Digi-SFT
User's Guide

Bes-Tech
Saving Energy. Building Comfort.

©2016 Bes-Tech, Inc. All rights reserved.

This document and the information contained within are the property of Bes-Tech, Inc. and delivered on the express condition that it is not to be disclosed, reproduced in whole or in part, or used for manufacturing purposes by anyone other than Bes-Tech, Inc. without its written consent. No right is granted to disclose or so use any information contained in said document.

Bes-Tech reserves the right to change or modify the information or product described without prior notice and without incurring any liability.

Use of the software is provided under a software license agreement. Unauthorized use of the software or related materials discussed in this manual can result in civil damages and criminal penalties. The terms of this license are included with the software installation.

Digi-SFT™ is a trademark of Bes-Tech, Inc.

All brand names, trademarks, and registered trademarks are the property of their respective owners.

Last updated: 12/14/2016

Bes-Tech, Inc.

4640 S 59th Street

Omaha, NE 68117

Phone: (402) 502-2340

Fax: (402) 502-2283

www.bes-tech.net

©2016 Bes-Tech, Inc. All rights reserved.

Table of Contents

1 Introduction

Digi-SFT is a supervisory control and data acquisition (SCADA) software that is a subsystem of the energy management and control system (EMCS). The SCADA software is used to manage building systems for the purpose of providing occupant comfort and reliable, energy efficient facilities.

With this software, the user can monitor and control the EMCS over an internet connection. A graphical view of the system displays real-time data and allows the user to modify the system operation. A table view of the system compliments the graphics with options to search the database. Custom algorithms can be written in the Distributed Control Language (DCL) and downloaded to execute on the field computers. A scheduling feature is provided to schedule when the equipment is depowered and the facilities are unoccupied.

The alarm management features include the ability to define, view, and acknowledge alarms. Custom algorithms can be created to trigger an alarm that is annunciated to the desktop of specified individuals or via email or text message.

Metering and performance system data is saved and can be easily reviewed for sensors, controllers, and equipment connected to the system. The operational data can be viewed in real-time in plots or on the graphical display. The interface to database is seamless from a users' perspective. Database replication is used for robust system operations when part of the network is unavailable. When the network is available again, the system will automatically synchronize the replicas to ensure the field computer contains all of the updated information in the server.

The security features include operation over an encrypted secure sockets layer, access control with various levels of authority to the system, and a log of system activity. The system utilizes 128-bit SSL protocol for initial client login; SSH secure tunnels for all other connections. The field computers communicate with the control devices over RS-485 serial communication.

1.1 Important Things to Know before Using this Guide

This user's guide is written with the expectation that the user has a general knowledge of the basic principles of building automation and HVAC systems and familiarity with Windows operating system. Certain advanced features are intended for use by a facility engineer or system administrator. For those features, advanced knowledge of the system and computers is expected.

Any innovative technologies provided by Digi-SFT will be thoroughly described so that you know how to use the technology and the purpose for applying the feature.

The base system is supplied with some recommended conventions so that it can be quickly and easily deployed in your facilities. This tool allows you the flexibility to establish your own conventions that meet your specific needs. These tools are described in this guide where appropriate.

1.2 More Information

This guide covers all of the features of Digi-SFT. For additional assistance on how to use the tool, please refer to the on-line help provided with Digi-SFT.

The tutorial explains how to use Digi-SFT for typical daily system operation.

The installation guide provides details for installing the system.

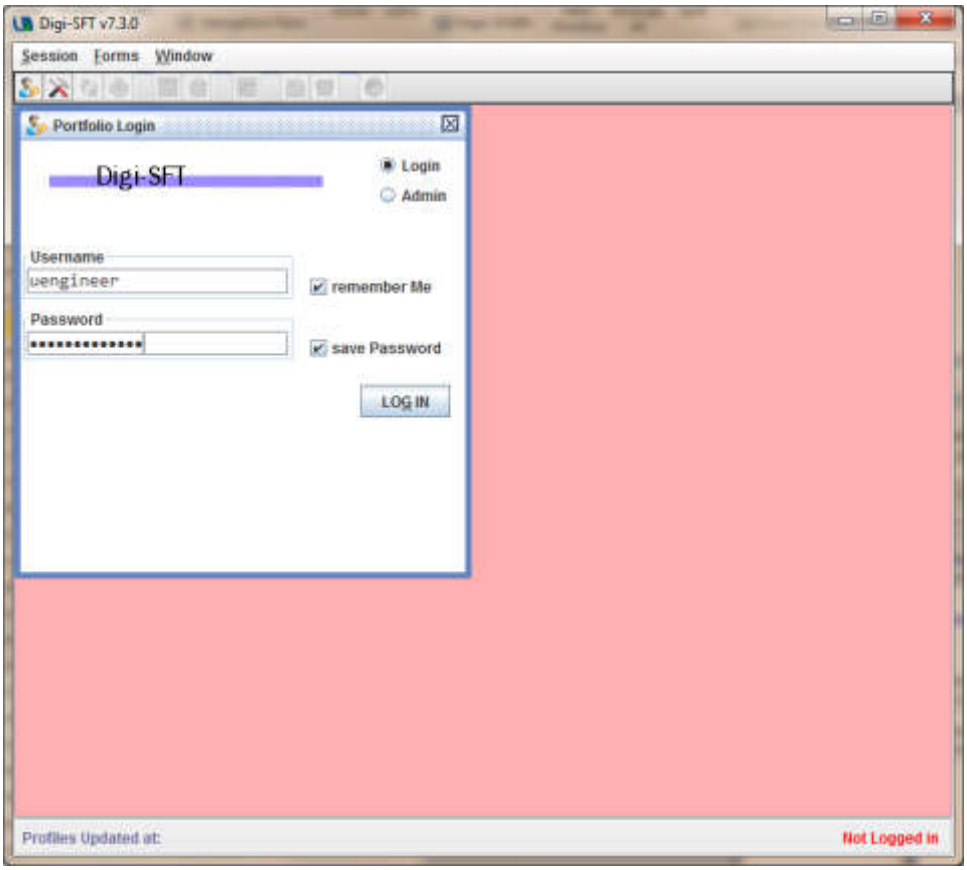
The quick start guide provides basic operation examples.

See the glossary in this document for the definition of terms.

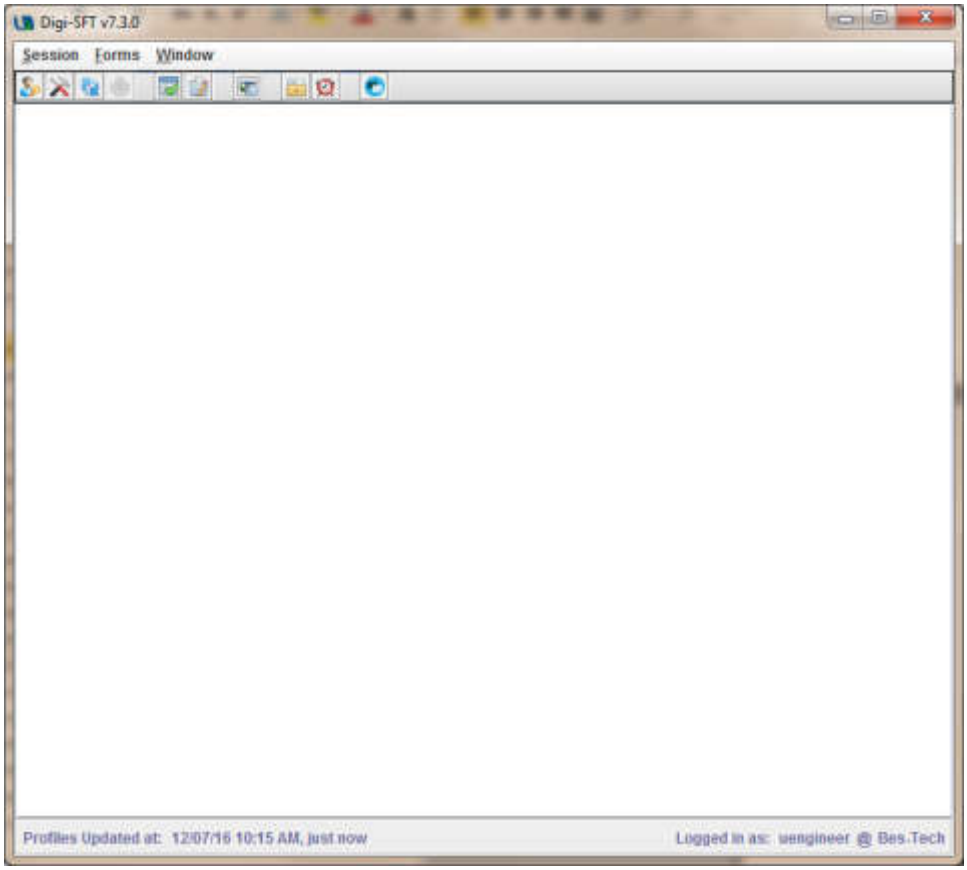
2 Using the User Interface

2.1 Login

The login window will pop up after software started. Enter the assigned username and password and click “LOG IN”



After user successfully logged in, background color will turn to white and login window disappeared.



2.2 Menus and Toolbars

The user interface includes menus and toolbars that enable user to quickly find and execute commands. The pull-down menus are shown below.

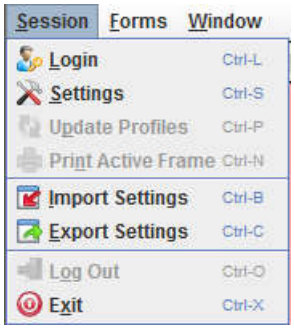


Figure 1 - Session Menu

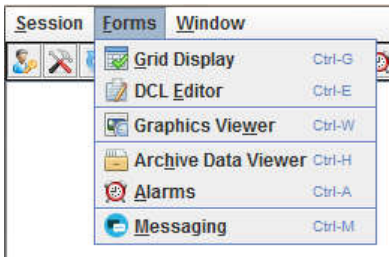


Figure 2 - Forms Menu

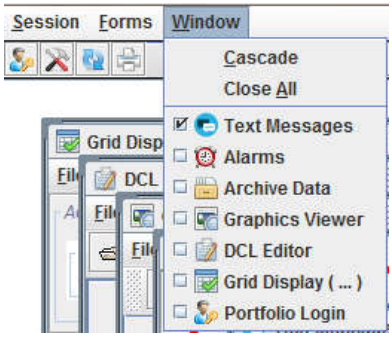


Figure 3 - Window Menu

In addition to "Cascade" and "Close All", the Window pull-down menu lists all of the open windows. When you select a window from the list, it will bring it to the top and make it active.

The toolbar includes the most commonly used commands. Figure 4 illustrates the commands that appear on the toolbar. Some are available only when your account at admin level.

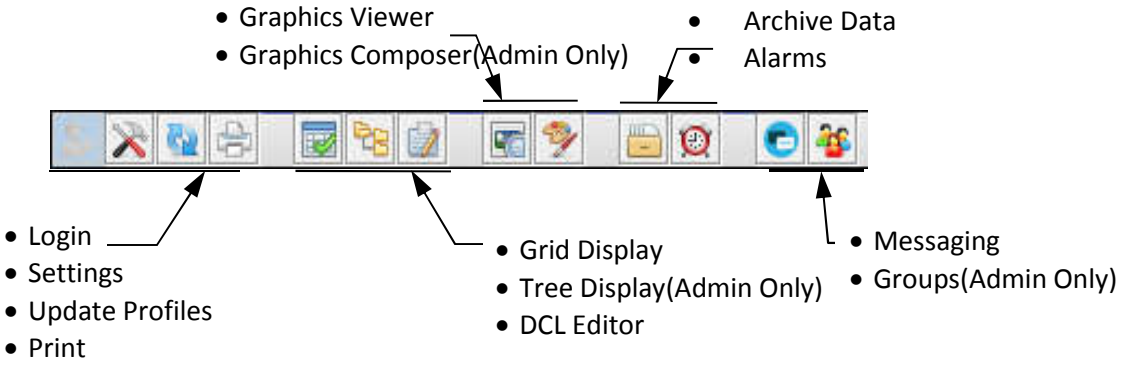


Figure 4 - Toolbar

2.3 Tree Display

The tree display provides a hierarchical view of the objects in the system.

2.3.1 Pop-Up Menu (right click)

The commands available from the pop-up menu on the tree display are batch commands. When multiple objects are selected, the command is performed on each object, one at a time.

2.3.1.1 Compile Control Block(s) (Admin Only)

You may compile control blocks by first selecting one or more control blocks and then selecting “compile control block(s)” command from the pop-up menu. The control block is compiled to binary stack code, called s-code. If successful, it will download and run on the appropriate hardware platform.

Control blocks are compiled one at a time. Their compilation status appears in a batch process results window.

An error window appears if no control blocks are selected.

2.3.1.2 Initialize Controller(s)

Use this command to initialize the controllers (hardware objects). First select one or more controllers then choose “initialize controller(s)” from the pop-up menu.

When a controller is initialized, all of the existing objects will be discarded from the controller. This is immediately followed by a download of all controller resident objects currently defined in the database to the controller. Once all objects are downloaded, the controller resumes operation. This is also useful when switching to a new controller, initializing will copy all needed information from server to new controller.

The controllers will be initialized one at a time and their initialization status will appear in a batch process results window.

Note: there is only one controller under per field computer, and should be created by Bes-Tech admin.

An error window will appear if no controllers are selected.

2.3.1.3 Reset Controller(s)

The reset command reboots the controller.

2.3.1.4 Set Controller(s) Online

Deprecated. This action will take no effect. The controller is always online.

2.3.1.5 Set Controller(s) Offline

Deprecated. This action will take no effect. The controller is always online.

2.4 Graphics Viewer

The graphics viewer provides an intuitive system interface that enables users to monitor and control the system based on a graphical representation.

A graphic drawing of each specific EMCS system needs to be created using the Composer (see Section 5) by Bes-Tech admin. The graphic image is to be named accordingly to the acronym of the graphic object it represents (see Section 5.3). The graphic can then be displayed in the Graphic Viewer showing actual values of the specific controls it displays (see Figure 5).

The user can select one or multiple graphic objects in the Grid display and then use the View/Control Graphic pop-up menu item to display them in the Graphics Viewer.

The Graphics Viewer can also be invoked from the toolbar, in which case the user needs to use the Open menu item or toolbar button in order to open the Object Selector Dialog for typing in the acronym of the graphic object that needs to be displayed (see Figure 6).

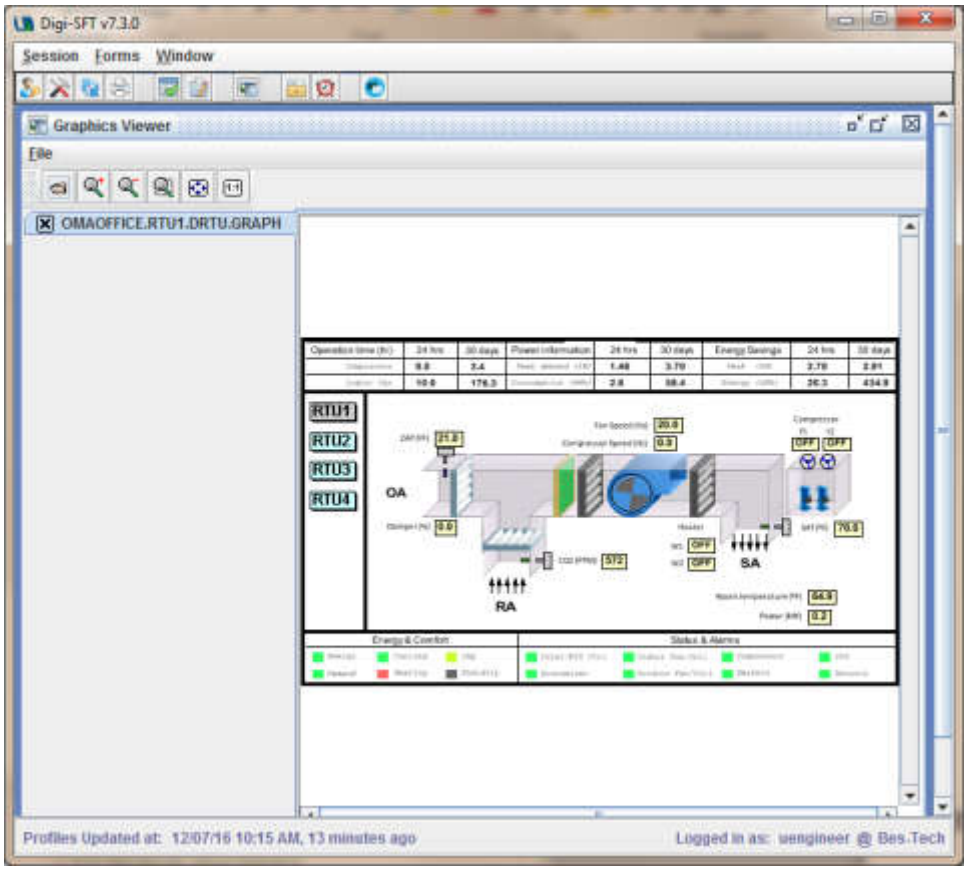


Figure 5: Graphics Viewer display

2.4.1 File Menu

'Open' menu item invokes the Object Selector Dialog for selecting the graphic object to be displayed in the Graphics Viewer.

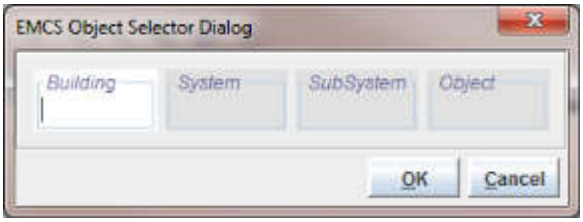


Figure 6: Object Selector Dialog

'Close' menu item closes the graphic displayed in the currently selected tab.

'Close All' menu item closes all the graphics currently displayed in the Graphics Viewer.

'Exit' menu item closes all the graphics displayed and then disposes the Graphics Viewer window as well.

2.4.2 Toolbar

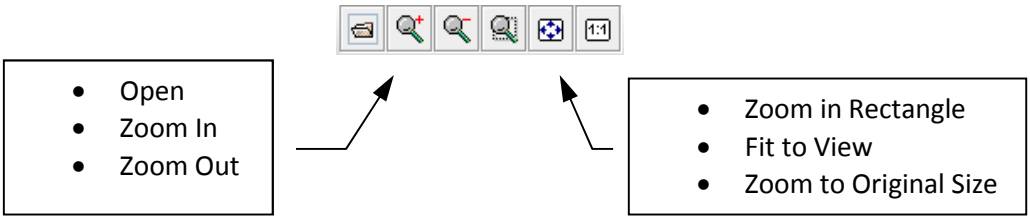


Figure 7: Graphics Viewer Toolbar

Toolbar buttons (see Figure 7) offer the possibility of opening new graphics in the viewer and scaling the opened graphic as the user prefers: zoom in, zoom out, zoom selected rectangle, fit to view, zoom to original graphic size.

2.4.3 Auto Refresh

The user can select to enable automatic graphic updates by clicking on the Enable Auto Refresh button. Default refresh rate is set to 60 seconds. If the user has admin rights, minimum refresh rate (in seconds) is 1s, otherwise 10s. Once the automatic refresh gets enabled, object values displayed in the active graphic are updated at the specified rate.

2.4.4 User actions

The user can click on object representations in the Graphic Viewer in order to perform specific actions.

In case a graphic link gets clicked (Figure 5: blue rectangles on the left) then the selected graphic gets opened in a new tab.

In case a value entry gets clicked (Figure 5: yellow or purple rectangles) the correspondent object's Field Information is displayed (see Section 3 for more information).

In case a value input gets clicked (Figure 5: green rectangles) the Input Value window (see Figure 8) gets displayed allowing the user to change the input value of the selected object.

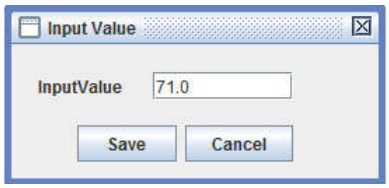


Figure 8: Input Value window

2.5 Grid Display

The grid display is a powerful system interface intended to give expert users quick access to objects and control of the system.

2.5.1 Menu

2.5.1.1 File

The file menu provides a command to print the information currently listed in the Grid Display.

2.5.1.2 Columns

The Grid Display is composed of numerous columns filled with data. Some of the columns can be turned off to reduce the amount of information on the grid and simplify the display. Turn a column off by deselecting it from the list of columns in the columns menu.

2.5.2 Columns

2.5.2.1 Sort

The columns in the Grid Display for database objects can be sorted in ascending and descending order.

The columns for the field objects cannot be sorted.

2.5.2.2 Changing Width and Position

The width of a column can be modified by moving the cursor to the top of the column and to the side until it changes to a bidirectional arrow. Grab the side of the column and drag it left or right.

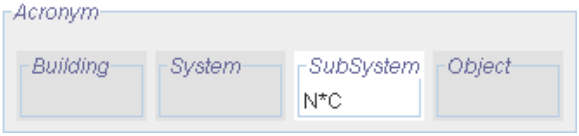
Change the position of a column so that it appears next to a different column by moving the cursor to the top of the column. Grab the top of the column and drag it left or right across the screen to move it next to a different column.

2.5.3 Acronym

The acronym is the four part alpha-numeric acronym assigned to the object. It includes a building acronym, system acronym, subsystem acronym, and an object acronym.

Any part of the acronym can be used as a filter to search for objects in the EMCS. Simply type in the partial or complete acronym and click search. The results will appear in the Grid Display.

In the following example, the filter lists only those objects that have a subsystem acronym starting with an “N” and third character “C”. The “*” is a wildcard that represents any character.



As characters typed into one of the four fields, a list of filtered objects will appear in a dropdown list. Select from this list to automatically populate the field with the acronym or use what you type.

2.5.4 Grid Options

2.5.4.1 Field

The grid display gives data on the objects from the field or from the database. Choose the field option to display data that includes information about the objects residing on hardware in the field. The field objects may include the following:

- Analog input
- Analog output

- Control block
- Digital input
- Digital output
- Graphic
- Hardware
- Loop
- Schedule

The grid display will provide the following information about field objects:

- Object acronym – The four part alpha-numeric acronym assigned to the object.
- Command, set point, input – Displays the numeric value of the input to an object and defines it is a command, set point, or other type of input.
- Feedback, measured value, output – Displays the numeric value of the output of an object. If the object is an analog output representing a sensor, for example, the measured value will be displayed.
- State, version – Displays the current state and allows the user to change the state to any of the following six states: Resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more on control block states.
- Status – Displays the current control block status and allows the user to change the status. Refer to section 4.1.2 for more on control block status.
- Lock, date – Displays the lock value. The lock value is one of the following four values: Normal, medium, high, and lock-out. It is assigned to controllable objects such as analog outputs, control blocks, Digital outputs, and loops.
- Key, time –Displays the key value. The key value is one of the following four values: Normal, medium, high, and lock-out, that can be assigned to an object to control other objects, such as control blocks and users.
- Report –Displays a message about the object when a report message has been written to the object.

2.5.4.2 Database

The grid display can show data about objects from the field or from the database. Choose the database option to display data that includes information about the objects saved in the database. The following objects types may reside in the database:

- Alarm (database grid option only)
- Analog input
- Analog output
- Control block
- Digital input
- Digital output
- Graphic
- Hardware
- Loop
- History (database grid option only)
- Schedule
- User (database grid option only)

The grid display will provide the following information about database objects:

- Type – The type of object, see section 4.2.4 for object types.
- Building, system, subsystem, and object acronym – The four part alpha-numeric acronym assigned to the object.
- Display sequence – The display sequence is a user defined, numeric value that may be used to sort objects in the grid display by their display sequence number in ascending or descending order.
- Object identifier – A three part numbering scheme that uniquely identifies objects with a top, middle, and bottom assigned by the system based on the type of object being created and the object locale.

2.5.5 Search/Filter

Several methods of filtering the data in the system is provided to help you quickly find the information you are looking for.

2.5.5.1 Object Types

The list of objects that appear in the grid display can be filtered according to the type of object, including the follows:

- Alarm (database grid option only)
- Analog input
- Analog output
- Control block
- Digital input
- Digital output
- Graphic
- Hardware
- Loop
- History (database grid option only)
- Schedule
- User (database grid option only)

The filter allows for the selection if multiple object types shown in the grid display.

2.5.5.2 Display (All, Normal, Critical)

The object definition window includes a display priority that can be defined when creating or editing objects. The display priority defines one of three possible selections, “DISPLAY_ALL”, “DISPLAY_NORMAL”, and “DISPLAY_CRITICAL”. Selecting to display the critical objects in the grid display will only show objects with the DISPLAY_CRITICAL priority. Selecting to display the normal objects will display objects with the DISPLAY_NORMAL and display critical priorities. Finally, selecting to display all objects will display all objects.

2.5.5.3 Search

When you have selected all filtering and search options, click the search button to search the system. This may include the object acronym, object type, and display priority. Objects that match your criteria will be presented in the grid display.

2.5.5.4 Refresh

The grid display presents static information to allow users to read the values even though the EMCS is constantly changing. Refresh will update the grid display to give the most current information.

Note that if you change the search and filtering options, you must click on the search button to regenerate the list of objects in the grid display. And a Search will automatically carry out Refresh action thereafter.

2.5.6 Views

A view is a snap-shot of the search and filter options. A view is provided to allow users to quickly search without having to enter in the search and filter options.

2.5.6.1 Save Current Filter as View List

You can save the current search and filter options as a view list by selecting the save icon under views. You will then be prompted for the view name, the scope, and description. The scope allows you to create a global view list that can be used by all system users. As an option, you can also create a personal view list that can only be used by you.

2.5.6.2 Show/Hide View List

You can view the pane for the view list by selecting the show/hide view list icon under views.

Double click on the desired view to apply the filter settings.

2.5.6.3 Search View List

The view list includes a search option that allows you to type in characters in order to narrow the list of displayed view names.

2.5.6.4 Delete View List

Delete a view by first highlighting it on the view list, then right click, and select “delete selected view”.

2.5.7 Enable Auto Refresh

Auto refresh periodically updates the data in the grid display. When you select to enable auto refresh, you are provided a field to define how often the display is updated.

2.5.8 Context Sensitive Pop-Up Menu (right click)

The pop-up menu provides various commands to allow users to manage objects and the EMCS from the grid display. In order to access the pop-up menu, first view the list of field objects in the grid display. Select the grid options you prefer and search. Then select one or more of the objects from the filtered list of objects displayed in the grid display, right-click, and the pop-up menu will appear. To select multiple objects, hold down the Shift or Ctrl key while selecting objects with the mouse. The following commands are available from the pop-up menu:

2.5.8.1 View/Modify Object Definition

The object definition window allows users to view and edit the static object information, which varies for different object types.

2.5.8.2 View/Modify Field Information

The field information window displays dynamic information for field objects. It also provide interface for user to control to these objects.

2.5.8.3 Edit Control Block/Schedule

A control block represents a piece of computer program written in the Distributed Control Language. A schedule is a specialized control block that outputs ON/OFF status based on current date and time. Control block will be opened in a text based editor, and Schedule will be opened in a graphical editor which looks like a calendar.

2.5.8.4 Real-Time Plot

The real-time plot graphs data of an object in real-time from the EMCS. If multiple objects are selected at the same time, they will appear in the same plot window.

2.5.8.5 Trend Plot

A trend plot graphs object data. If multiple objects are selected at the same time, they will appear in the same plot window.

2.5.8.6 History Plot

A history plot graphs the data of a History object. Users can choose to plot multiple objects or a single object in the same window.

2.5.8.7 Save Multiple Histories

The saving Multiple Histories data option is available for history objects. Users can save the data by selecting multiple history objects. The saved data can be exported to a spreadsheet in csv file format, with columns including:

- the date and time the data was recorded
- the acronym of the object
- the text message or history string
- the data of the object

2.5.8.8 Save Multiple Trends

The multiple Trends function exists for certain objects, such as CB, AI, AO, DI and DO. You can save the data by selecting multiple objects. The trend data can be exported to a csv file. Such data includes:

- the date and time the data was recorded
- the acronym of the object
- the text message or trend string
- the data of the object

2.5.8.9 Save ALL CBs (Admin Only)

Users can save all CBs by selecting this function. The saved data can be exported to txt files in a folder.

2.5.8.10 View/Control Graphic

A graphic is an object that is an interactive illustration used to display real-time information on any system part and to control the system output devices.

2.5.8.11 Get Archive Data

The Archive Data function exists for certain objects, such as history objects and alarms. Retrieve and view the data by selecting an object and choosing to get the archive data. The data presented depends on the type of object. For example, the archive data for a history object may give the following information:

- the date and time the data was recorded
- the acronym of the object
- the text message or trend string
- the data of the object

The archive data for an alarm object will show:

- the date and time the alarm was recorded
- the object that reported the alarm
- the text message or alarm string
- the number of times the alarm was triggered
- the number of notifications
- the user who acknowledged the alarm
- the time and date the alarm was acknowledged
- the cascading alarm

If multiple objects are selected at the same time, they will appear in different archive data windows.

2.5.8.12 Create New Object

See section 3.1.

2.5.8.13 Create Similar Object

See section 3.1.

2.5.8.14 Create I/Os from XML

See section 3.1.

2.5.8.15 Delete Object

See section 3.1.

2.5.8.16 Check Dependencies

A control block may be dependent upon other objects if they are referenced in the control block. Digi-SFT will not allow deletion of any referenced objects where such

dependencies exist. When an object is highlighted, check dependencies will list the objects upon which the selected object has a dependency.

2.5.8.17 Batch Process Commands

The batch commands available from the pop-up menu on the grid display are provided to automate the execution of the commands on multiple objects. The command will be performed on each object, one at a time.

2.5.8.17.1 Find Controller(s)

Use this command to check the status of controllers (hardware objects) by first selecting one or more controllers and then choosing “find a controller” from the pop-up menu.

The status of the controllers will be checked one at a time and appear in a batch process results window.

2.5.8.17.2 Set Selected HW Online/Offline

See section 2.3.1.4 and 2.3.1.5.

2.5.8.17.3 Initialize Selected Controllers

See section 2.3.1.2.

2.5.8.17.4 Reset Selected Controllers

See section 2.3.1.3.

2.5.8.17.5 Compile Selected Control Blocks

See section 2.3.1.1.

3 Managing Objects

Digi-SFT is composed of building blocks known as objects. The objects represent physical hardware, such as computers and controllers. In some cases, the objects represent virtual concepts such as alarms and control blocks. This section describes how to manage objects. Managing objects is necessary, for example, when first installing a new controller in the energy management control system so that Digi-SFT can recognize the additional hardware.

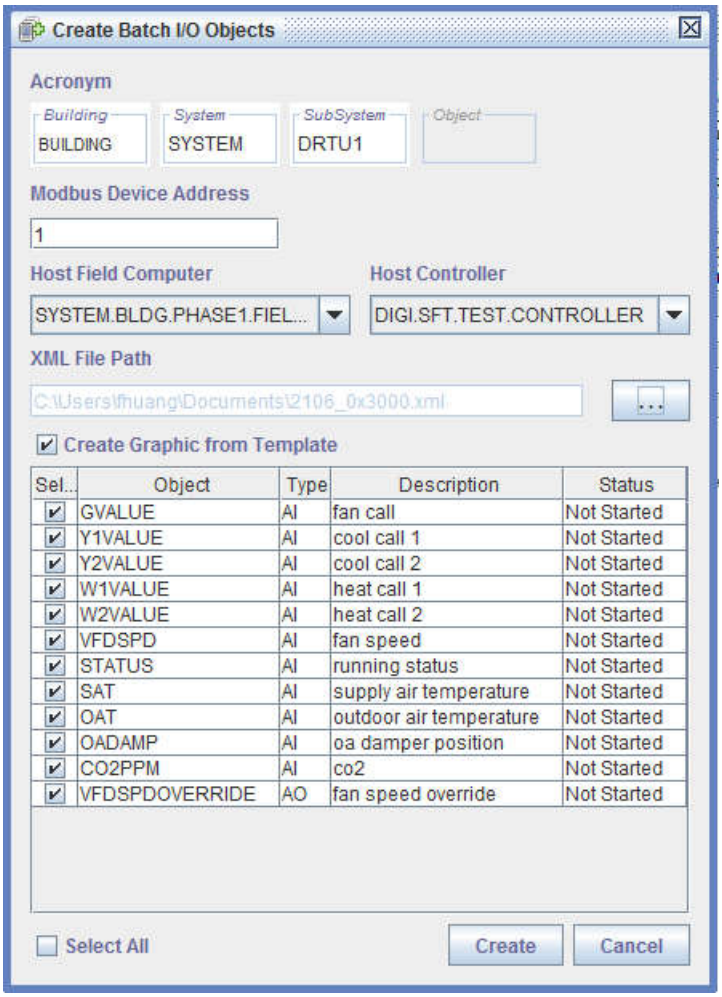
3.1 Create, Create Batch I/O, Create Similar, Delete

Access the pop-up menu from the grid display to create or delete objects. Select the grid options you prefer and search. Then select one or more of the objects from the filtered list of objects displayed in the grid display, right-click, and the pop-up menu will appear. To select multiple objects, hold down the Shift or Ctrl key while selecting objects with the mouse.

You can create a new object of any type by choosing to create a new object. The object definition window enables users to edit the settings of the new object. Different types of objects have different definitions. For more information, see the specific object type in this section.

You can create a batch of I/O points from a pre-defined xml mapping file. This XML file is typically made for a Bes-Tech product, such as Digi-RTU, Digi-VAV. User has to input a combination of Acronym without the Object field, which will be generated from xml entries. A

valid Modbus address is also required, and will be applied to every point in the list. User can choose only to create a subset of all the points within the xml file. After choosing the xml file, the file will be parsed and a list of base information will be filled into table below. If a file of extension “ivl” also exists in the same directory, then user has an option of creating a graphic object along with the points. The “.ivl” file is the predefined template file, distributed along with xml file.



When you select an object in the grid display and choose to create a similar object, the system will open an object definition window that is a copy of the original object selected. The object definition window allows the user to edit the settings of the new object. You will need to change the object acronym in order to save the new object. Different types of objects have different definitions. For more information, see the specific object type in this section.

Delete an object by selecting it in the grid display and choosing “delete object”. Digi-SFT will not allow the deletion of an object that another object depends upon. Therefore, the dependency must first be eliminated before an object can be deleted.

3.2 Shared Object Definition

All objects are required basic information including:

3.2.1 Acronym

The acronym is the four part alpha-numeric acronym assigned to the object. It is comprised of a building acronym, system acronym, subsystem acronym, and an object acronym, 15 characters max for each field.

3.2.2 Type

The type defines the selected object type.

3.2.3 Description

The description is an optional field that may be used to describe the object.

3.2.4 Display Priority

The display priority defines one of three possible selections used to filter objects in the grid display. These are “DISPLAY_ALL”, “DISPLAY_NORMAL”, and “DISPLAY_CRITICAL”. This attribute determines whether the object will be shown in Grid display with desired display level.

3.2.5 Display Sequence

The display sequence is a user defined, numeric value that may be used to sort objects in the grid display by their display sequence number in ascending or descending order. When listing a list of point, Grid display will use this number after compared Building and System, and before comparing Subsystem and Object.

3.2.6 Display Format

The display format defines how the numeric value is displayed in the grid display. For example, it indicates the number of decimal points and whether it is in scientific notation.

3.3 Alarms

An alarm is a type of object that provides an audible or visual warning of a problem or condition. For more information on alarms, see section 6.

3.3.1 View/Modify Object Definition

The object definition window shown in Figure 9 provides you the ability to view and edit the settings of an object. To view this window, open the grid display, select DB option and check on Alarm type in Object Type Selector, then Search.

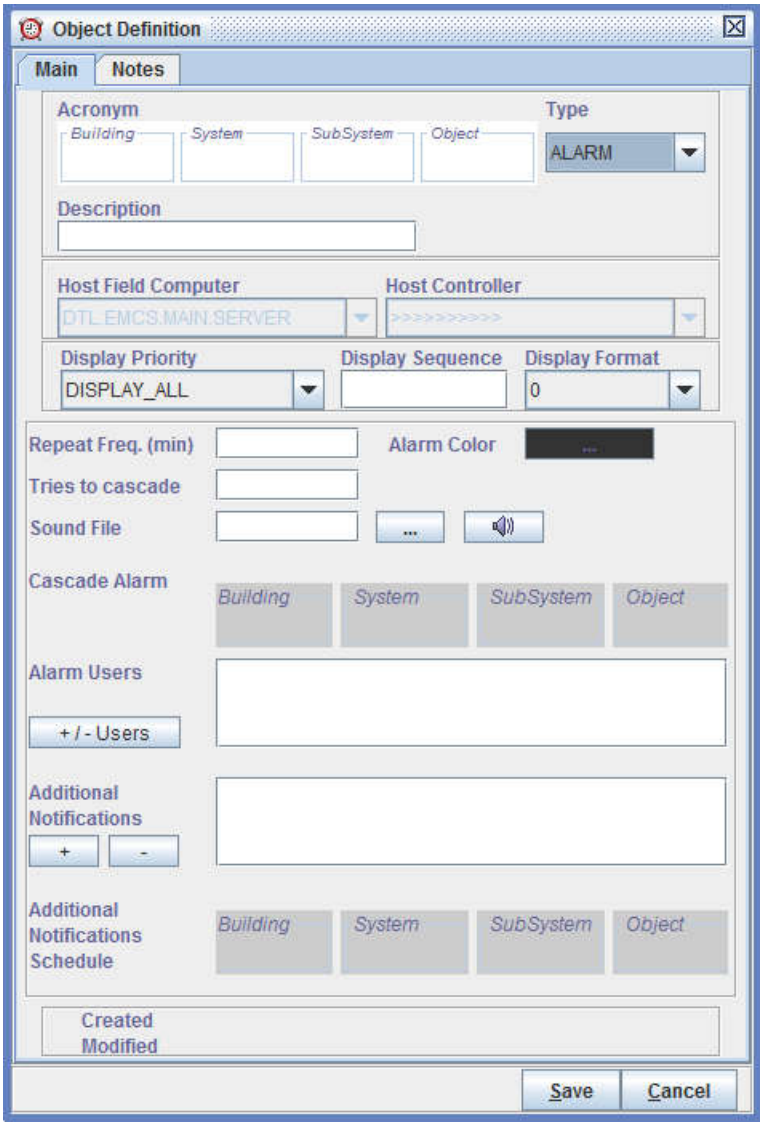


Figure 9 - Alarm Object Definition Window

3.3.1.1 Repeat Frequency

Alarm users will periodically be notified of an alarm that has been triggered at the repeat frequency until it is acknowledged. The repeat frequency can range in time from 1 minute to 2147483647 minutes.

3.3.1.2 Tries to Cascade

Notification of the alarm will reoccur at the repeat frequency until the alarm has been acknowledged. If the number of notifications equals the number of tries to cascade and the alarm still has not been acknowledged, it is then escalated by cascading or triggering another alarm object.

3.3.1.3 Alarm Color

When the alarm is triggered the alarm text color is displayed in the alarm window. Each alarm is assigned a different color as a quick visual indicator of which alarm has been set.

3.3.1.4 Sound File

The sound file is a waveform (.wav) audio format that plays when the alarm is triggered.

3.3.1.5 Cascade Alarm Acronym

If the alarm has not been acknowledged and the number of notifications equals the number of tries to cascade, the alarm is escalated by cascading or triggering another alarm object. The cascade alarm acronym is the four part alpha-numeric acronym assigned to the other alarm object.

3.3.1.6 Alarm Users

Triggered alarms are automatically routed to those users listed as the alarm users. Users are notified of triggered alarms via an alarm window. Alarm users can be selected from the list of users in the database. Note enlisted users will not get email even if they already have email on profile. Use Additional Notification instead.

3.3.1.7 Additional Notification

The additional notification is a list of email addresses that will be sent a message when the additional notifications schedule is applied. This same method can be utilized to send text pages. Your text message system must be capable of sending text pages upon receipt of an email.

3.3.1.8 Additional Notification Schedule

The notification schedule defines when the addition notification should be sent. This may be useful for notifying key personal of alarms when they are scheduled to be on-call and may not be working on a computer.

3.4 Control Blocks

A control block is an object that is a computer program written in the Distributed Control Language.

3.4.1 View/Modify Object Definition

An object is managed using the object definition window, shown in Figure 10. To view this window, open the grid display, select the grid options you prefer, and search.

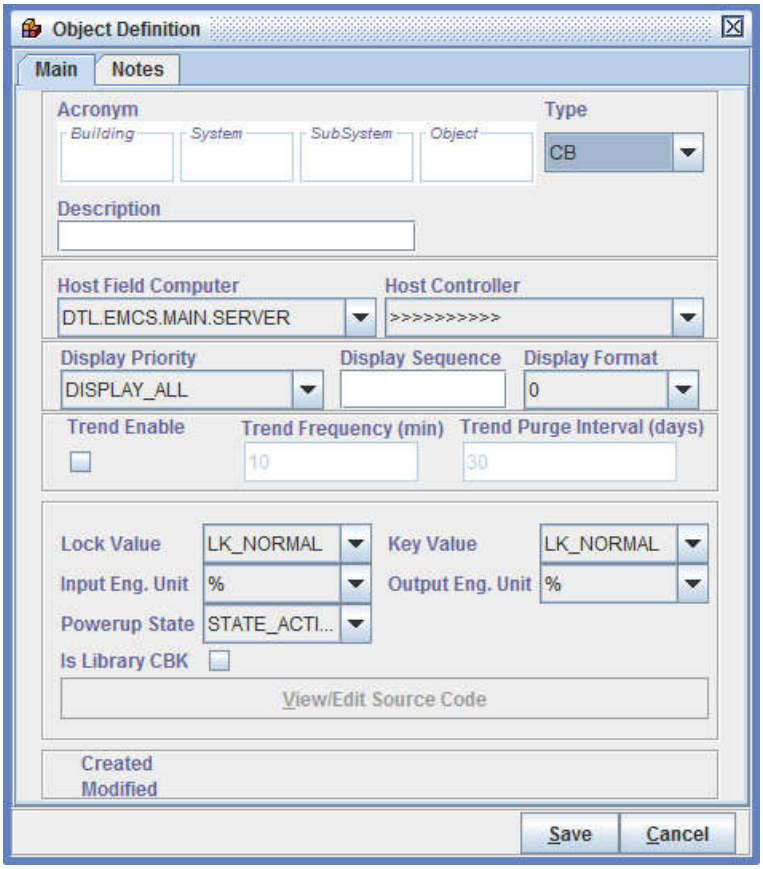


Figure 10 - Control Block Object Definition Window

3.4.1.1 Host Field Computer

The host field computer is used to define the acronym of the field computer that manages the hardware where the object resides. The object can also reside on the field computer itself. Control blocks may reside on the server, or a field computer.

