	*BOOL*	*Decimal*	*Reset Run Hrs Cmd*
RstStartNumReq	*BOOL*	*Decimal*	*Reset No.of Starts Request*
RstStartNumConfirm	*BOOL*	*Decimal*	*Reset No.of Starts Confirm*

RstStartNum	*BOOL*	*Decimal*	*Reset No.of Starts Command*
ResetFault	*BOOL*	*Decimal*	*Reset Fault*
Cancel	*BOOL*	*Decimal*	*Cancel Requests*
StartRead	*BOOL*	*Decimal*	*Start Para Read*
StopRead	*BOOL*	*Decimal*	*Stop Para Read*
NOS_Enter	*BOOL*	*Decimal*	*New NOS SP*
NOS_AlmSP	*DINT*	*Decimal*	*New NOS Alm SP*
Sts	**uMotorE300_DS_Sts**		**General Status**
AutoAck	*BOOL*	*Decimal*	*Auto Acknowledge*
ManualAck	*BOOL*	*Decimal*	*Manual Acknowledge*
LocalAck	*BOOL*	*Decimal*	*Local Acknowledge*
StartCSB_Ack	*BOOL*	*Decimal*	*Start CSB Ack*
StartUSB_Ack	*BOOL*	*Decimal*	*Start USB Ack*
StopAck	*BOOL*	*Decimal*	*Stop Ack*
ForceCSB_Ack	*BOOL*	*Decimal*	*Force CSB Ack*
RstRunHrsAck	*BOOL*	*Decimal*	*Reset Run Hrs Ack*
RstStartNumAck	*BOOL*	*Decimal*	*Reset No.of Starts Acknowledge*
AckReqd	*BOOL*	*Decimal*	*Ack.Required*
AutoLock	*BOOL*	*Decimal*	*Locked into Auto*
Manual	*BOOL*	*Decimal*	*Manual Mode*
Local	*BOOL*	*Decimal*	*Local Mode*
Duty	*BOOL*	*Decimal*	*Normal Duty*
Fault	*BOOL*	*Decimal*	*Fault (Alarm Sum)*
Warning	*BOOL*	*Decimal*	*Warning (Alm Sum)*
DeviceConfig	*BOOL*	*Decimal*	*Device Config Warn*
Interlock	*BOOL*	*Decimal*	*Interlocked*
Permissive	*BOOL*	*Decimal*	*Permitted to Run*
CSB_OK	*BOOL*	*Decimal*	*CSB Supply OK*
USB_OK	*BOOL*	*Decimal*	*USB Supply OK*
ForceCSB	*BOOL*	*Decimal*	*Force CSB Supply*
UseUSB	*BOOL*	*Decimal*	*Use USB Supply*
USB_Inhib	*BOOL*	*Decimal*	*Inhibit USB Supply*
aReady	*BOOL*	*Decimal*	*Ready to aRun*
mReady	*BOOL*	*Decimal*	*Ready to mRun*
Running	*BOOL*	*Decimal*	*Running*
RunningCSB	*BOOL*	*Decimal*	*Running from CSB*
RunningUSB	*BOOL*	*Decimal*	*Running from USB*
RunReq	*BOOL*	*Decimal*	*Run Required*
iIsolated	*BOOL*	*Decimal*	*Isolator Open*
iESPB_OK	*BOOL*	*Decimal*	*ESPB Healthy*
State	*DINT*	*Decimal*	*Drive State*
RunHrs	*DINT*	*Decimal*	*Hours Run*
StopHrs	*DINT*	*Decimal*	*Stopped Hours*
StopDays	*DINT*	*Decimal*	*Stopped Days*
StartNum	*DINT*	*Decimal*	*Number of Starts*
NOS	*DINT*	*Decimal*	*No.of Starts pHr*