3.4.1.2 Host Controller

The host controller is used to define the acronym of the controller that manages the object. If the control block is saved on a server or field computer this field should be left blank. Host Controller should always be default blank, starting from centralized version. No CB should reside in controller.

3.4.1.3 Trend Enable

The trend enable defines whether the input and output value of the object is saved to the trending database.

3.4.1.4 Trend Frequency

The trend frequency defines how often the I/O value of the object is saved to the trending database. The minimum number is 10 for normal user and 10 for system admin.

3.4.1.5 Trend Purge Interval

The trend purge interval defines the maximum amount of time the I/O value of the object is saved in the trending database before it is deleted. The trend purge interval is set in increments of one day and can range from 30 to 65,000.

3.4.1.6 Lock Value

The lock value is one of four values: normal, medium, high, and lock-out. These can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops.

3.4.1.7 Key Value

The key value is one of four values: normal, medium, high, and lock-out. These can be assigned to an object to control other objects, such as control blocks and users. An object can control the controllable object when the key value is greater than or equal to the lock value.

3.4.1.8 Input Engineering Unit

The "input engineering unit" is used to define the engineering unit of the standard predefined input variable of the object. The grid display shows the value of the standard predefined input variable of the object and its units.

3.4.1.9 Output Engineering Unit

The "output engineering unit" is used to define the engineering unit of the standard predefined output variable of the object. The grid display shows the value of the standard predefined output variable of the object and its units.

3.4.1.10 Power-up State

When a control block is first executed, it will be initialized to the power-up state. This can be one of the following six states: resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more information about control block states.

3.4.1.11 Library Control Block

Algorithms can be reused throughout the EMCS by defining a control block as a library control block object and making reference to the library using the #include statement and the respective function call in another control block. In order to create a library control block, select the library control block option in the object definition window.

A library control block is saved on the server and will not be executed. A control block that is not a library can still be #include[d] in other control blocks. However, it will be executed and appear in the Grid Display as a stopped control block since it does not have a main() function.

3.4.1.12 View/Edit Source Code

Selecting "view/edit source code" opens the control block in the DCL Editor. User needs EditCB permission to be able to modify the code, and DBModify permission to view it. For more information, see section 4.11.

3.4.2 View/Modify Field Information

The field information window, shown in Figure 11, displays the values for the control block field information and allows users to set or change some of those values. User needs to have "Control" permission to be able to control the point, otherwise it's read only.

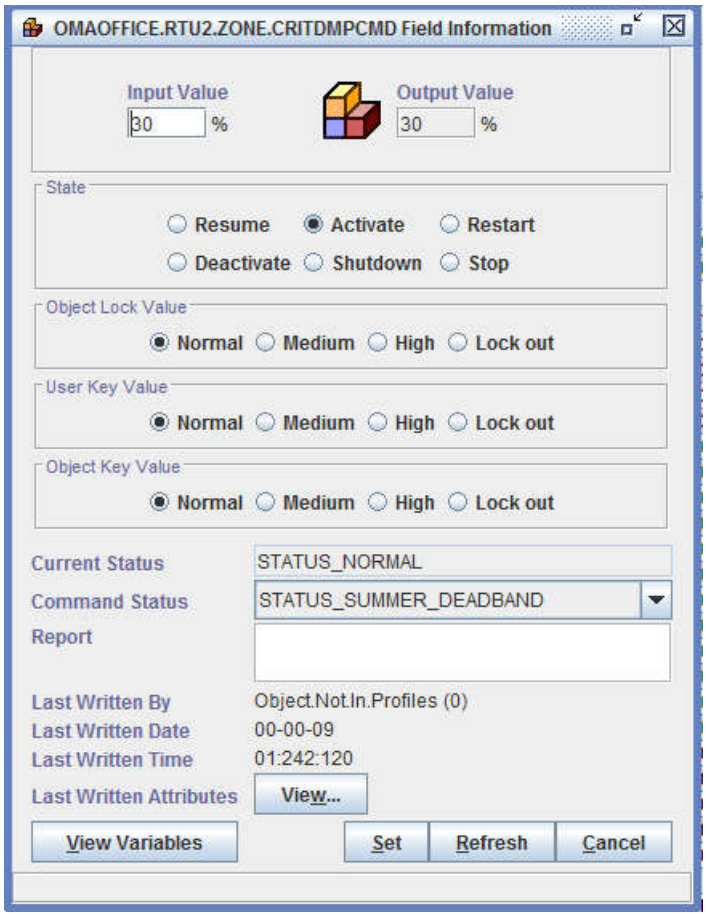


Figure 11 - Control Block Field Information Window

3.4.2.1 Input Value

Displays the current value of the standard predefined input variable and allows you to change that value.

3.4.2.2 Output Value

The output value field displays the current value of the standard predefined output variable.

3.4.2.3 State

Displays the current state and allows the user to change the state to any of the following six states: resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more about control block states.

3.4.2.4 Object Lock Value

The lock value is one of four values: normal, medium, high, and lock-out. These can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object. Users can change the object lock value in this field.

3.4.2.5 User Key Value

The user key value is one of four values: normal, medium, high, and lock-out. It can be assigned to an object to control other objects, such as control blocks and users. An object can control the controllable object when the key value is greater than or equal to the lock value. The current key value is displayed for the object. Users can change the key to any value in this field.

3.4.2.6 Object Key Value

Similar to User Key Value, the Object Key Value is the key when this CB is writing to other points.

3.4.2.7 Current Status

Display of the current control block status. Refer to section 4.1.2 for more information about control block status.

3.4.2.8 Command Status

Displays the current command status and allows the user to change this status.

3.4.2.9 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more information about the report object variable method.

3.4.2.10 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that were most recently defined is displayed by selecting to view the last written attributes.

3.4.2.11 View Variables

If the user selects the view variables option, a list of functions that are used by the control block is provided. All of the defined variables in each function are given along with their current values. The updated values can be obtained by selecting the refresh button. The user may select any one of the values and type over the current value to change the value of the variable.

3.4.2.12 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.4.2.13 Refresh

The field information window presents static information to allow you to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.4.2.14 Last Written By, Date, Time

Users who made changes to data in the field information window are displayed along with the date and time those changes were set.

3.5 Graphics

A graphic is a type of object that is an interactive illustration. It can be used to display real-time information for any part of the system and control the system output devices.

3.5.1 View/Modify Object Definition

An object is managed using the object definition window shown in Figure 12. In order to view this window, first open the grid display, select the database, "DB", as the grid option along with any other preferred options and click search.

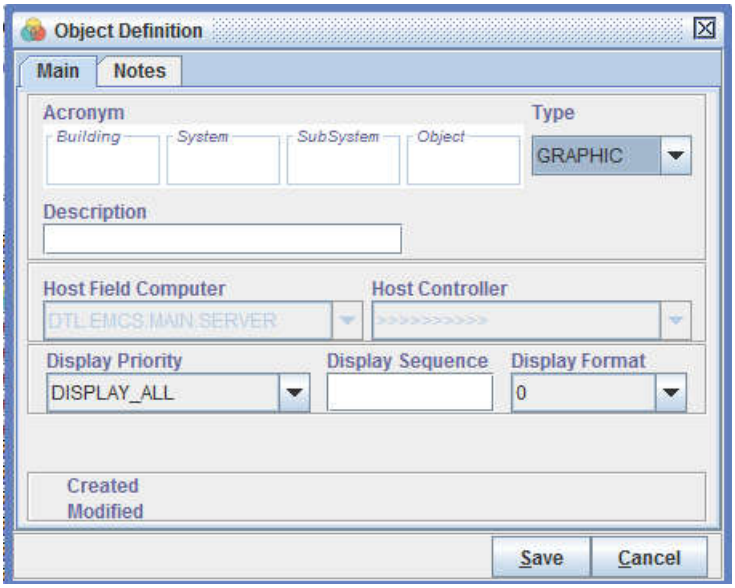


Figure 12 - Graphic Object Definition Window

3.5.1.1 Host Field Computer

The host field computer is used to define the acronym of the field computer that manages the hardware where the object resides or if the object resides on the field computer itself. Graphics objects are only managed on the computer where the server software resides.

3.5.1.2 Host Controller

The host controller is used to define the acronym of the controller that manages the object. Graphics objects are only managed on the computer where the server resides. Therefore, the host controller field will be blank.

3.5.2 View/Modify Field Information (Not Applicable)

Graphics objects do not have any field information.

3.6 Hardware

A hardware object represents an instance of physical hardware, such as

- Field Computer
- Controller
- Server

3.6.1 View/Modify Object Definition

An object is managed using the object HW type corresponding definition window, shown in Figure 13, Figure 14, Figure 15 and **Error! Reference source not found.**.

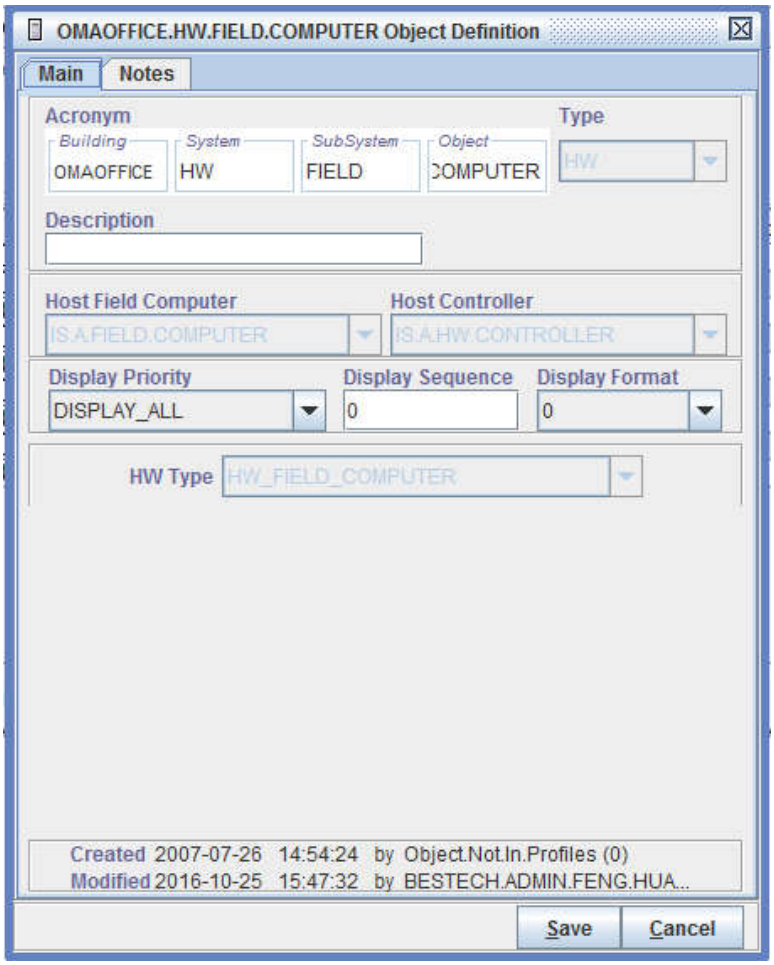


Figure 13 – FC Hardware Object Definition Window

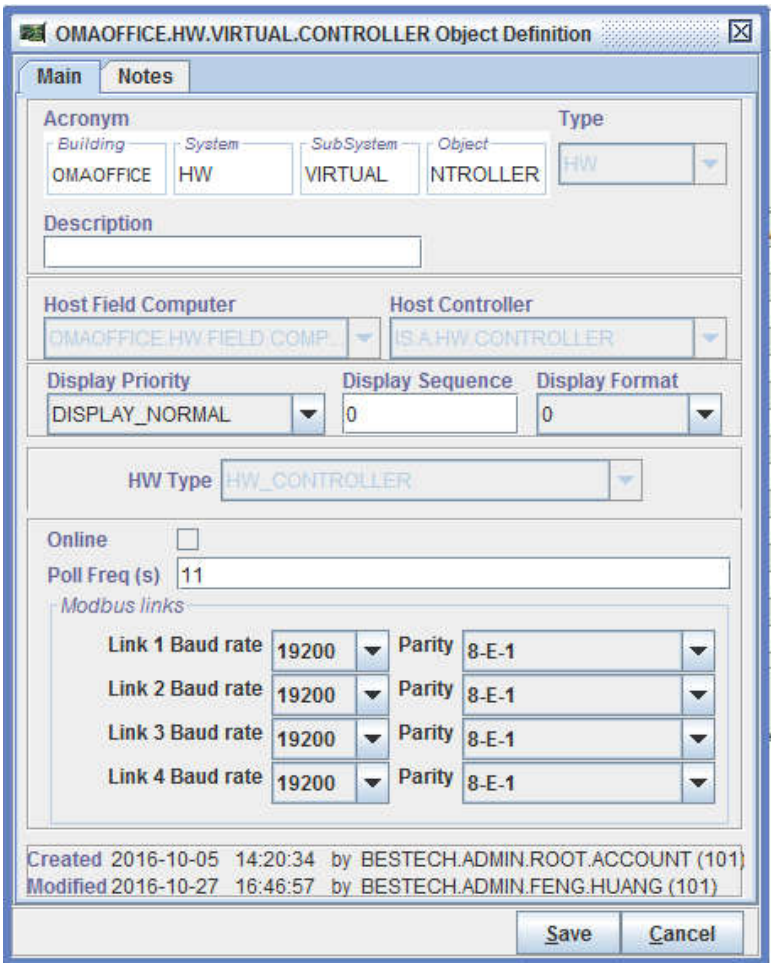


Figure 14 – Controller Hardware Object Definition Window

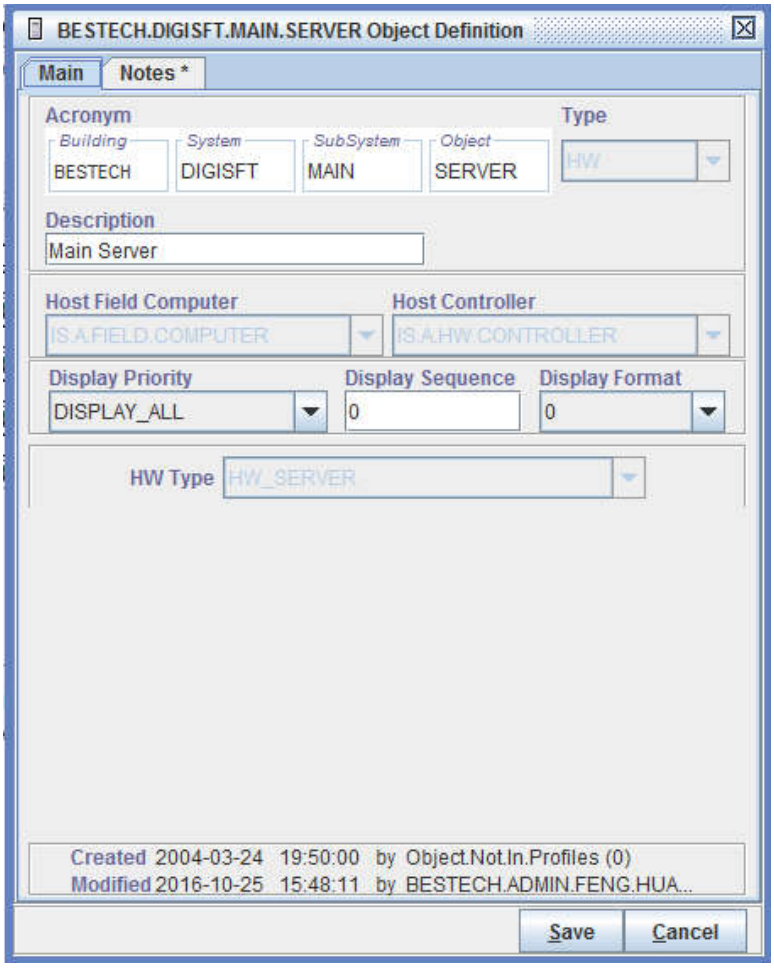


Figure 15 – Server Hardware Object Definition Window

3.6.1.1 Host Field Computer

The host field computer is used to define the acronym of the field computer. If the hardware is a field computer, the field is populated with "IS.A.FIELD.COMPUTER".

3.6.1.2 Host Controller

The host controller is used to define the acronym of the controller that manages the object. For hardware objects other than the server or field computer, this field is automatically populated with "IS.A.HW.CONTROLLER".

3.6.1.3 Hardware Type

This field is used to describe the type of hardware the object represents. For example,

- Field Computer, this represents the minicomputer installed in each building
- Controller, this represents the process running inside FC. For centralized version, each FC only has one virtual controller.
- Server, this represents the centralized server, maintained at Bes-Tech.

3.6.1.4 Controller: Online

A controller's Online/Offline is not supported in this version. You can check object status in Grid display for communication status of FC and Controller.

3.6.1.5 Controller: Poll Frequency

This is the minimum number of seconds between starting of 2 consecutive polling. For example when polling frequency is 10 s, and controller takes 4 seconds to read all residing points, then it will wait 6 second after and begin next batch. Any read to these points within 10s will return the cached values. Change this number requires "Reset" the controller through batch commands.

3.6.1.6 Controller: Modbus links

The "Modbus links" dropdowns allow users to view and edit the RS-485 link settings of a controller. In this version, each link represents a USB-Serial converter plugged in with FC.

3.6.2 View/Modify Field Information

The Controller has field information. Other hardware types, such as the field computer and server hardware objects do not have any field information.

The field information window shown in Figure 16 displays field information of a Controller, you can check FC's date and time, software version, and put note on it. Its status should be initialized in batch process menu.

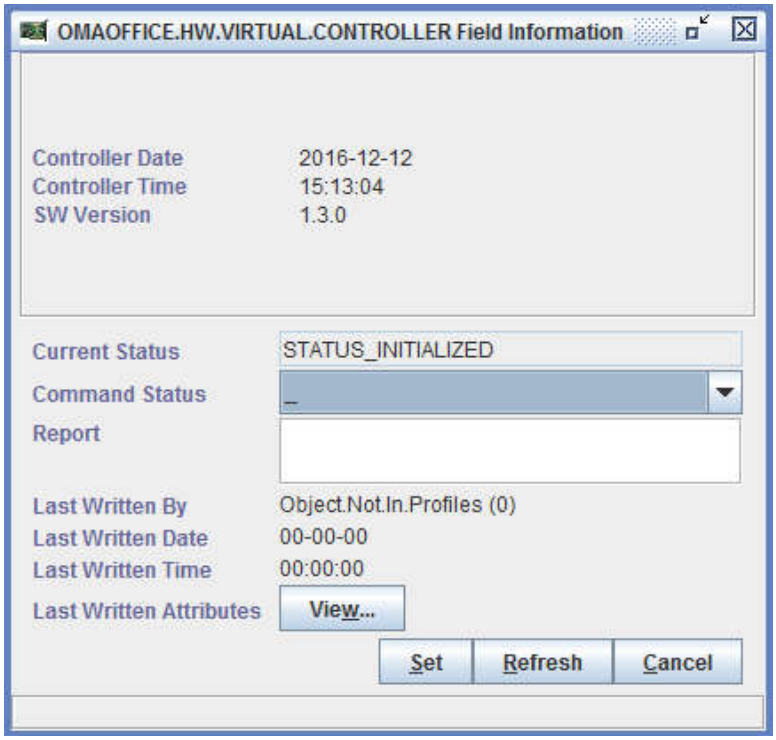


Figure 16 - Hardware Field Information Window

3.6.2.1 Controller Date

EMCS supports time and date expressions and necessitates that controllers keep track of the time and date. The controller date displays the date as known by the controller, and it's used by resident Schedule object.

3.6.2.2 Controller Time

The controller time displays the time as known by the controller.

3.6.2.3 Software Version

The software version refers to the embedded firmware installed in the controller.

3.6.2.4 Current Status

The field displays the current controller status.

3.6.2.5 Command Status

Status should be initialized in batch process menu, other command status will take no effect.

3.6.2.6 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more information about the report object variable method.

3.6.2.7 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.6.2.8 Set

Set will send any changes that you have made in the field information window to the hardware.

3.6.2.9 Refresh

The field information window presents static information to allow you to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.6.2.10 Last Written By, Date, Time

Users who made changes to data in the field information window are displayed along with the date and time those changes were set.

3.7 Inputs/Outputs

Input and output objects represent an instance of physical hardware, such as sensors or actuators. They are divided into four different object types:

- Analog input
- Analog output
- Digital input
- Digital output

3.7.1 Shared attributes

All I/O has these attribute:

3.7.1.1 Host Field Computer

I/O should always reside in a FC.

3.7.1.2 Host Controller

I/O will be managed by the Controller in a FC.

3.7.1.3 Trend Enable

The trend enable defines whether the input and output (if applicable) value of the object is saved to the trending database.

3.7.1.4 Trend Frequency

The trend frequency defines how often the I/O value of the object is saved to the trending database. The minimum number is 10 for normal user and 10 for system admin.

3.7.1.5 Trend Purge Interval

The trend purge interval defines the maximum amount of time the “output” value of the object is saved in the trending database before it is deleted. The trend purge interval is set in increments of one day and can range in number from 30 to 65,000.

3.7.1.6 Modbus

The Modbus tab shown in allows users to view and edit the Modbus settings of an AI object.

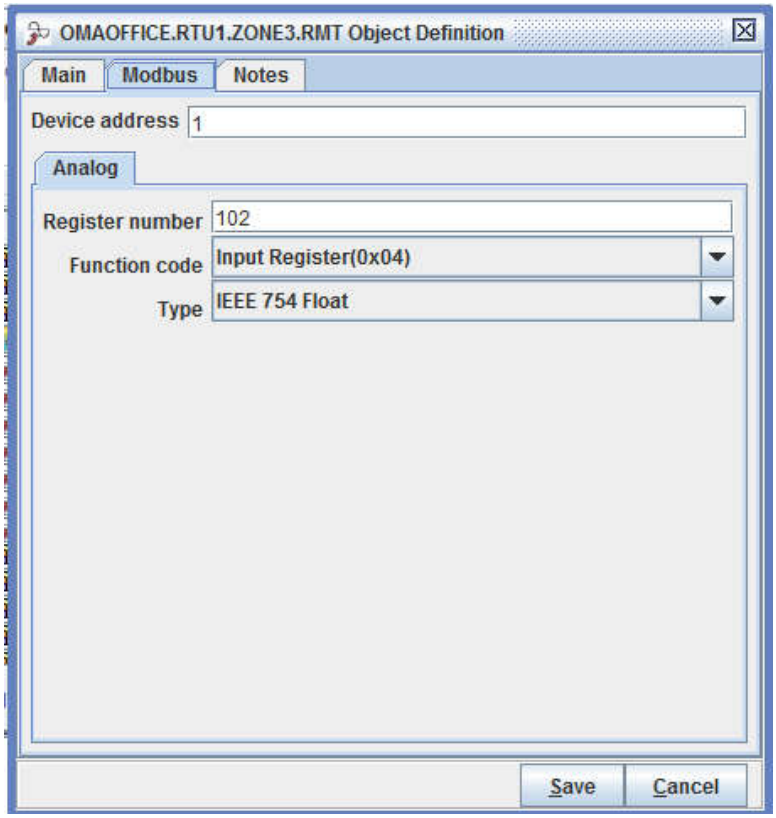


Figure 17 - Analog Input Object Definition Modbus Window

3.7.1.6.1 Device Address

The “Device Address” is used to define the slave address of the device, in decimal.

3.7.1.6.2 Analog

3.7.1.6.2.1 Register Number

The “Register Number” is the 2 byte register number, in Hex.

3.7.1.6.2.2 Function Code

The “Function Code” is the 1 byte command for Modbus message.

3.7.1.6.2.3 Type

The “Type” is used to define the type of the input value.

3.7.2 Analog Input

An analog input is an object that provides the control system an interface to the hardware, such as an analog sensor for measuring the temperature or humidity.

3.7.2.1 View/Modify Object Definition

The object definition window, as shown in Figure 18, allows users to view and edit the settings of an object.

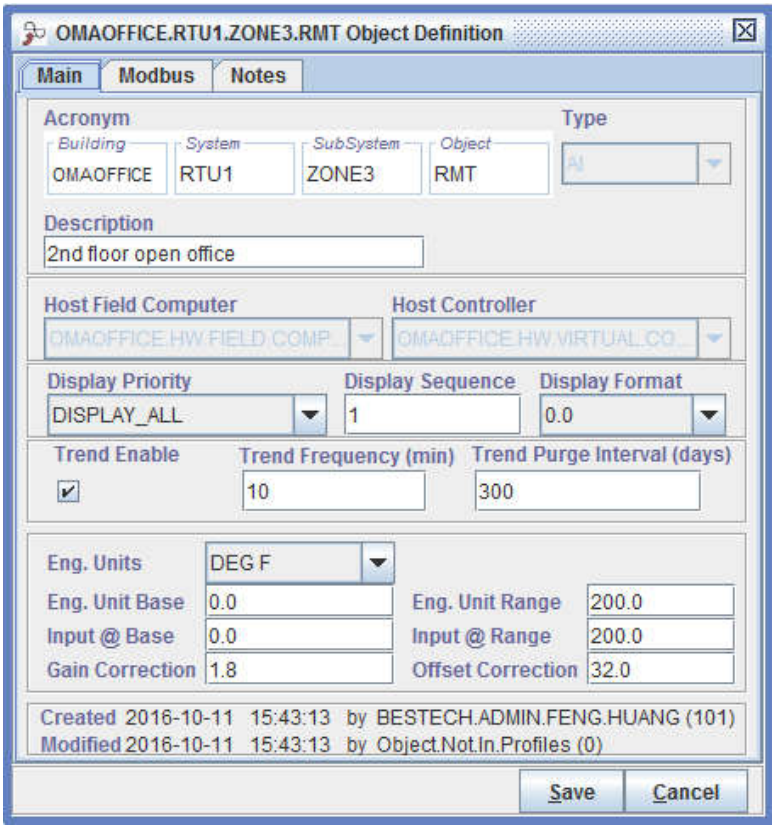


Figure 18 - Analog Input Object Definition Window

3.7.2.1.1 Engineering Units

The “engineering units” is used to define the engineering units of the sensed value, such as pressure, temperature, or humidity.

3.7.2.1.2 Engineering Unit Base

The “engineering unit base” is used for a linear transfer function to define the lower range limit value of the signal in engineering units.

3.7.2.1.3 Engineering Unit Range

The “engineering unit range” is used for a linear transfer function to define the upper range limit value of the signal in engineering units.

3.7.2.1.4 Input at Base

The “input at base” is used for a linear transfer function to define the lower range limit value of the raw input signal.

3.7.2.1.5 Input at Range

The “input at range” is used for a linear transfer function to define the upper range limit value of the raw input signal.

3.7.2.1.6 Gain Correction

The “gain correction” is a value that is multiplied by the transfer function to apply a slope correction factor.

3.7.2.1.7 Offset Correction

The “offset correction” is a value that is added to the minimum range limit of the transfer function to compensate for signal bias.

3.7.2.2 View/Modify Field Information

The field information window shown in Figure 19 displays the field values and allows you to set or change some of those values.

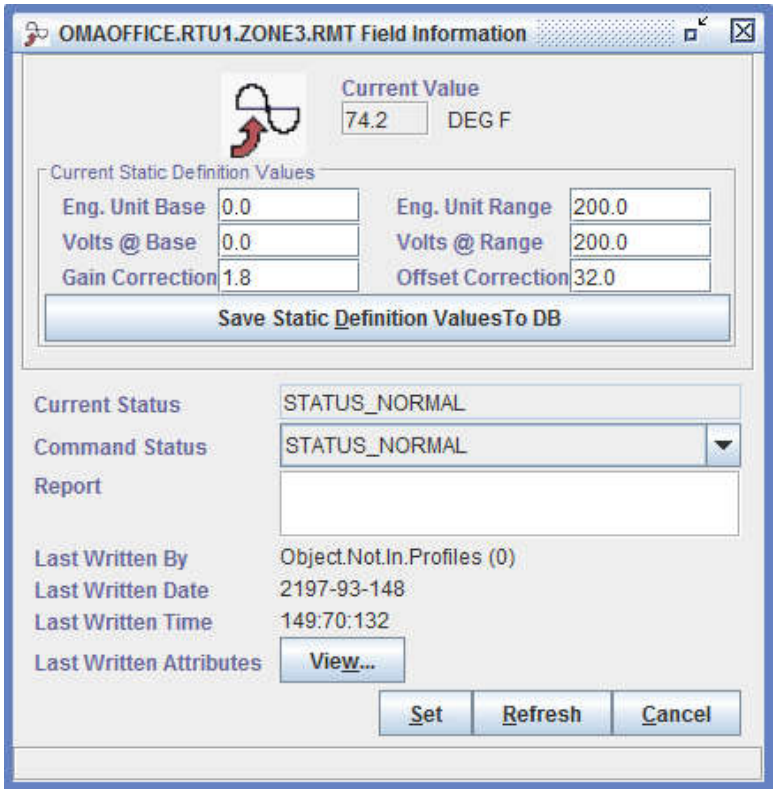


Figure 19 - Analog Input Field Information Window

3.7.2.2.1 Current Value

The current value displays the measured value from the analog input.

3.7.2.2.2 Current Static Definition Values

These are the same as those in Object Definition.

3.7.2.2.3 Save Static Definition Values to Database

This will save static information, so user doesn't need to open Object Definition to modify them, if needed.

3.7.2.2.4 Current Status

The field displays the current object status.

3.7.2.2.5 Command Status

Displays the current object command status and allows the user to change this status.

3.7.2.2.6 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more about the report object variable method.

3.7.2.2.7 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.7.2.2.8 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.7.2.2.9 Refresh

Refresh will update the field information window to display the current information.

3.7.2.2.10 Last Written By, Date, Time

Displays users who made changes to the data in the field information along with the date and time those changes were set.

3.7.3 Analog Output

An analog output is an object that provides the control system an interface to hardware, such as an analog actuator that modulates a valve or controls the speed of a variable speed motor.

3.7.3.1 View/Modify Object Definition

The object definition window, as shown in Figure 20, allows users to view and edit the settings of an object.

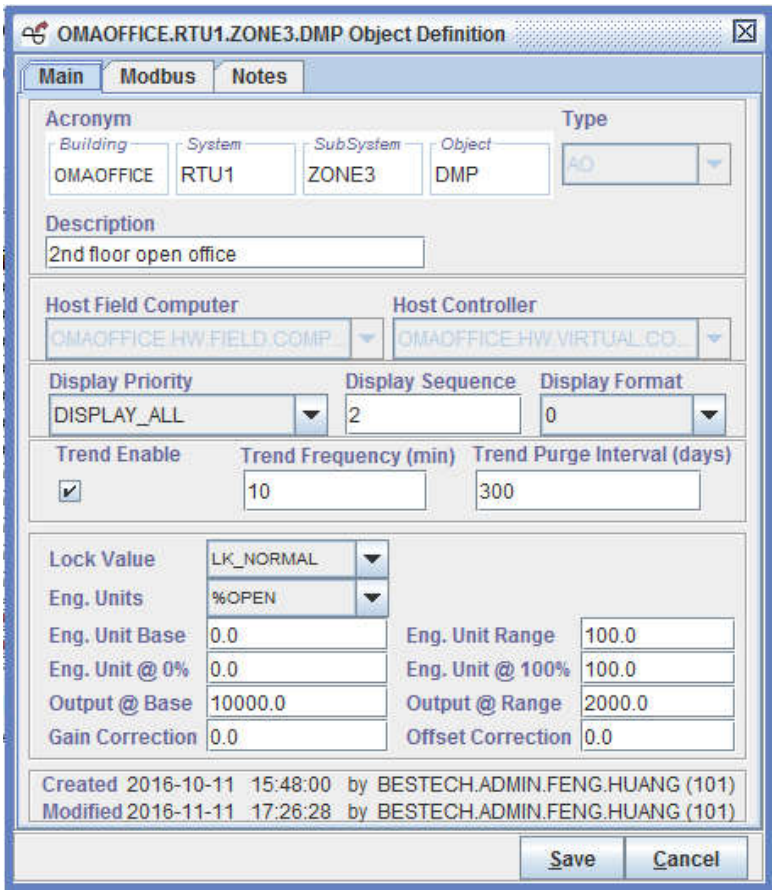


Figure 20 - Analog Output Object Definition Window

3.7.3.1.1 Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. Other object can control this object when key value is greater than or equal to the lock value here.

3.7.3.1.2 Engineering Units

The “engineering units” is used to define the engineering units of the command, such as degrees (angle).

3.7.3.1.3 Engineering Unit Base

The “engineering unit base” is used for a linear transfer function to define the lower range limit value of the command signal in engineering units.

3.7.3.1.4 Engineering Unit Range

The “engineering unit range” is used for a linear transfer function to define the upper range limit value of the command signal in engineering units.

3.7.3.1.5 Engineering Unit at 0%

The “engineering unit at 0%” is used for a linear transfer function to define the value of the command signal in engineering units at its zero percent value.

3.7.3.1.6 Engineering Unit at 100%

The “engineering unit at 100%” is used for a linear transfer function to define the value of the command signal in engineering units at its one-hundred percent value.

3.7.3.1.7 Output at Base

The “output at base” is used for a linear transfer function to define the lower range limit value of the command signal sent to the hardware.

3.7.3.1.8 Output at Range

The “output at range” is used for a linear transfer function to define the upper range limit value of the command signal sent to the hardware.

3.7.3.1.9 Gain Correction

The “gain correction” is a value that is multiplied by the transfer function to apply a slope correction factor.

3.7.3.1.10 Offset Correction

The “offset correction” is a value that is added to the minimum range limit of the transfer function to compensate for signal bias.

3.7.3.2 View/Modify Field Information

The field information window, as shown in Figure 21, displays the field values and allows users to set or change some of those values.

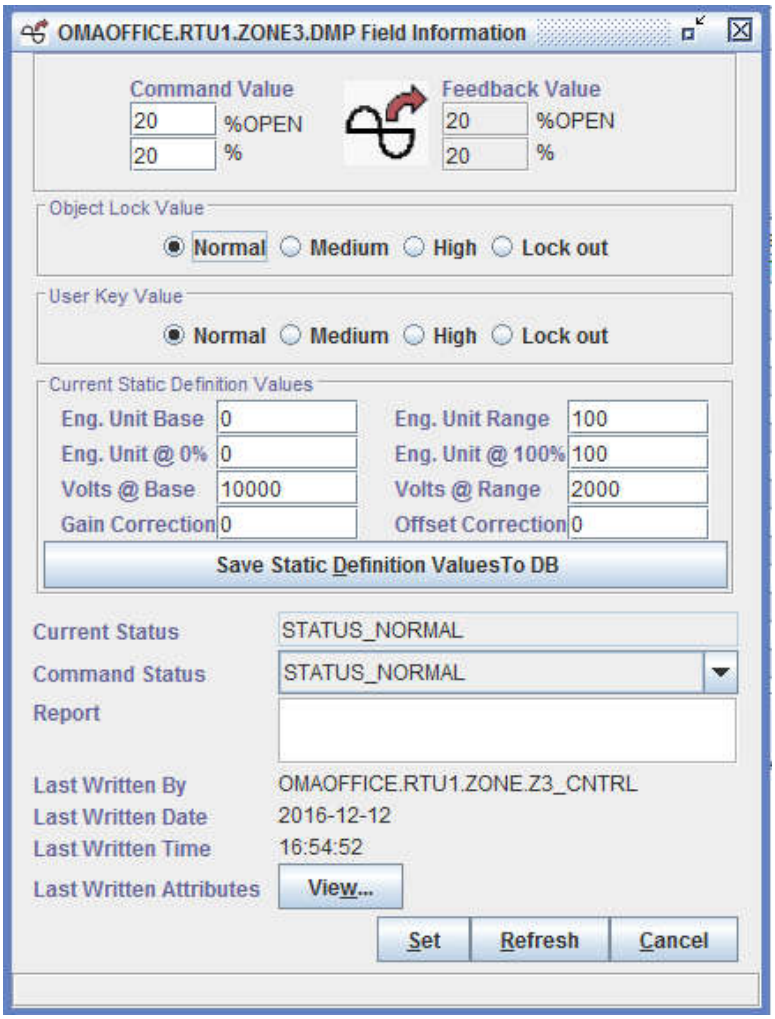


Figure 21 - Analog Output Field Information Window

3.7.3.2.1 Command Value (Engineering Unit, %)

The current value of the command sent to the analog output device is displayed in both engineering units and in percent.

3.7.3.2.2 Feedback Value (Engineering Unit, %)

The current value of the feedback from an analog output device is displayed in both engineering units and in percent.

3.7.3.2.3 Object Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object and allows the user to change the lock value.

3.7.3.2.4 User Key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object that can control other objects, such as control blocks and users.

An object can control the controllable object when key value is greater than or equal to the lock value. A user can use any level of key when using Digi-SFT to control points.

3.7.3.2.5 Current Status

The field displays the current object status.

3.7.3.2.6 Command Status

Displays the current object command status and allows the user to change this status.

3.7.3.2.7 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more about the report object variable method.

3.7.3.2.8 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.7.3.2.9 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.7.3.2.10 Refresh

Refresh will update the field information window to display the current information.

3.7.3.2.11 Last Written By, Date, Time

Displays users who made changes to the data in the field information window along with the date and time those changes were set.

3.7.4 Digital Input

A Digital input is an object that provides a discrete signal from a switch or limit device.

3.7.4.1 View/Modify Object Definition

The object definition window, shown in Figure 22, allows users to view and edit the settings of an object.

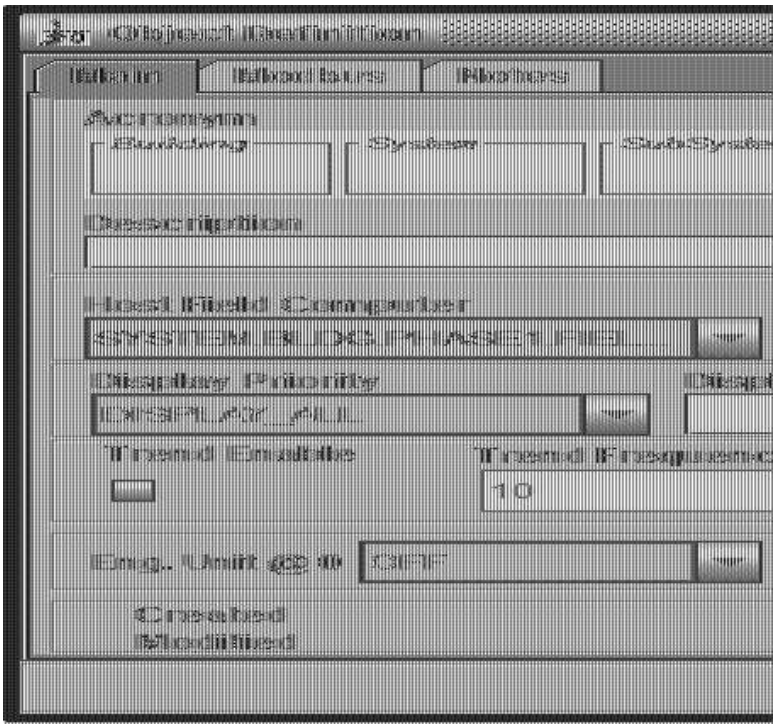


Figure 22 - Digital Input Object Definition Window

3.7.4.1.1 Engineering Unit at 0

The “engineering unit at 0” is used to define the value of the Digital input signal in engineering units at its value zero.

3.7.4.1.2 Engineering Unit at 1

The “engineering unit at 1” is used to define the value of the Digital input signal in engineering units at its value of one.

3.7.4.2 View/Modify Field Information

The field information window, as shown in Figure 23, displays the field values and allows users to set or change some of those values.

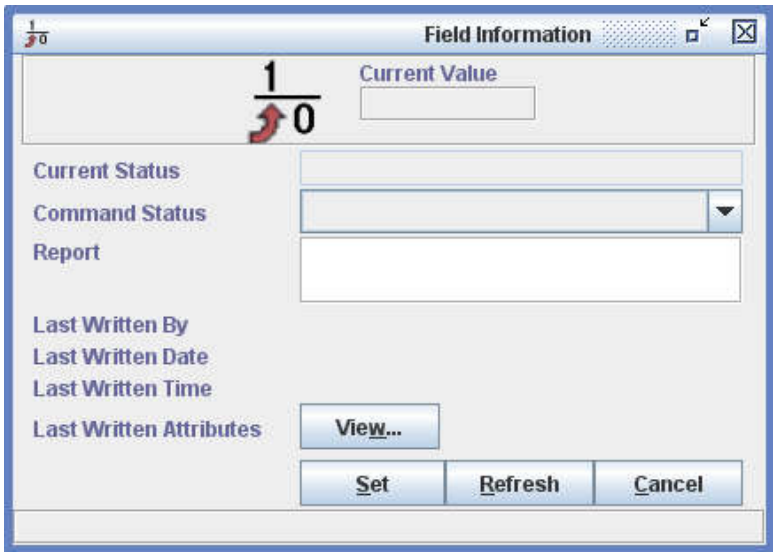


Figure 23 - Digital Input Field Information Window

3.7.4.2.1 Current Value

The current value displays the measured value from the Digital input.

3.7.4.2.2 Current Status

The field displays the current object status.

3.7.4.2.3 Command Status

Displays the current object command status and allows the user to change this status.

3.7.4.2.4 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more about the report object variable method.

3.7.4.2.5 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.7.4.2.6 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.7.4.2.7 Refresh

The field information window presents static information to allow you to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.7.4.2.8 Last Written By, Date, Time

Displays users who made changes to data in the field information window along with the date and time those changes were set.

3.7.5 Digital Output

A Digital output is an object that provides a discrete signal sent to an on/off device.

3.7.5.1 View/Modify Object Definition

The object definition window, shown in Figure 24, allows users to view and edit the settings of an object.