NOS_AlmSP	*DINT*	*Decimal*	*No.of Starts Alm SP*
TagName	*String_08*		*Tag Name*
TagDescrip	*String_32*		*Tag Description*
Alm	**uMotorE300_DS_Alm**		**Alarm Bits**
Tripped	*BOOL*	*Decimal*	*E300 Tripped*
ESPB	*BOOL*	*Decimal*	*Emergency Stop*
TestTrip	*BOOL*	*Decimal*	*Test Trip*
Overload	*BOOL*	*Decimal*	*Overload Trip*
PhaseLoss	*BOOL*	*Decimal*	*Phase Loss Trip*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbal.Trip*
NV_MemFlt	*BOOL*	*Decimal*	*Non-Vol.Memory Flt*
HW_Flt	*BOOL*	*Decimal*	*Hardware Fault*
Warning	*BOOL*	*Decimal*	*E3+ Warning*
wOverload	*BOOL*	*Decimal*	*Overload Warning*
wCurrentImbal	*BOOL*	*Decimal*	*Current Imbalance Warning*
FTRc	*BOOL*	*Decimal*	*Failed to Run CSB Alarm*
FTRu	*BOOL*	*Decimal*	*Failed to Run USB Alarm*
FTS	*BOOL*	*Decimal*	*Failed to Stop Alm*
NOS_H	*BOOL*	*Decimal*	*No.of Starts High*
CommsFlt	*BOOL*	*Decimal*	*Comms Fault*
E300	**uMotorE300_Para**		**E300 Parameters**
TimeToTrip	*INT*	*Decimal*	*Time to Trip*
TimeToRst	*INT*	*Decimal*	*Time to Reset*
L1_Current	*REAL*	*Float*	*L1 Current (A)*
L2_Current	*REAL*	*Float*	*L2 Current (A)*
L3_Current	*REAL*	*Float*	*L3 Current (A)*
AveCurrent	*REAL*	*Float*	*Average Current*
L1_FLA	*REAL*	*Float*	*L1 FLA (%)*
L2_FLA	*REAL*	*Float*	*L2 FLA (%)*
L3_FLA	*REAL*	*Float*	*L3 FLA (%)*
Ave_FLA	*REAL*	*Float*	*Average FLA (%)*
Current_Imbal	*REAL*	*Float*	*Current Imbalance*
ThermUtil	*REAL*	*Float*	*% Thermal Utilised*
Status	**uMotorE300_Status**		**Device Status**
Tripped	*BOOL*	*Decimal*	*Tripped*
Warning	*BOOL*	*Decimal*	*Trip Warning*
OutputA	*BOOL*	*Decimal*	*Output A On*
OutputB	*BOOL*	*Decimal*	*Output B On*
Input1	*BOOL*	*Decimal*	*Input 1 On*
Input2	*BOOL*	*Decimal*	*Input 2 On*
Input3	*BOOL*	*Decimal*	*Input 3 On*
Input4	*BOOL*	*Decimal*	*Input 4 On*
MotorCurrent	*BOOL*	*Decimal*	*Motor Current*
GndFltCurrent	*BOOL*	*Decimal*	*Ground Flt Current*
Spare10	*BOOL*	*Decimal*	*Spare 10*
Spare11	*BOOL*	*Decimal*	*Spare 11*

Spare12	*BOOL*	*Decimal*	*Spare 12*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
Warn	**uMotorE300_Warn**		**Trip Warnings**
Spare0	*BOOL*	*Decimal*	*Spare 0*
Overload	*BOOL*	*Decimal*	*Overload*
Spare2	*BOOL*	*Decimal*	*Spare 2*
GroundFlt	*BOOL*	*Decimal*	*Ground Fault*
Spare4	*BOOL*	*Decimal*	*Spare 4*
Jam	*BOOL*	*Decimal*	*Jam*
Underload	*BOOL*	*Decimal*	*Underload*
PTC	*BOOL*	*Decimal*	*PTC*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbalance*
CommFlt	*BOOL*	*Decimal*	*Comms Fault*
CommIdle	*BOOL*	*Decimal*	*Comms Idle*
Spare11	*BOOL*	*Decimal*	*Spare 11*
DeviceConfig	*BOOL*	*Decimal*	*Device Config*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
Trip	**uMotorE300_Trips**		**Trip Status**
TestTrip	*BOOL*	*Decimal*	*Test Trip*
Overload	*BOOL*	*Decimal*	*Overload*
PhaseLoss	*BOOL*	*Decimal*	*Phase Loss*
GroundFlt	*BOOL*	*Decimal*	*Ground Fault*
Stall	*BOOL*	*Decimal*	*Stall*
Jam	*BOOL*	*Decimal*	*Jam*
Underload	*BOOL*	*Decimal*	*Underload*
PTC	*BOOL*	*Decimal*	*PTC*
CurrentImbal	*BOOL*	*Decimal*	*Current Imbalance*
CommFlt	*BOOL*	*Decimal*	*Comms Fault*
CommIdle	*BOOL*	*Decimal*	*Comms Idle*
NV_MemFlt	*BOOL*	*Decimal*	*Non-Vol.Memory Flt*
HW_Flt	*BOOL*	*Decimal*	*Hardware Fault*
Spare13	*BOOL*	*Decimal*	*Spare 13*
Spare14	*BOOL*	*Decimal*	*Spare 14*
Spare15	*BOOL*	*Decimal*	*Spare 15*
AutoLast	BOOL	Decimal	Auto Previously
ManualLast	BOOL	Decimal	Manual Previously
ResetFltProg	BOOL	Decimal	PLC Fault Reset
ResetFlt	BOOL	Decimal	Reset Fault
ControlActive	BOOL	Decimal	Control Active
CurrentFlow	BOOL	Decimal	Current Detected
aRunProg	BOOL	Decimal	Auto Run Prog.
StartPulse	BOOL	Decimal	Start Pulse