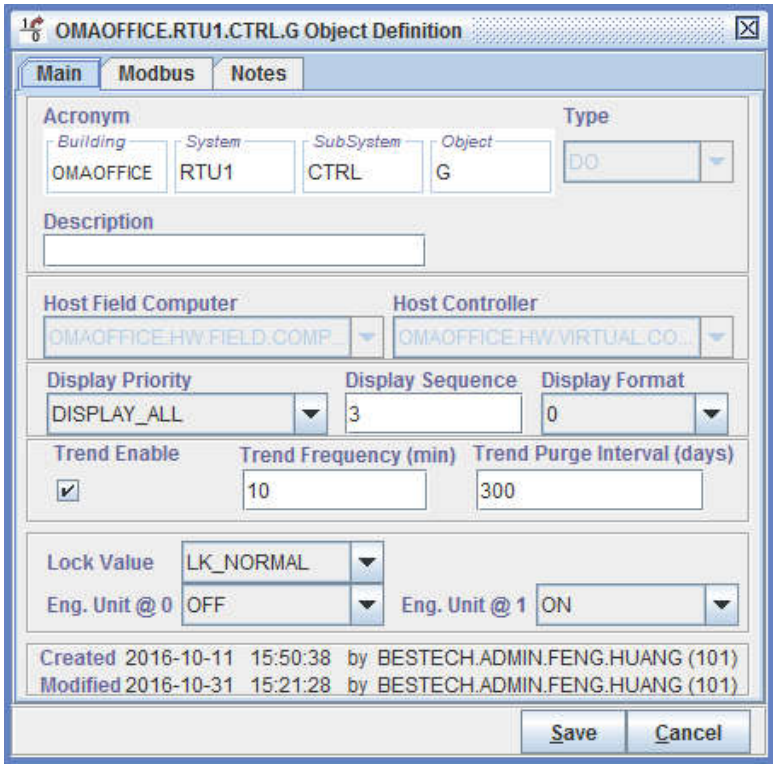


Figure 24 - Digital Output Object Definition Window

3.7.5.1.1 Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object and allows the user to change the lock value.

3.7.5.1.2 Engineering Unit at 0

The “engineering unit at 0” is used to define the value of the Digital output signal in engineering units at its value zero.

3.7.5.1.3 Engineering Unit at 1

The “engineering unit at 1” is used to define the value of the Digital output signal in engineering units at its value of one.

3.7.5.2 View/Modify Field Information

The field information window, as shown in Figure 25, displays the field values and allows users to set or change some of those values.

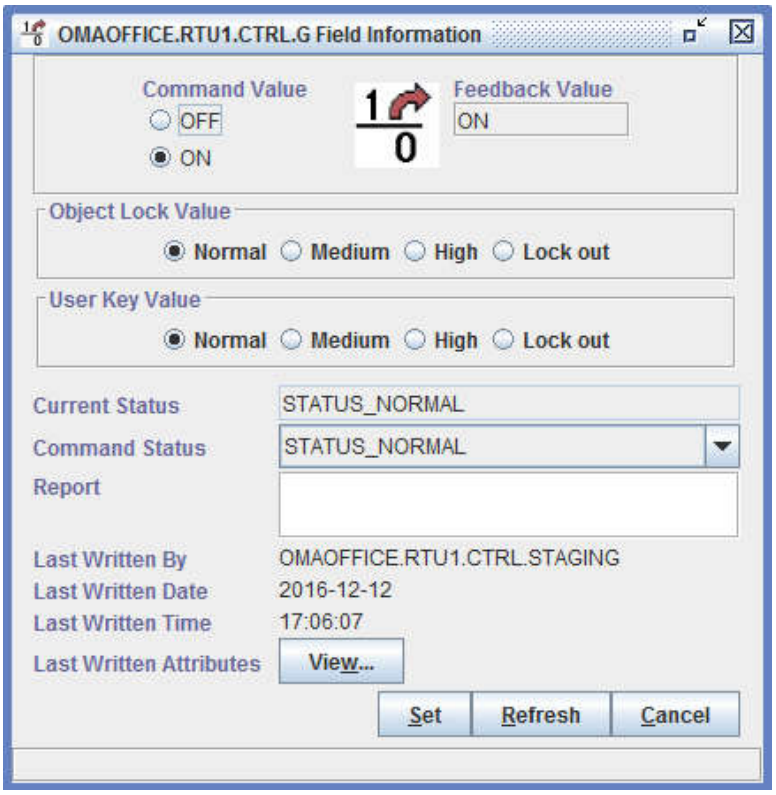


Figure 25 - Digital Output Field Information Window

3.7.5.2.1 Command Value

The current value of the command sent to the Digital output device is displayed.

3.7.5.2.2 Feedback Value

The current value of the feedback from a Digital output device is displayed.

3.7.5.2.3 Object Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object and allows the user to change the lock value.

3.7.5.2.4 User Key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object that can control other objects, such as control blocks and users. An object can control the controllable object when key value is greater than or equal to the lock value. The current key value for the user is displayed and allows the user to change the key value.

3.7.5.2.5 Current Status

The field displays the current object status.

3.7.5.2.6 Command Status

Displays the current object command status and allows the user to change this status.

3.7.5.2.7 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more about the report object variable method.

3.7.5.2.8 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.7.5.2.9 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.7.5.2.10 Refresh

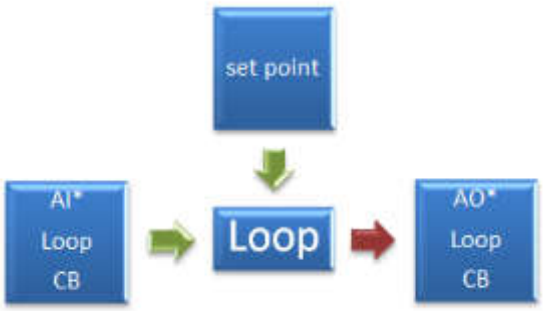
The field information window presents static information to allow users to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.7.5.2.11 Last Written By, Date, Time

Display users who made changes to the data in the field information window along with the date and time those changes were set.

3.8 Loops

A control loop is an object that performs a proportional, integral, derivative (PID) control on a single input.



* Most often used as inputs and outputs to the loop object.

Figure 26 - Loop Object

3.8.1 View/Modify Object Definition

The object definition window, as shown in Figure 27, allows users to view and edit the settings of an object.

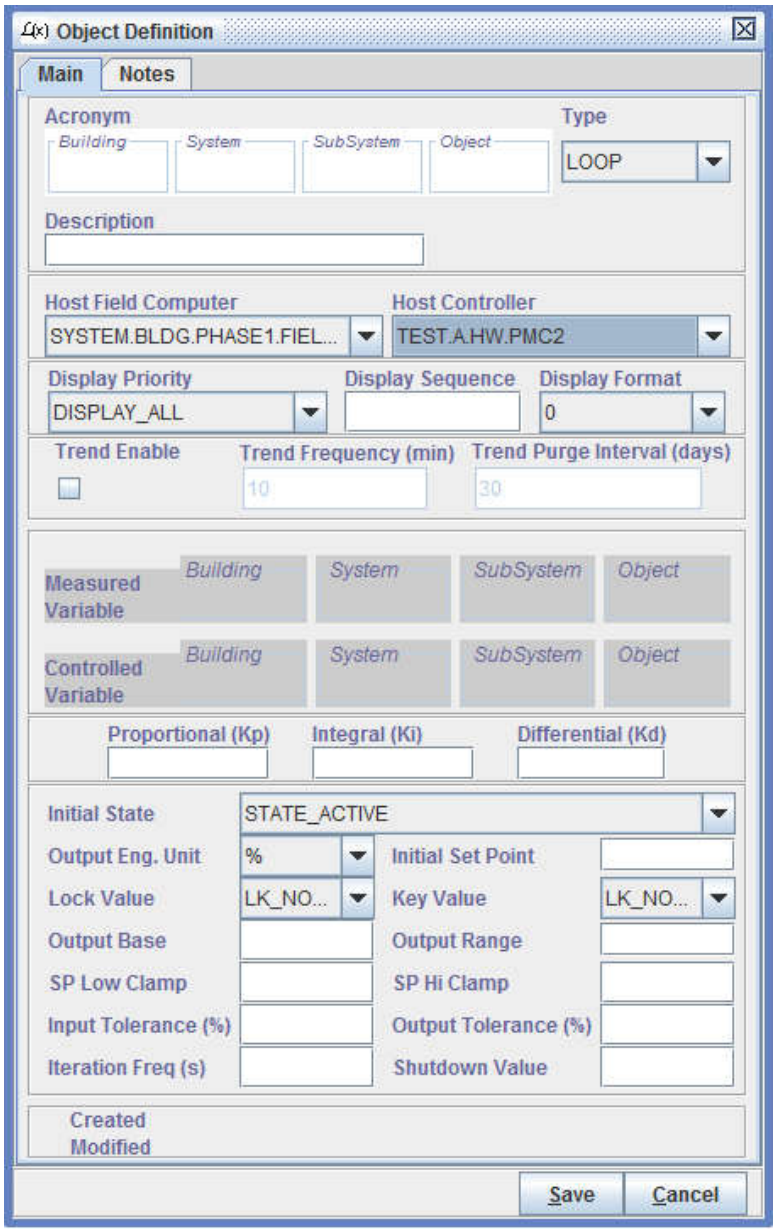


Figure 27 - Loop Object Definition Window

3.8.1.1 Measured Variable Acronym

The input to the PID loop is the measured variable. The four part alpha-numeric acronym assigned to the analog input object is used to define the source of the measured variable.

3.8.1.2 Controlled Variable Acronym

The output of the PID loop is the controlled variable. The four part alpha-numeric acronym assigned to the analog output object is used to define the source of the controlled variable.

3.8.1.3 Proportional Gain

The proportional gain, Kp, of the PID control is calibrated in the object definition window.

3.8.1.4 Integral Gain

The integral gain, Ki, of the PID control is calibrated in the object definition window.

3.8.1.5 Differential Gain

The differential gain, Kd, of the PID control is calibrated in the object definition window.

3.8.1.6 Initial State

When a loop is first executed, it will be initialized to the initial state. These are any of the six states: resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more information about control block states.

3.8.1.7 Output Engineering Unit

The "output engineering unit" is used to define the engineering unit of the standard predefined output variable of the object. The grid display shows the value of the standard predefined output variable of the object and its units.

3.8.1.8 Initial Set Point

The set point is the target value the PID controller aims to reach. When a loop is first executed, the PID control will use the initial set point.

3.8.1.9 Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops.

3.8.1.10 Key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object that can control other objects, such as control blocks and users. An object can control the controllable object when key value is greater than or equal to the lock value.

3.8.1.11 Output Base

The loop object has a base and range (lower limit and upper limit). The output of the loop is scaled to be between the base and the range. The output is not truncated or clipped between these values. If the (base, range) is set to (0, 1000) and the output of the loop is 600. If the loop output is connected to a control block the value used by the control block is 600. However, it the loop output is connected to an analog output (AO), the AO would be moved to 60%.

3.8.1.12 Output Range

The loop object has a base and range (lower limit and upper limit). The output of the loop is scaled to be between the base and range. The output is not truncated or clipped between these values.

3.8.1.13 SP Low Clamp (Set Point Lower Range Limit)

A lower range limit can be defined to prevent the set point from being assigned a value less than the set point lower range limit.

3.8.1.14 SP Hi Clamp (Set Point Upper Range Limit)

An upper range limit can be defined to prevent the set point from being assigned a value greater than the set point upper range limit.

3.8.1.15 Input Tolerance

The input tolerance is a value in percent that functions as a deadband on the input to the PID loop. The input value will not be used if it is within the input tolerance of the last accepted value. Once outside the input tolerance, the new input value is accepted and used in the PID control calculation. This function helps prevent small changes in the input signal (such as changes due to signal noise) from cascading into the PID loop calculation.

3.8.1.16 Output Tolerance

The output tolerance is a value in percent that functions as a deadband on the output command of the PID loop. The output value will remain constant if the command is within the output tolerance of the last accepted value. Once outside the output tolerance, the command is accepted and used. This function is provided to prevent small oscillations and reduce wear on the actuators.

3.8.1.17 Iteration Frequency

This field displays the frequency of the loop executions.

3.8.1.18 Shutdown Value

When the loop object is set to the shutdown state, the command or output value of the loop is forced to the shutdown value.

3.8.1.19 Notes

The “notes” tab is provided as an option. A text-based note of up to 1024 characters can be included with the object.

3.8.2 View/Modify Field Information

The field information window shown in Figure 28 displays the field values and allows users to set or change some of those values.

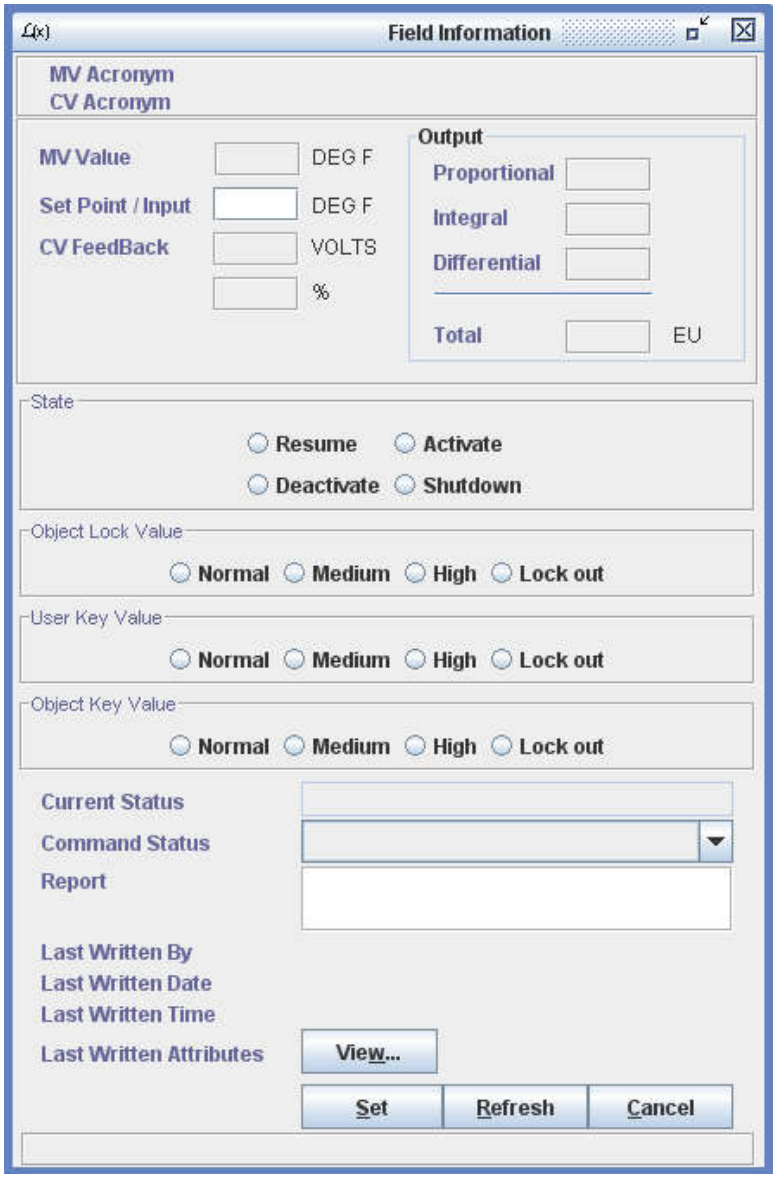


Figure 28 - Loop Field Information Window

3.8.2.1 Measured Value

The input to the PID loop is the measured variable. The four part alpha-numeric acronym assigned to the analog input object is used to define the source of the measured variable.

3.8.2.2 Set Point/Input

The set point is the target value the PID controller will aim to reach. The current value is displayed for the loop object and allows the user to change the set point.

3.8.2.3 Controlled Value Feedback (Volts, %)

The output of the PID loop is the controlled variable. The four part alpha-numeric acronym assigned to the analog output object is used to define the source of the controlled variable.

3.8.2.4 Proportional Output

The value of the calculated proportional term is displayed.

3.8.2.5 Integral Output

The value of the calculated integral term is displayed.

3.8.2.6 Differential Output

The value of the calculated differential term is displayed.

3.8.2.7 Total Output

The value of the calculated PID loop is displayed in percent.

3.8.2.8 State

Displays the current state and allows the user to change the state to any of the six states: resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more information about control block states.

3.8.2.9 Object Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object and allows the user to change the lock value.

3.8.2.10 User Key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object to control other objects, such as control blocks and users. An object can control the controllable object when the key value is greater than or equal to the lock value. The current key value for the user is displayed and allows the user to change the key value.

3.8.2.11 Object key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object that can control other objects, such as control blocks and users. An object can control the controllable object when key value is greater than or equal to the lock value. The current key value for the object is displayed and allows the user to change the key value.

3.8.2.12 Current Status

The field displays the current object status.

3.8.2.13 Command Status

Displays the current object command status and allows the user to change this status.

3.8.2.14 Report

This will display a message about the object when a report message has been written to the object.

See section 4.2.9.1 for more about the report object variable method.

3.8.2.15 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set are displayed by selecting to view the last written attributes.

3.8.2.16 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.8.2.17 Refresh

The field information window presents static information to enable users to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.8.2.18 Last Written By, Date, Time

Displays users who made changes to data in the field information window along with the date and time those changes were set.

3.9 History

History is an object that offers a data acquisition mechanism that requires programming a control block and is used to save user defined variables at a programmable data acquisition rate. This requires more effort to record the data, but it is more flexible then automatic trending. The trend interval is the same as the task rate of the control block, set by using delay(). Logic can be used to trigger trending or stop trending. Users may even choose to use two different control blocks to change the trend interval.

3.9.1 View/Modify Object Definition

The object definition window, as shown in Figure 29, provides you the ability to view and edit the settings of an object. To view this window, open the grid display, select database, DB, as the grid option along with other options you prefer, and search.

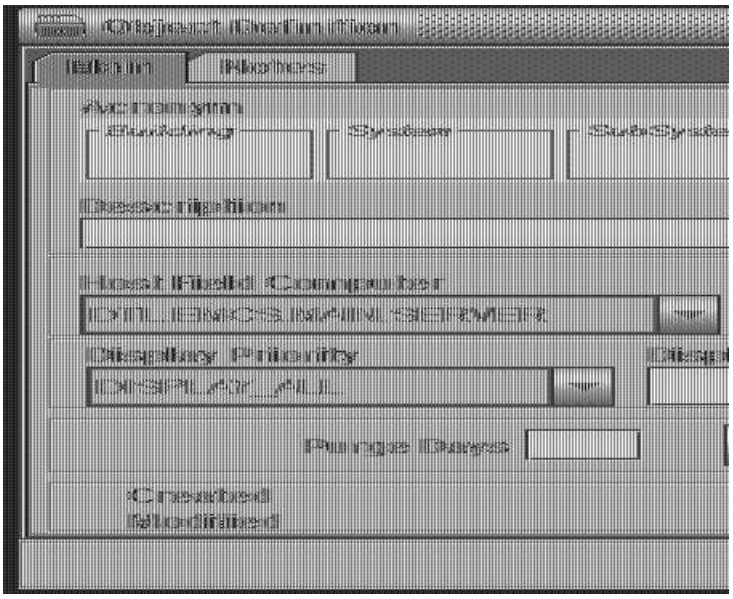


Figure 29 - History Object Definition Window

3.9.1.1 Host Field Computer

History objects are only managed on the computer where the server software resides.

3.9.1.2 Host Controller

History objects are only managed on the computer where the server resides. Therefore, the host controller field will be blank.

3.9.1.3 Purge Days

The purge interval defines the maximum amount of time the trend data is saved in the trending database before it is deleted. The trend purge interval is set in increments of one day and can range from 30 to 65,000.

3.9.1.4 View History Data

The history data can be uploaded from the system database and viewed in a table by selecting to view the history data.

3.9.1.4.1 Acronym

The acronym is the four part alpha-numeric acronym assigned to the object. It includes a building acronym, system acronym, subsystem acronym, and an object acronym.

3.9.1.4.2 Start Date and Time

A subset of the data stored in the system database can be uploaded from the server. The beginning of the data set is defined by the start date and time.

3.9.1.4.3 End Date and time

A subset of the data stored in the system database can be uploaded from the server. The end of the data set is defined by the end date and time.

3.9.1.4.4 Get Data

This will load and display the data in the current window.

3.9.1.4.5 Save to Disk

This will save the data to a comma-separated value, text file. The file can be opened with a text editor or spreadsheet program.

3.9.2 View/Modify Field Information (Not Applicable)

History objects do not have any field information.

3.10 Schedules

A schedule is an object that allows you to define what time events occur and what days, such as weekdays, weekends, and holidays.

3.10.1 Alarm Schedule (Not supported)

3.10.2 Generic Schedule

A generic schedule is an object that allows users to define what time events occur and what days, such as weekdays, weekends, and holidays.

3.10.2.1 View/Modify Object Definition

The object definition window, shown in Figure 30, enables users to view and edit the settings of an object.

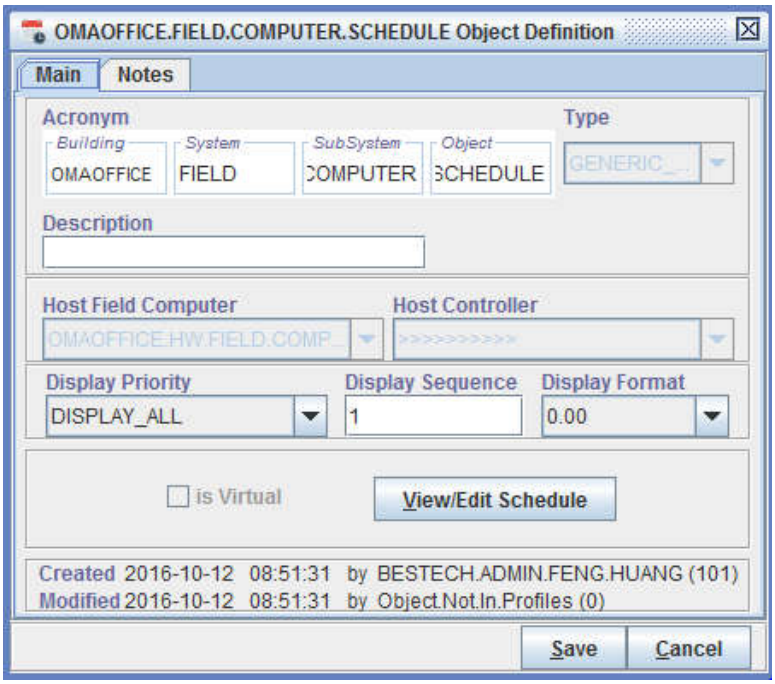


Figure 30 - Generic Schedule Object Definition Window

3.10.2.1.1 Host Field Computer

Schedule usually resides in Field Computer, representing a field schedule on site, but can also in Server.

3.10.2.1.2 Host Controller

This field should be blank, since Schedule are in FC/Server.

3.10.2.1.3 Is Virtual

The field displays if the generic schedule is virtual or not. Only a virtual schedule can be added as a master schedule to a non-virtual one (see Section 4.12.1.3.1)

3.10.2.1.4 View/Edit Schedule

Selecting “view/edit schedule” will open the schedule in the Schedule Editor. For more information, see section 4.12.

3.10.2.2 View/Modify Field Information

The field information window, as shown in Figure 31, displays the field values and allows users to set or change some of those values.

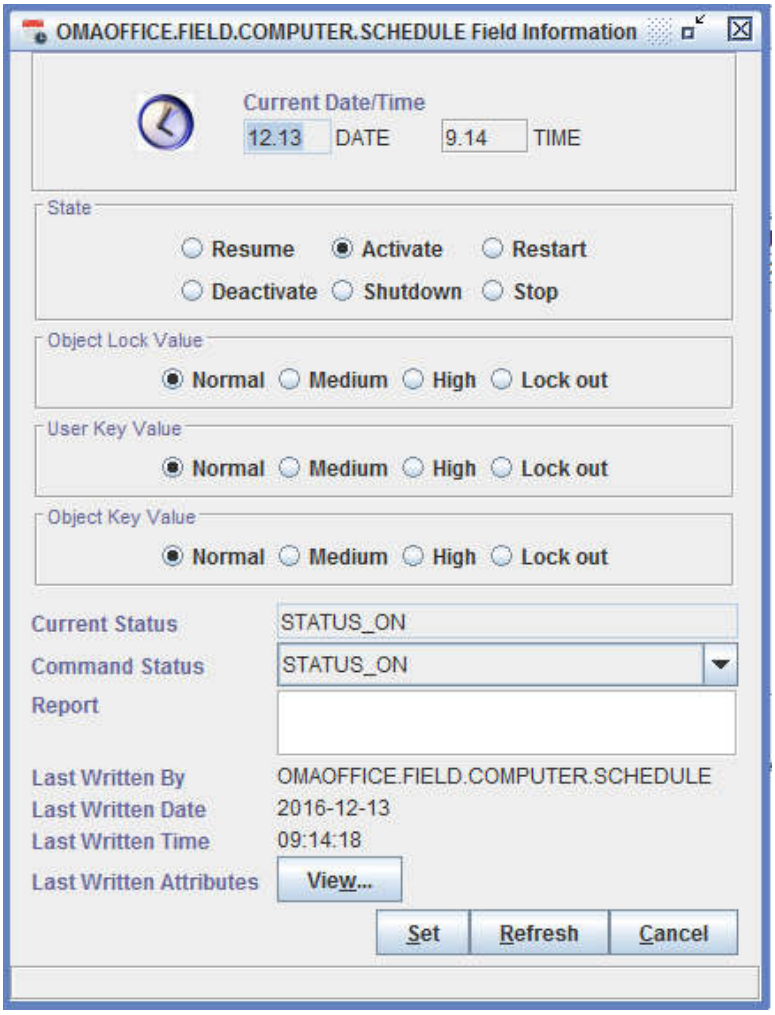


Figure 31 - Generic Schedule Field Information Window

3.10.2.2.1 Current Date/Time

The schedule field information window displays the current date and time as known by its Hosting Field Computer.

3.10.2.2.2 State

Displays the current state and allows the user to change the state to any of the following six states: resume, activate, restart, deactivate, shutdown, and stop. Refer to section 4.1.1 for more about control block states.

3.10.2.2.3 Object Lock Value

The lock value is one of four values, normal, medium, high, and lock-out, that can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops. The current lock value is displayed for the object and allows the user to change the lock value.

3.10.2.2.4 User Key Value

The key value is one of four values, normal, medium, high, and lock-out, that can be assigned to an object that can control other objects, such as control blocks and users. An object can control the controllable object when key value is greater than or equal to the lock value. The current key value for the user is displayed and allows the user to change the key value.

3.10.2.2.5 Object key Value

Schedule inherits this information from CB, but it should not use it because it does not control other points.

3.10.2.2.6 Current Status

The field displays the current object status.

3.10.2.2.7 Command Status

Displays the current object command status and allows the user to change this status.

3.10.2.2.8 Report

This will display a message about the object when a report message has been written to the object. See section 4.2.9.1 for more about the report object variable method.

3.10.2.2.9 View (Last Written Attributes)

The fields displayed in the field information window represent the attributes of the object. A list of the attributes or fields that have last been set is displayed by selecting to view the last written attributes.

3.10.2.2.10 Set

Selecting set will apply the changes to the field attributes and refresh the values in the field information window.

3.10.2.2.11 Refresh

The field information window presents static information to enable users to read the values even though the EMCS is constantly changing. Refresh will update the field information window to display the current information.

3.10.2.2.12 Last Written By, Date, Time

Displays users who made changes to data in the field information window along with the date and time those changes were set.

3.11 Users

A user object defines an individual's account settings.

3.11.1 View/Modify Object Definition

The object definition window, as shown in Figure 32, allows users to view and edit the settings of an object.

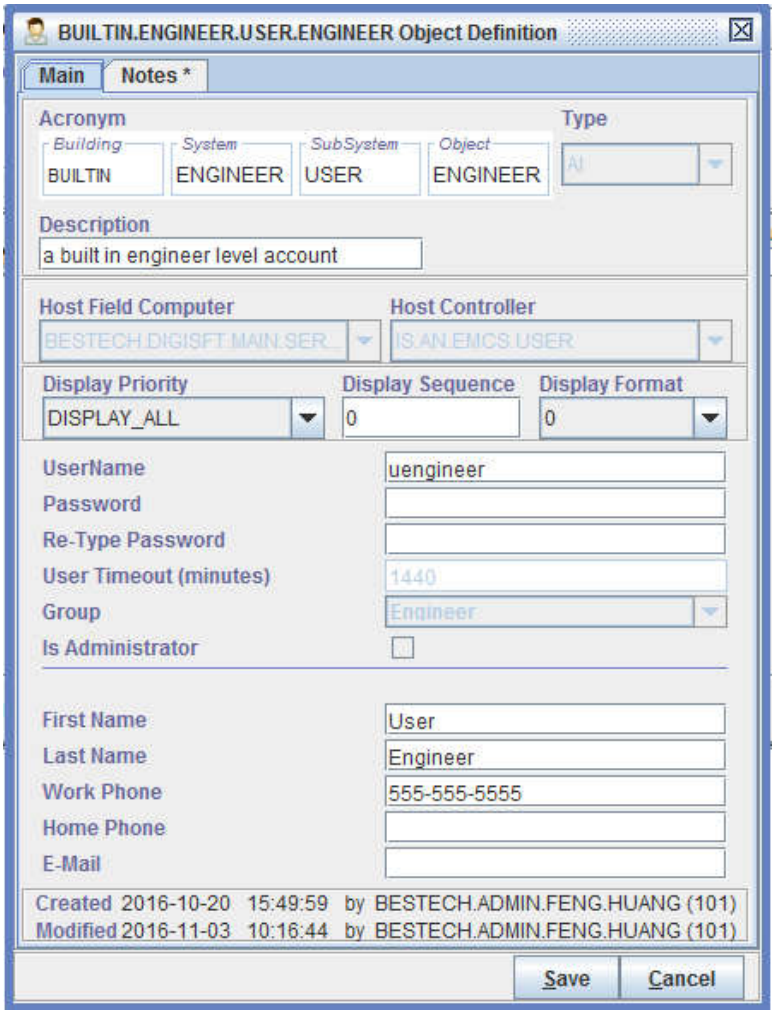


Figure 32 - User Object Definition Window

3.11.1.1 Host Field Computer

User objects are only managed at Server.

3.11.1.2 Host Controller

The host controller field should be blank.

3.11.1.3 User Name

The user name is an alpha-numeric string that is assigned to an individual and used to login into Digi-SFT. It must contain at least 5 characters.

3.11.1.4 Password

The user assigns a password, an alpha-numeric string that the user enters to login into Digi-SFT.

3.11.1.5 Re-type Password

The user re-types the password when it is assigned to confirm that it was entered correctly.

3.11.1.6 User Time-Out

The user is automatically logged-out when no compunction since last time for user time-out period.

3.11.1.7 Group

Users are assigned to a group to define their access control. See section 10.1 for more information on groups and access control.

3.11.1.8 Is Administrator

A user classified as an administrator has more privileges than a typical user. See section 10.2 for more information on administrators and access control.

3.11.1.9 First Name

The first name of the individual is required for that individual's account settings.

3.11.1.10 Last Name

The last name of the individual is required for that individual's account settings.

3.11.1.11 Work Phone

The work phone number of the individual may be entered for that individual's account settings.

3.11.1.12 Home Phone

The home phone number of the individual may be entered for that individual's account settings.

3.11.1.13 Email

The email address of the individual may be entered for that individual's account settings.

3.11.2 View/Modify Field Information (Not Applicable)

User objects do not have any field information.

3.12 More about Lock and Key

The lock and key concept is like a permissions or access control for modifications of controllable objects. For example, if a user wanted to set the value of a Digital output to test the actuator they could set the value in the object definition window. A control block that normally commands a Digital output would soon overwrite the value. Therefore, the user can change the lock on the Digital output to medium and set their key value to medium, while the control block remains at normal. Then, the control block does not have the level of key privileges to update the Digital output, but the user does. When the user sets the output of the Digital output, it does not get overwritten by the control block.

By convention, the lockout is primarily meant to prevent any other object from controlling that object. However, other objects can be set to have a lockout key value to control it. By convention, objects are not giving lockout key privilege. The lockout is meant for situations, for example, if a pump is failed and we want to prevent it from being commanded.

Controllable objects can be assigned lock values (normal, med, high, lockout), such as control blocks, loops, and Digital outputs. Objects that act or control other objects can be assigned key values (low, med, high, lockout), such as control blocks and users.

4 Distributed Control Language Programming

4.1 Introduction

This document domain specific language called the "Distributed Control Language" (DCL). DCL was designed to make control programs that monitor and control building systems. It is used in Bes-Tech energy management control system (EMCS) software, and front end application, Digi-SFT.

Since the DCL syntax is familiar to C++ languages, persons familiar with these "object oriented" languages will immediately feel comfortable with DCL.

Individual DCL source code files are edited via the Digi-SFT user interface. Once editing is complete, the source code file is sent to the central EMCS server where it is compiled in an intermediate form called s-code (short for "stack" code).

Note that there is a one-to-one correspondence between DCL source code files and compiled s-code files. In other words, DCL does not support the compilation of multiple source code files into one "executable" code. Both DCL source code and s-code are saved in the EMCS database for future use.

Each s-code file instance is hereafter referred to as a control block. Instances of control blocks can be downloaded to EMCS field computers where they are executed (run) by an s-code interpreter. This interpreter is referred to as a Control Block Manager (CBMgr). CBMgr's can execute control blocks simultaneously using a round robin scheduling approach.

Control blocks can interact with each other across hardware platforms as well as with various I/O points (analog / Digital inputs and outputs). Conceptually, the system intelligence provided by control blocks is distributed across the system, hence the "distributed" moniker in DCL. Figure 33 illustrates some of the mechanisms and interactions just discussed.

The lower part of Figure 33 shows running control blocks interacting with each other as well as the following types of "real world" interface objects:

- Analog Inputs (AI's)
- Analog Outputs (AO's)
- Digital Inputs (DI's)
- Digital Outputs (DO's)

DCL also allows the manipulation of simple control algorithms referred to as LOOPS. It is important to note that static object parameters are not defined or changed using DCL. Rather, these are set using the Digi-SFT object definition. Examples of such static parameters include object acronyms, sensor base and range values, etc.

This document is organized as follows: First, the actual specification of the language are presented, starting with variables and working up to expressions, statements, and functions. Then, the actual structure of a DCL program (control block) is discussed and several examples are presented. Finally, the Appendix offers a reference for keywords and built-in functions.

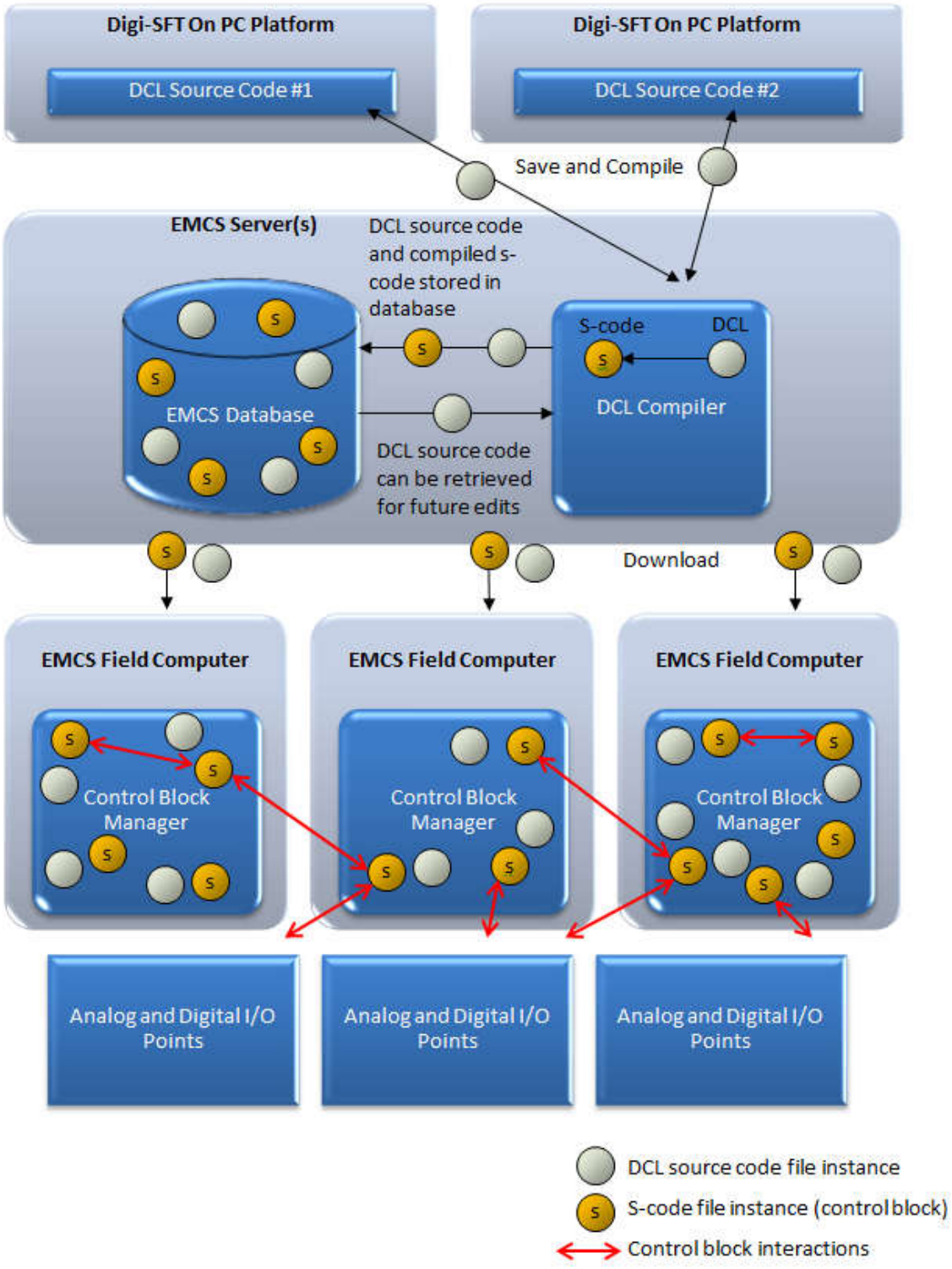


Figure 33 - DCL and S-Code / Control Block Interaction Diagram

4.1.1 Control Block States

Once a DCL file is compiled and the resulting control block is downloaded to a specific hardware platform (i.e., field computer), it is in one of the following six states:

Active Control block execution is started using the ***activate*** method. The ***main*** method executes once this method is complete if the ***activate*** method is in front of it.

Resumed Control block execution is started using the ***resume*** method. The ***main*** method executes once this method is complete if the ***resume*** method exists.

Deactivate The control block executes the ***deactivate*** method and then execution is terminated. The ***deactivate*** method works in combination with the ***activate*** method.

Shutdown The control block executes its ***shutdown*** method and execution is terminated.

Stopped Control block execution is terminated immediately. This state is entered in one of four ways:

- DCL code was written without any "infinite" or continuous loops. The code finished executing normally.
- Control block execution hit an unconditional `stop` statement (section 4.6.7).
- A user sent a stop command to the control block for debugging purposes or some other reason.
- An error occurred during control block execution. Its *status* (section 4.1.2) will be set so as to give an indication of what kind of error caused the execution to stop (divide by zero, array out of bounds, etc.).

When the ***stopped*** state is entered, the control block stops execution immediately. The control blocks program counter remains on any stop instruction that was encountered or on the next statement that will be executed if the control block is ***restarted***.

Restarted The control block is running. The ***restarted*** state can only be initiated by a restart command generated via the EMCS user interface. The purpose of this command is to "restart" the control block from the ***stopped*** state so program execution can continue where it left off without having to enter through ***activate*** or ***resume***.

The intent of the first four states is to provide two potential user-defined points of entry and exit in the execution of each control block. This is illustrated in .

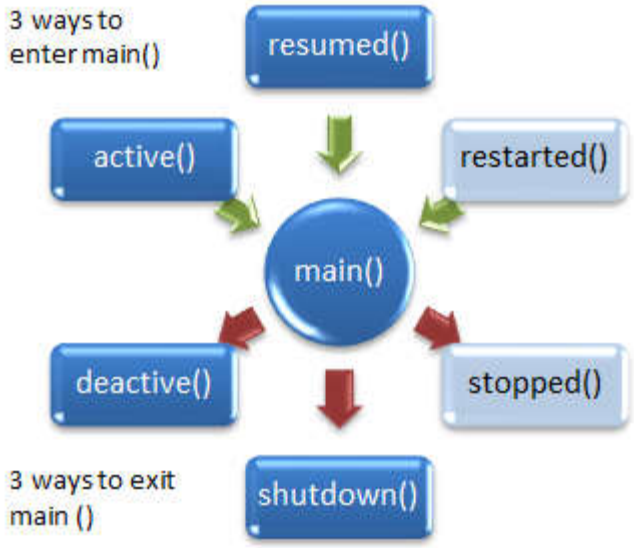


Figure 34- Control Block Execution: Normal Entry and Exit States

These first four states can be initiated by users or other control blocks using object "methods" as described in section 4.2.9.6. Users can define specific functions that will execute when entering ***active***, ***resumed***, ***deactivate*** and ***shutdow*n** states.

Upon initial download or hardware device power up, the control blocks reside in either the ***active***, ***deactivate***, shutdown, or resume states depending on how the control block object was defined in the system. If a control block is deleted from the system, the CBMgr will remove the control block from those it is presently running. The state can be defined in several ways: One way to define a state is by selecting it under the "Powerup State" option in the Object Definition window as shown in Figure 35. For other methods, see section 4.2.9.6.

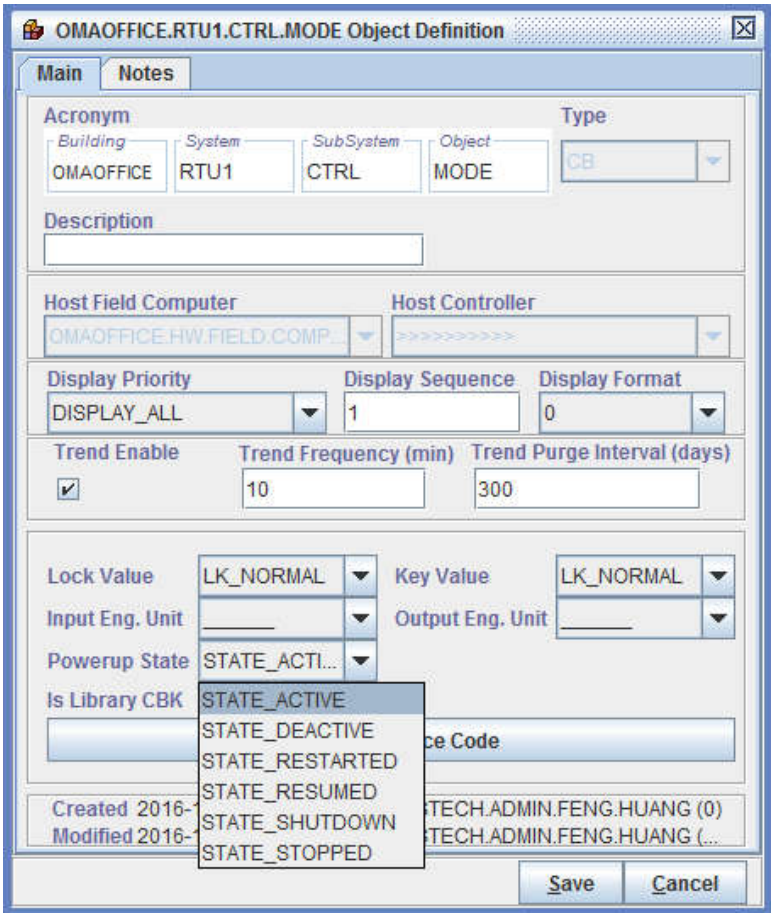


Figure 35- Using the Object Definition Window to Select the Control Block State

4.1.2 Control Block Status

Aside from a control block's state as just discussed in the previous section, a control block has another characteristic referred to as its *status*. Status values are enumerated constants stored in the system database. These values convey additional information about a control block including error conditions. Status values can be set in one of the following ways:

- Upon initial download to the Control Block Manager (CBMgr), the control block status will be set to the value defined during the control block definition in the Field Information window as shown in Figure 36.
- A user can set the status of a control block via the Digi-SFT.
- Another control block can set the status of a different control block.
- A control block can set its own status via a function call.
- The CBMgr can set the status of a control block on some error condition (i.e., divide by zero, array out of bounds, etc.).

The status of a control block can be in the ***Normal, On, Off, Night, Day and others*** conditions predefined in the database. After the status is defined in the Field Definition window, the status of the control block will display in the Grid Display of the Digi-SFT. In Figure 36 the status of the control block is defined as NORMAL and displayed as NORMAL.

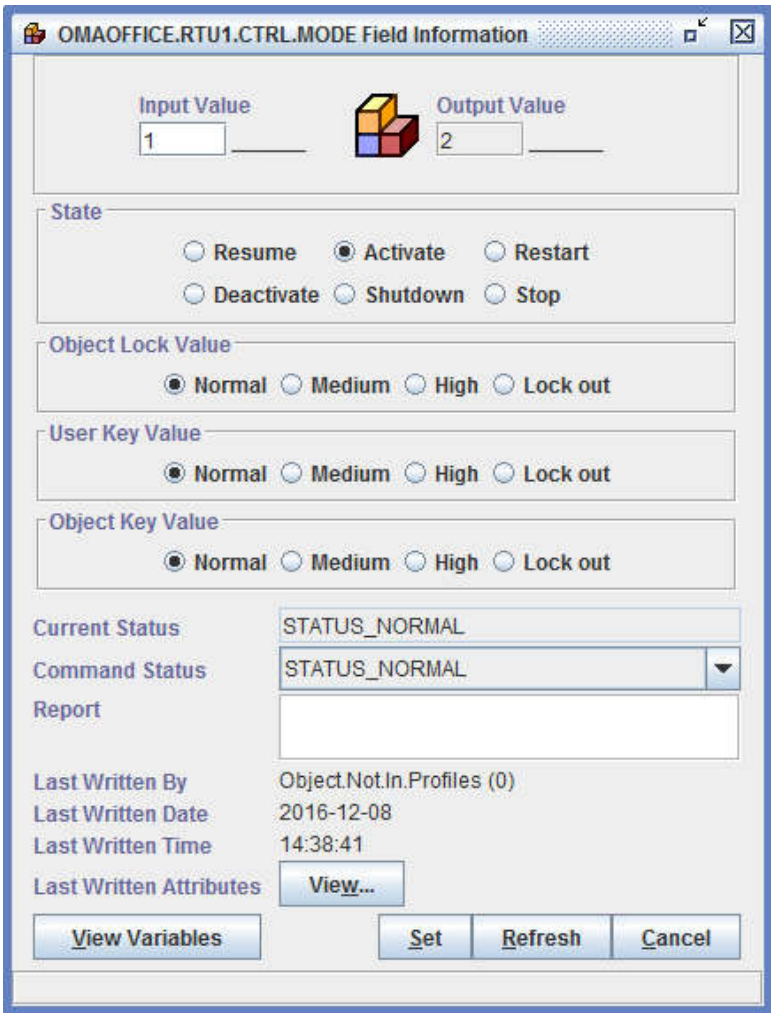


Figure 36- Control Block Status in the Field Information Window

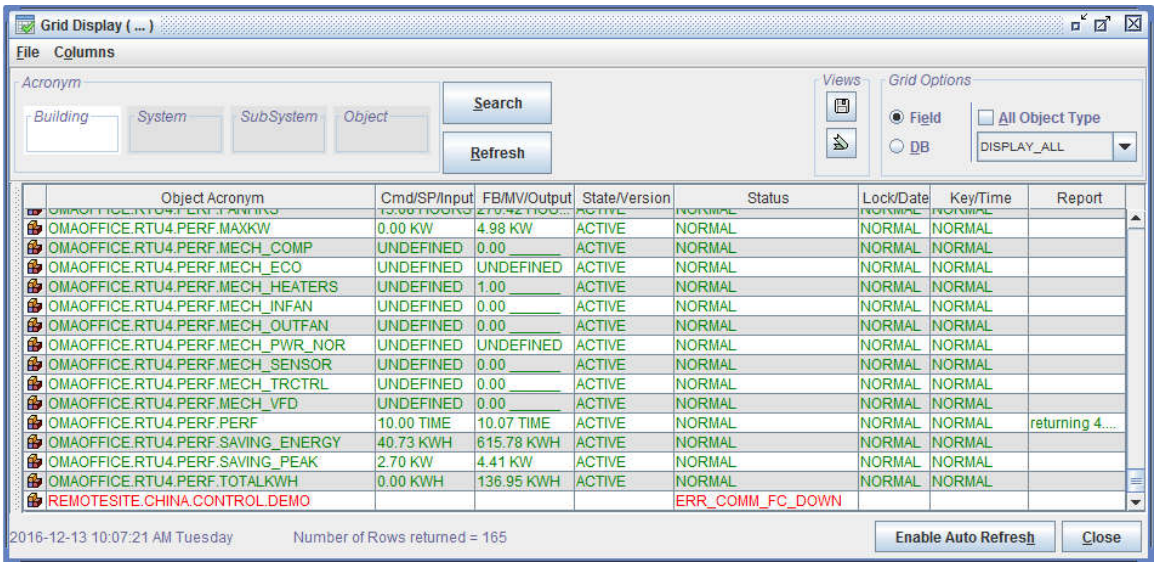


Figure 37- Control Block Status in Grid Display Window

4.1.3 Font Conventions

In order to clearly distinguish DCL code from narrative, all code examples in the book are in courier font.

4.1.4 Source Code Comments

DCL source code commenting is presented at this point in the document because many of the code examples that follow are commented. Comments are short, concise narratives provided by the DCL programmer in order to document information such as the reason that a specific action was taken, the purpose of certain lines of code, the user who wrote a specific control block, etc. The compiler ignores all comments and therefore comments exist only in the DCL source files.

From a code maintenance standpoint, it is good programming practice to comment on the source code. Commenting standards are discussed in more detail in section 13.1 of the Appendix. Two styles of comments are supported. Both styles begin with two "reserved" characters.

1. C++ Style

```
// This is the "two-slash" or C++ style comment.
// It is valid to the end of the current line.
If you tried to enter some more comments here without
another "two-slash" you would get a compiler error.
// This style is better for single-line comments
```

2. C Style

```
/* This is the "slash-star" or C style comment.
It is valid all the way from the first "slash-star"
through any number of carriage returns to the next "starslash".
This style of comment is good for multiple-line
comments.
*/
```

Note that comments may begin at any location on a line. In other words, the comment "start "delimiters (`//` or `/*`) do not have to be placed at the beginning of a line.

4.2 Variables

Variables serve as "placeholders" for information. A variable's value is referred to by its name. DCL supports numeric and certain object type variables. Alphanumeric string variables are not supported. Variables must be declared before they are used.

Variables can only be used in a manner that is consistent with the variable type with which they were declared. For example, if you declare a variable as an integer, you cannot later assign a floating-point value to it.

All variables are declared and used within the DCL "file" in which they were created.

DCL distinguishes between *primitive* and *object* variables. Basically, primitive variables are used to store numbers, dates or times, whereas object variables are used to represent EMCS objects such as I/O points, other control blocks, users, etc. User defined variable types such as structures and classes are not supported in DCL. A variable usage summary table is provided in section 14.2 of the Appendix.

4.2.1 Variable Names

A variable name can contain letters, numbers, or underscores. However, all variable names must start with a letter or underscore. Valid examples include:

```
x;
AHU1_MixedAirTemp;
__My1_2Custom3_4Variable5;
```

There is essentially no limit to the maximum length of a variable name. However, external variable names are limited to 16 characters (see section 4.2.5 for a discussion on external variables). Variable names are case sensitive. For example, AHU1 aHu1 and ahu1 are all considered to be different variables. DCL promotes standardization in variable naming schemes. A common technique used for long variable names is to use lower case for the first complete word, and then capitalize the first letter of each next complete word such as:

```
mixedAirTemp
pressureSensor
returnAirFan
```

All EMCS enumerated constants used in DCL (section 4.3.1) are in Uppercase. This serves to differentiate constants from variables. Variable names cannot be an enumerated constant, function or language reserved words listed in section 14.1 of the Appendix.

4.2.2 Variable Scope

DCL variables can be used inside of a control block, referred to as internal variables; or accessed from other control blocks with certain restrictions, and changed by other control blocks, referred to as external variables. External variables are discussed in more detail in section 4.2.7. Variables must be declared in one of two places before they can be used; otherwise, the control block will not be compiled:

- At the beginning of the DCL file (global scope).
- At the beginning of function definitions before any other executable statements (function scope).

DCL variables are considered to have “global” scope if they are declared outside of any user defined or reserved functions. This means that they can be accessed and manipulated by any function within the DCL file.

DCL variables have “local” scope if they are declared within a particular function. These variables can only be accessed from within the function they are declared. Having “local” scope variables promotes code reuse. For example, certain user defined control functions can be tested, optimized, and then copied into other control blocks without having to declare any additional “global” scope variables.

Note that no additional key words are required to define the variable scope as “global” or “local”. The compiler assigns scope based on where it finds the variable declaration.

4.2.3 Primitive Variable Types

The following primitive variable types are supported:

Primitive Type Name	Representation

Primitive Type Name	Representation
Int	signed integer
Float	signed floating point (IEEE Standard 754-185)
Bool	boolean (0/1)
Time	hr:min:sec
Date	mon\day\year

4.2.4 Object Variable Types

EMCS object manipulation with DCL takes place through the following object type variables:

AI Analog Input

AO Analog Output

DI Digital Input

DO Digital Output

CB Control Block

HIST A history object is a repository for user defined variables at a user defined data acquisition rate.

LOOP This object repeatedly measures an AI, compares its engineering value to a given set point value, then modulates an AO object using a proportional, integral, derivative algorithm.

ALARM The purpose of alarm objects is to capture and disseminate information concerning adverse or harmful situations in the system. Examples include a coil freeze condition; a triggered building fire alarm, etc. Alarm objects are "triggered" by control blocks based on certain conditional statements within the control block source code.

HW A HW object represents a particular controller or field computer. As far as DCL is concerned, this object has the fewest capabilities. It is used to determine if a HW platform is available (i.e., it can be communicated with).

Object variable types have specific actions (methods) that can be invoked (section 4.2.9). These actions either get the value of specific attributes of the object (read), or alternatively, change the value of an object attribute (write).

4.2.5 Variable Declaration

Variables must be declared before they can be used. If the initialization of a variable is in the same line of code with the declaration, than this variable will represent a constant value and the assigned value cannot be changed later in the code. For a true variable meaning, the initialization should follow the declaration on a different line of code.

Variable declaration can occur in one of two places:

- At the start of a DCL file before any function definitions (global scope). As will be discussed in section 4.2.5.2, object variables must be declared with global scope.

- Within a function definition before any statements (local scope).

4.2.5.1 Primitive Variable Declaration

Primitive type variable declaration has one of the following two syntaxes:

```
// A number of un-initialized variables
VariableType VariableName1, VariableName2;

// One initialized variable set to constant value
VariableType VARIABLE_NAME = value;

VariableType can be one of the following:

int          // 32 bit signed integer
float  // 32 bit signed floating point
bool         // Boolean 0=FALSE, 1=TRUE
time         // Time in hr:min:sec
date         // Date in mon:day:year
```

The first type of declaration assigns the value of zero by default to the declared variable.

The second declaration is a combination of declaration and assignment statement where the value on the right hand side of the equals sign is assigned to the variable. It is important to note that any variable declared in this manner is automatically considered a **constant** by the DCL compiler, meaning that the variable cannot be assigned a new value anywhere in the DCL code.

The value assigned to a variable must be consistent with the `VariableType`. For example, the following variable declaration would cause a compiler error message since the value assigned is not an integer.

```
int i, j, k; // Declaration of three integers
j = 2.5; // Will not compile! Value assigned is float
```

It is allowable to use arithmetic expressions as well as other previously declared and initialized variables in declaration and assignment statements. For example:

```
float SLOPE = 10;
float INTERCEPT = 5;
float FACTOR = SLOPE * 20 + INTERCEPT;
```

All variable names in the above example have been capitalized in order to signify that they are constants (i.e., declaration with assignment). However, capitalization is not enforced by the compiler.

A variable must be assigned a value before it can be used in subsequent declaration and assignments as shown in the following example.

```
float SLOPE;
float INTERCEPT = 5;

// Will not compile! Variable "slope" has not been
// initialized with any value
float FACTOR = SLOPE * 20 + INTERCEPT;
```

4.2.5.2 Object Variable Declaration

Object variable declarations must have the following syntax:

```
// Object Variable initialized with referenced object
ObjectType VariableName = [Building.System.Subsystem.Object];
```

`ObjectType` can be one of the following:

```
// Valid Object Variable Types
AI            // Analog Input
ALARM  // Alarm
AO            // Analog Output
CB            // Control Block
DI            // Digital Input
DO            // Digital Output
HW            // Hardware Device
HIST          // History
LOOP          // Control Loop
```

Note the special syntax and brackets of the referenced object:

```
[Building.System.Subsystem.Object]
```

This represents the complete name of the object that was defined via Digi-SFT during object creation.

All object variables must have global scope and must be assigned values during declaration.

This restriction serves two purposes:

1. It allows the compiler to check the database for the existence of the referenced object (right hand side of the equals sign) before generating any s-code. The program will not compile unless the referenced object has been previously defined and entered into the database.

   ```
   AI ahuDischAirTemp = [BURNETT.AH1.MA.TP];
   ```

   A control block with this assignment will not compile unless `[Burnett.AH1.MA.TP]` has previously been entered into the EMCS database and it has been defined as an AI object.

   Compilation will also fail if the `ObjectType` does not match the type of the referenced object.

   It is also worth noting that after a successful DCL compile, the control block is now considered *dependent* upon any referenced objects. The EMCS will not allow deletion of any referenced objects where such dependencies exist. For example, consider a control block successfully compiled with the above assignment for `ahuDischAirTemp`. The object called `[BURNETT.AH1.MA.TP]` now cannot be deleted from the EMCS until the dependent control block is either deleted or recompiled with a different assignment. All such dependencies must be removed before an object can be deleted (i.e., the object could be referred to in more than one control block).

2. Object declaration with global scope promotes code reuse. If a control block is developed that serves a generic purpose (for example, mixed air control), this control block can be reused multiple times using different equipment in different buildings merely by changing the right hand side of the declaration / assignment statement. Furthermore, since all variable declarations must be located at the beginning of the DCL file (i.e., global scope), changing these is easy to do without wading through a lot of code.

4.2.6 Arrays

It is often convenient to group variables together so that an operation can be performed on all of them. DCL includes arrays for this purpose. Arrays are represented by variable names followed by brackets that contain an index. For example, the following expression assigns the value of 23 to the array at index = 5.

```
// Example array variable assignment myArray with index 5
myArray[5] = 23;
```

Each index is a separate placeholder for information and can be an arithmetic expression, variable, function call, or any combination of those. All of the following are valid array assignments.

```
// All are valid array assignments
myArray[i] = 23;              // index = i
myArray[i - j] = thisArray[2];// index = i - j
myArray[i + function()] = 10; // index = i + value from function
```

Arrays are indexed *starting at zero*. The very first element of the above array is `myArray[0]` *not* `myArray[1]`. The DCL compiler stores the maximum index of the array in s-code so that array bounds checking can occur at run time. This will prevent access to any index less than 0 or greater than its maximum index minus one (index is set at compile time). For example, an array of size 5 has valid indices 0 through 4 (0, 1, 2, 3, and 4).

Similar to the variable declarations previously discussed, array declarations for primitive variable types may be one of two possible types: *Declaration Without Assignment* and *Declaration With Assignment*.

4.2.6.1 Array Declaration without Assignment

```
// Array declaration without assignment for int's, float's,
// bool's, times, and dates... Note that int represents any
// valid integer expression
VariableType VariableName[int];
```

In this type of declaration, no assignments are made and all array members will be set to zero (similar to non-array variables in section 4.2.3). Assignments may then subsequently be made within functions subject to variable scoping rules. DCL grammar requires that the user "declare" the array size within the brackets ("`[ ]`"). The value inside the brackets must be an integer or a mathematical expression made up of previously initialized variables. For example:

```
// An array of ten int's.  No assignment
int j[10];

// An array of ten float's.  Size based on a previously
```

```
// initialized variable
int MAX_SIZE = 10;
float f[MAX_SIZE]; // Array size = MAX_SIZE = 10;

// The following will not compile since MAX_SIZE has not
// been initialized with a value
int MAX_SIZE;
float f[MAX_SIZE];
```

4.2.6.2 Array Declaration with Assignment

In this type of declaration, values are assigned to the array at compile time as follows:

```
// Array declaration with assignment
VariableType VARIABLE_NAME[] = {val1, val2, .. valn};
```

As with non-array primitive variable declaration and assignment statements, the above array is considered a constant by the compiler. Therefore, members of the array cannot be assigned new values in DCL code.

Note that DCL grammar requires that the user omit the size of the array when using array declaration with assignment. For example, given the following declaration:

```
float A[] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

The compiler will allocate space for five (5) array variables and will initialize these values as follows:

```
A[0] = 1.1
A[1] = 2.2
A[2] = 3.3
A[3] = 4.4
A[4] = 5.5
```

Arrays of objects are also allowed and encouraged. However, same as with non-array object variables, all assignments must occur at variable declaration as shown below.

```
// Object array declaration must include assignment
ObjectType VariableName[] = {Acronym1, Acronym2, etc.};
```

For example:

```
// Object array declaration
AI temps[] = {     [BURNETT.AH1.MA.TP],
                  [BURNETT.AH2.MA.TP],
                  [BURNETT.AH1.OA.TP],
                  [BURNETT.AH2.OA.TP]};
```

This statement will create an AI object array of size 4, initialized to the AI points listed on the right. The declaration and assignment statement can be “tabbed” and spread over multiple lines to promote better readability.

4.2.7 External Variables

As previously discussed in section 4.2.2, external variables may be accessed by other control blocks, thereby providing an additional information sharing mechanism.

For example, one control block could set an external boolean variable called "`winter`" to TRUE or FALSE. Other control blocks could then periodically check the value of this external variable and base some action on whether it was set to TRUE or FALSE.

The syntax for declaring an external variable is as follows:

```
// Declares an external floating point variable called
// "cutOffTemp"
extern float cutOffTemp;
```

By making `cutOffTemp` "external," the declaring control block is allowing other control blocks to read and write to this variable. Figure 38 is an example of external variable sharing among three different control blocks.

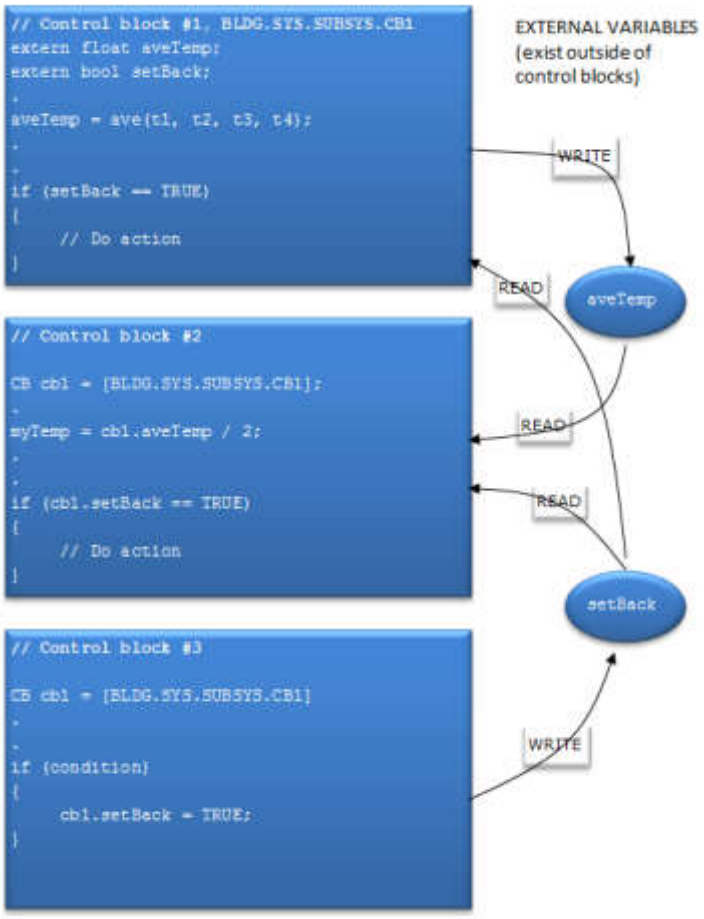


Figure 38 - External Variables

CB #1 has declared two external variables (`aveTemp` and `setBack`) which are accessed by other control blocks using the syntax:

```
controlBlockName.externalVariableName
```

In Figure 38, CB #2 is using `aveTemp` and `setBack`, whereas CB #3 is only using `setBack`.

While external variables offer added functionality and flexibility, there is an inherent danger in allowing more than one control block to "write" a value to an external variable. This is

because the variable can only contain the last written value. In the preceding example, only CB #1 is writing to `aveTemp` and only CB #3 is writing to `setback`. This makes these two variables safe from such race conditions. However, if a value is assigned to `aveTemp` in CB #3, an undesirable race condition between CB #3 and CB #1 would be created.

The following types of external variables are not allowed:

- Arrays of any variable type.
- Any object type variable.
- Any variable with local scope. All external variables must have global scope (i.e., they must be declared at the beginning of the DCL source code file).

External variables are accessible across hardware platforms (i.e., server, field computers, and controllers).

In conjunction with the EMCS database, the DCL compiler verifies that any external variable referenced with the `cb.varName` syntax has been previously declared in the referenced Control Block; otherwise, compilation will fail.

The DCL compiler also checks for instances of the following undesirable scenario. Consider a control block called CB#1 that has been successfully compiled with the declaration:

```
// Declaration in CB#1
extern float temp;
```

Subsequently, CB#2 is written and successfully compiled with the following statement:

```
// Temp is referenced in CB #2
x = cb1.temp;
```

A dependency now exists between CB#1 and CB#2. Suppose that after CB#2 has been compiled and downloaded, someone edits CB#1. Since the `extern` variable `temp` is used by CB#2, the compiler will not allow the user to delete `extern` variable `temp` in order to satisfy its "external variable dependencies."

If any other control blocks on the hardware platform are using external variables "owned" by the control block being compiled; and if there are unmet dependencies, the compilation will fail. Then the appropriate error messages that describe the dependencies will be presented.

In order to delete the variable temp from CB#1, `cb1.temp` would have to be deleted from CB#2 to break the dependency before CB#1 could be successfully compiled without the statement:

```
extern float temp;
```

4.2.8 Input/Output Variables

All control blocks have two predefined floating-point variables called `input` and `output`. The values of the input and output can also be viewed from the grid display when looking at the field information (cmd/SP/input and BD/MV/Output). When trending a control block or loop, the value of the output variable is the data that is saved to the database.

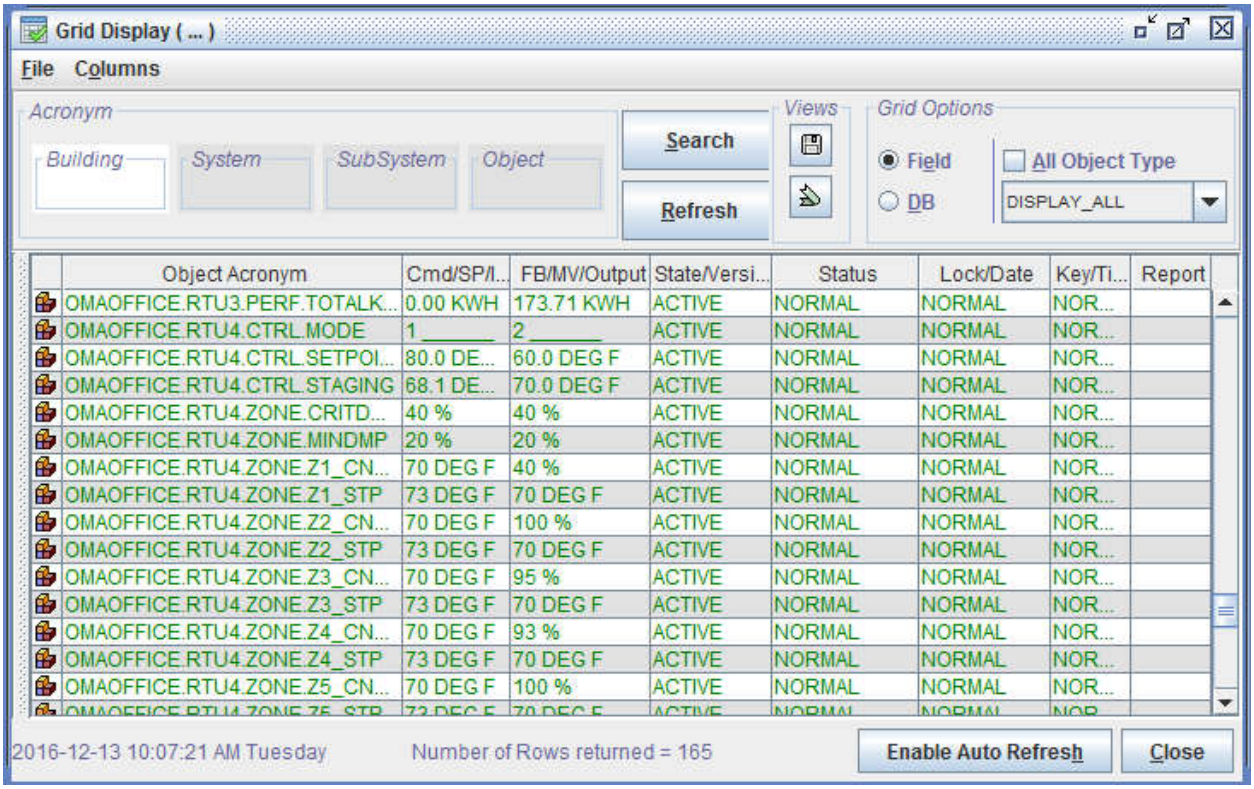


Figure 39 - Input value and output value in the Grid Display Window

The `input` value and `output` value may be viewed/modified from the field information as described in section 3.4.2.1 and section 3.4.2.2. There DCL methods described in section 4.2.9.6 can also be used for `input` and `output`.

Note that the `input` and `output` do not and should not be explicitly declared (compilation will fail otherwise). These two variables can be accessed from within a control block as any other variable.

```
DuctPressure = input;
output = AverageValue;
```

`input` and `output` are floating-points. When they are used for the Boolean application, the value of TRUE /ON is 123; and the value of FALSE/OFF is 124.

4.2.9 Object Variable Methods

EMCS objects can be manipulated by invoking certain actions on the declared object variables. In other object oriented programming languages, these actions are commonly referred to as "methods." The syntax for using all object variable methods is:

```
ObjectVariable.MethodName(argument);
```

All of the following are valid method invocations.

```
Sensor1.getValue();                // Get value on AI
Actuator1.setCommandPercent(50);   // Set value to 50% on AO
Loop1.resume();                          // Invoke resume on LOOP
```

```
t = mixedAir.getValue();          // Get the value of mixedAir
```

Note that some methods require arguments such as `setCommandPercent()` as seen in the above example. Other methods require a return a value. Each object type has certain specific methods that can be invoked. These are further categorized and described in the following sections. At least one example is given for each method. If a method returns a value, its type is indicated before the method name. Any required argument types are listed within parenthesis after the method name.

EMCS object manipulation within DCL takes place through object type variables. There are two groups of object variable types. The first group of object variable types is called *field objects.* Object variables falling into the field object category are illustrated in the following Figure.

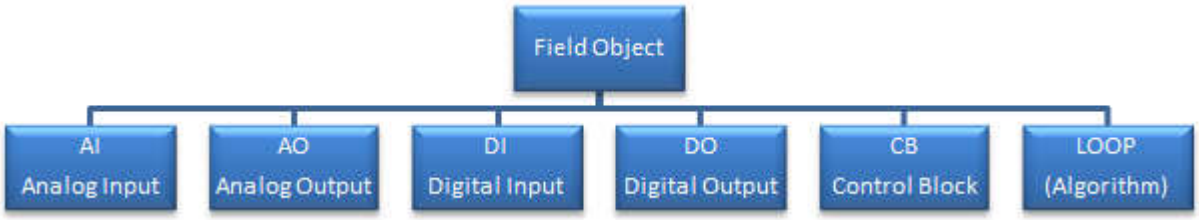


Figure 40 - Field Object Hierarchy

A second group of object variable types are not descended from the field object type.

Figure 41 - Non-Field Object Types

Object variables have their own unique methods that can be invoked as described in the following section.

4.2.9.1 Methods for Any Field Objects (AI/AO, DI/DO, CB, LOOP)

This section discusses methods that may be invoked on all types of field objects including AI, AO, DI, DO, CB, and LOOP.

getStatus()

This call returns an enumerated integer value which represents the current operating status of the object. It will return FALSE (0) if the object does not respond. Enumerated values are predefined and stored in the EMCS database as integer / capitalized string value pairs. Although displayed in string format, in the database they have predefined values. See section 4.3.1 for further discussion on enumerated constants. The system database keeps a mapping of these integer values to their string representations so that the user does not have to deal with the integer representation.

```
// Get status of controller

if (controller.getStatus() == FALSE)
{
```

```
      // Controller did not respond!
}
else
{
 // Other action
}
```

setStatus(int)

Sets object's status to a specified enumerated value (section 4.3.1).

```
// Set status of supplyFan to OUT_OF_SERVICE
supplyFan.setStatus(OUT_OF_SERVICE);
```

report(int, . . .)

This method allows the control block to send a "report". The report consists of strings, numeric values, or both strings and values to a particular object.

The first integer is an enumerated constant stored in the EMCS database that represents the type or severity of the report. Some objects use this constant to decide upon a particular course of action to take when they receive the report.

Any combination of strings or expressions can follow the first integer. The following are some examples. The string argument needs to be in the double quote string delimiter. Non-string and string values are separated by a comma.

```
// All the following report method calls are valid
monitor.report(SEVERITY_LOW,
                  "Water temperature is: ", currentTemp);
supplyFan.report(SEVERITY_LOW,
                  "Speed set to : ", 50, " by Joe");
joe.report(SEVERITY_MEDIUM,
            "Float switch status=", switch.getStatus());
```

Note that the last example used another method that calls for one of the expressions.

All non-string argument values are converted to alphanumeric strings (ASCII) when the actual report is sent. For example:

```
supplyFan.report(SEVERITY_LOW, "Speed set to : ",
                   50, " by Joe");
```

The actual report sent to `supplyFan` is: "Speed set to 50 by Joe" as shown in Figure 42. The integer value of 50 will be converted to a string and then concatenated with the strings on either side of the integer argument.

The report method is used for a number of different purposes depending on the type of object it is used with. This will become clearer as methods for other objects are presented.

When used with any field object, this method will set the object's report attribute. The report attribute can then be displayed at the EMCS user interface.

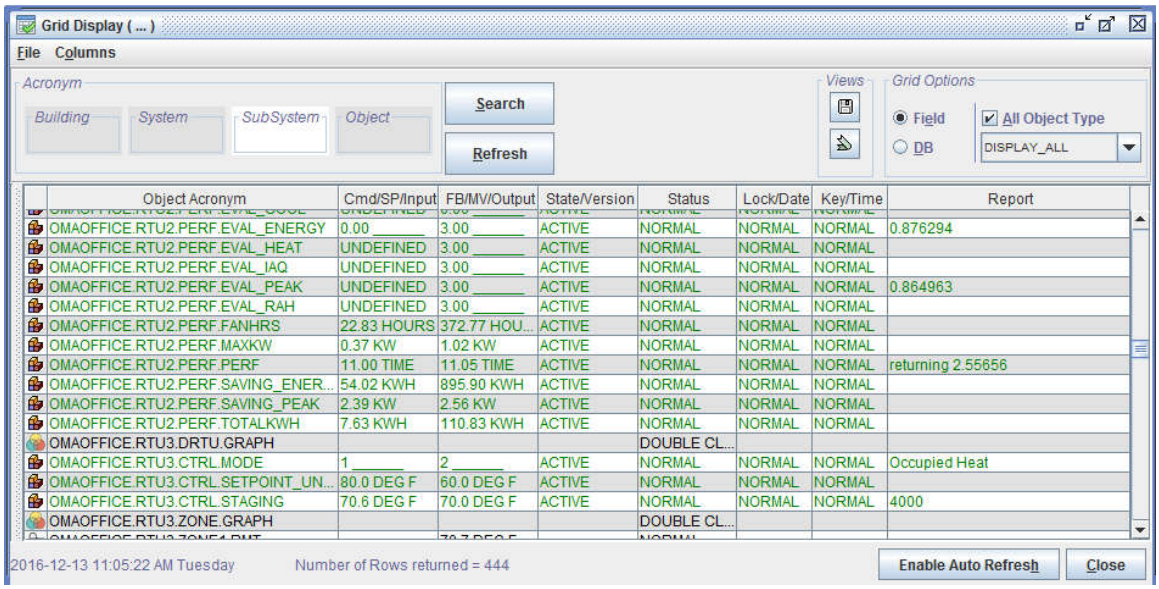


Figure 42 - Report in the Grid Display Window

4.2.9.2 AI Objects

getValue()

Returns the current floating point engineering unit value of an AI object. The returned value is computed using AI object's base and range properties set during object definition.

```
// Get value of returnTemp
currentTemp = returnTemp.getValue();
```

4.2.9.3 DI Objects

getValue()

Returns the current integer value of a DI object. The returned value will be one of the enumerated constants chosen for 0-Value or 1-Value at the DI object definition. See section 4.3.1 for a discussion of enumerated constants.

```
// Get value of floatSwitch
switch = floatSwitch.getValue();
```

4.2.9.4 AO Objects

getValuePercent()

Returns a floating point number between 0 and 100 that is representative of the AO object's current feedback value in percent. This number is computed using the AO object's base and range properties.

```
// Get percentage value of airTerminal
percent = airTerminal.getValue();
```

getValueEU()

Returns a floating point number that is the representative of the AO object's current feedback value in engineering units. This number is computed using an AO object's base and range properties.

```
// Get EU value of airTerminal
airflow = airTerminal.getValueEU();
```

getCommandPercent()

Returns a floating point number between 0 and 100 that is representative of the AO object's current command value in percent.

```
// Get command value of airTerminal in percent
cmdPercent = airTerminal.getCommand();
```

getCommandEU()

Returns a floating point number that is representative of the AO object's current command value in engineering units.

```
// Get command value of airTerminal in EU
cmdAirflow = airTerminal.getCommandEU();
```

setCommandPercent(cmd)

Sets the command value of an AO object to a given value that represents a percentage. The cmd value must be a floating point value between 0 and 100.

```
// Set command value of airTerminal to 100%
airTerminal.setCommandPercent(100);
```

setCommandEU(cmd)

Sets the command value of an AO object to a given value that represents engineering units.

```
// Set command value of airTerminal to 500 CFM
airTerminal.setCommandEU(500)
```

Note that the base and range of the object to which `airTerminal` refers should have been defined such that: base <= 500 <= range.

getLock()

Returns an integer that is representative of the AO object's current lock value.

```
// Get lock value from damper1
lock = damper1.getLock();
```

Lock (and key) values are a series of enumerated constants stored in the EMCS database. Valid constant values that should be used are as follows:

```
LK_LOW
LK_MED
LK_HIGH
LK_LOCKOUT
```

setLock(lock)

Sets the AO object's lock value to a specified value. Lock (and key) values are a series of enumerated constants stored in the EMCS database. Valid constant values that should be used are as follows:

```
LK_LOW
LK_MED
LK_HIGH
LK_LOCKOUT

// Set lock value at damper1 to LK_MED
damper1.setLock(LK_MED);
```

4.2.9.5 DO Objects

getValue()

Returns the current feedback integer value of a DO object. The value returned will be one of the enumerated constants chosen for the 0-Value or 1-Value set when the DO object was defined. See section 4.3.1 for a discussion of enumerated constants.

```
// Get feedback value of fanStart
fbVal = fanStart.getValue();
```

getCommand()

Returns the current command integer value of a DO object. The value returned will be one of the enumerated constants chosen for the 0-Value or 1-Value when the DO object was defined. See section 4.3.1 for a discussion on enumerated constants.

```
// Get command value of fanStart
cmdVal = fanStart.getCommand();
```

setCommand(cmd)

Sets the command value of the DO object to the specified enumerated constant value. For example:

```
// Set command value of fanStart
fanStart.setCommand(ON);
```

The DCL compiler will check the EMCS database to make sure that the method argument is one of the 1-Value or 0-Value constants defined for the referenced DO object.

getLock()

Returns an integer that is representative of the DO object's current lock priority value. Valid lock values are between 0 and 255.

```
// Get lock value of fanStart
lock = fanStart.getLock();
```

setLock(lock)

Sets the DO object's lock priority to a specified value. Valid lock values are between 0 and 255.

```
// Set lock value of fanStart to value contained in
// variable lock
fanStart.setLock(lock);
```

4.2.9.6 CB and LOOP Objects

getInput()

Returns the current floating point value of a CB or LOOP object's input variable.

```
// Get input value from mixedAirCB
currentInput = mixedAirCB.getInput();
```

setInput(input)

Sets a CB or LOOP object's input variable to a specified floating point value.

```
// Set input value to 55
mixedAirCB.setInput(55);
```

getOutput()

Returns the current floating point value of a CB or LOOP object's output variable.

```
// Get output value from mixedAirCB
currentOutput = mixedAirCB.getOutput();
```

getLock()

Returns an integer that is representative of the CB or LOOP object's current lock value. Valid lock values are between 0 and 255.

```
// Get lock value from pressureLoop
lock = pressureLoop.getLock();
```

setLock(lock)

Sets the CB or LOOP object's lock priority to a specified value. Valid lock values are integers between 0 and 255.

```
// Set lock value at pressureLoop to 20
pressureLoop.setLock(20);
```

getKey()

Returns an integer that is representative of the CB or LOOP object's current key priority value. Lock (and key) values are a series of enumerated constants stored in the EMCS database. Valid constant values that should be used are as follows:

```
LK_LOW
LK_MED
LK_HIGH
LK_LOCKOUT
```

```
// Get key value from pressureLoop
key = pressureLoop.getKey();
```

setKey(key)

Sets the CB or LOOP object's key priority value to specified value. Lock (and key) values are a series of enumerated constants stored in the EMCS database. Valid constant values that should be used are as follows:

```
LK_LOW
LK_MED
LK_HIGH
LK_LOCKOUT

// Set key value at pressureLoop to LK_LOW
pressureLoop.setKey(LK_LOW);
```

getState()

Returns an enumerated integer value that represents the CB or LOOP object's state. For a LOOP this can be one of for possible values (active, resumed, deactivate, shutdown).

```
// Get pressureLoop's current state
currentState = pressureLoop.getState();
```

State values are stored as enumerated constants in the database as follows:

```
STATE_ACTIVE
STATE_RESUMED
STATE_DEACTIVE
STATE_SHUTDOWN
STATE_RESTARTED
STATE_STOPPED
```

Note the last two states are only valid for a CB. A CB can be stopped at its current instruction by a user if error conditions are encountered (i.e., divide by zero, array out of bounds, etc.). A CB can be restarted at its next instruction by a user.

shutdown()

If the referenced object is a CB and its current state is ***restarted***, ***active*** or ***resumed***, the following will occur:

1. The `shutdown()` function within the referenced CB will run.
2. The CB state is set to ***shutdown***. CB execution is terminated even though the CB still exists

If the referenced object is a LOOP, and its current state is ***active*** or ***resumed***, the following will occur:

1. The LOOP's controlled object (typically of type AO) is adjusted such that it is equal to the LOOP's defined shutdown value.
2. Iteration of the LOOP is stopped and its state is set to ***shutdown***.

```
// Invoke shutdown method in CB1 if it exists
CB1.shutdown();

// Shutdown pressureLoop
pressureLoop.shutdown();
```

activate()

If the referenced object is a CB in the ***stopped***, ***deactivate*** or ***shutdown*** states, the following will occur:

1. The CB's state is set to ***active***.
2. Any user defined `activate()` function within the referenced CB will run. If no such function was defined, skip to 3.
3. Any user defined `main()` function within the referenced CB will run. If no such function was defined, skip to step 4.
4. After running `main()`, execution of the CB will stop. The CB will be moved to the inactive queue and its state set to ***stopped***.

If the referenced object is a LOOP and its state is ***deactivate*** or ***shutdown***, the following will occur:

1. The LOOP's state sets to ***active***.
2. Iteration of the LOOP will be started with zeroed out integral, derivative, and proportional terms.

```
// Invoke activate method in CB1 if it exists
CB1.activate();

// Activate pressureLoop
pressureLoop.activate();
```

deactivate()

If the referenced object is a CB and its current state is ***restarted***, ***active*** or ***resumed***, the following will occur:

1. Any user defined `deactivate()` function within the referenced CB will run. If no such function was defined, skip to 2.
2. The CB is moved to the inactive queue by the CBMgr and its state is set to ***deactivate***.

If the referenced object is a LOOP, and its current state is ***active*** or ***resumed***, the following will occur:

1. Iteration of the LOOP is stopped and its state is set to ***deactivate***.

```
// Invoke deactivate method in CB1 if it exists
CB1.deactivate();

// Deactivate pressureLoop
pressureLoop.deactivate();
```

resume()

If the referenced object is a CB and its current state is ***stopped***, ***deactivate*** or ***shutdown***, the following will occur:

1. The CB's state is set to ***resumed***.
2. Any user defined `resume()` function within the referenced CB will run. If no such function was defined, skip to 3.
3. Any user defined `main()` function within the referenced CB will run. If no such function was defined, skip to step 4.
4. After running `main()`,execution of the CB will stop. The CB will be moved to the inactive queue and its state set to stopped.

If the referenced object is a LOOP in the ***deactivate*** or ***shutdown*** state, the following will occur:

1. The LOOP's state is set to ***resumed***.
2. Iteration of the LOOP will start using any saved integral, derivative, and proportional terms.