StopPulse	BOOL	Decimal	Stop Pulse
aRun	BOOL	Decimal	Auto Run
mRunCSB	BOOL	Decimal	Manual Run CSB
mRunUSB	BOOL	Decimal	Manual Run USB
iLocalStart	BOOL	Decimal	Local Start
iLocalStop	BOOL	Decimal	Local Stop
oStartCSB	BOOL	Decimal	Start CSB Output
oStartUSB	BOOL	Decimal	Start USB Output
oStop	BOOL	Decimal	Stop Output
oResetFlt	BOOL	Decimal	Fault Reset
oLocal	BOOL	Decimal	Local Mode Output
oROCF	BOOL	Decimal	Run On Comms Flt
E300_Bits	INT	Decimal	E300 Status Bits
NOS_Array	DINT[6]	Decimal	No.of Starts Array
NOS_Ctrl	CONTROL		No.of Starts Ctrl
OneShot	DINT	Decimal	One Shot
FTRc_T	TIMER		Failed to Run CSB Timer
FTRu_T	TIMER		Failed to Run USB Timer
FTS_T	TIMER		Failed to Stop Tmr
LocalTripRst_T	TIMER		Local Trip Reset Timer
TripRst_T	TIMER		Trip Reset Timer
Running_T	TIMER		Running Timer
Stopped_T	TIMER		Stopped Timer
RunOn_T	TIMER		Motor Run On Timer
ControlOn_T	TIMER		Control Active Tmr
AutoReq_T	TIMER		Auto Request Tmr
ManualReq_T	TIMER		Manual Request Tmr
LocalReq_T	TIMER		Local Request Tmr
StartCSB_Req_T	TIMER		Start CSB Req.Tmr
StartUSB_Req_T	TIMER		Start USB Req.Tmr
StopReq_T	TIMER		Stop Request Tmr
ForceCSB_Req_T	TIMER		Force CSB Req Tmr
RstRunHrsReq_T	TIMER		Reset Run Hrs Request Timer
RstStartNumReq_T	TIMER		Reset No.of Starts Request Timer

PV+ Write
Remote SCADA Write
HMI Read

A4. Single Coil Valve UDT (uValveSC_NC)

This UDT is for standard normally closed single coil valves.