```
// Resume pressureLoop
pressureLoop.resume();
```

4.2.9.7 HIST Objects

history(date, time, string literal, float)

This method enables the logging of a *single* object attribute at selected times. History objects provide more flexibility than automatic trending. The history object must be programmed in a control block and is used to save user defined variables at a programmable data acquisition rate. Each history method call will generate a new entry into the trending table within the EMCS database for the referenced HIST object. If one wanted to use the system date and time values, the following history statement is valid:

```
HISTObject.history(sysdate(), systime(), "chws", chwst);
```

Alternatively, if we wanted to indicate the exact top of the hour, the minutes and seconds could be zeroed out.

```
HISTObject.history(sysdate(),
                   hour():0:0,
                   "ANDREWS CHWS",
                   chwst);
```

4.2.9.8 ALARM Objects

trigger(int, . . .)

This method is used to "trigger" an alarm. The first argument represents an enumeration value that must be defined by the CB programmer. It is used to distinguish between different triggers. For example, a CB may want to trigger the same alarm object for both a freeze and flood condition. The remainder of the arguments are identical to the report arguments discussed in section 4.2.9.1. An example of how to use this method is as follows.

```
ALARM highSeverityAlarm = [UNL.ALARM.HIGH.SEVERITY];

// Alarm object enumerations
int FREEZE_ALARM = 0;
int FLOOD_ALARM = 1;

// After some conditional statements that detect a freeze
// condition. The variable "mixedAirTemperature" contains
// the value of the current mixed air temperature at some
// air handling unit
highSeverityAlarm.trigger(     FREEZE_ALARM,
                               "Coil freeze temp = ",
                               mixedAirTemperature);

// After some conditional statements that detect a flood
// condition
highSeverityAlarm.trigger(     FLOOD_ALARM,
                               "Flood at And. Hall");
```

4.2.9.9 HW Objects

getStatus()

The syntax of this method has been previously discussed in section 4.2.9.1. This function will return FALSE (0) if no response was received. It can therefore be invoked on HW objects to see if they are "on-line" or otherwise functioning. For example, one could call `getStatus()` on a controller before reading a number of points that reside there. If `getStatus()` on a HW object returned FALSE, this fact could then be used to trigger an alarm.

report(int, . . .)

This method is similar to the one having the same name described in section 4.2.9.1 and can be used to set the report attribute of a HW object.

4.3 Constants and Literals

In addition to variables, DCL includes constants and literals. Constants are enumerated values stored in the EMCS database. Literals are explicit numbers or text strings. Both constants and literals are described in this section.

4.3.1 Enumerated Constants

Enumerated constants have the following uses in DCL:

- To indicate engineering units on DI and DO objects (i.e., Value @State 1 , Value @State 0).
- To represent the criticality of reports (section 4.8.5).
- To represent object status. (section 4.1.2).
- To represent the state of control blocks and LOOP's. (section 4.1.1).
- As truth values (TRUE = 1, FALSE = 0).
- Other user defined uses.

Constants are stored in the EMCS database as integer and capitalized string value pairs. The DCL compiler will first check if a constant is some user defined constant within the CB (i.e., some user defined variable). If it is not user defined, the compiler will attempt a EMCS

database look up of the identifier. If the identifier is found, the compiler will substitute its string representation with an integer representation used by the s-code. However, source code will always be presented using the string representation.

4.3.2 Number Literals

For example, all the following are valid number literal assignments assuming that the variable "f" is a floating point type.

```
f = 1;          // Ok. Note no decimal point
f = 1.23;             // Ok. Note decimal point
f = 1.0e-6;     // Ok. Note scientific notation
```

Number literal assignments are more restrictive with `int` variables.

```
int j;
bool b;
j = 1;          // Ok
j = 1.23;             // Decimal point not allowed!
j = 1.0e-6;     // Scientific notation not allowed!
```

Assignments to `bool` variables are the most restrictive.

```
b = 1;                // Ok
b = 0;                // Ok
b = AnythingElse      // Not Allowed!
```

4.3.3 String Literals

These are used in conjunction with the `history()` and `report()` functions and methods (section 4.8.5). String literals must begin and end with double quotes.

```
"This is a valid string literal"
"This is not because it's missing an ending double quote
```

Note that DCL does not support escape characters to allow placing of quotes within strings.

4.4 Operators

4.4.1 Arithmetic Operators

The following operations are supported for `float` and `int` variables.

Name	Operator	Use
Addition	+	op1 + op2
Subtraction	-	op1 – op2
Multiplication	*	op1 * op2
Division	/	op1 / op2
Exponentiation	^	op1 ^ op2
Modulo	%	op1 % op2
Unary sign	+ or -	-op1, +op1
Increment / Decrement	++ or --	op1++ or op1--

The modulo operator can only be used with integers. It returns the result of integer division. For example, 7 % 5 = 2 (the remainder of 7 / 5 is 2).

Note that the result of any expression with mixed `int` and `float` operands will be promoted to `float`. Remainders of integer division will be dropped. Math operations on Booleans are not allowed.

Add and subtract operations are allowed on time variables. When adding, the result may represent the next day. For example, if adding 11:50:0 + 1:0:0, the result will be 0:50:0. Similar behavior may result when subtracting.

4.4.2 Relational Operators

The following relational operators are supported for Booleans, integers, floats, dates and times. The value returned from all relational operations is Boolean (0 or 1, `TRUE` or `FALSE`).

Operator	Use	Return TRUE if
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2

Mixed `int`, `float` and `bool` relational operands are allowed. Mixed operands of other types are not allowed.

4.4.3 Logical Operators

The following logical operators are supported for Booleans.

Operator	Mnemonic	Use	Return TRUE if
AND	“and”	op1 AND op2	op1 and op2 are both TRUE
OR	“or”	op1 OR op2	op1 or op2 are TRUE
!	“not”	!op1	op1 is FALSE

The value returned from all logical operations is Boolean (`TRUE` or `FALSE`).

4.4.4 Assignment Operator

`floats`, `ints`, `bools`, `dates` and `times`. It cannot be used with object variables. For example, assuming that the following variables are all object variables, the assignments made are not allowed.

```
myAI = 7.5;            // Not ok!
myAI_1 = myAI_2;       // Not ok!
```

4.4.5 Operator Precedence

Unless parenthesis are used, default operator precedence is as follows (from highest to lowest):

1. unary sign operator (+ or -)
2. exponentiation (^) (this operator is “right” associative: 2^3^2 = 2^(3^2) = 2^9)
3. multiplication and division (* or /)
4. addition, subtraction and modulo (+ , - or %)

5. relational (< > <= >=)
6. equality (== !=)
7. logical (AND, OR)
8. assignment (=)

4.5 Expressions

The basic building blocks of DCL are expressions. Expressions can be various combinations of variables, arithmetic operators, function calls, etc. The job of an expression is two-fold, performing a computation and returning a value. The data type of the value returned is dependent on the elements used in the expression.

The following are examples of expressions:

```
f * 1.08 / g              // Returns result of computation
press1 <= press2  // Returns TRUE or FALSE
z + min(x, y)             // Returns result of computation (i.e., Adds
                        // value of z to minimum of x and y)
```

4.5.1 Arithmetic Expressions

These expressions perform some computation subject to the precedence constraints discussed in section 4.4.5. Precedence can be "overridden" through the use of parenthesis. For example:

```
// Multiplication will happen first since it has higher
// precedence
x + y * x

// The preceding is equivalent to x + (y * z). We can change
// precedence so as to perform the add first
(x + y) * z
```

DCL allows mixed mode arithmetic, so arithmetic expressions can be comprised of any combination of floating point numbers, integers, and Booleans. The resultant variable type of an expression depends on the expression operands. The following table summarizes mixed mode behavior in DCL.

Mixed Mode Operands of Type	**Resultant Type**
`float, int, bool`	`float`
`float, int`	`float`
`float, bool`	`float`
`Int, bool`	`int`

Basically, `int's` and `bool's` are "promoted" if they are found in an expression with `float`, and `bool's` are promoted if found in an expression with an `int`.

Functions calls can also be operands in arithmetic expressions.

```
// Function call in arithmetic expression
x + min(x, y, z) – 10
```

Time and date variables cannot be used in arithmetic expressions.

4.5.2 Logical Expressions

Logical expressions are typically used as a test before branching program flow. Logical expressions will evaluate to TRUE or FALSE. Valid examples include:

```
// Valid logical expressions
x <= y          // TRUE if x is less than or equal to y
time1 != time2  // TRUE if time1 is not equal to time2
date1 > date2   // TRUE if date1 is greater than date2
x == ln(y)      // TRUE if x is equal to natural log of y
```

Logical expression operands can include any primitive variable type including dates and times.

Just as arithmetic expressions can be combined with additional operators (i.e., `+`, `–`, `*`, etc.), logical expressions can be combined with the operators and, or, and not (`and`, `or`, `!`).

```
// Logical expressions combined with AND, OR and ! (NOT)
x < y and z == 5      // TRUE if x less than y and z equal to 5
s1 or s2              // TRUE if s1 or s2 are TRUE
!(x > y)              // TRUE if x is less than or equal to y
                       // (i.e., x is not greater than y)
```

4.5.3 Time Expressions

Time expressions represent time values and are used in conjunction with time variables as well as time ranges (section 4.5.5) and time functions (section 4.8.4). The format of a time expression is: `hour:minute:seconds` where `hours`, `minutes`, and `seconds` can be other integer expressions including integer literals. Integer literals must be within the ranges of 0-23, 0-59 and 0-59 respectively. For example:

```
10:30:0   // Ok.
0:0:0     // Ok. This represents midnight
5:25:55   // Ok.
24:0:0    // Not Ok! Hour value out of range
10:60:60  // Not Ok! Minute and Second value out of range
12:25.2:0       // Not Ok! Minute is not an integer
```

All of the preceding time expressions had integer literals as their arguments. Therefore, the value of the expression can be evaluated at compile time. An error message will result if the compiler finds that any of the integer literal values are not within the expected range. Time expressions of any of the following forms are also valid.

```
// Valid time expressions not computable at compile time
i:10:sec()      // hours = i, min = 10, sec = current seconds
12:i + j:0      // hours = 12, min = i + j, sec = 0
j:min() + 1:k   // hours = j, min = current min + 1, sec = k
```

Note that none of the three preceding time expressions can be evaluated at compile time. Rather, evaluation occurs during the execution of the CB. This means that if any of the resulting hour, minute, or second values are out of range, the CBMgr will stop execution of the CB and set its state and status values accordingly.

Time expressions can be also assigned to time variables.

```
time t, midnight;     // Declaration of two time variables
```

```
.
.
midnight = 0:0:0;     // Assignment
.
.
t = hour():30:0              // Assignment
```

Note that the declaration of midnight could have included the assignment on the same line if desired since the time expression contained all literals. Furthermore, if the user intended to use midnight as a constant, declaration could be as follows.

```
// Declaration of midnight as constant.  Note that all caps
// are used to indicate constant status
time MIDNIGHT = 0:0:0;
```

4.5.4 Date Expressions

Date expressions represent date values and are used in conjunction with date variables as well as date ranges (section 4.5.6) and date functions (section 4.8.4). The format of a date expression is: `month\day\year` where the `month`, `day`, and `year` are integer numbers or expressions within specified ranges.

`Month` 1 to 12
`day` 1 to maximum number of days per month based on standard calendar
`year` 1 to 9999

```
1\25\2010 // Ok. January 25, 2010
9\31\2001 // Not Ok! September only has 30 days
```

Note the backslash (\) has been chosen in lieu of the typical forward slash date delineation character (/) in order to distinguish between a date and division operations.

In all of the preceding examples, the value of the date expression can be evaluated by the compiler at compile time (similar to time expressions with all literals discussed in section 4.5.3). An error message will result if the compiler finds that any of the integer literal values are not within the expected range. However, date expressions of any of the following forms are also valid.

```
// Valid date expressions not computable at compile time
i\1\2002        // month = i, day = 1, year = 2002
12\i + j\2002   // month = 12, day = i + j, year = 2002
1\day()\2010    // month = 1, day = current day, year = 2010
```

All of the preceding date expressions must be evaluated during CB execution. If any of the resulting month, day, or year values are out of range, execution of the CB is stopped and its state and status values are set accordingly.

4.5.5 Time Range

One important function of the control blocks is to start and stop equipment based on time criteria (i.e., time scheduling). Therefore, a logical Boolean construct called a “time range” is provided in order to make this task easier for the DCL programmer and to improve program legibility. The syntax of a time range is:

```
TimeExpression_1 -> TimeExpression_2
```

This expression returns a Boolean TRUE or FALSE based on whether the current system time is greater than or equal to `TimeExpression_1` and less than or equal to `TimeExpression_2`. For example:

```
8:0:0 -> 17:0:0       // Will return TRUE if current time is between
                  // 8 AM and 5 PM, FALSE otherwise
```

This expression is meant to replace the following equally valid expression:

```
8:0:0 <= time() AND time() <= 5:0:0
```

4.5.6 Date Range

Date ranges are analogous in purpose and syntax to time ranges. Their syntax is:

```
DateExpression_1 -> DateExpression_2
```

For example:

```
12\1\2010 -> 12\31\2010 // Will return TRUE if current date
                        // falls in the month of December, 2010
```

4.6 Statements

DCL statements are comprised of one or more expressions. Single line statements must end with a semicolon (`;`). As will be seen, statements that can potentially contain other statements are delineated with opening and closing brackets. This is to allow the compiler to "recover" at the next statement after a syntax error. The DCL compiler ignores all source code "white space" (spaces, tabs, and line feeds) so that single statements can be spread over multiple lines if necessary to improve the readability of the DCL. With the exception of variable declarations (covered in section 4.2.5), all statements must be part of a function (either user defined or reserved). The overall CB file structure is described in more detail in section 4.10.1. This section illustrates the various statement types in the DCL.

4.6.1 Assignment Statement

Assignment statements are used to assign a value to a specific variable. Numerous examples have previously been presented. Valid examples include:

```
f = 12 + 3;
currentPos = valve1.getValuePercent();
t = hour():0:0;
```

4.6.2 Function or Object Method Call

If a function or an object method call does not return a value, the call itself is a statement.

```
// Function and Object Method calls as a statement
delay(10);
damper.setCommandPercent(50);
```

Note that if a user defined or built in function is supposed to return a value, but the compiler finds the function call on a line by itself, CB compilation will fail. Functions that return a value must be used in an expression or as part of an assignment statement.

4.6.3 If-else Statement

If-else statements are used for branching the flow of program execution into two paths. *If* statement syntax is as follows:

```
if (boolean expression)
{
     statements;
}
```

If combined with *else*:

```
if(boolean expression)
{
     statements;
}
else
{
     statements;
}
```

All statements enclosed in the braces are considered to be part of the *if* or *else* clause. Note that in order to encourage the use of the switch-case statement (section 4.6.6), an "else-if" construct is not provided.

4.6.4 for Statement

For statements can be used when the number of looping iterations are known. Examples of this include iterating through an array. The syntax of *for* statements is as follows:

```
for (initialization; boolean expression; increment)
{
     statements;
}
```

The initialization is an assignment statement that is used to initialize the loop control variable. The boolean expression determines when the loop stops. The increment defines how the loop control variable changes each time the loop is executed. Again, braces are used to indicate the beginning and end of *for* loops. The following are valid examples of *for* statements:

```
// j starts at 0, increment j by one each time thru loop, stop
// before j reaches 10
for(j = 0; j < 10; j = j + 1)
{
     statements;
}

// j starts at 10, decrement j by 2 each time through loop, stop
// after j = 0
for(j = 10; j >= 0; j = j - 2)
{
     statements;
}
```

Note that the increment and decrement operators (++ and --) can be used on the increment expression to save typing as follows:

```
// j starts at 0, increment j by one each time through loop,
// stop before j reaches 10
for(j = 0; j < 10; j++)
{
     statements;
}
```

4.6.5 while and do-while Statement

A *while* statement performs some action "while" a certain condition remains TRUE. The general syntax of the *while* statement is:

```
while(boolean expression)
{
     Delay(5)
     statements;
}
```

The reader should now be familiar with the curly brace demarcation of the statement. The *do-while* is similar. However, the conditional test is performed at the end of the loop rather than at the beginning. Thus, a *do-while* loop will always be executed at least once.

The syntax of the *do-while* loop is:

```
do
{
     statements;
} while (boolean expression);
```

Note the semi-colon required at the end. A concrete example that includes a `while` and a `do-while` loop is as follows:

```
while (j < 10)
{
     do
     {
           k = j + 1;
     } while(k < 20); // End of do-while
     j = j + 1;
} // End of while !
```

4.6.6 switch-case Statement

A *switch-case* statement is intended to replace multiple *if - else if* statements in order to improve program legibility and understanding. Note that DCL *switch-case* statements differ from normal C / C++ / Java syntax in that a Boolean expression is evaluated at each *case*.

The basic syntax of a DCL *switch-case* statement is as follows:

```
switch
{
     case (boolean expression1)
     {
           statements;// If expression1 is true
           break;
     }
     case (boolean expression2)
```

```
    {
          statements;// If expression2 is true
          break;
    }
    default
    {
          statements;// If all preceding expressions are
                     // false
  }
}
```

Note that any number of case clauses can be included and that unlike C / C++ and Java, a colon is not necessary after the boolean expression or the default statement. The parenthesis around the boolean expressions may also be omitted if desired.

The *default* clause enables the triggering of some default action if all the preceding *case* clauses are FALSE. If the *break* statement is omitted within a case, then program execution will continue with the next case statement that evaluates TRUE. If the *break* statement is included as shown, program execution will jump out of the switch statement. The following example illustrates these points.

```
switch
{
    case (OutsideAirTemp < 70)
    {
          AH1_Economizer.activate();
          // No more cases evaluated
          break;
    }
    case (OutsideAirTemp < 20)
    {
          // With the preceding switch statement,
          // program execution will never reach here
          AH1_Economizer.deactivate();
    }
    default
    {
          ; // No Op
    }
} // End switch
```

In the preceding example, the program will exit the *switch-case* statement if the first case statement is TRUE and execution never reaches the second case statement. In this particular instance, it would have been better to reverse the order of the case statements.

4.6.7 stop Statement

A *stop* statement causes the CB to stop execution unconditionally. Its state will then remain ***stopped*** until ***activated*** or ***resumed*** by another CB, or until ***activated***, ***resumed*** or ***restarted*** by a user. Its syntax is simply:

```
stop;
```

This could be used to stop CB execution if some error condition arises. For example:

```
while(TRUE)
```

```
{
     // normal code
     if (failure_mode == TRUE){stop;}
     // more normal code
}
```

4.6.8 break Statement

The usage of the *break* statement has already been shown in conjunction with the *switch-case* statement. Basically, a *break* statement causes program execution to leave the immediate surrounding for, *while* loop, *do-while* loop or *switch-case* statement and go to the statement after the loop. For example:

```
// Note the break statement causes the loop to
// terminate if k > 10
while(j < 10)
{
     k = -5 + j;
     if (k > 10)
     {
           break;
     } // End if
} // End while
k = 0; // Program will jump to here after break statement
```

4.6.9 continue Statement

A *continue* statement can be considered to be a special case of *break*. A *continue* statement causes program execution to skip to the end of a *for, while,* or *do-while loop* and then evaluates the conditional expression at the beginning of the loop. For example:

```
for(j = 0; j < 100; j = j + 1) // conditional expression
{
     if (j / 10 = 0)
     {
           y = j;
           continue;
     } // End if
     k = k + j;
     z = k;
}   // Program jumps to here after continue statement and then
     // back to conditional expression at the top of the for
     // loop.  If a break statement had been used instead, the
     // conditional expression would not be evaluated.
```

4.6.10 return Statement

A *return* statement is used to initiate a return from a user defined function (see section 4.7). This statement can have one of two possible forms:

```
return(expression);   // The function was defined as returning
                       // some value
return;   // The function was defined as returning
           // no value
```

The returned value used in the first form can be any valid expression. Valid examples include:

```
return(10);
```

```
return(i);
return(command + 0.5);
return(min(x,y));
```

Note that in the last example, we are returning the result of the `min()` library function call (section 4.8.1).

4.7 User Defined Functions

Statements within a CB are grouped into one or more functions. These functions can be classified into two types:

- Reserved functions that can be invoked both from within the CB and externally by users and other control blocks. These are: `activate`, `resume`, `main`, `deactivate`, and `shutdown`.
- Those that can be invoked only from other functions within the CB. The general syntax of a function is:

```
returnValue functionName(varType varName1, varType varName2 … )
{
   // Local Variable Declarations
   // Statements to do the work of the function
   // Return Statement
}     // End of function
```

The first item, `returnValue`, indicates what type of variable the function returns (if any). This can be any primitive or object type variable. If the function does not return a value, then `returnValue` is omitted.

The next item, `functionName`, is a valid identifier given by the programmer. This name can be: `activate`, `resume`, `main`, `deactivate`, `shutdown` or some other valid identifier subject to the same rules for naming variables.

Any arguments that will be passed to the function are included within the parenthesis. Each argument is identified with a variable type and variable name. The variable type is necessary so that the compiler can check these types against any subsequent calls to this function from other functions. Variable type can be any primitive or object type. If the function does not take any arguments, then the space between the parentheses are left blank.

A simple example will help clarify the preceding discussion.

```
// This function called "adder" adds two floats and returns the
// result as a float
float adder(float x, float y)
{
   return (x + y);
}
// "adder" is called from main() function
main()
{
   // Some code
   result = adder(2.3, z);
   // Some more code
}
```

In this simple example, the function "adder" has been defined. It takes two floating point arguments and returns a floating point value. In function "main", the "adder" function is called with arguments 2.3 and z. The return value of the function is then assigned to the variable called result.

Note that the five reserved functions (activate, resume, main, deactivate, and shutdown) do not return any values nor do they need any arguments.

Since DCL supports mixed mode arithmetic, the preceding code will compile successfully even if the arguments to adder were integers or booleans. Essentially, any integer or boolean arguments would have been "promoted" (or cast) to floating point values as part of the function call. Similarly, a bool will be "promoted" to an integer if an integer is expected as a function argument. However, when passing times, dates, and objects, the argument type must match the function definition exactly. For example, passing an AO object to a function that expects an AI will not compile.

4.7.1 Local Function Variables

Any local variables must be declared before any other statements within the function. Therefore, the following function definition will not compile successfully.

```
main()
{
     int i = 1; // Variable definition ok
     i = i + 1; // Valid statement
     AI a; // ERROR: variable declaration
     // out of place, it should be before other statements
}
```

4.7.2 Before Calling Functions

Any function definition that is not one of the five predefined functions (i.e., activate(), resume(), deactivate(), shutdown(), and main()) must come before any calls are made to the function. For example, the following code will not compile:

```
// This will not compile!
main()
{
     // myFunction is called before its definition
     myFunction();
}
myFunction()
{
     // myFunction operations
}
```

The definition of myFunction must come before it is called.

```
// This will compile
myFunction()
{
     // myFunction operations
}
main()
{
```

```
    // myFunction is called before its definition
    myFunction();
}
```

4.7.3 Function Parameter Passing

4.7.3.1 Primitive Variables

When passing non-array primitive type variables to functions, DCL uses a default mechanism that is commonly known as "pass by value." What this means is that the contents of any variables passed to a function are not changed when the function returns. The following example illustrates this point.

```
// This function attempts to swap two values but does not
// work as intended!
swap(float x, float y)
{
      float temp; // local variable!
      temp = x;
      x = y;
      y = temp;
      return;
}
main()
{
      float a = 1; // a and b declared with local scope
      float b = 2;
      swap(a, b); // swap called with a and b
      // More operations on a and b
}
```

When `swap` returns from being called in `main`, `a` and `b` will still be set to their original values (i.e., 1 and 2). Note that this is true even if `a` and `b` were declared with global scope. While this may seem to be limitation in this particular example, there are other instances where pass-by-value is indeed the desired behavior.

In order to obtain the desired effect with our "swap" function, DCL supports a second parameter passing mechanism that is commonly referred to as "pass-by-reference." Pass-by-reference behavior is indicated by placing an ampersand (`&`) in front of the variable name in the function declaration. In the preceding example, the intended behavior can be achieved as follows:

```
// This function swaps two values using pass-by-reference
// behavior
swap(float &x, float &y)
{
      float temp; // local variable!
      temp = x;
      x = y;
      y = temp;
      return;
}
main()
{
      float a = 1; // a and b declared with local scope
      float b = 2;
```

```
	swap(a, b); // swap called with a and b
	// More operations on a and b
}
```

Now when swap returns, the desired behavior will be achieved: `a` will be equal to 2 and `b` will be equal to 1. Note that the same function can have both pass-by-value and pass-by reference parameters. For example, the following function definition is perfectly valid.

```
doSomething(float &ductPress, int count)
{
	// Code
}
```

In this example, any argument passed in for `ductPress` will be pass-by-reference, whereas `count` will default to pass-by-value behavior.

4.7.3.2 Object Variables

In order to support code reuse, DCL also supports passing object variables as function arguments. The syntax of passing object variables is entirely analogous to that of other primitive variable types. For example, the following function requires two object variables.

```
// Function which takes AI and AO object as arguments
genericLoop(AI measuredVariable, AO controlledVariable)
{
	// Code
}
```

DCL always passes object variables as pass-by-reference. An ampersand is not required. If the programmer places an ampersand in front of an object variable within a function definition, the compiler will generate a warning, but the program will still compile.

```
// Function which takes AI and AO object as arguments
// Pass-by-reference ampersand on measuredVariable
// is not necessary and will generate warning
genericLoop(AI &measuredVariable, AO controlledVariable)
{
	// Code
}
```

4.7.3.3 Arrays of Primitive and Object Variables

Individual array elements can be passed to functions as any other primitive or object variable as discussed in the previous sections. DCL also supports the passing of entire arrays as function parameters. The syntax required to pass an array is illustrated by the following example.

```
// Array function parameter example
float averageTemp(AI temps[])
{
	float total;
	int i;
	total = 0;
	for (i = 0; i < arraySize(temps[]); i++)
	{
		total = total + temps[i].getValue();
	}
```

```
	return(total / arraySize(temps[]));
}
main()
{
	// Variables and code
	ave = averageTemp(temps[]);
}
```

Note the syntax required to define an array parameter and to pass an array argument are identical: `arrayVariableName[]`.

The built-in library function `arraySize` (section 4.8.6) can be used to get the size of any arrays and is therefore very useful when used in conjunction with `for` loops. The above example illustrates the ability of DCL to facilitate code reuse in that the function `averageTemp` is generic. Users can pass in an AI array of any size that represents any combination of AI objects in the system and the function will compute their average value.

If the function definition calls for an array parameter but the function call does not provide one, or vice-versa, the program will not compile.

```
// Will not compile since function requires an array but
// function call does not provide one!
someFunction(float a[])
{
	// Code
}
someOtherFunction()
{
	// Variable declarations and code
	//someFunction is called without array argument
	someFunction(value);
}
```

Arrays are always passed in DCL using pass-by-reference behavior. Similarly, an ampersand is not required. If a user places an ampersand in front of an array function parameter, a compiler warning message will be generated but the program will compile successfully.

4.8 Library Functions

This section describes several “built-in” or library functions which are supported by DCL.

4.8.1 Math Functions

abs(float), abs(int)

Returns the absolute value of a floating point or integer number. The type returned (float or int) is the same as the argument type.

max(float, float, int, …), max(int, int …), max(arrayName[])

Returns a floating point or integer value that represents the largest of a list of floating point or integer numbers. Mixed lists are allowed, but in this case the function will return a floating point value. Note that a single float or integer array can be passed as an argument. The function will then return the maximum value of the array. However, a mixed list of individual numbers and one or more arrays is not allowed.

min(float, float, int, ...), min(int, int ...), min(arrayName[])

Returns a floating point or integer value that represents the smallest of a list of floating point or integer numbers. Mixed lists are allowed, but in this case the function will return a floating point value. Note that a single float or integer array can be passed as an argument. The function will then return the minimum value of the array. However, a mixed list of individual numbers and one or more arrays is not allowed.

ave(float, float, int, ...), ave(int, int ...), ave(arrayName[])

Returns a floating point or integer value that represents the average of a list of floating point or integer numbers. Mixed lists are allowed, but in this case the function will return a floating point value. Note that a single float or integer array can be passed as an argument. The function will then return the average value of the array. However, a mixed list of individual numbers and one or more arrays is not allowed.

ln(float)

Returns a floating point value that represents the natural logarithm (base = e) of a floating point argument. Note that the exponentiation operator can be used as the inverse of `ln()`.

```
// ln() and exponentiation
const float E = 2.718;
float f;
float g;
g = ln(12);
// Now we can get make f = 12
f = E ^ g;
```

4.8.2 Flow Control Functions

delay(float)

Delays the program execution for a specified number of seconds. All DCL programmers should be fully aware that *delay* functions implicitly impose priorities among the control blocks on the same hardware platforms. Those control blocks with high delays have an implied lower priority than those control blocks with short delays. A continuously running CB will typically have a *while* statement with a *delay* function call as follows:

```
while (TRUE)
{
     // Do some stuff
     delay (10); // Delay 10 seconds
}
```

4.8.3 Status Functions

getIOErr()

Returns an integer value that is representative of the current I/O error condition flag. Returns zero (i.e., FALSE) if there is no error. Error conditions are enumerated values that represent a particular I/O error condition. These allow the programmer to take different actions within the control block based on the particular error condition.

getStatus()

This gets the status of the calling control block. The value returned is an enumerated value. This is similar to using the `getStatus()` object method call discussed in section 4.2.9.1. However, no object variable is required.

setStatus(status)

This sets the status of the of the calling control block to an enumerated value. This is similar to using the `setStatus()` object method call discussed in section 4.2.9.1. However, no object variable is required.

4.8.4 Time and Data Functions

Note that all time and data functions return integer values except for weekday functions. Weekday functions return Boolean type values.

systime()

Returns an integer time value which represents the current system time.

hour()

Returns an integer (0 – 23) that corresponds to the current hour.

minute()

Returns an integer (0 – 59) that corresponds to the current minute.

second()

Returns an integer (0 – 59) that corresponds to the current second.

sysdate()

Returns a Date value which represents the current system date.

month()

Returns an integer (1 - 12) that corresponds to an enumerated value for the current month of the year (i.e., JAN, FEB, MAR, etc.)

day()

Returns an integer (1 - 31) that corresponds to the current day of the month.

year()

Returns an integer that corresponds to the current year.

sun(), mon(), tue(), wed(), thu(), fri(), sat()

These each return a boolean (TRUE/FALSE) based on the condition:

```
sun(); // TRUE if today is Sunday, FALSE otherwise
mon(); // TRUE if today is Monday, FALSE otherwise
etc. . .
```

4.8.5 Reporting Functions

report(severity, . . .)

This "reports" to the calling control block. Its usage is identical to the object method call discussed in section 4.2.9.1. However, no object variable is required since the implied recipient is the calling control block. If the calling control block is subsequently displayed at the EMCS user interface, the last written "report" values are displayed. This function can be thought of as a "print" statement for the control block.

4.8.6 Array Functions

arraySize(array name)

This function returns the number of declared elements in an array. It is useful when included as part of loop constructs and when passing arrays to user defined functions.

```
// Declare and array of 10 AO elements
AO speedControl[10];

// Other code

// This loop will execute 10 times since speedControl is a 10
// element array i will be incremented from 0 to 9!
for (i = 0; i < arraySize(speedControl[]); i++)
{
     // code
}
```

4.9 Compiler Directives

DCL provides the capability to embed certain instructions to the compiler within the source code. These instructions are referred to as compiler "directives" and are discussed in this section. All such directives are preceded with the "pound" sign (#).

4.9.1 #include

The included directive provides DCL with the capability for control block reuse. Reusable or library type control blocks can be developed with generic code that can be combined into larger control blocks. The syntax of this directive is as follows.

```
// Include directive syntax. Note no semicolon is used.
#include [building.system.subsystem.object]
```

Note the object acronym has the same form as that used in the object variable definitions discussed in section 4.2.5.2. The rationale for reusable control blocks and the mechanism by which they are achieved can best be illustrated by an example.

Suppose we want to develop a "library" type function that computes the average value of an array of Analog Inputs (e.g., average outdoor air temperature). Suppose that the control block called `UNL.LIB.AI.AVE` has been defined for this purpose and it contains the following code.

```
// [UNL.LIB.AI.AVE] Library function
float aiAverage(AI aiArray[])
{
     int i;
```

```
    float average;
    average = 0;
    for (i = 0; i < sizeof(aiArray); i++)
    {
          aiAverage = aiAverage + aiArray[i].getValue();
    }
    aiAverage = aiAverage / sizeof(aiArray);
    return (aiAverage);
}
```

The function `aiAverage()` takes an array of `AI` objects, computes their average value, and returns this value as a `float`. Note that the function is self contained in that it does not require any information about what the objects represent, the size of the actual array, etc. It is therefore a good candidate for reuse.

Suppose we had a second control block where we would like to use this `aiAverage()` function. We could do this as follows.

```
// Include aiAverage() function definition by compiler
// directive.
// Notes:
// 1) The definition should be included before the actual
// function is called.
// 2) The #include directive used in this manner must be
// located outside of function definitions
#include [UNL.LIB.AI.AVE]

..

Statements; // Other code

..

// Call aiAverage function from within second control block
// at any place after the #include directive
ave = aiAverage(freshAirTemp[]);
```

When the compiler hits the `#include` directive, it "copies-and-pastes" the source code of `UNL.LIB.AI.AVE` into the second control block at the exact location of the directive. Any function defined with the "included" control block is thus able to be used. In order to promote code maintenance, the preferable location for `#include` directives is directly after the global scope variable declarations (see section 4.2.2).

4.10 Control Block Structure

4.10.1 Elements of a Control Block

A valid control block is comprised of the following elements:

- Zero or more global variable declarations. These are declared outside of any functions and therefore have global scope (section 4.2.2). A control block will still compile if there are no global variables declared.
- Zero or more #include directives.
- One or more user function definitions. Note that at least one function is required or the control block will not compile. If only one function is defined, it should be one of the five

pre-defined function names (`activate`, `deactivate`, `resume`, `shutdown` or `main`). Since the control block has no defined entry point, it will otherwise never run.

A typical control block will have a `main` function and perhaps one or more other functions.

The following control block layout is suggested:

```
// Control block documentation comments
// See commenting standard in the Appendix
// Global variable declarations. All variables have global
// scope!
int j, k;
float f;
AI myAI = [Building.System.SubSystem.Object];

// User defined functions
// User function documentation comments
// See commenting standard in the Appendix
myFunction1()
{
     // Local variables needed
     int j, k;
     time t;

     statements;
}

// User function documentation comments
// See commenting standard in the Appendix
myFunction2()
{
     statements;
}

// Activate and resume
activate()
{
     statements;// If activated.
}

resume()
{
     statements;// If resumed.
}

// The main function.
main()
{
     // Infinite while loops go here.
     while(TRUE)
     {
           // Call myFunctions.
           myFunction1();
           myFunction2();
           if (failure_mode = severity_8)
           {
```

```
                shutdown();
            }
            if (failure_mode >= severity_9)
            {
                stop;
            }
            // Delay 1 minute between iterations.
            delay(60);
        } // End of infinite while loop.
} // End of main().