Tag Name	Type	Display	Description
Cmd	**uValveSC_NC_Cmd**		**Valve Command Bits**
AutoReq	*BOOL*	*Decimal*	*Auto Request*
AutoConfirm	*BOOL*	*Decimal*	*Auto Confirm*
Auto	*BOOL*	*Decimal*	*Auto Command*
ManualReq	*BOOL*	*Decimal*	*Manual Request*
ManualConfirm	*BOOL*	*Decimal*	*Manual Confirm*
Manual	*BOOL*	*Decimal*	*Manual Command*
OpenReq	*BOOL*	*Decimal*	*Open Request*
OpenConfirm	*BOOL*	*Decimal*	*Open Confirm*
Open	*BOOL*	*Decimal*	*Open Command*
CloseReq	*BOOL*	*Decimal*	*Close Request*
CloseConfirm	*BOOL*	*Decimal*	*Close Confirm*
Close	*BOOL*	*Decimal*	*Close Command*
RstOpenNumReq	*BOOL*	*Decimal*	*Rst No.of Open Req*
RstOpenNumConfirm	*BOOL*	*Decimal*	*Reset No.of Open Confirm*
RstOpenNum	*BOOL*	*Decimal*	*Rst No.of Open Cmd*
Cancel	*BOOL*	*Decimal*	*Cancel Requests*
ResetFault	*BOOL*	*Decimal*	*Reset Fault*
Sts	**uValveSC_NC_Sts**		**Valve NO Status Bits**
AutoAck	*BOOL*	*Decimal*	*Auto Acknowledge*
ManualAck	*BOOL*	*Decimal*	*Manual Acknowledge*
OpenAck	*BOOL*	*Decimal*	*Open Acknowledge*
CloseAck	*BOOL*	*Decimal*	*Close Acknowledge*
RstOpenNumAck	*BOOL*	*Decimal*	*Rst No.of Open Ack*
AckReqd	*BOOL*	*Decimal*	*Ack.Required*
Manual	*BOOL*	*Decimal*	*Manual Mode*
AutoLock	*BOOL*	*Decimal*	*Locked into Auto*
Fault	*BOOL*	*Decimal*	*Faulted (Alarm)*
Moving	*BOOL*	*Decimal*	*Moving*
Interlock	*BOOL*	*Decimal*	*Interlocked*
Permissive	*BOOL*	*Decimal*	*Permitted to Open*
iClosed	*BOOL*	*Decimal*	*Closed*
iOpened	*BOOL*	*Decimal*	*Open*
oOpen	*BOOL*	*Decimal*	*Open Output*
State	*DINT*	*Decimal*	*Valve State*
OpenNum	*DINT*	*Decimal*	*Number of Opens*
OStroke	*REAL*	*Float*	*Opening Time*
CStroke	*REAL*	*Float*	*Closing Time*
TagName	*String_08*		*Tag Name*
TagDescrip	*String_32*		*Tag Description*

Alm	**uValveSC_Alm**		**Valve Alarm Bits**
FTO	*BOOL*	*Decimal*	*Failed to Open Alm*
FTC	*BOOL*	*Decimal*	*Failed to Close Alm*
FB_Error	*BOOL*	*Decimal*	*Feedback Error*
CommsFlt	*BOOL*	*Decimal*	*Comms Fault*
aOpenProg	BOOL	Decimal	Auto Open Prog.
OpenPulse	BOOL	Decimal	Open Pulse
ClosePulse	BOOL	Decimal	Close Pulse
aOpen	BOOL	Decimal	Auto Open
mOpen	BOOL	Decimal	Manual Open
oOpen	BOOL	Decimal	Open Output
OneShot	DINT	Decimal	One Shot
FTO_T	TIMER		Failed to Open Tmr
FTC_T	TIMER		Failed to Close Tmr
FB_Err_T	TIMER		Feedback Error Tmr
eAlm_T	TIMER		Extension Alarm Timer
Open_dT	TIMER		Open Debounce Tmr
Close_dT	TIMER		Close Debounce Tmr
Stroke_T	TIMER		Stroke Timer
AutoReq_T	TIMER		Auto Request Tmr
ManualReq_T	TIMER		Manual Request Tmr
OpenReq_T	TIMER		Open Request Tmr
CloseReq_T	TIMER		Close Request Tmr
RstOpenNumReq_T	TIMER		Rst No.of Open Tmr

PV+ Write
Remote SCADA Write
HMI Read

A5. Dual Coil Valve UDT (uValveDC)

This UDT is for standard dual coil valves.

Tag Name	Type	Display	Description
Cmd	**uValveDC_Cmd**		**Valve Command Bits**
AutoReq	*BOOL*	*Decimal*	*Auto Request*
AutoConfirm	*BOOL*	*Decimal*	*Auto Confirm*
Auto	*BOOL*	*Decimal*	*Auto Command*
ManualReq	*BOOL*	*Decimal*	*Manual Request*
ManualConfirm	*BOOL*	*Decimal*	*Manual Confirm*
Manual	*BOOL*	*Decimal*	*Manual Command*
OpenReq	*BOOL*	*Decimal*	*Open Request*
OpenConfirm	*BOOL*	*Decimal*	*Open Confirm*
Open	*BOOL*	*Decimal*	*Open Command*
CloseReq	*BOOL*	*Decimal*	*Close Request*
CloseConfirm	*BOOL*	*Decimal*	*Close Confirm*