// Deactivate and shutdown
deactivate()
{
    statements;// If deactivated.
}

shutdown()
{
    statements;// If shutdown.
}
```

All functions, including the five predefined types (i.e., `activate`, `resume`, `deactivate`, `shutdown`, and `main`) can be listed in any arbitrary order if they exist in the CB. However, user functions must be defined before they are called. If the definition for `myFunction1` in the above example had been listed after `main`, the control block would not have compiled since `myFunction1` is called from `main` before it is defined.

4.11 DCL Editor

The DCL editor is a component of Digi-SFT used for editing, compiling, and downloading control blocks to controllers.

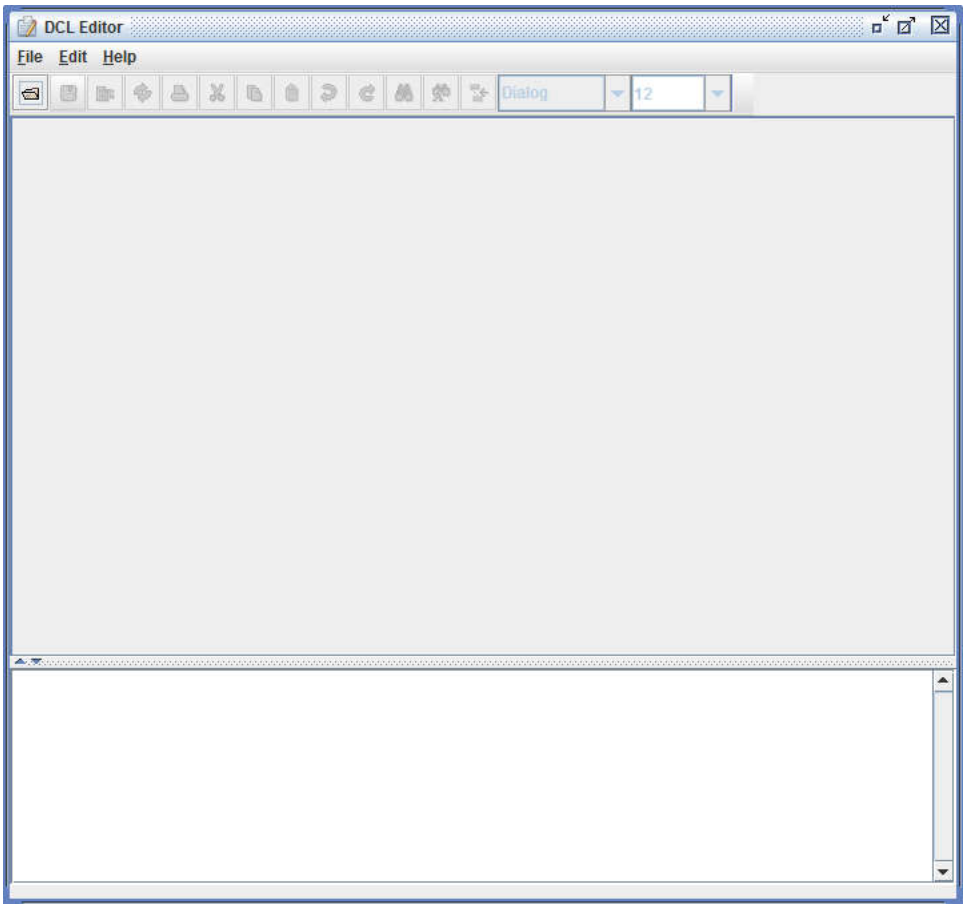


Figure 43 - DCL Editor Window

The #include directive will appear as "+include". Click on the plus (+) symbol to open the referenced file in a read-only window. The #including will appear as "-include". Click on the minus (-) symbol to close the referenced file.

4.11.1 Menu

The pull-down menus from the DCL Editor include File, Edit, and Help. The menu commands are described below.

4.11.1.1 File

The file menu provides commands for managing DCL control block files.

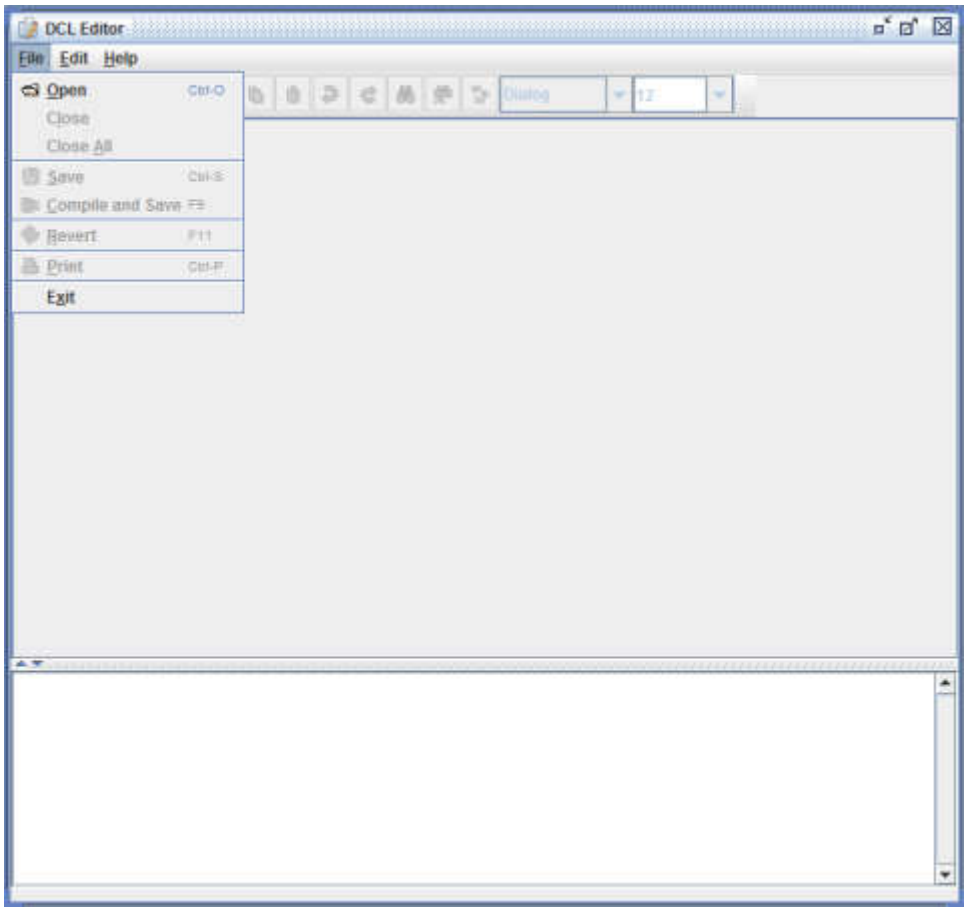


Figure 44 - File menu in the DCL Editor Window

4.11.1.1.1 Open

The Open command allows users to open a control block by entering the four part alpha-numeric acronym assigned to the object. If users enter an acronym that does not exist, the EMCS Object Selector Dialog window will remain open to allow you to enter an existing acronym for a control block object. If the acronym you enter is not associated with a control block object, an error message will appear, "Incorrect object type selected." The EMCS Object Selector Dialog window will remain open to allow users to enter an existing acronym for a control block object.

4.11.1.1.2 Close (All)

The Close command closes the control block displayed in the active editor window without saving the changes.

4.11.1.1.3 Save

The Save command saves the control block source code displayed in the active editor window without compiling the source code.

4.11.1.1.4 Compile and Save

Compile and Save saves the control block source code displayed in the active editor window to the system database. It then compiles the source code, saves it in the database, and downloads it to the host controller.

4.11.1.1.5 Revert

Revert undos changes that were made since the last compile. Note that when using Revert any changes made since the last compile will be lost. There is no redo command under the File menu.

4.11.1.1.6 Print

The Print command opens a print dialog box for modifying the print settings. It allows users to print the control block source code in the active window to the selected printer.

4.11.1.1.7 Exit

The Exit command prompts users to save their work and close the DCL Editor.

4.11.1.2 Edit

The edit menu provides commands for editing the DCL source code in the DCL Editor.

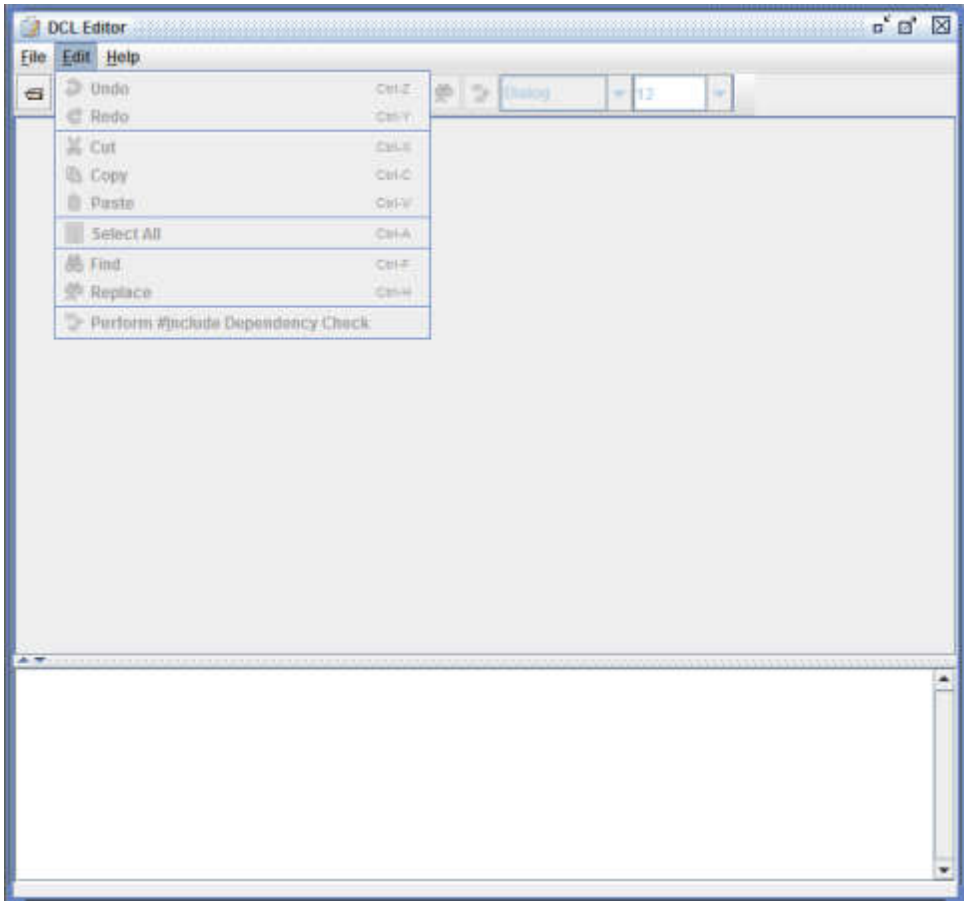


Figure 45 - Edit Menu in the DCL Editor Window

4.11.1.2.1 Undo

The Undo command will remove the last changes made to the source code, one change at a time, and in order of the most recent change.

4.11.1.2.2 Redo

The Redo command adds the changes to the DCL source code removed using the Undo command. This is done one change at a time in the order of the most recent Undo.

4.11.1.2.3 Cut

Cut deletes the selected text from the DCL Editor window and saves a copy of the text in the Windows Clipboard so that it can be pasted in another location, file, or application.

4.11.1.2.4 Copy

Copy saves the selected text from the DCL Editor window to the Windows Clipboard. The copied text can then be pasted to another location, file, or application. The original selected text in the DCL Editor window is not deleted.

4.11.1.2.5 Paste

Paste inserts text from the Windows Clipboard to the DCL Editor window. The text will be inserted after the position where the cursor is located.

4.11.1.2.6 Select All

Select All will select all of the text in the active DCL Editor window.

4.11.1.2.7 Find

Find opens a Find and Replace window. The Find command searches the current DCL Editor window for text input into the “Find what” field. The search will be performed starting where the cursor is currently located in the DCL Editor window.

4.11.1.2.8 Replace

Replace opens a Find and Replace window. The Replace command searches the current DCL Editor window for the text input into the “Find what” field. That text will be deleted and text input into the “Replace” field inserted in its place. The search will be performed starting where the cursor is currently located in the DCL Editor window.

4.11.1.2.9 Perform #include Dependency Check

The #include Dependency Check queries the database for any other control blocks that reference the current control block with the #include directive. A window will open listing all of the control blocks that reference the current control block with the #include directive. The object acronym and compile message is displayed for each control block.

4.11.1.3 Help

The help tab under the DCL Editor opens the DCL Manual document that describes how to use the Distributed Control Language.

4.11.2 Tool Bar

The toolbar includes commonly used commands in the DCL Editor. They function the same as the commands available in the pull-down menus. The toolbar commands are shown below. See section 4.11.1 for a discussion on how to use the commands.

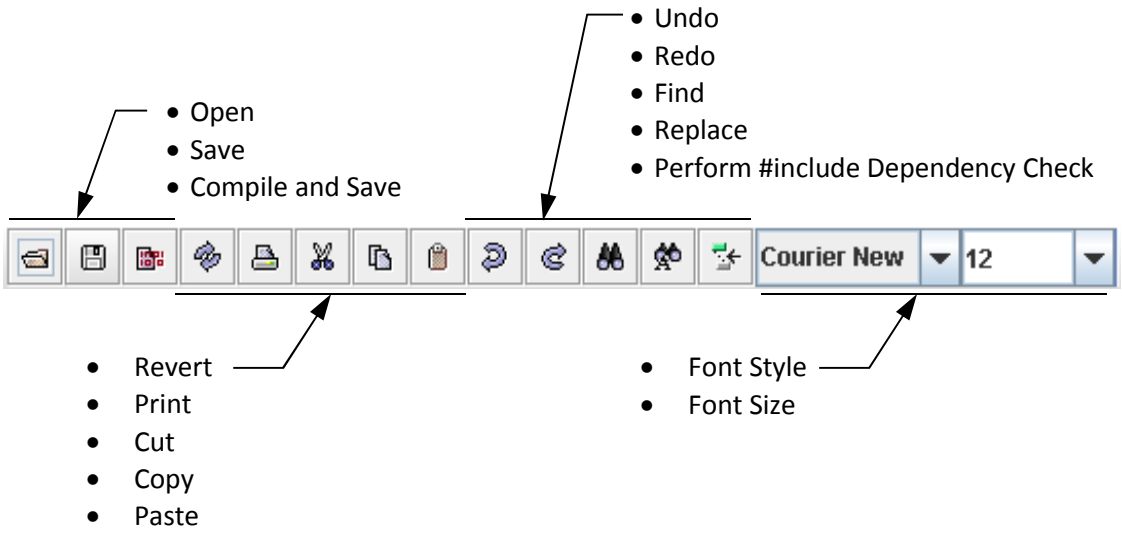


Figure 46 - Tool Bar in the DCL Editor Window

4.11.3 Pop-Up Menu

Open a control block and click the right mouse button to view the pop-up menu in the DCL Editor.

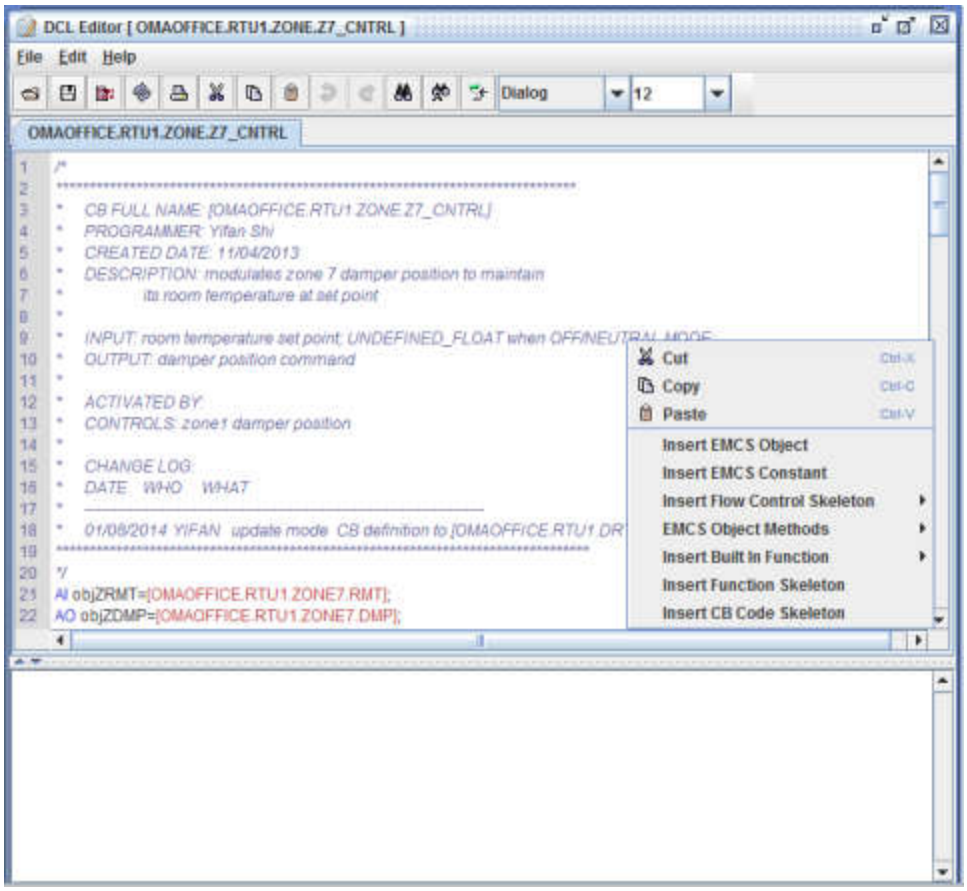


Figure 47 - Pop-up menu in the DCL Editor Window

4.11.3.1 Cut, Copy, Paste

The cut, copy, and paste commands function the same as in the edit menu; see sections 4.11.1.2.3, 4.11.1.2.4, and 4.11.1.2.5, respectively.

4.11.3.2 Insert EMCS Object

EMCS objects can be used and manipulated by control blocks as described in sections 4.2.4 and 4.2.9. The syntax used to reference an object is, for example:

```
[Building.System.Subsystem.Object]
```

Recall that this represents the complete four-part acronym of the object defined in the EMCS user interface during object creation. Selecting to insert an EMCS object from the pop-up menu will automatically create this syntax in a control block. A prompt will appear asking for the four-part acronym.

4.11.3.3 Insert EMCS Constant

There are predefined constants that are available for use in a control block algorithm. The constants are categorized into 6 sections: Status, User Settable Status, Lock/Key, Undefined Primitives, Report Severity, and CB or Loop State. See section 14.3 for more details on the available constants. A constant can be inserted into a control block by selecting the constant from a list in the pop-up menu in the DCL Editor.

4.11.3.4 Insert Flow Control Skeleton

A template or skeleton of certain DCL code statements will be inserted into a control block in the DCL Editor by selecting to insert a flow control skeleton from the pop-up menu. This includes the following statements:

- if (see section 4.6.3)
- if..else (see section 4.6.3)
- for (see section 4.6.4)
- while (see section 4.6.5)
- do..while (see section 4.6.5)
- switch (see section 4.6.6)

4.11.3.5 EMCS Object Methods

The methods that are available to manipulate control blocks are described in section 4.2.9. They can be inserted into a control block in the DCL Editor by selecting the EMCS object method from the pop-up menu.

They are listed here for reference:

- AI
  - getStatus()
  - setStatus()
  - report()
  - getValue()
- AO
  - getStatus()
  - setStatus()
  - report()
  - getValuePercent()
  - getValueEU()
  - getCommandPercent()
  - getCommandEU()
  - setCommandPercent()
  - setCommandEU()
  - getLock()
  - setLock()
- DI
  - getStatus()
  - setStatus()
  - report()
  - getValue()
- DO
  - getStatus()
  - setStatus()
  - report()
  - getValue ()
  - getCommand()

    - setCommand()
    - getLock()
    - setLock()
- CB
    - getStatus()
    - setStatus()
    - report()
    - getInput()
    - setInput()
    - getOutput()
    - getLock()
    - setLock()
    - getKey()
    - setKey()
    - getState()
    - activate()
    - resume()
    - deactivate()
    - shutdown()
- LOOP
    - getStatus()
    - setStatus()
    - report()
    - getInput()
    - setInput()
    - getOutput()
    - getLock()
    - setLock()
    - getKey()
    - setKey()
    - getState()
    - activate()
    - resume()
    - deactivate()
    - shutdown()
- HIST
    - history()
    - getLastValue()
- ALARM
    - trigger()
- HW
    - getStatus()
    - report()

4.11.3.6 Insert Built-In Function

Functions can be inserted into a control block in the DCL Editor by selecting to insert a built-in function from the pop-up menu. Functions are categorized as math, flow control,

status, time/date, report, and array (see below). See section 4.8 for more information about the built-in library functions.

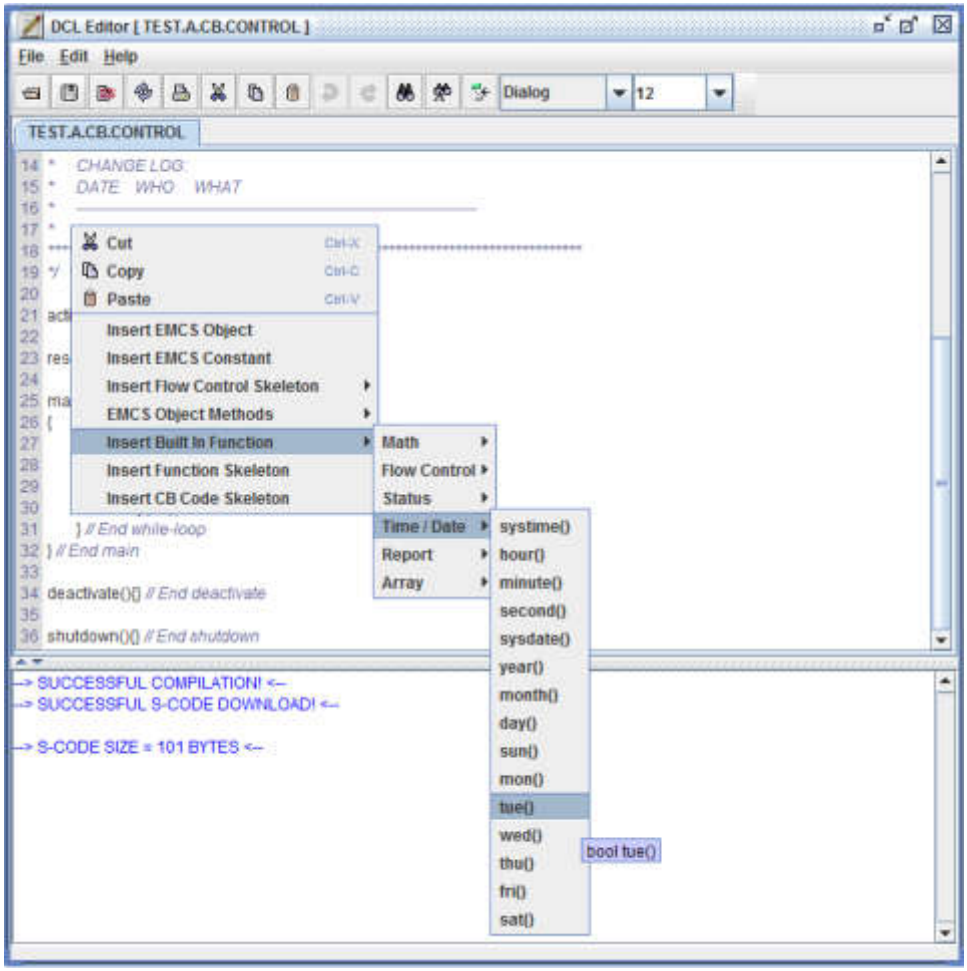


Figure 48 - Built-in functions in the DCL Editor Window

- Math
  - abs()
  - max()
  - min()
  - ave()
  - ln()
- Flow Control
  - delay()
- Status
  - getIOErr()
  - getStatus()
  - setStatus()
- Time/Date
  - systime()
  - hour()
  - minute()

- second()
- sysdate()
- year()
- month()
- day()
- sun()
- mon()
- tue()
- wed()
- thu()
- fri()
- sat()

- Report
  - report()
- Array
  - arraySize()

4.11.3.7 Insert Function Skeleton

A template or skeleton of a function will be inserted into a control block in the DCL Editor by selecting to insert a function skeleton from the pop-up menu.

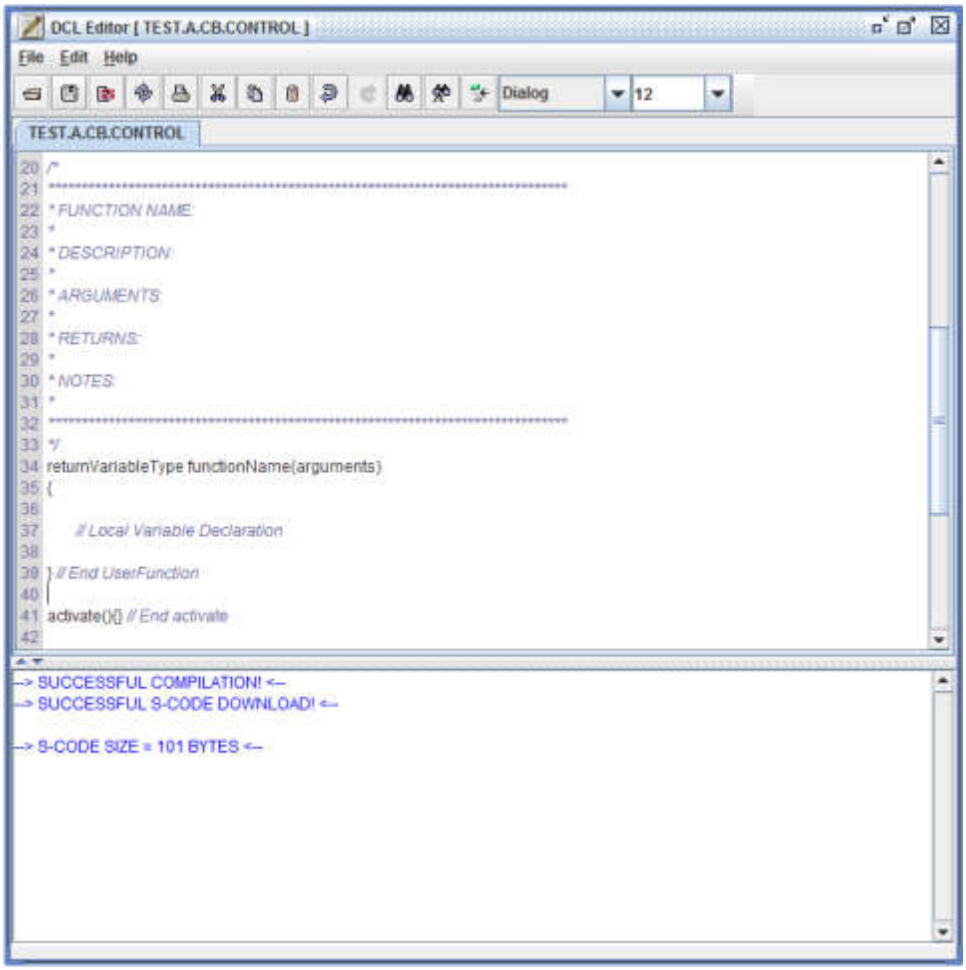


Figure 49 - Function Skeleton in the DCL Editor Window

4.11.3.8 Insert Control Block Code Skeleton

Insert a template or skeleton of an entire control block into a control block in the DCL Editor by selecting to insert a control block code skeleton from the pop-up menu.

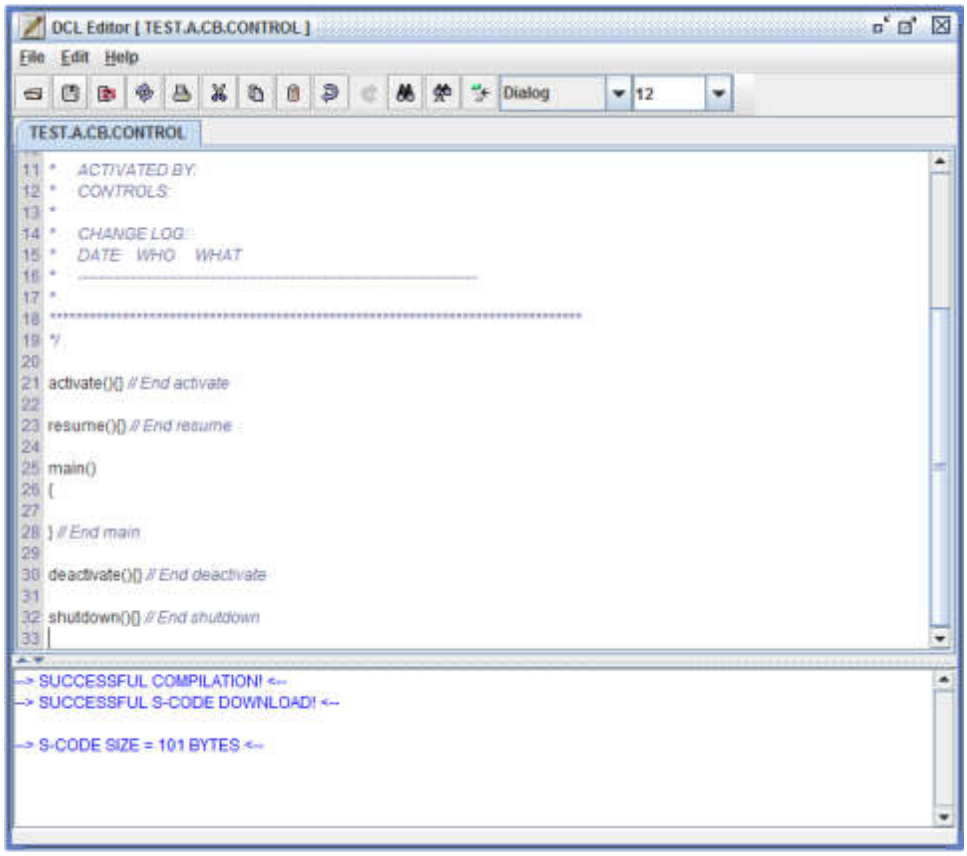


Figure 50 - Control block skeleton in the DCL Editor Window

4.12 Schedule Editor

Digi-SFT provides a schedule object so that users can schedule events. Users define the particular days (weekdays, weekends, and holidays) that events occur. The Schedule Editor is a tool used to modify a schedule object. This section describes the various features provided by the Schedule Editor.

Right clicking on a schedule object in the Grid display and selecting the Edit Control Block/ Schedule menu item opens the Schedule Editor as shown in the figure below.

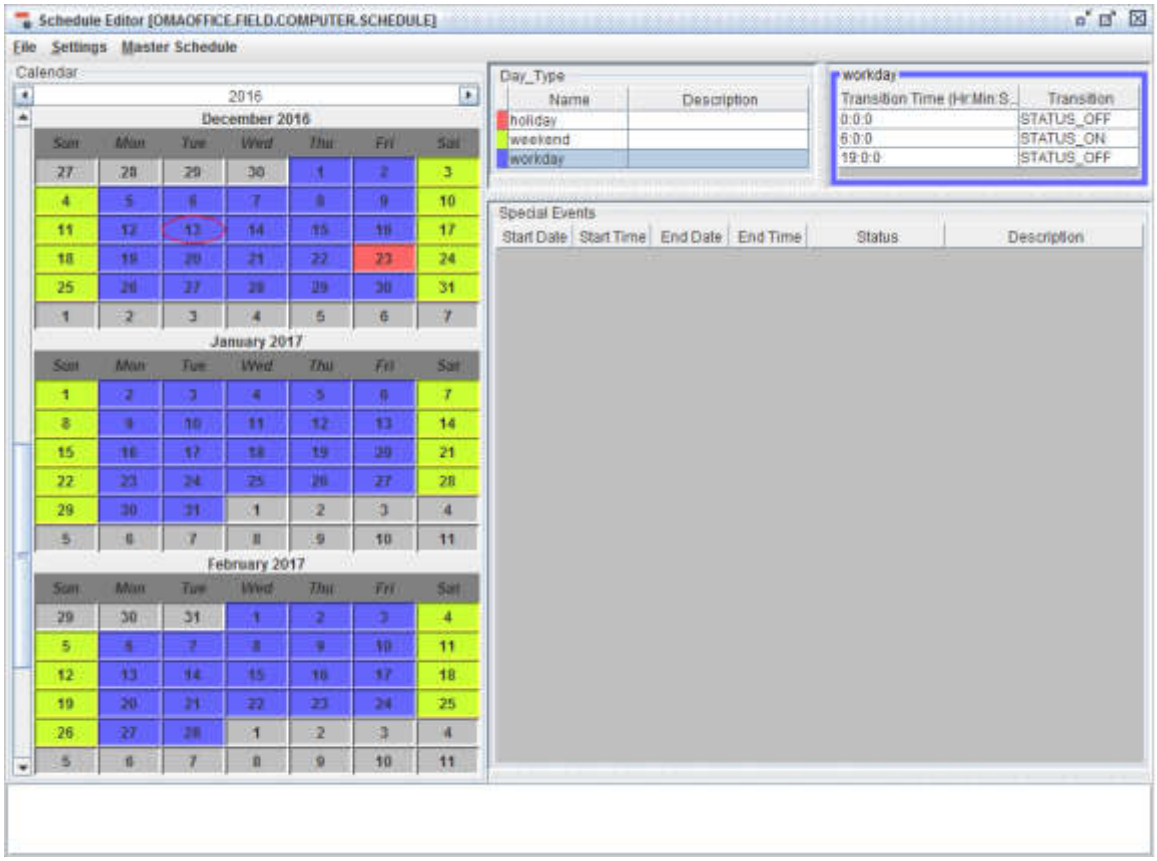


Figure 51 - Schedule Editor Window

4.12.1 Menu

The pull-down menus from the Schedule Editor display the File, Settings, and Master Schedule. The menu commands are described below.

4.12.1.1 File

The file menu provides commands for managing the schedule.

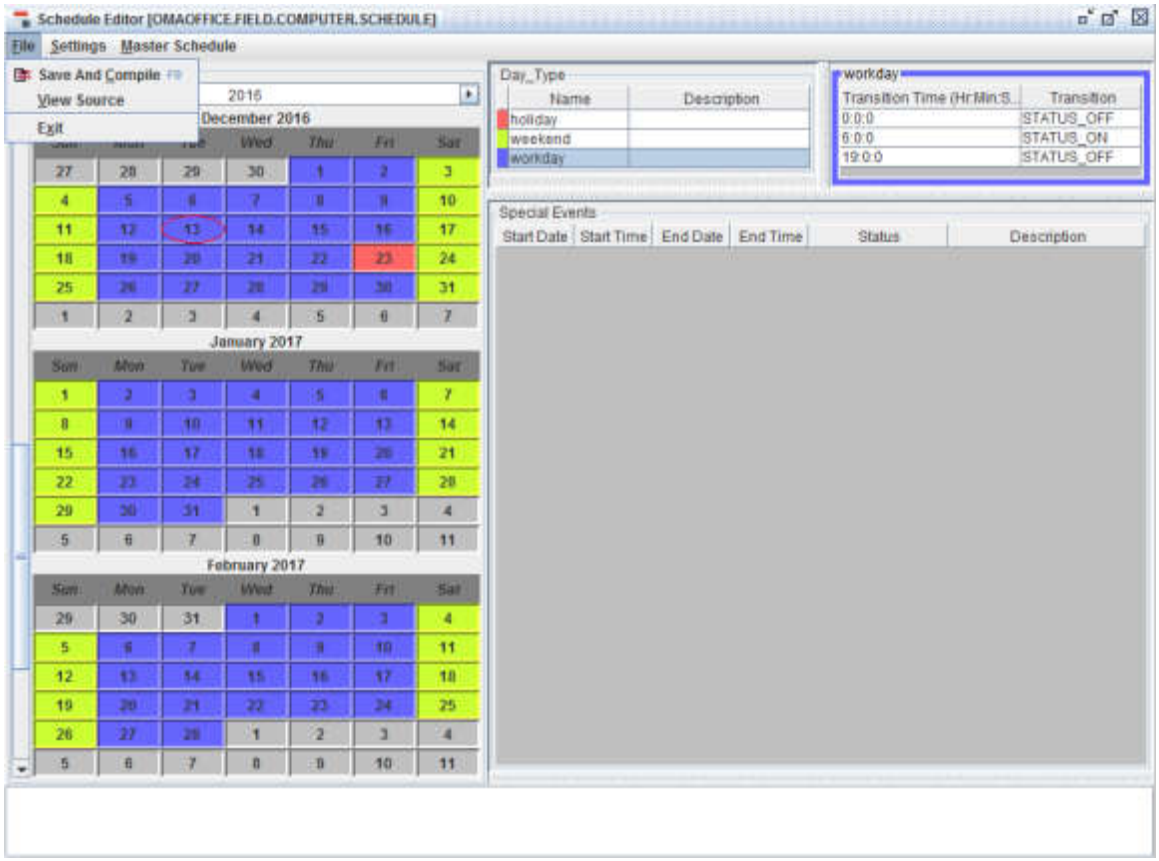


Figure 52 - File menu in the Schedule Editor Window

4.12.1.1.1 Save and Compile

Save and Compile will first save the schedule to the system database. Then the source code is compiled and the compiled code is saved in the database and downloaded to the host controller.

4.12.1.1.2 View Source

A schedule is a type of control block that the DCL automatically generates by the Schedule Editor. When users select to view the source code, the DCL source code for the schedule is shown in a read-only window.

4.12.1.1.3 Exit

This option closes the Schedule Editor.

4.12.1.2 Settings

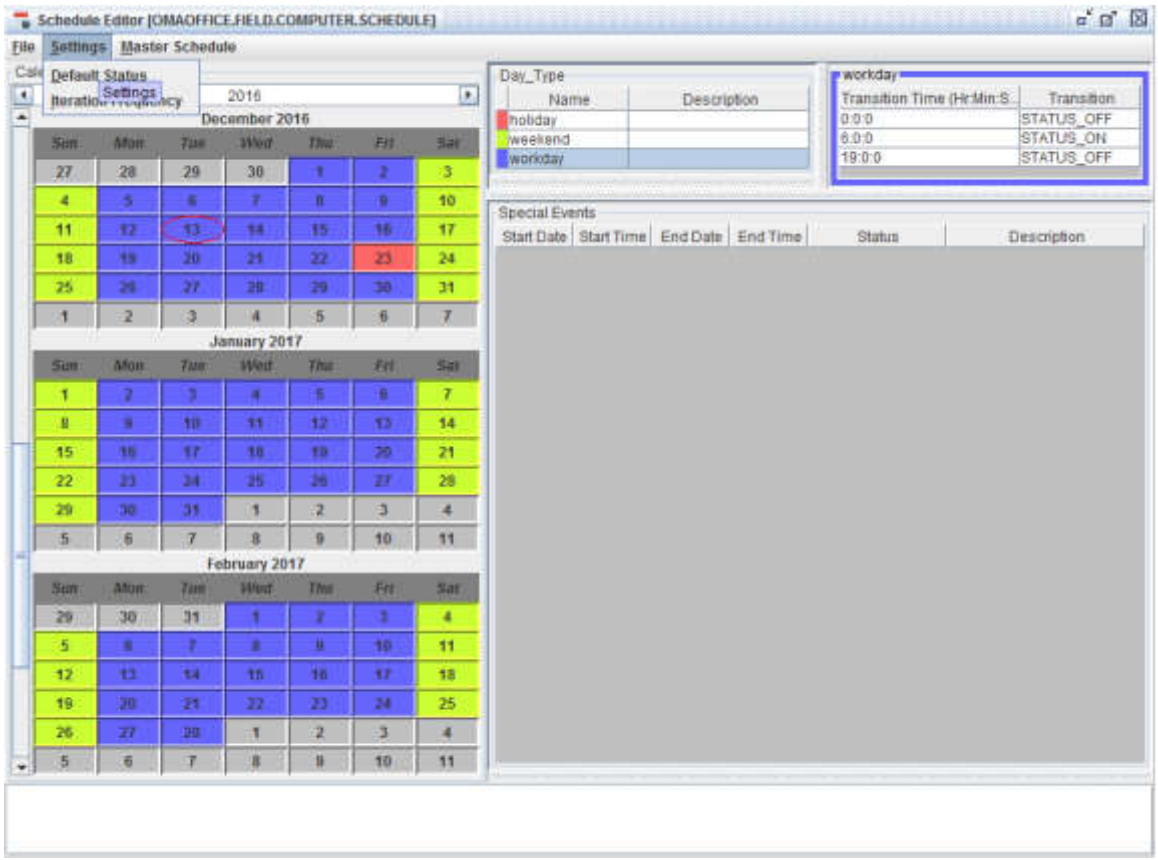


Figure 53 - Settings Menu in the Schedule Editor Window

4.12.1.2.1 Default Status

Without explicitly declaring the default schedule status, a schedule uses the default schedule status of STATUS_OFF. The default status can be changed to any of the following using the Default Status command from the Settings menu.

STATUS_CYCLE_PUMPS
STATUS_DAY
STATUS_DEADBAND
STATUS_DEADBAND_NO_REHEAT
STATUS_DEFROST
STATUS_EXTERNAL_RESET
STATUS_HIGH_SPEED
STATUS_INTERNAL_RESET
STATUS_LOW_SPEED
STATUS_MANUAL
STATUS_MANUAL_OVERRIDE
STATUS_NIGHT
STATUS_NORMAL
STATUS_NORMAL_NO_REHEAT
STATUS_NULL_POINT

STATUS_OCCSENS
STATUS_OFF
STATUS_ON
STATUS_RUN_BOTH_PUMPS
STATUS_RUN_PUMP1
STATUS_RUN_PUMP2
STATUS_STAFF_HOLIDAY
STATUS_STUDENT_HOLIDAY
STATUS_SUMMER_DEADBAND
STATUS_SUMMAR_NORMAL
STATUS_TEMP_OCCUPIED
STATUS_UNOCC
STATUS_WINTER_DEADBAND
STATUS_WINTER_NORMAL
STATUS_WINTER_SHUTDOWN

4.12.1.2.2 Iteration Frequency

The execution of the program can be set to a fixed rate by choosing the iteration frequency as 10, 20, 30, 60, 120, 240, or 300 seconds. This is achieved using the delay() function. See Section 4.8.2 for more information.

4.12.1.3 Master Schedule

4.12.1.3.1 Add or Remove Master Schedule

A master schedule is a schedule used to define the schedule status for all schedules that reference it as the master schedule. Although there can be more than one master schedule in the system, only one master schedule can be referenced per schedule. The master schedule must be defined as a virtual object.

When a new schedule object is created, Add Master Schedule shows in the menu. When you choose to add a master schedule, an EMCS Object Selector Dialog box will prompt you to type in the four part acronym of the mater schedule.

When opening the existing schedule, Remove Master Schedule shows in the menu instead of the Add Master Schedule. Click on it to remove the master schedule.

4.12.1.3.2 Perform Dependency Check

A dependency check will produce a list of schedules that reference the current schedule as a master schedule. If the master schedule is modified, the schedules with a dependency can be compiled to reflect the changes. This is done by first selecting to perform a dependency check. Then select one or more schedules from the list of schedules with a dependency, right-click, and choose to compile the control block. Hold down the <Shift> or <Ctrl> key while selecting objects to select multiple schedules.

4.12.2 Calendar Pop-Up menu

Right click on the Calendar to open the Calendar pop-up menu.

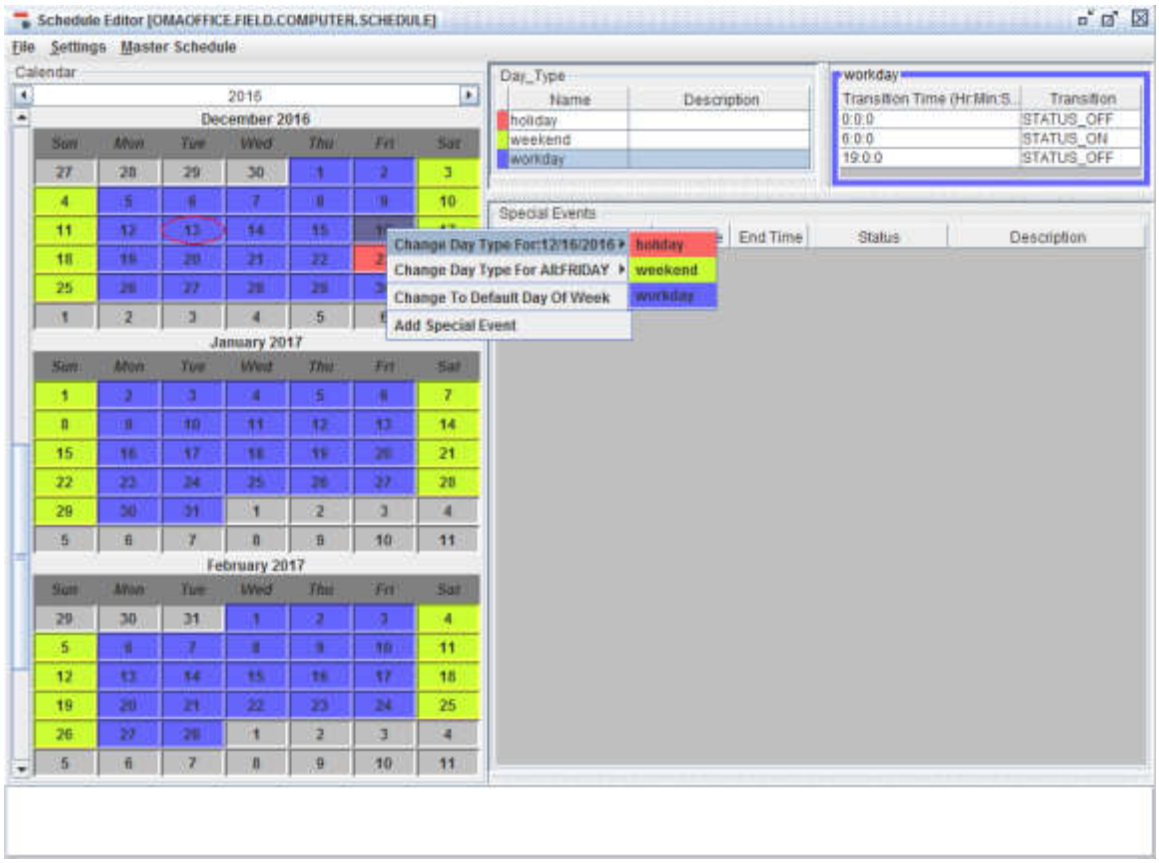


Figure 54 - Calendar Pop-up Menu in the Schedule Editor Window

4.12.2.1 Change Day Type for DATE

The day of a selected date can be changed to Weekend, Holiday, Weekday or other day types as defined in the Day Type window (see Add Day Type in section 4.12.3.1)

4.12.2.2 Change Day type for all

All Day of Week can be changed to any predefined day type.

4.12.2.3 Change to default Day of Week

The default Day of the Week is the actual day of that date.

4.12.2.4 Add Special Event

Click on Add Special Event to add a special event to the special events window on the right as shown in the above figure. To edit the time or day of a special event, double click on the day start time and end time.

4.12.3 Day Type Pop-Up Menu

Right clicking on the “Weekend”, “Holiday”, or “Weekday” field opens the day type pop-up menu.

4.12.3.1 Add Day Type

Using Add Day Type, a new day type will be added to the Day Type. For example, if the user wants to add a Rainday day type, the following procedure should be followed: When the user types Rainday in the pop-up input window, Rainday will be added to the Day Type window. Another day type can then be selected in the calendar.

4.12.3.2 Delete Selected Day Type

Click the delete selected Day Type to delete it.

4.12.3.3 Day Type Transition Pop-Up Menu

Right click on the transition window area to open the pop-up menu.

4.12.3.4 Delete Selected Transition

Click to select the transition time and use the Delete Selected Transition option in the pop-up menu to delete the selection.

4.12.3.5 Add New Transition

Click the transition time window (the gray area) to add a new row for the transition time. Double click the transition time to edit the time.

4.12.4 Status Pane

The status pane displays the current transition status, for example, STATUS_ON, as shown in the following figure. Double-clicking on the selected transition status reveals the status pane containing a drop-down field. The status can then be changed.

4.12.5 Transition Time Pane

The transition time pane displays the current transition time value, for example 0:0:0, as shown in the following figure. Double-clicking on the selected transition time reveals the transition time pane containing hour, minutes and seconds selector fields. The transition time can then be changed.

5 Using the Graphics Editor (Admin Only)

5.1 Introduction

Composer is a graphics editor that allows users to create images and link them with EMCS objects for the purpose of monitoring and controlling the EMCS system from a system illustration. Figure 55 provides an example of a simple graphic of an air handler unit with various EMCS objects, such as analog inputs, analog outputs, and control blocks. Most of these objects are associated with actual hardware with real-time values displayed on the illustration to allow users to control the hardware from the graphics.

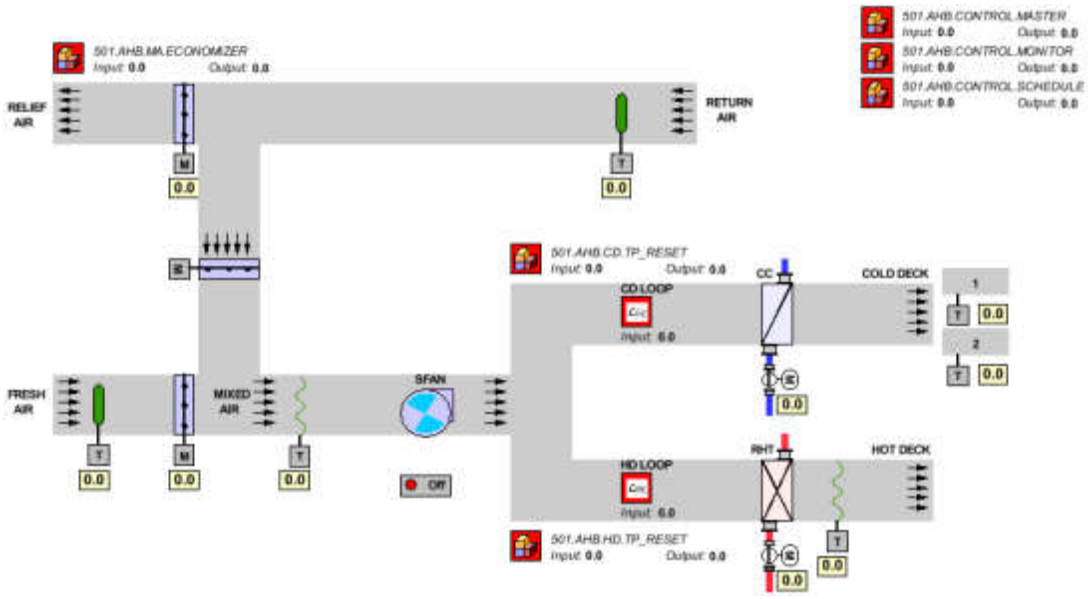


Figure 55 - Sample Graphics

5.1.1 Requirements

To create, and edit graphics using Composer, the path to the graphics server needs to be mapped as a network drive to your computer. Only user as admin will be able to see the button to invoke Composer. For more information see section 5.2.1.

5.1.2 Additional Documentation

More information is available on this topic in the ILOG Composer Manual.

5.2 Setup

Before using the graphics editor, Composer, the area network drive needs to be mapped with the path to the graphics server. Then, the prototype libraries need to be loaded.

5.2.1 Mapping a Network Drive

A network drive needs to be mapped to the graphics server to view, create, and edit using Composer. The server IP address is needed to create a network drive for Digi-SFT. Contact the network system administrator for the IP address.

1. Open “My Computer”
2. Open the “Tools” menu (skip this step in Windows 7)
3. Select “Map Network Drive”
4. Select a Drive, such as “J:”
5. In the Folder field box, type \\<IP ADDRESS>\EMCSGraphics, except replace <IP ADDRESS> with the IP address of the server provided by your network system administrator.
6. Select “Reconnect at login” so that the network drive will be mapped to the computer at every login time.
7. Click “OK” or “Finish”. The new drive should open immediately.

5.2.2 Loading Prototypes

Images and symbols for common HVAC equipment are provided in graphics known as prototypes. The prototypes are supplied so that users can insert them into graphics without having to create their own images.

The first time that Composer starts on the user's computer, the prototypes frame is unoccupied. After the prototype libraries have been opened once, they will load automatically into the prototype frame. To load the prototype libraries for the first time:

1. Select "File" from the Composer pull-down menu.
2. Select "Open".
3. Select the network drive for EMCSGraphics, see section 5.2.1.
4. Select the "prototypes" folder.
5. Select the files with .ivl extension. Use the <Shift> key or <Ctrl> key to select multiple files.
6. Click "Open". The prototypes should be opened in the prototype frame.

5.3 File Management

When managing files with Composer, it is important to know where the files are saved and the naming convention for the files. The Composer files are organized on the graphics server and mapped to a network drive for EMCSGraphics. See section 5.2.1 for more information. The files are named using the four part alpha-numeric acronym assigned to the object in Digi-SFT. Then, the files are organized in directories that have the same name as the first part of the acronym. For example, a graphic object with the acronym "bldg501.Chiller.Pump.Graphic" is named "bldg501.Chiller.Pump.Graphic.ivl". This file is saved in the directory "bldg501".

5.3.1 Open

To open a graphics file in Composer,

1. Select "File" from the Composer pull-down menu.
2. Select "Open".
3. Select the network drive for EMCSGraphics, see section 5.2.1.
4. Select the directory, for example "bldg501".
5. Select the file with an .ivl extension, for example "bldg501.Chiller.Pump.Graphic.ivl".
6. Click "Open". The file should be opened.

5.3.2 Save

To save a graphics file in Composer,

1. Select "File" from the Composer pull-down menu.
2. Select "Save".
3. Select the network drive for EMCSGraphics, see section 5.2.1.
4. Select the directory, for example "bldg501". If the directory does not exist the user must create it using the "Create New Folder" icon. See .
5. Type in the file name and include .ivl, and the file name extension; for example, "bldg501.Chiller.Pump.Graphic.ivl". Composer will not automatically add the file name extension.
6. Click "Save". The file should be saved.

5.4 Prototypes

5.4.1 Connecting a prototype to an object

1. Place a prototype on a new graphic by left clicking the prototype on the prototype library toolbar, then clicking where it will be placed on the graphic.
2. Select the EMCSObjects prototype library.
3. Select the EMCS object to be used.
4. Place it in the graphic (this will be invisible when a user views the graphics, EMCS objects are only visible in Composer).

Figure 56 - EMCSObjects prototype library

5. Right click on the EMCS object (without selecting it, because in that case the popup-menu gets displayed).
6. Enter the EMCS acronym that the EMCS object is to represent.
7. Select the Connect tool (a green arrow connecting a red circle to a white one).
8. Drag a line between the EMCS object and the prototype it will control or be controlled by.
9. Select the properties that will be connected (like the command valuePercent on an AI and rotatePeriod on a fan prototype) from the available prototypes by double clicking on the connecting line.

6 Alarm Manager

The alarm manager allows users to define, view, and acknowledge alarms.

Alarms are typically used to notify users of undesirable events or conditions. Several options for communicating the alarms are provided, such as a visual and audible warning on a computer's desktop, email, or text page. Significant events or conditions that trigger an alarm are recorded as archived data. An alarm must be acknowledged so that alarms cannot be missed or ignored.

Routing alarms to specific users and defining how they are routed are standard features of alarms.

6.1 Defining Alarms

The first step in setting up an alarm is to create the alarm object. Open the grid display, select the grid options you prefer, and search. Then select one or more of the objects from the filtered list of objects displayed in the grid display, right-click, and the pop-up menu will appear. Then choose to create a new object. The object definition window will appear and provides you the ability to edit the settings of an object.

For example, use the acronym MyBldg.Sys_A.SubSys_1.MyAlarm and select "Alarm" as the type of object. See section 3.2 for more information.

Next, the alarm object is referenced in a control block and custom logic is used for triggering the alarm. For example, see the DCL algorithm below. See section 4.2.9.8 for more information.