Close	*BOOL*	*Decimal*	*Close Command*
RstOpenNumReq	*BOOL*	*Decimal*	*Reset No.of Opens Request*
RstOpenNumConfirm	*BOOL*	*Decimal*	*Reset No.of Opens Confirm*
RstOpenNum	*BOOL*	*Decimal*	*Reset No.of Opens Command*
Cancel	*BOOL*	*Decimal*	*Cancel Requests*
ResetFault	*BOOL*	*Decimal*	*Reset Fault*
Sts	**uValveDC_Sts**		**Valve Status Bits**
AutoAck	*BOOL*	*Decimal*	*Auto Acknowledge*
ManualAck	*BOOL*	*Decimal*	*Manual Acknowledge*
OpenAck	*BOOL*	*Decimal*	*Open Acknowledge*
CloseAck	*BOOL*	*Decimal*	*Close Acknowledge*
RstOpenNumAck	*BOOL*	*Decimal*	*Reset No.of Opens Acknowledge*
AckReqd	*BOOL*	*Decimal*	*Ack.Required*
Manual	*BOOL*	*Decimal*	*Manual Mode*
AutoLock	*BOOL*	*Decimal*	*Locked into Auto*
Fault	*BOOL*	*Decimal*	*Faulted (Alarm)*
Moving	*BOOL*	*Decimal*	*Moving*
Interlock	*BOOL*	*Decimal*	*Interlocked*
oPermissive	*BOOL*	*Decimal*	*Permitted to Open*
cPermissive	*BOOL*	*Decimal*	*Permitted to Close*
OpenReq	*BOOL*	*Decimal*	*Open Required*
CloseReq	*BOOL*	*Decimal*	*Close Required*
iClosed	*BOOL*	*Decimal*	*Closed*
iOpened	*BOOL*	*Decimal*	*Open*
oOpen	*BOOL*	*Decimal*	*Open Output*
oClose	*BOOL*	*Decimal*	*Close Output*
State	*DINT*	*Decimal*	*Valve State*
OpenNum	*DINT*	*Decimal*	*Number of Opens*
OStroke	*REAL*	*Float*	*Opening Time*
CStroke	*REAL*	*Float*	*Closing Time*
TagName	*String_08*		*Tag Name*
TagDescrip	*String_32*		*Tag Description*
Alm	**uValveDC_Alm**		**Valve Alarm Bits**
FTO	*BOOL*	*Decimal*	*Failed to Open Alm*
FTC	*BOOL*	*Decimal*	*Failed to Close Alm*
FB_Error	*BOOL*	*Decimal*	*Feedback Error*
SupplyFault	*BOOL*	*Decimal*	*Power Supply Fault*
CommsFlt	*BOOL*	*Decimal*	*Comms Fault*
aOpenProg	BOOL	Decimal	Auto Open Prog.
aCloseProg	BOOL	Decimal	Auto Close Prog.
OpenPulse	BOOL	Decimal	Open Pulse
ClosePulse	BOOL	Decimal	Close Pulse
aOpen	BOOL	Decimal	Auto Open
aClose	BOOL	Decimal	Auto Close
mOpen	BOOL	Decimal	Manual Open
mClose	BOOL	Decimal	Manual Close

oOpen	BOOL	Decimal	Open Output
oClose	BOOL	Decimal	Close Output
OneShot	DINT	Decimal	One Shot
FTO_T	TIMER		Failed to Open Tmr
FTC_T	TIMER		Failed to Close Tmr
FB_Err_T	TIMER		Feedback Error Tmr
eAlm_T	TIMER		Extension Alarm Timer
Open_T	TIMER		Open Timer
Close_T	TIMER		Close Timer
Stroke_T	TIMER		Stroke Timer
AutoReq_T	TIMER		Auto Request Tmr
ManualReq_T	TIMER		Manual Request Tmr
OpenReq_T	TIMER		Open Request Tmr
CloseReq_T	TIMER		Close Request Tmr
RstOpenNumReq_T	TIMER		Reset No.of Opens Timer

PV+ Write
Remote SCADA Write
HMI Read

APPENDIX B – STANDARD DEVICE ROUTINES

This appendix contains a list of various standardised logic routines used in SHL PLC applications.