```
ALARM myAlarm = [MyBldg.Sys_A.SubSys_1.MyAlarm];
.
.
.
main()
```

```
{       .
        .
        if (output > 100)
        {
              myAlarm.trigger(1, "Output > 100.  output = ",output);
        }
        if (output < -100)
        {
              myAlarm.trigger(0, "Output < -100. output = ",output);
        }
    .
    .
}
```

6.2 Viewing Alarms

When an alarm is triggered, the alarm will be automatically routed to the specific users listed as the alarm users. Users are notified with an alarm window. Users may also open the alarm window at any time to view the active alarms. An active alarm is a triggered alarm in which the conditions that caused the triggering still exist. If an alarm has not been acknowledged, notification of the alarm will reoccur at the repeat frequency until the alarm has been acknowledged. If the notification exceeds the number of tries to cascade, the alarm is escalated by cascading or triggering another alarm object.

The alarm window displays a list of all active alarms with the following information:

- The date and time the alarm was first triggered
- The reporter object acronym and control block object that triggered the alarm
- The alarm object acronym
- The alarm string; a message provided when the alarm is triggered
- The number of times the user list has been notified that the alarm has been triggered
- The date and time of the last notification that the alarm was triggered
- Indicates if the alarm has been acknowledged

The alarm window has an option to mute the sound.

The alarm window has the ability to filter alarms or hide alarms based on whether or not they have been acknowledged.

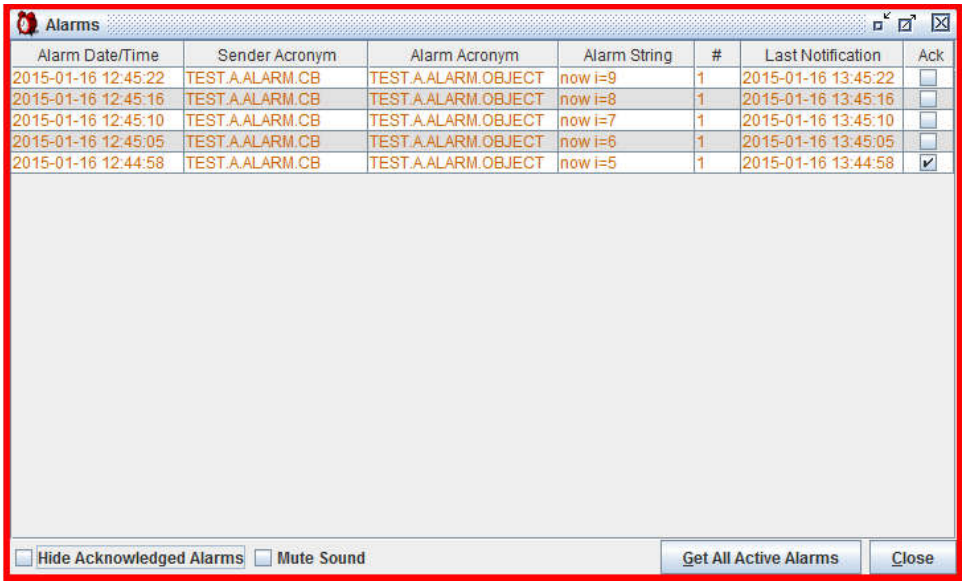


Figure 57 – Alarms window

6.3 Acknowledging Alarms

When an alarm is triggered, the user is automatically notified with an alarm window at the repeat frequency until the alarm has been acknowledged or the conditions that triggered the alarm no longer exist. The alarm window displays a list of all active alarms and provides options to view and acknowledge current alarms.

Acknowledge an alarm by double clicking on a specific alarm in the alarm window and opening an alarm acknowledgement window.

The alarm acknowledgement window displays:

- The alarm object acronym
- The reporter object acronym (the control block object that triggered the alarm).
- The date and time the alarm was first triggered
- The date and time of the last notification that the alarm was triggered
- The number of times the user list has been notified that the alarm has been triggered
- The number of times the alarm was triggered due to a reoccurrence of the condition
- The user ID used to acknowledged the alarm
- The date and time the alarm was acknowledged
- The alarm status
- The alarm string

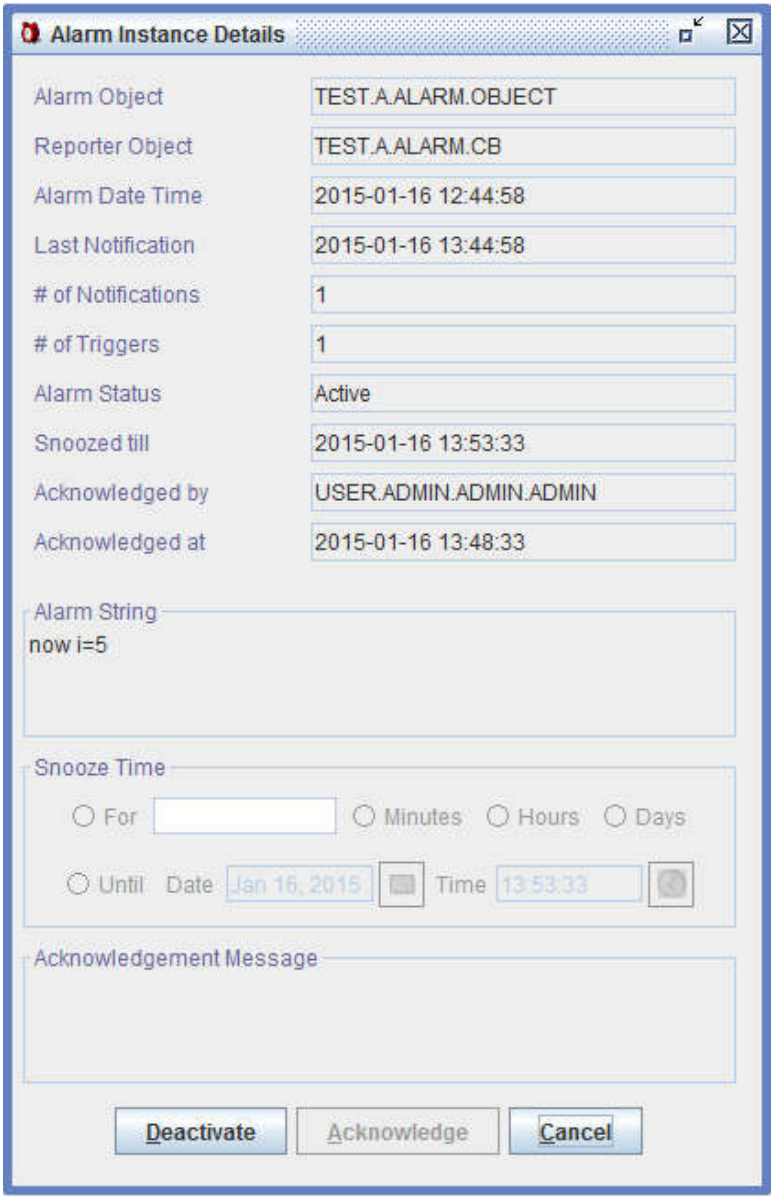


Figure 58 - Alarm Details window

A snooze time needs to be provided when acknowledging the alarm. The alarm does not notify a user during the snooze time when the alarm was acknowledged. If the problem that triggered the alarm still exists and the snooze time has been exceeded, the user notification resumes. The alarm acknowledgement window provides options that allow users to define the length of the snooze time in terms of the

- minutes, hours, or days
- selected start and end time and date

The alarm acknowledgement window includes a field for users to enter a description about the acknowledgement.

6.4 Get Archived Data

An alarm represents a significant event or condition in the EMCS, so the instance is recorded as archived data. To view the archived data for an alarm, open the grid display, select the grid options you prefer, and search. Then select one or more of the alarm objects from the filtered list of objects displayed in the grid display, right-click, and the pop-up menu will appear. Then choose to get the archive data. An archive data window will allow you to select the date range to retrieve the alarm data and choose to get data to load the data in the window. The following information will be presented:

- The date and time the alarm was first triggered
- The reporter object acronym, the control block object that triggered the alarm
- The alarm string (a message provided when the alarm is triggered)
- The number of times the alarm was triggered due to a reoccurrence of the condition
- The number of times the user list has been notified that the alarm has been triggered
- The user ID used to acknowledge the alarm
- The date and time the alarm was acknowledged
- The acronym of the cascade alarm

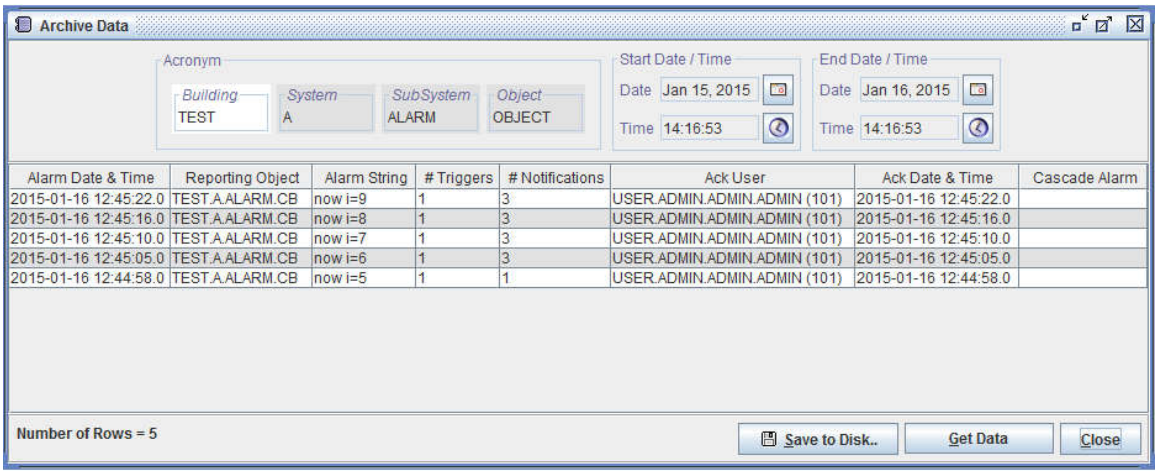


Figure 59 – Alarm Archive Data window

Selecting the "Save to Disk" command on the Archive Data window will save the data to a comma-separated value text file. The file can be opened with a text editor or spreadsheet program.

7 Trend Manager

The Trend Manager provides two methods to plot trended data: Real Time Plot and Trend Plot.

7.1 Trend Plot

The Trend Plot function allows users to graph the data of one or multiple objects. After trend enable gets selected and the trend frequency and trend purge interval are set up in the Field Definition window (See sections 3.4.1.3, 3.4.1.4 and3.4.1.5), the object data is automatically trended.

To display data in the Trend Plot window:

1. Right click the selected object or objects in the Grid Display window. A pop-up menu appears.
2. Click the Trend Plot command in the pop-up menu.

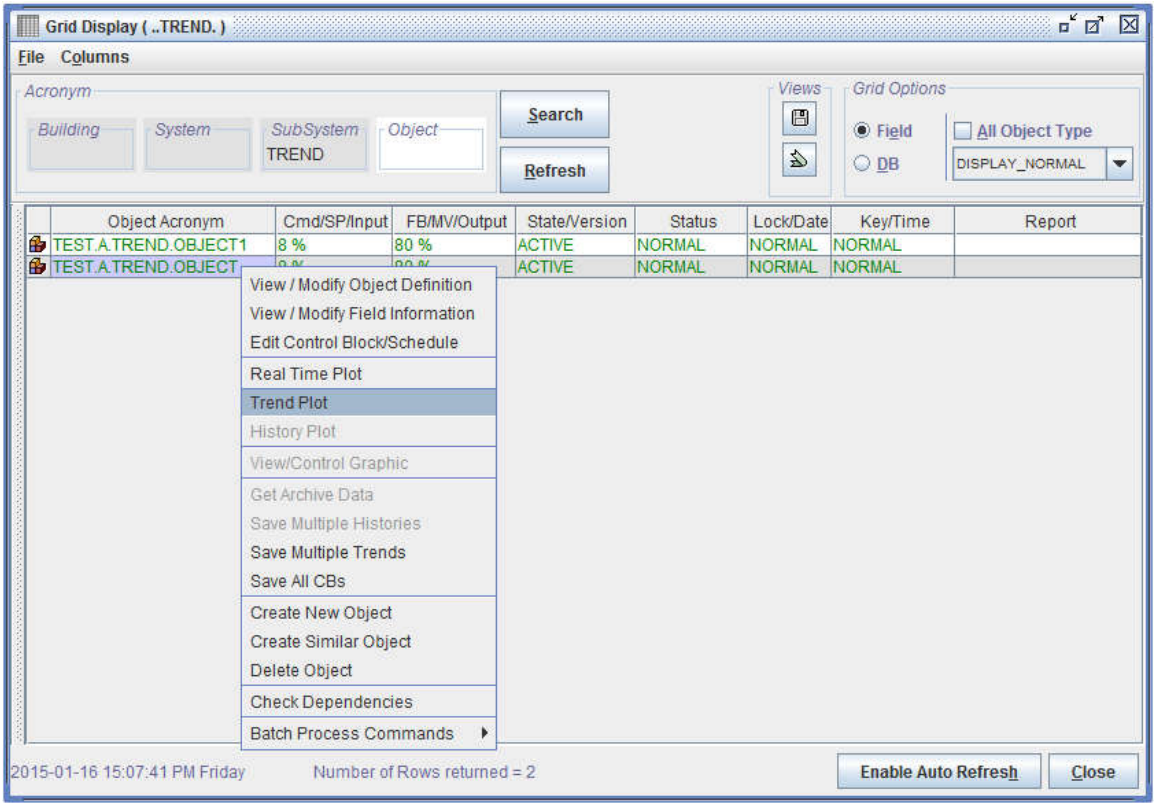


Figure 60 - Trend Plot in the Grid Display Window

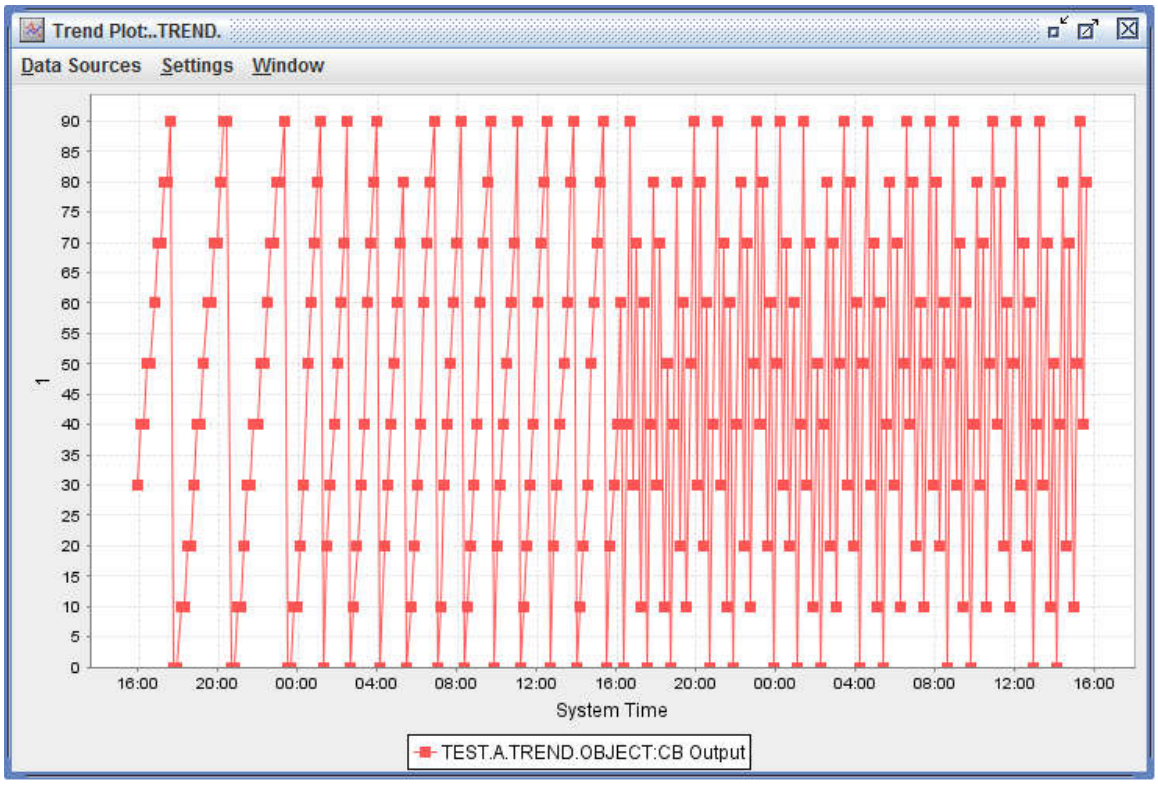


Figure 61 - Trend Plot Window

7.1.1 Data Sources

There are two commands in the Data Source menu:

1. Add Data Source
2. Edit Data Source

7.1.1.1 Add Data Source

Data of more than one object can be plotted.
Click on Add Data Source to open the EMCS Object Selector Dialog window (Figure 62) in order to add the object data to be displayed in the Trend Plot window:

1. Type the object acronym
2. Click the OK button to confirm the selection or the Cancel button to void the selection

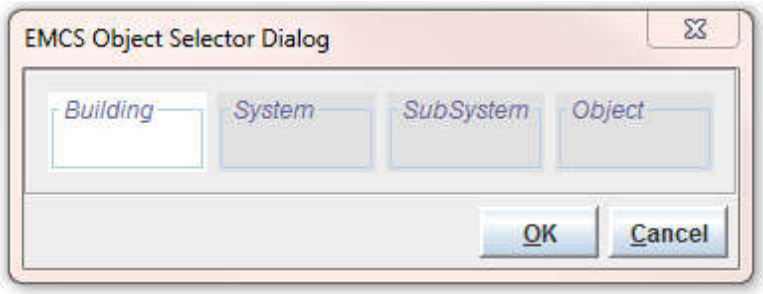


Figure 62 - EMCS Object Selector Dialog

7.1.1.2 Edit Data Source

The Trend Plot window can display the following attributes of an object:

1. Command or input value
2. Feedback or output value

Users can select the attribute or attributes to be plotted. The data of different attributes are plotted in different sub-windows. The Edit Data Source window is used to allocate the data to the selected sub-window. See section 7.1.1.2.1, 7.1.1.2.2 for details.

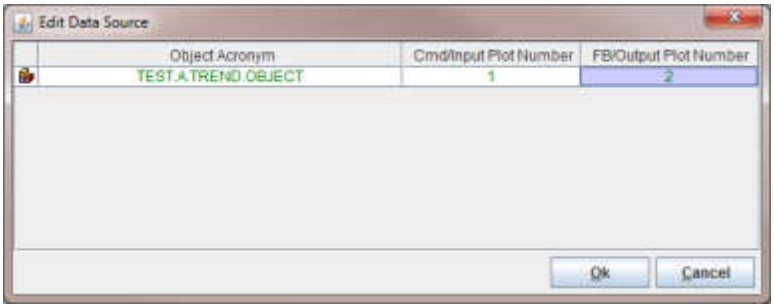


Figure 63 - Edit Data Source Window

7.1.1.2.1 Cmd/Input Plot Number

The Cmd/Input Plot Number determines whether the command or input data of an object as shown in the Object Acronym column is plotted. It also assigns the number or name of the plot window of the plotted attribute.

The plot window number, or window name (section 7.1.2.2) is selected from the scroll-down menu. The plot number is displayed on the left side of the Y-axis of a plot. See Figure 64.

If “None” is selected, the plot for the data of the command or input will not appear.

7.1.1.2.2 FB/Output Plot Number

The FB/Input Plot Number determines whether the feedback or output data of an object as shown in the Object Acronym column is plotted. It also assigns the number or name of the plot window to the plotted attribute.

The plot window number, or window name (section 7.1.2.2) is selected from the scroll-down menu. The plot number is displayed on the left side of the Y-axis of a plot. See Figure 64.

If “None” is selected, the plot for the data of the feedback or output will not appear.

Figure 63 is an example of assigning plot numbers. In this example,

1. The value of the Cmd/Input Plot Number is “1”
2. The value of the FB/Output Plot Number is “2”

Figure 64 demonstrates the plot results.

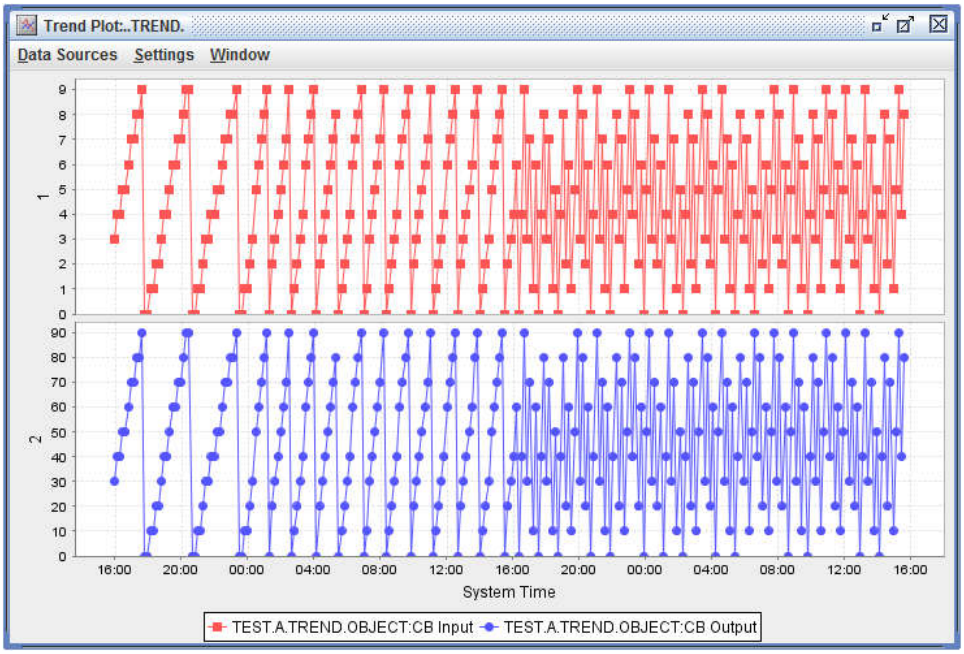


Figure 64 - Multiple Plots in Edit Data Source Window

7.1.2 Settings

Figure 65 shows the Settings menu in the Trend Plot window. With the Settings menu, a user can do the following:

1. Select Show Lines by checking the box for this option.
2. Select Show Shapes by checking the box for this option
3. Define trending start and end time by clicking the Trend Plot Window command. See section 7.1.2.1 for details.
4. Edit individual plot settings including plot names, auto scale options, y-axis bases and ranges (see sections 7.1.2.2, 7.1.2.3, 7.1.2.4 and 7.1.2.5).

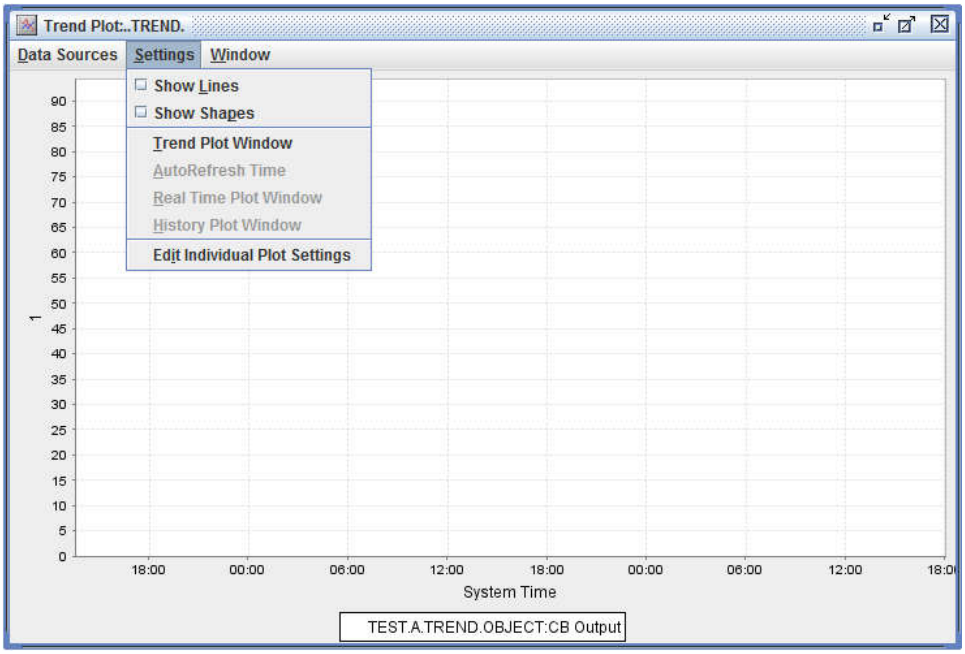


Figure 65 - Settings Menu in the Trend Plot Window

7.1.2.1 Start Time/Date and End Time/Date

To define the start date/time and end date/time

1. Click the Trend Plot Window command as show in Figure 65 to open the Object Trend Plot Window (Figure 66).
2. To set the plot start date
   - Click the calendar icon to open the calendar (Figure 67)
   - Select the plot start date
   - Click Ok to confirm the selection or Cancel to void the selection
3. To set the plot end date, open the end date calendar and follow the same steps as those for setting the plot start date.

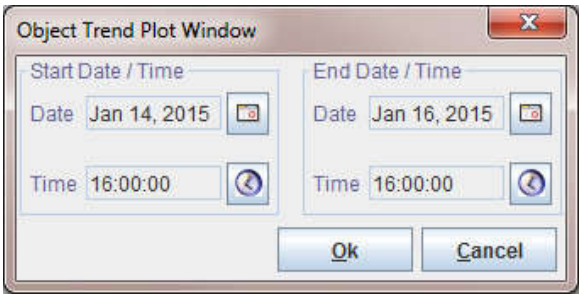


Figure 66 - Object Trend Plot Window

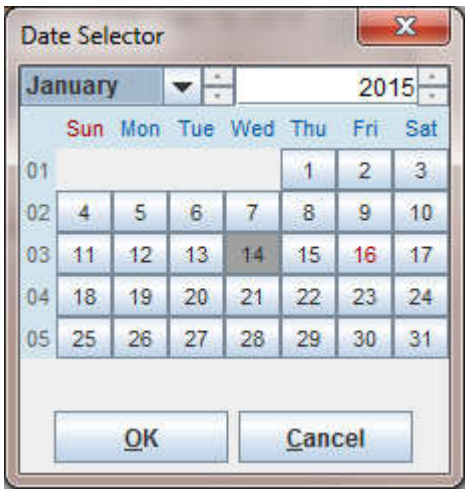


Figure 67 - Start Date Calendar

4. To set the plot start/end time
    - Click the time icon to open the time selector (Figure 68)
    - Select the plot start/end time
    - Click Ok to confirm the selection or Cancel to void the selection

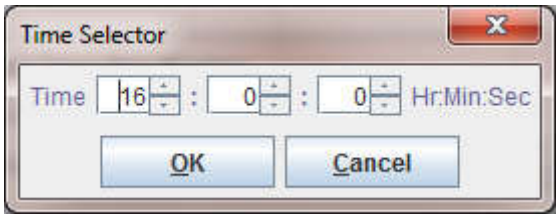


Figure 68 - Time Selector window

7.1.2.2 Plot Name

A trend plot window can plot up to four plots. Each plot can have a user defined name. The name can be assigned in the Plot Data Settings window as shown in Figure 69. The default name of each plot is the same as its plot number.

To give the plot name of a plot:

1. Open the Settings menu.
2. Click Edit Individual Plot Settings (Figure 65) to open the Plot Data Settings window (Figure 69).
3. Click the cell in the Plot Name column in which you want to assign a name
4. Type in the plot name. In Figure 69, the name of Plot 1 is “Testing” which is displayed on the left side of the y-axis.
5. Click a cell other than the one with the just typed plot name
6. Click Ok to save the plot name change or Cancel to void the plot name change

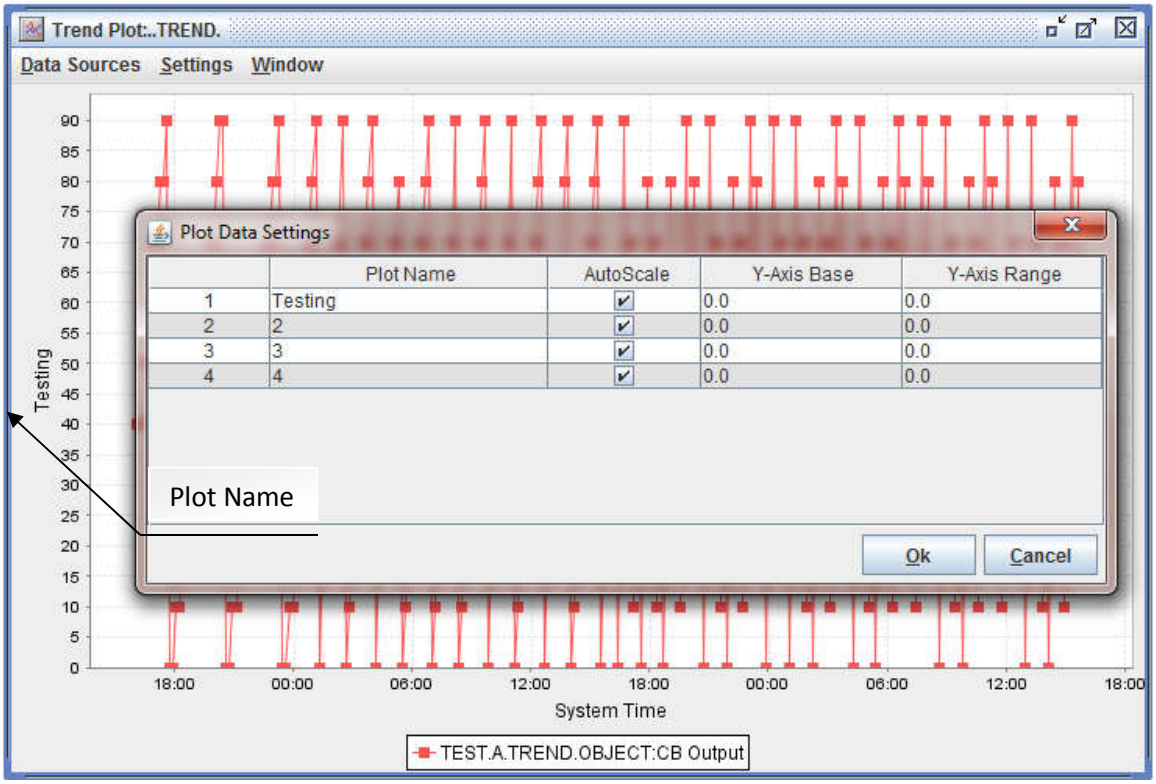


Figure 69 - Plot Data Settings Window

7.1.2.3 Auto Scale

If the AutoScale option box is checked (Figure 69), the y-axis scale will be automatically scaled based on the maximum value of the data. Otherwise, the user needs to define the Y-Axis Base and Y-Axis Range of the plot.

7.1.2.4 Y-Axis Base

If not using AutoScale option for the plot, use Y-Axis Base to determine the origin of the Y-Axis.

To define the origin of the Y-Axis of a plot:

1. Open the Plot Data Setting window (section 7.1.2.2)
2. Click the cell in the Y-Base column
3. Type in the value for the origin of the Y-Axis (Figure 69)
4. Click any other cell
5. Click Ok to save the value change or Cancel to void the change

7.1.2.5 Y-Axis Range

If not using AutoScale option for the plot, use Y-Axis Range to determine the plot range of the Y-Axis.

To define the Y-Axis range of a plot, take similar steps to inputting the Y-Axis Base. See section 7.1.2.4 for details.

7.1.3 Window

The Window menu contains the Exit menu item. If chosen, the Trend Plot window gets closed, releasing resources.

7.1.4 Pop-up Menu in the Trend Plot window

Right-click on any portion of the plot area to open the pop-up menu indicated in Figure 70.

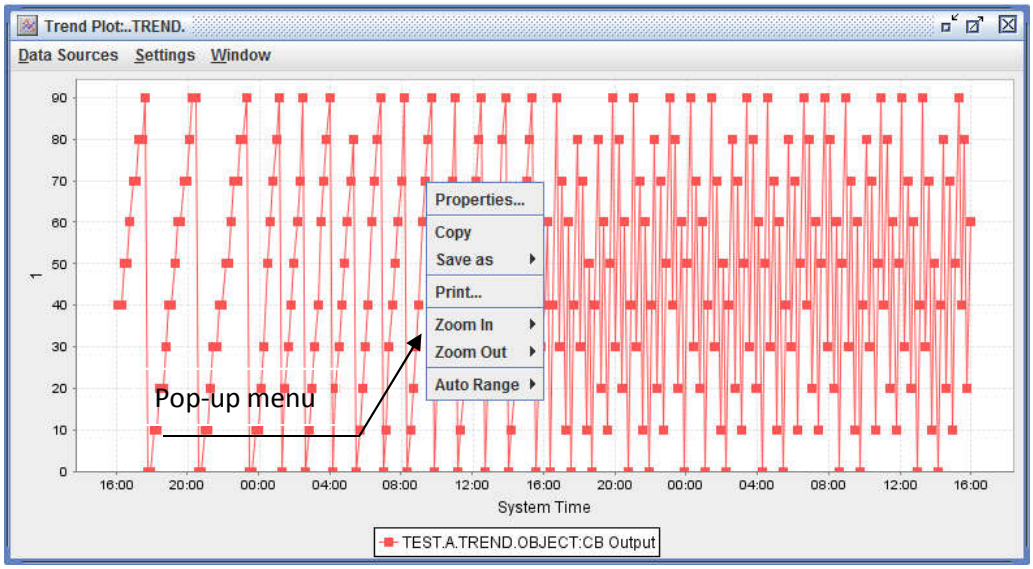


Figure 70 - Pop-up menu in the Trend Plot window

7.1.4.1 Properties

Click Properties in the pop-up menu to open the Chart Properties window as shown in Figure 71. Then you can select the plot properties such as the title, font, color, tick settings, etc.

7.1.4.1.1 Title Tab

Figure 71 shows the function of the Title tab.

1. Check the Show Title option to display the plot title on the top of the plot.
2. Click the Select buttons to define the title font and color.

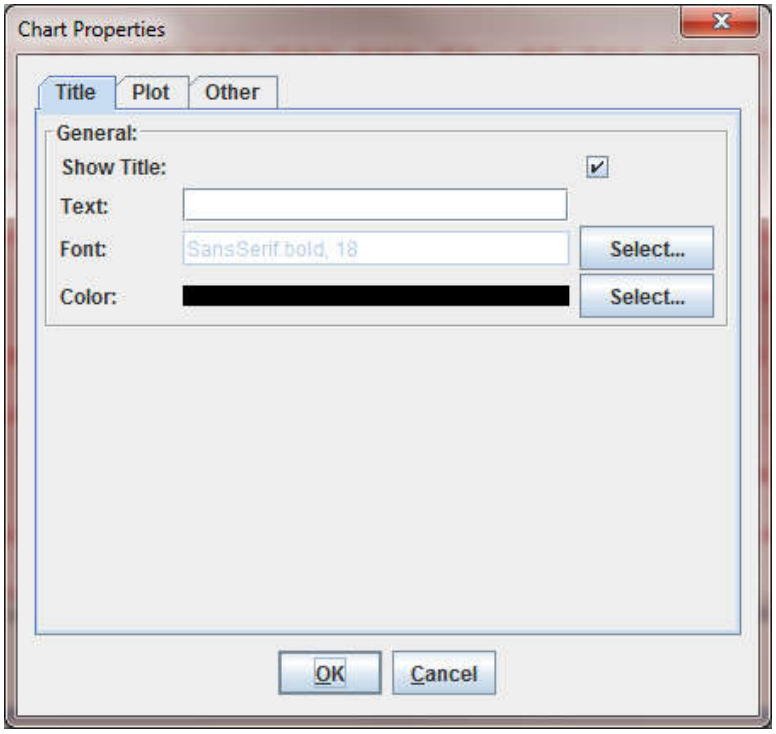


Figure 71 - Title tab in the Chart Properties window

7.1.4.1.2 Plot Tab

Click the Plot tab to specify the Domain Axis (X-Axis) properties displayed in Figure 72.

To set the plot appearance properties, click the Appearance tab. The functions of the Appearance tab are shown in Figure 73.

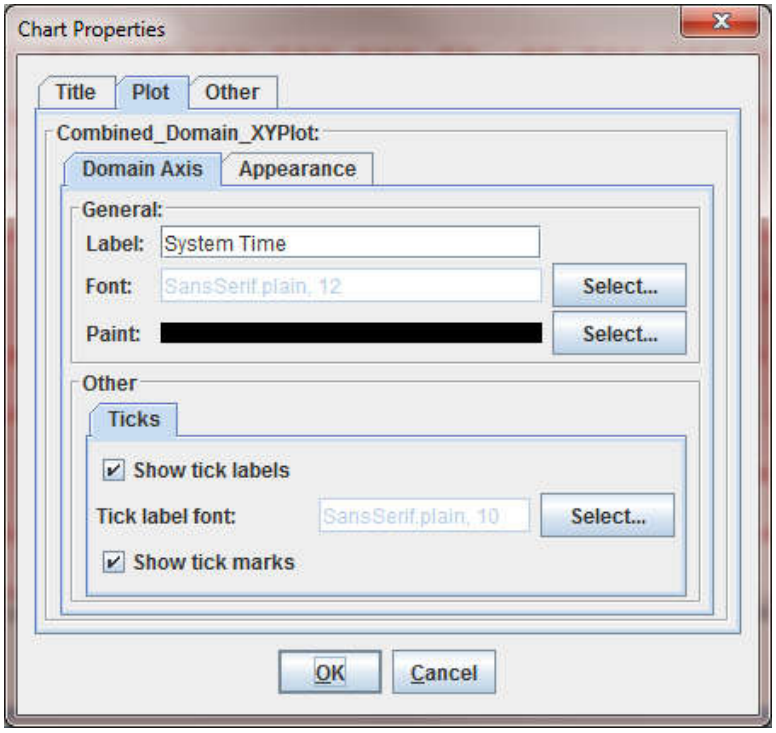


Figure 72 - Plot tab in the Chart Properties window

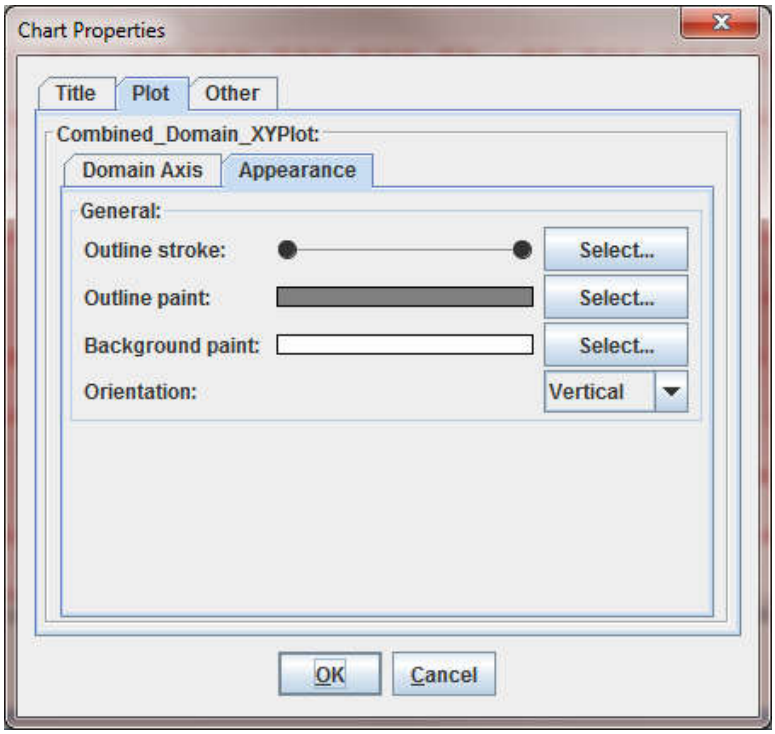


Figure 73 - Appearance tab in the Chart Properties window

7.1.4.1.3 Other Tab

The functions under the Other tab are shown in Figure 74.

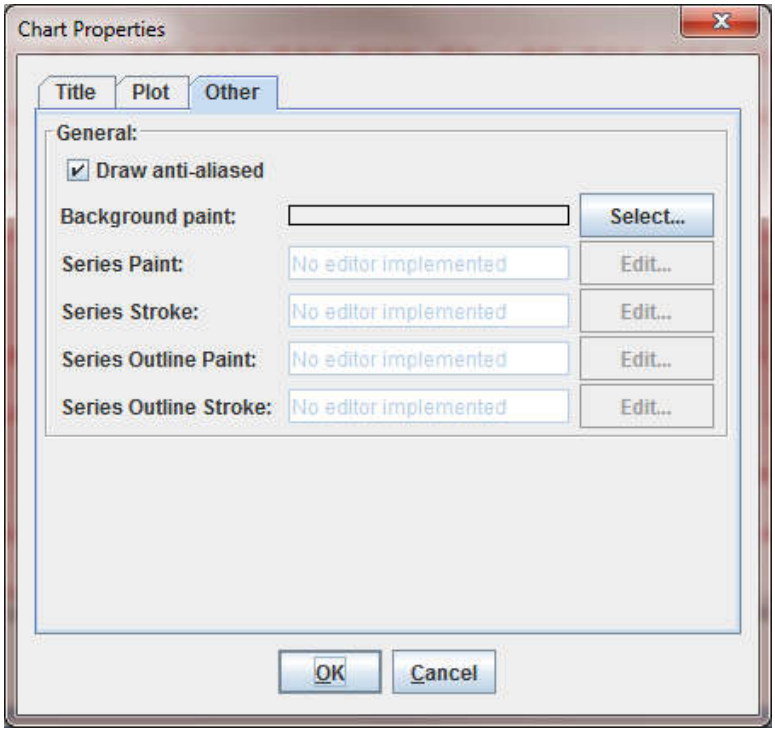


Figure 74 - Other tab in the Chart Properties window

7.1.4.2 Copy

The Copy command places the chart in clipboard memory. It can be pasted then in a user document.

7.1.4.3 Save as

The Save As command is used to save the selected plot to a file. To save the plot to a PNG file:

1. Click Save As PNG
2. Find or create the folder to store the file
3. Type the file name
4. Click Save to save the file or Cancel to not save the file (Figure 75).

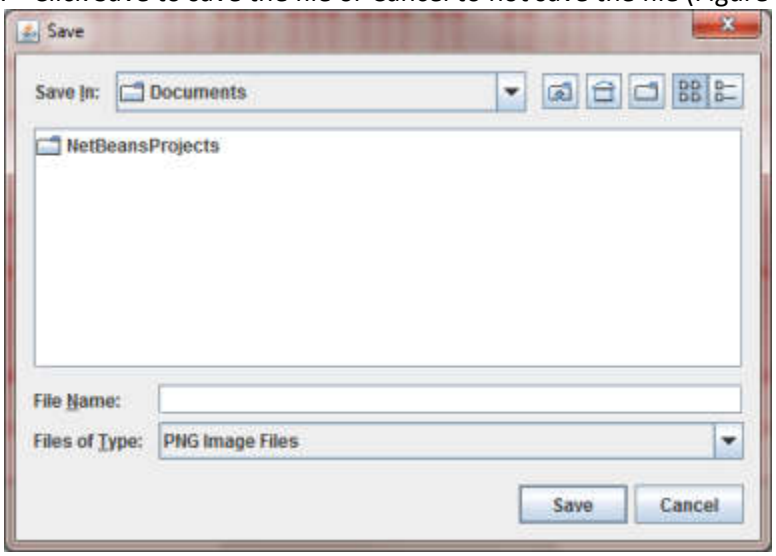


Figure 75 - File Save window

7.1.4.4 Print

The Print command is used to print the plot.

7.1.4.5 Zoom In

Take the following steps to enlarge the plot:

1. Click Zoom In (Figure 76)
2. Click Both Axes to zoom in on the X and Y-axis
3. Or click Domain Axis to zoom in on the X-axis
4. Or click Range Axis to zoom in on the Y-axis

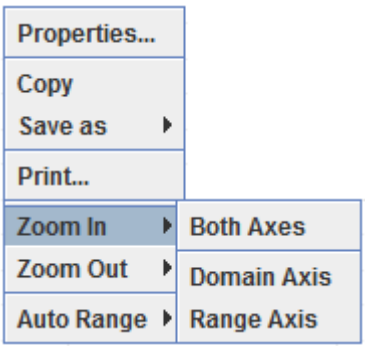


Figure 76 - Zoom In

7.1.4.6 Zoom Out

Take the following steps to zoom out of the plot:

1. Click Zoom Out (Figure 77)
2. Click Both Axes to zoom out of both axes
3. Or click Domain Axis to zoom out of the X-axis
4. Or click Range Axis to zoom out of the Y-axis

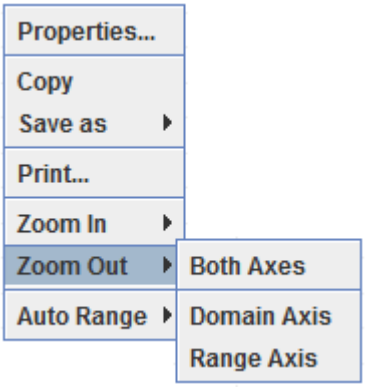


Figure 77 - Zoom Out

7.1.4.7 Auto Range

Take the following steps to automatically set up the range of the plot:

1. Click Auto Range (Figure 78)
2. Click Both Axes to automatically set up the ranges of both axes
3. Or click Domain Axis to automatically set up the ranges of the X-axis
4. Or click Range Axis to automatically set up the ranges of the Y-axis

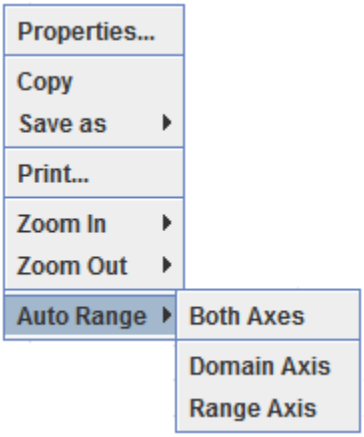


Figure 78 - Auto Range