Recent project examples containing standardised logic routines are available on request from SHL.

- Digital input setup and processing
- Analogue input setup
- Analogue input processing
- Analogue input sub-routine (non-latched alarms)
- Generator Heater starter with E300 overload relay
- Single supply motor starter with E300 overload relay
- Dual supply motor starter with E300 overload relay
- Single coil valve
- Dual coil valve
- Electrical switchgear (monitoring only)
- GSV module fault capture and logging

APPENDIX C – PROGRAMMING BEST PRACTICES

Computer based systems such as PLCs are being used in all application sectors to perform non-safety functions and, increasingly, to perform safety functions. If computer system technology is to be effectively and safely exploited, it is essential that those responsible for making decisions have sufficient guidance's on the safety aspects on which to make those decisions.

When designing a new control system it is important to follow some guidelines and methods to produce reliable, safe and easy to understand software.

The following sections draw on some IEC standards specifically the IEC 61508 which make recommendations for designing and developing safe and reliable software.

C1. Software Language

The programming software used for developing the PLC programs will be RSLogix 5000. This development software is designed to work with Rockwell Automation's Logix platforms. RSLogix 5000 Enterprise Series software is an IEC 61131-3 compliant software package that offers ladder, structured text, function block diagram, and sequential function chart editors to develop application programs.

Ladder logic will be used for majority of the programming. It is the language most understood by maintenance personnel who may have to interpret code in order to troubleshoot the machine or process. It is also the best language suited for continuous or parallel execution of multiple operations.

Function blocks will generally be used only when doing any PID loop or continuous process control. Calculations in circuit flows and totalising are also easily done in function blocks.

C2. Software Design and Development

Software design and development should conform to standards and suggestions outlined in the standards document IEC 61508-3. This document defines "Detailed Design" to mean software system design – the partitioning of the major components in the architecture into a system of software modules, individual software module design, and coding. In small applications, software system design and architectural design may be combined.

It is good practice to design software is a structured way, including organising the software into a modular structure that separates out (as far as possible) safety-related parts; including range checking and other features that provide protection against input mistakes; using previously verified software modules; and providing a design that facilitates future software modifications.

Some recommendations for software design and development are as follows:

- Using structured methods
- Using semi-formal methods
- Formal methods
- Computer-aided design tools
- Defensive programming
- Modular approach
- Design and coding standards
- Structured programming
- Use of trusted / verified software modules and components

C2.1. Structured Methods

The main aim of structured methods is to promote the quality of software development by focusing attention on the early parts of the lifecycle. The methods aim to achieve this through both precise and intuitive procedures and notations, to determine and document requirements and implementation features in a logical order and a structured manner.

Structured methods are essentially “thought tools” for systematically perceiving and partitioning a problem or system. Their main features are the following:

- A logical order of thought, breaking a large problem into manageable stages;
- Analysis and documentation of the total system, including the environment as well as the required system;
- Decomposition of data and function in the required system;
- Checklists, i.e. lists of the sort of things that need analysis;
- Low intellectual overhead – simple, intuitive, pragmatic;

Another benefit of structured notations is their visibility, enabling a specification or design to be checked intuitively by a user, against his powerful but unstated knowledge.

C2.2. Semi-Formal Methods

Semi-formal methods entail any of the following techniques to assist in the design of the software:

- Logic/Functions Block Diagrams
- Sequence Diagrams
- Data Flow Diagrams
- Finite State Machine / State Transition Diagrams
- Time Petri Nets
- Decision / Truth Tables

Logic / function block diagrams and sequence diagrams are described in IEC61131-3.

C2.3. Formal Methods

The aim of using formal methods is to express a specification unambiguously and consistently, so that mistakes and omissions can be detected.

Formal methods provide a means of developing a description of a system at some stage in its specification or design. The resulting description takes on a mathematical form and can be subjected to mathematical analysis to detect various classes in inconsistency or incorrectness.

C2.4. Computer-Aided Design Tools

The aim of the computer-aided design tools is to use formal specification techniques to facilitate automatic detection of ambiguity and completeness.

The technique produces a specification in the form of a database that can be automatically inspected to assess consistency and completeness.

C2.5. Defensive Programming

The aim of defensive programming is to produce programs which detect anomalous control flow, data flow or data values during their execution and react to these in a predetermined and acceptable manner.

Many techniques can be used during programming to check for control or data anomalies. These can be applied systematically throughout the programming of a system to decrease the likelihood of erroneous data processing.

The following list some of the defensive techniques:

- Variables should be ranged checked including new set points received from a HMI;
- Where possible, values should be checked for plausibility;
- Parameters to procedures should be type, dimension and range checked at procedure entry.

C2.6. Modular Approach

The aim of developing software with modular approach in mind is to decompose a software system into small comprehensible parts in order to limit the complexity of the system.

A modular approach or modularisation contains several rules for the design, coding and maintenance phases of a software project. These rules vary according to the design method employed during design. Most methods contain the following rules.

- A software module should have a single well-defined task or function to fulfil;
- Connections between software modules should be limited and strictly defined, coherence in one software module shall be strong;
- Collection of subprograms should be built providing several levels of software modules;
- Subprogram sizes should be built providing several layers of software modules;
- Subprogram sizes should be restricted to some specified value, typically two to four screen sizes;
- Subprograms should have a single entry and single exit only;
- Software modules should communicate with other software modules via their interfaces – where global or common variables are used they should be well structured, access should be controlled and their use should be justified in each instance;
- All software module interfaces should be fully documented;
- Any software module’s interface should contain only those parameters necessary for its function.

C2.7. Design and Coding Standards

The aim of having design and coding standards is to facilitate verifiability, to encourage a team-centred, objective approach and to enforce a standard design method.

The rules to be adhered to are agreed at the outset of the project between the participants. These rules are made to allow for ease of development, software module testing, verification, assessment and maintenance.

- Typical rules comprise of the following:
- Details of modularisation, for example, interface shapes, software module sizes;
- Use of encapsulation, inheritance (restricted in depth) and polymorphism, in the case of object oriented languages;
- Limited use or avoidance of certain language constructs or mnemonics, for example, “jmp”, “goto”, “equivalence”, dynamic objects, dynamic data, dynamic data structures, recursion, pointers, exits, etc;
- Restrictions on interrupts enabled during the execution of safety-critical code;
- Layout of coding (listing);
- No unconditional jumps (for example, “goto”) in program in higher level languages.

C2.8. Structured Programming

The aim of structures programming is to design and implement the program in a way that it is practical to analyse without it being executed. The program may contain only an absolute minimum of statistically un-testable behaviour.

The following principles should be applied to minimise structural complexity:

- Divide the program into appropriately small software modules, ensuring they are decoupled as far as possible and all interactions are explicit;
- Compose the software module control flow using structured constructs - that is sequences, iterations and selection;
- Keep the number of possible paths through a software module small, and the relation between the input and output parameters as simple as possible;
- Avoid complicated branching and, in particular, avoid unconditional jumps (goto) in higher level languages;

- Where possible, relate loop constraints and branching to input parameters;
- Avoid using complex calculations as the basis of branching and loop decisions.

C2.9. Trusted / Verified Software Modules

Trusted or verified software modules avoid the need for software modules and hardware components designs to be extensively revalidated or redesigned for each new application. Take advantage of designs which have been formally or rigorously verified, but for which considerable operational history is available.

A component or software module can be sufficiently trusted if it is already verified to the required safety integrity level, or of it fulfils the following criteria:

- Unchanged specification;
- Systems in different applications;
- At least one year of service history;
- Operating time according to the safety integrity level or suitable number of demands; demonstration of a non-safety-related failure rate of less then
    - 10-2 per demand (year) with a confidence of 95% requires 300 operational runs (years).
    - 10-5 per demand (year) with a confidence of 99.9% requires 690,000 operational runs (years)
- All of the operating experience must relate to a known demand profile of the functions of the software module, to ensure that increased operating experience genuinely leads to an increased knowledge of the behaviour of the software module relative to that demand profile;
- No safety-related failures.

To enable verification that a component or software module fulfils the criteria, the following must be documented:

- Exact identification of each system and its components, including versions numbers (for both software and hardware);
- Identification of users, and time of application;
- Operating time;
- Procedure for the selection of the user-applied systems and application cases;
- Procedures for detecting and registering failures, and for removing faults.