7.2 Real Time Plot

Real-Time Plot (Figure 79) will graph the data from an object or objects in real time. The start time of the plot is the moment that the Real Time Plot menu item is clicked. A set of old data is discarded after the predetermined time interval has passed and a new set of real time data starts to display.

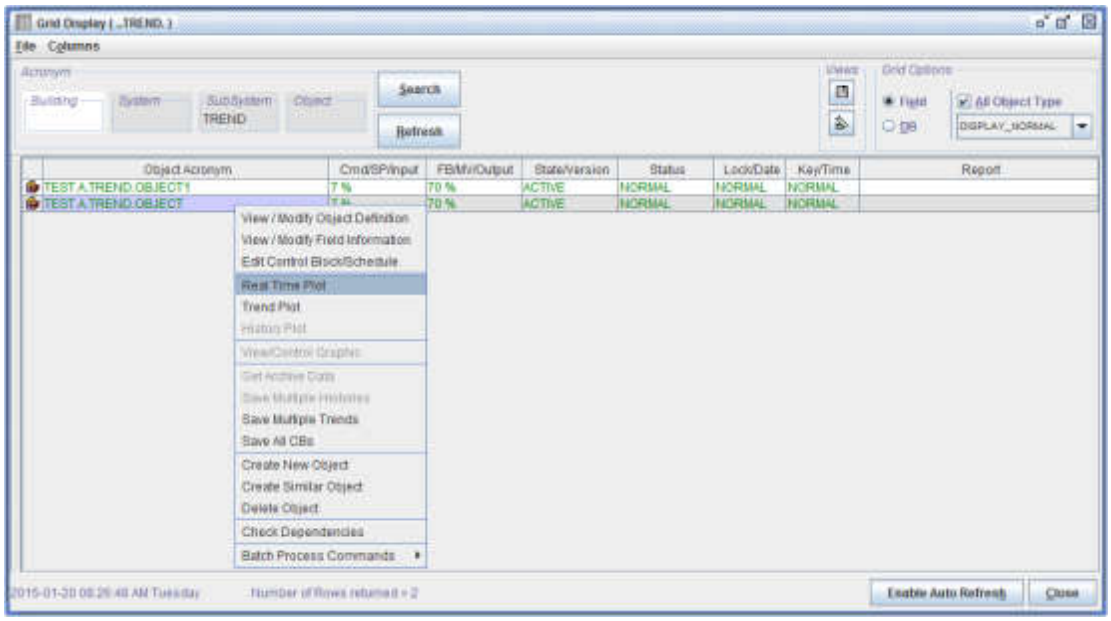


Figure 79 - Real Time Plot

7.2.1 Data Sources

There are two commands in the Data Source menu:

1. Add Data Source
2. Edit Data Source

7.2.1.1 Add Data Source

Data of more than one object can be plotted.

Click on Add Data Source, an EMCS Object Select Dialog window pop up (Figure 62). To add the object data to be displayed in the Real Time Plot window:

1. Type the object acronym
2. Click OK button to confirm the selection, or Cancel button to void the selection

7.2.1.2 Edit Data Source

The Real Time Plot window can display the following attributes of an object:

1. Command or input value
2. Feedback or output value

Users can select the attribute or attributes to be plotted. The data of different attributes are plotted in different sub-windows. The Edit Data Source window is used to allocate the data to the selected sub-window. See section 7.1.2.1, 7.1.2.2 for details.

7.2.1.2.1 Cmd/Input Plot Number

The Cmd/Input Plot Number determines whether the command or input data of an object as shown in the Object Acronym column is plotted. It also assigns the number or name of the plot window of the plotted attribute.

The plot window number, or window name (section 7.2.2.3) is selected from the scroll-down menu. The plot number is displayed on the left side of the Y-axis of a plot. See Figure 64.

If “None” is selected, the plot for the data of the command or input will not appear.

7.2.1.2.2 FB/Output Plot Number

The FB/Input Plot Number determines whether the feedback or output data of an object as shown in the Object Acronym column are plotted. It also assigns the number or name of the plot window of the plotted attribute.

The plot window number, or window name (section 7.2.2.3) is selected from the scroll-down menu. The plot number is displayed on the left side of the Y-axis of a plot. See Figure 64.

If “None” is selected, the plot for the data of the feedback or output will not appear.

7.2.2 Settings

Figure 80 shows the Settings menu in the Real Time Plot window. In the Settings menu, a user can do the following:

1. Select Show Lines by checking the box for this option
2. Select Show Shapes by checking the box for this option
3. Define the trending interval of real time data displayed in the window by clicking AutoRefresh Time to open the input window. See section 7.2.2.1 for details.
4. Define the time interval needed to discard the old data and start to plot the new data by clicking Real Time Plot Window and opening an input window. See section 7.2.2.2
5. Edit individual plot settings including plot names, auto scale options, y-axis bases, and ranges (Sections 7.2.2.3, 7.2.2.4, 7.2.2.5 and 7.2.2.6).

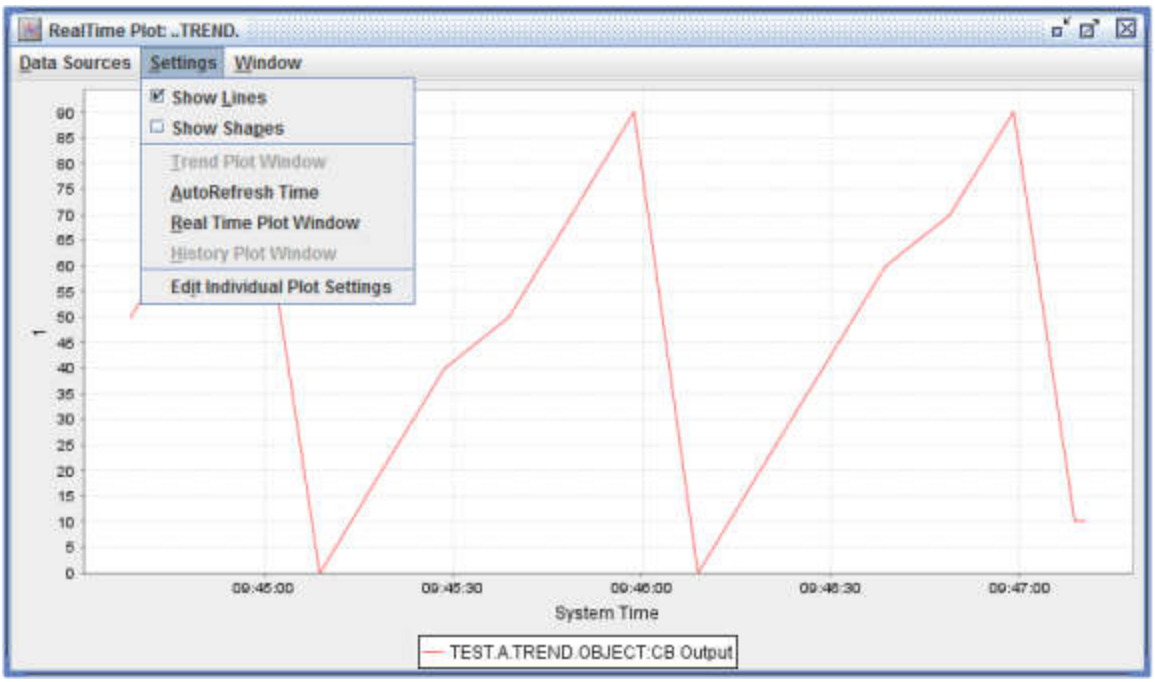


Figure 80 - Settings menu in the Real Time Plot window

7.2.2.1 Auto Refresh Time

The Auto Refresh Time command is used to define the next moment to refresh the value of the object.

To define the Auto Refresh Time:

1. Select an object or several objects for the real time plot in the Grid Display window
2. Right click the selected object or objects
3. Click Real Time Plot
4. Click Settings
5. Click Auto Refresh Time to open the AutoRefresh Time window (Figure 81)
6. Input the time interval in seconds and click OK button

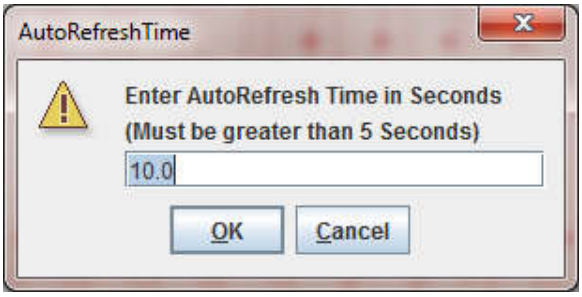


Figure 81 - AutoRefresh Time window

7.2.2.2 Real Time Plot Window Data Purge Time

Use the Real Time Plot Window to define the data purge time in the real time plot window.

To define the data purge time:

1. Select an object or several objects for the real time plot in the Grid Display window
2. Right click the selected object or objects
3. Click Real Time Plot
4. Click Settings
5. Click Real Time Plot Window (Figure 82)
6. Input the time interval in minutes and click OK button

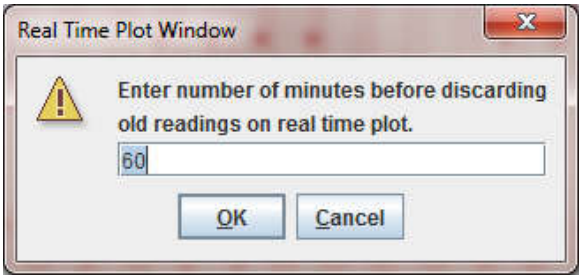


Figure 82 -Real Time Plot Window

7.2.2.3 Plot Name

A Real Time Plot window can plot up to four plots. Each plot can have a user defined name. The name can be assigned in the Plot Data Settings window as shown in Figure 69. The default name of each plot is the same as its plot number.

To give the plot name of a plot:

1. Open the Settings menu.
2. Click Edit Individual Plot Settings (Figure 65) to open the Plot Data Settings window (Figure 69).
3. Click the cell in the Plot Name column which you want to assign a name
4. Type in the plot name. In Figure 69, the name of Plot 1 is “Testing” which is displayed on the left side of the y-axis.
5. Click a cell other than the one with the just typed plot name
6. Click Ok to save the plot name change or Cancel to void the plot name change

7.2.2.4 Auto Scale

If the AutoScale option box is checked (Figure 69), the y-axis scale will be automatically scaled based on the maximum value of the data. Otherwise, the user needs to define the Y-Axis Base and Y-Axis Range of the plot.

7.2.2.5 Y-Axis Base

If not using AutoScale option for the plot, use Y-Axis Base to determine the origin of the Y-Axis.

To define the origin of the Y-Axis of a plot:

1. Open the Plot Data Setting window (section 7.2.2.3)
2. Click the cell in the Y-Base column
3. Type in the value for the origin of the Y-Axis (Figure 69)
4. Click any other cell
5. Click Ok to save the value change or Cancel to void the change

7.2.2.6 Y-Axis Range

If not using the AutoScale option for the plot, use the Y-Axis Range to determine the plot range of the Y-Axis.

To define the Y-Axis range of a plot, take similar steps to input the Y-Axis Base. See section 7.2.2.5 for details.

7.2.3 Window

The Window menu contains the Exit menu item. If chosen, the Real Time Plot window gets closed, releasing resources.

7.3 History Plot

The History Plot function allows users to graph data from a history object or multiple history objects.

To display data in the History Plot window:

1. Right click the selected history object or history objects in the Grid Display window. A pop-up menu appears.
2. Click the History Plot command in the pop-up menu (Figure 83).
3. Fill in Date/Time information in History Date/Time Selector window (Figure 84) and click Ok button

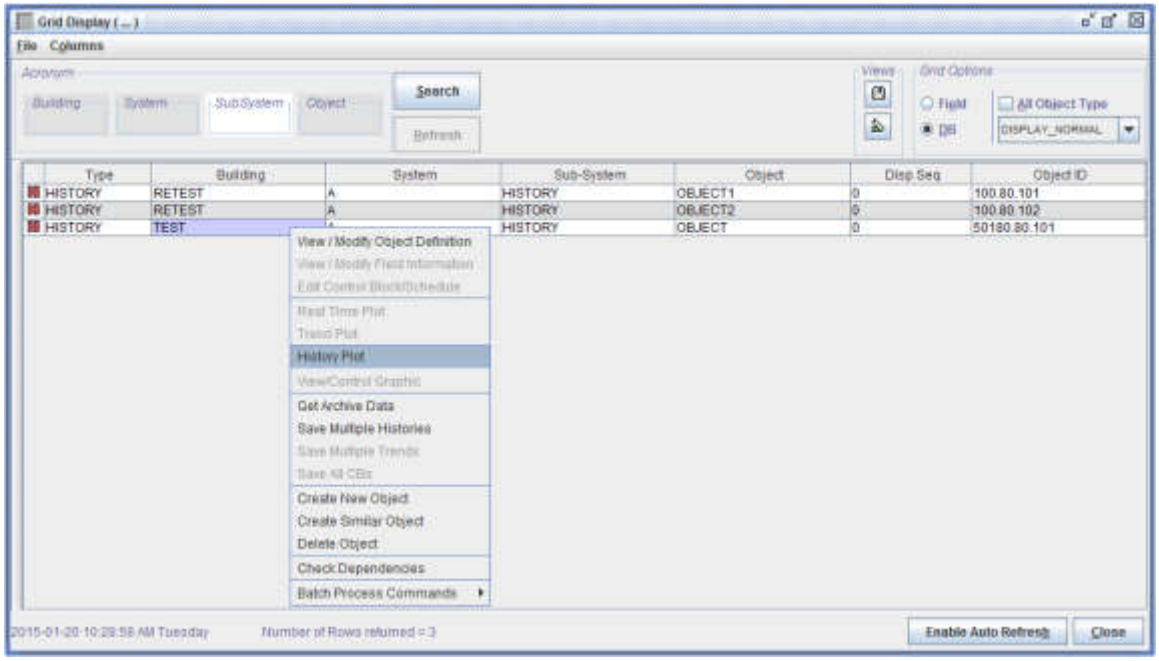


Figure 83- History Plot in the Grid Display window

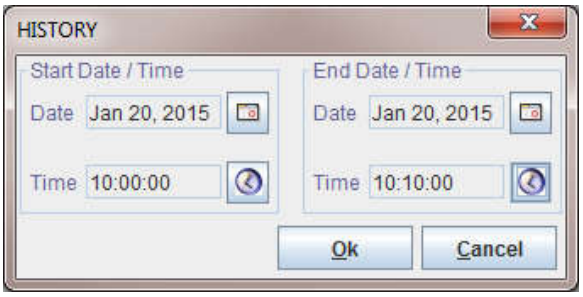


Figure 84: History Date/Time Selector window

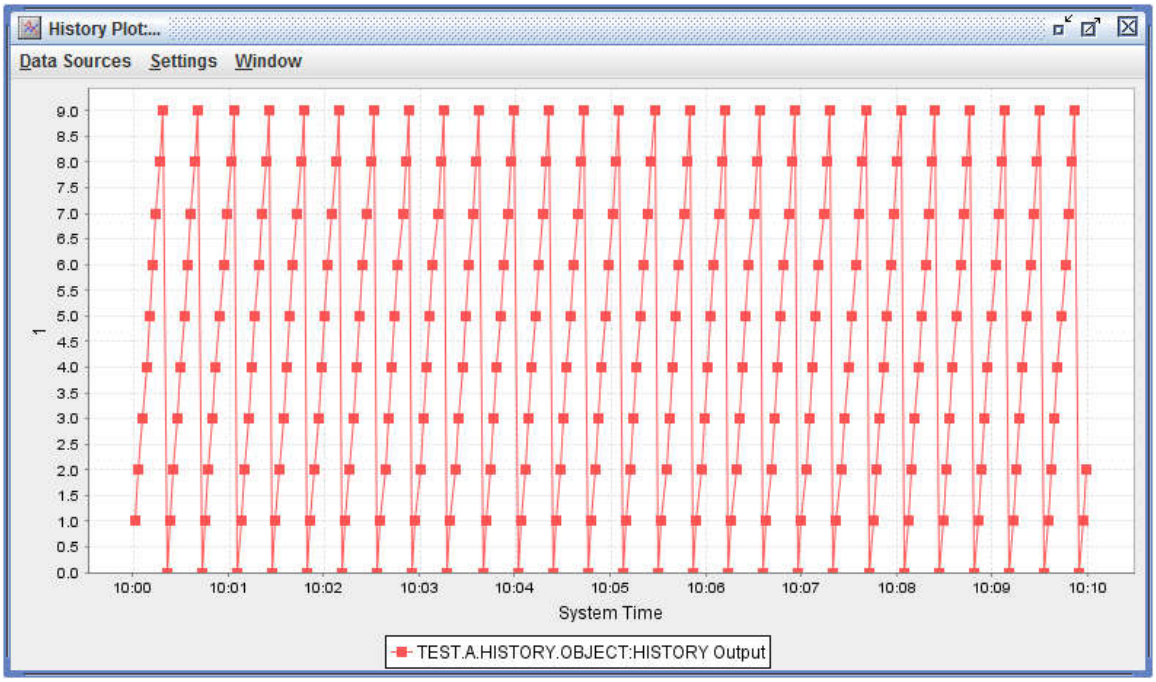


Figure 85- History Plot window

7.3.1 Data Sources

There are two commands in the Data Source menu:

1. Add Data Source
2. Edit Data Source

7.3.1.1 Add Data Source

Data of more than one object can be plotted.
Click on Add Data Source to open the EMCS Object Selector Dialog window (Figure 86).
Add the object data to be displayed in the History Plot window:

1. Type the object acronym
2. Click the OK button to confirm the selection, or the Cancel button to void the selection

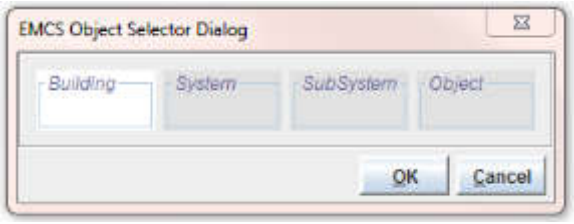


Figure 86 - EMCS Object Selector Dialog

7.3.1.2 Edit Data Source

The History Plot window can display the following attributes of an object:

1. Command or input value
2. Feedback or output value

7.3.1.2.1 Cmd/Input Plot Number

The Cmd/Input Plot Number determines whether the command or input data of an object as shown in the Object Acronym column is plotted. It also assigns the number or name of the plot window if the attribute will be plotted.

If “None” is selected, the plot for the data of the command or input will not appear.

7.3.1.2.2 FB/Output Plot Number

The FB/Input Plot Number determines whether the feedback or output data of an object as shown in the Object Acronym column are plotted. It also assigns the number or name of the plot window if the attribute will be plotted.

If “None” is selected, the plot for the data of the feedback or output will not appear.

7.3.2 Settings

Figure 87 shows the Settings menu in the History Plot window. Using the Settings menu, a user can do the following:

1. Select Show Lines by checking the box for this option.
2. Select Show Shapes by checking the box for this option
3. Define history plot start and end date/time settings by clicking the History Plot Window command.
4. Edit individual plot settings including plot names, auto scale options, y-axis bases and ranges.

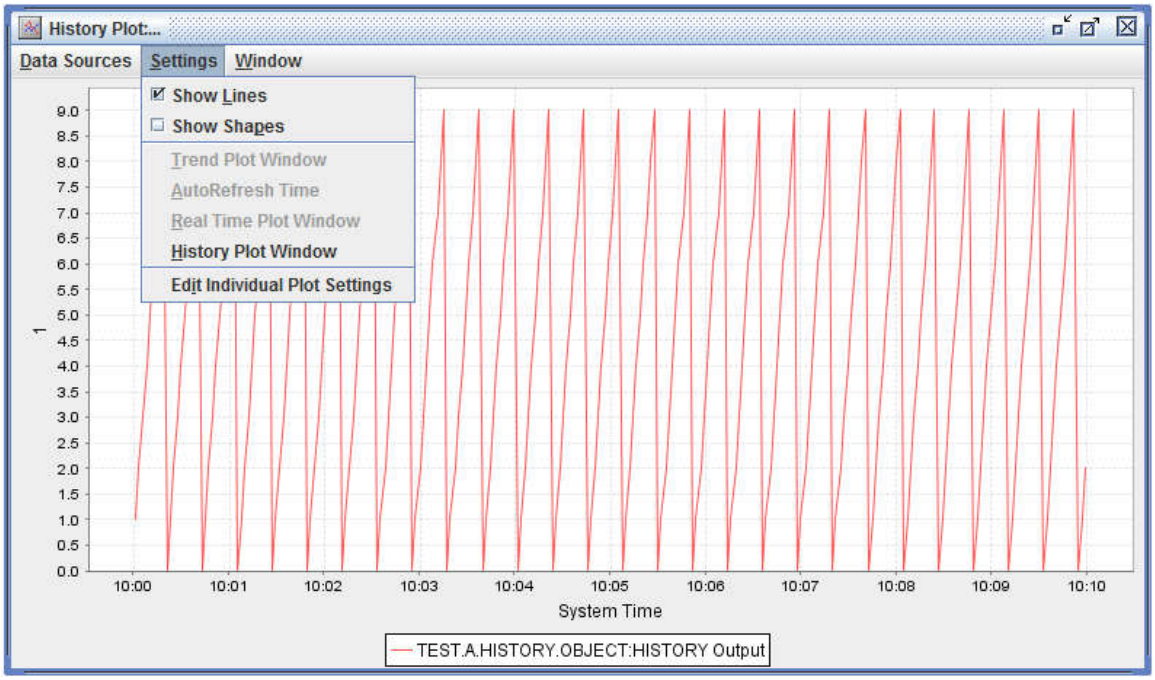


Figure 87 - Settings menu in the History Plot window

7.3.2.1 Start Time/Date and End Time/Date

To define the start date/time and end date/time

1. Click the History Plot Window command as show in Figure 87 to open the Object History Plot Window (Figure 88).
2. To set the plot start date
   - Click the calendar icon to open the calendar (Figure 89)
   - Select the plot start date
   - Click Ok to confirm the selection or Cancel to void the selection
3. To set the plot end date, open the end date calendar and follow the same steps as those for setting the plot start date.

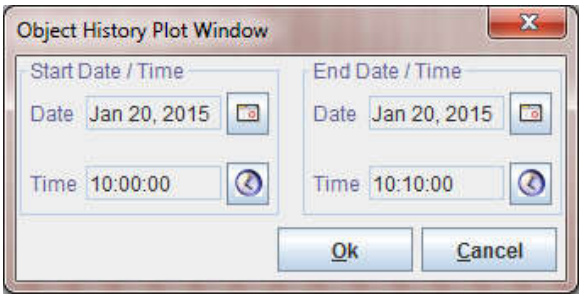


Figure 88 - Object History Plot Window

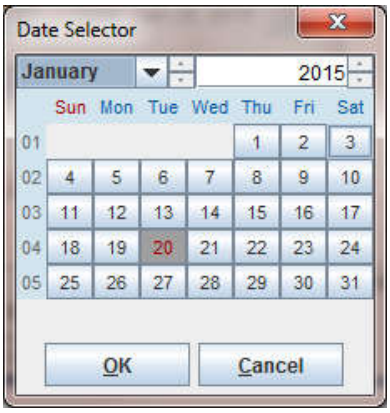


Figure 89 - Start Date calendar

4. To set the plot start/end time
    - Click the time icon to open the time selector (Figure 90)
    - Select the plot start/end time
    - Click Ok to confirm the selection or Cancel to void the selection

Figure 90 - Time Selector window

7.3.2.2 Plot Data Settings

Plot Data Settings functionality is similar to the one described for Trend Plot in sections 7.1.2.2, 7.1.2.3, 7.1.2.4 and 7.1.2.5

7.3.3 Window

The Window menu contains the Exit menu item. If chosen, the History Plot window gets closed, releasing resources.

8 History

After a History object is created (section 3.9), the archived data can be acquired using the Get Archive Data command under the pop-up menu of the Grid Display, or by using the toolbar. The data can be viewed in the Archive Data window and saved as a file.

8.1 Get Archive Data

To get the archive data from the Grid Display window:

1. Highlight the History object
2. Right click the selected object
3. Click Get Archive Data from the pop-up menu to open the Archive Data window as shown in Figure 91
4. Click the Start date/time icons to set the start date and time of the historic data
5. Click the End date/time icons to set the end date and time of the historic data

6. Click Get Data at the bottom to display the archive data
7. Click Save to Disk to save the data to a comma-separated value text file. The file can be opened with a text editor or spreadsheet program.

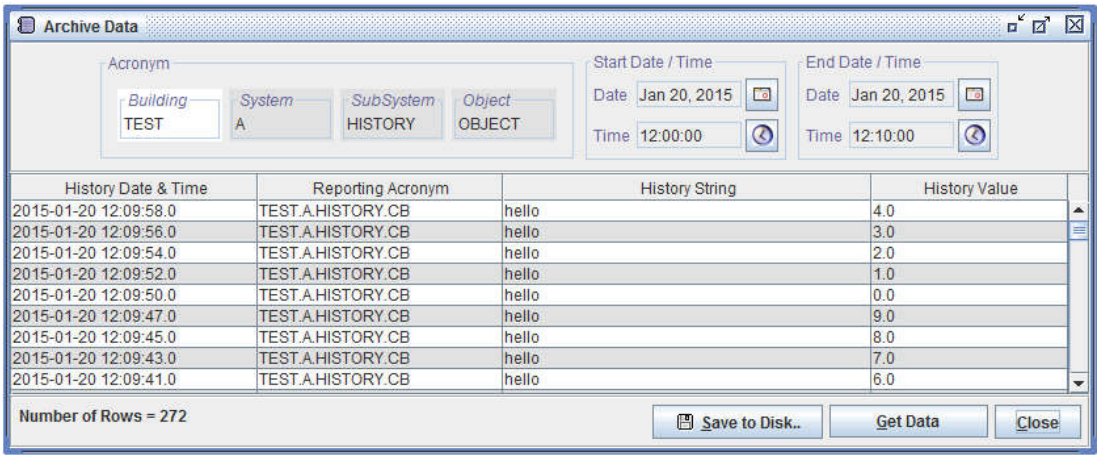


Figure 91 – History Archive Data Window

Get the archive data by clicking the toolbar (Figure 4). A window similar to Figure 91 appears. The only difference is the user needs to type in the acronym of an alarm or history object.

8.2 Save Multiple Histories

To save multiple histories from the Grid Display window:

1. Highlight one or several History objects
2. Right click the selected object(s)
3. Click Save Multiple Histories from the pop-up menu to open the Save History window as shown in Figure 92
4. Click the Start date/time icons to set the start date and time of the historic data
5. Click the End date/time icons to set the end date and time of the historic data
6. Click Ok at the bottom to open the File Selector window.
7. Give a name and click Save to save the data to a file.

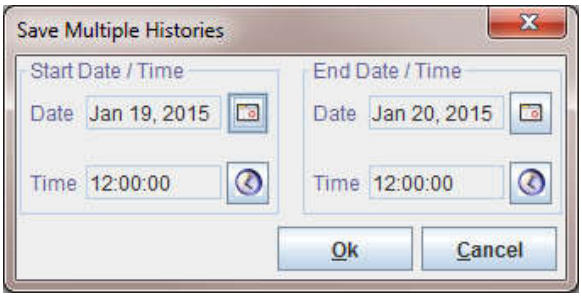


Figure 92- Save History Window

9 Text Messaging

The Text Messaging function provides a way for EMCS system users to communicate. A text message is input in the Text Messaging window as shown in Figure 93.

To open the Text Messaging window, click Messaging in the Forms menu (Figure 2), or click the Messaging toolbar (Figure 4).

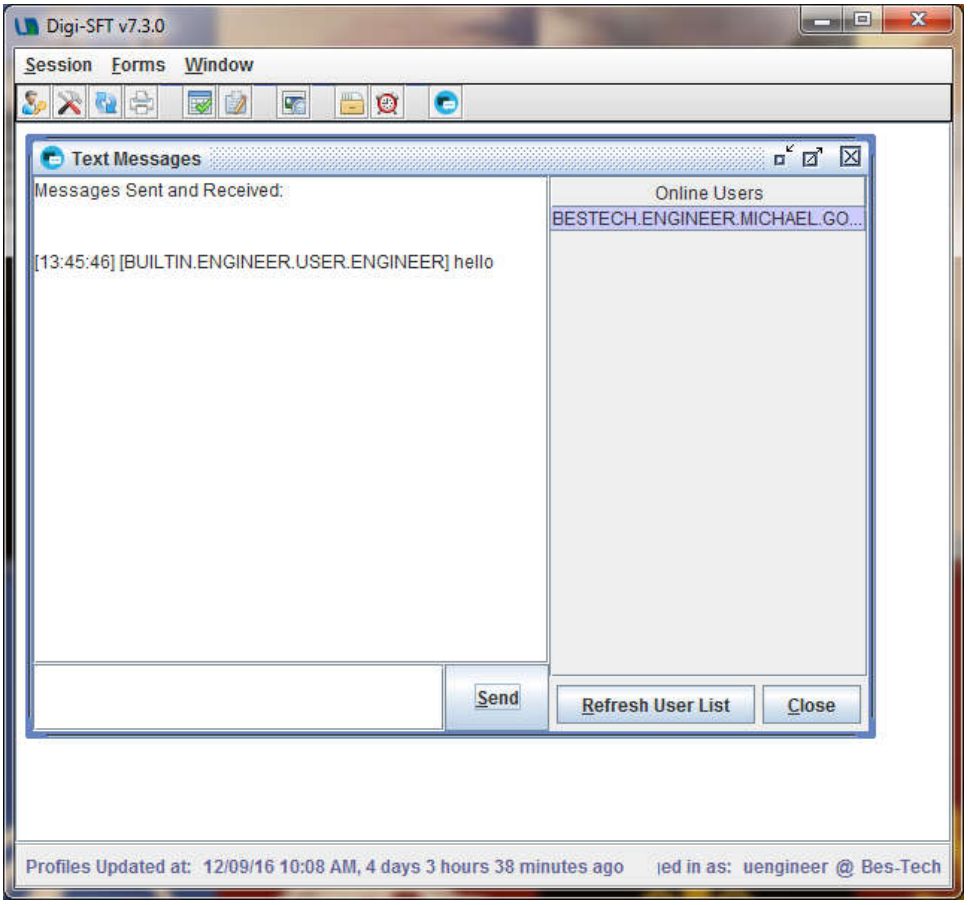


Figure 93 - Text Messaging window

9.1 Refresh User List and Select User

Click the Refresh User List button at the bottom right to refresh the Online Users list at the top right.

Click the user in the online user list to highlight and select the user.

9.2 Type Message

Type messages in the space at the bottom of the messaging window.

9.3 Send Message

Select the message recipient from the online user list at the top right; and click Send at the bottom right of the Text Messaging window (Figure 93) to send the message.

9.4 Read Message

A user can read received messages from the Text Messaging window (Figure 93).

10 Access Control (Admin Only)

Access control determines which users are allowed to log on the system. Permission is granted to specific users to determine what they can control after logging into Digi-SFT.

A user account is set up in the Object Definition window (section 3.11). A user is identified by the user name. To log in the system, a password is also required, which is also entered in the Object Definition window.

When defining a user object (section 3.11), a user may be assigned to a predefined user group (section 10.1). The permissions of a user are determined by the group to which the user belongs.

The permissions of a group are granted in group definition (section 10.1). Only an administrator level user can assign the rights of a user group (section 10.2).

A user group's rights may enable users to engage in as little as none to all of the following actions:

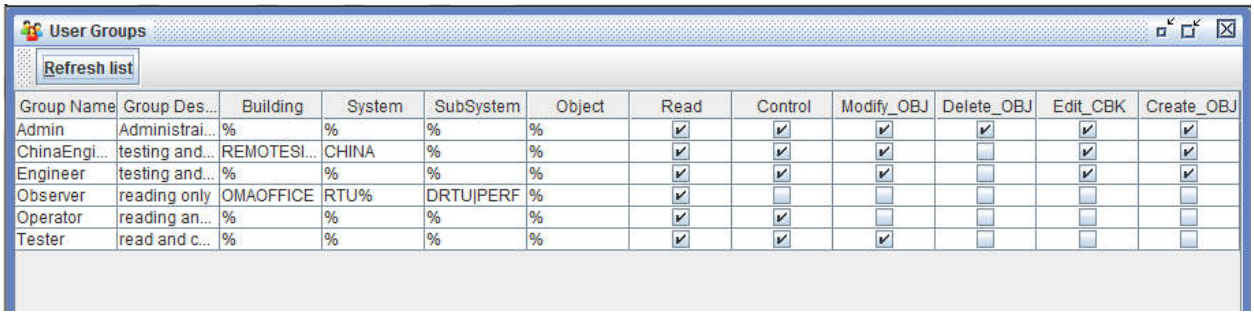
- Read
- Control
- Modify_OBJ
- Delete_OBJ
- Edit_CBK
- Create_OBJ

10.1 User Groups

User can be assigned to different groups. The group can be used for access control by assigning different privileges to the group.

To create a new group, edit or delete an existing group:

1. Click Groups in the Forms menu (Figure 2),or click the Groups toolbar (Figure 4) to open the User Groups window
2. Highlight a group
3. Right click the highlighted group to open a pop-up menu
4. Click the corresponding command to edit, delete, create new or create similar group
5. Click Refresh list on the top left corner of the window to update the User Groups window
6. Click Close in the bottom to dispose the User Groups window.



Group Name	Group Des...	Building	System	SubSystem	Object	Read	Control	Modify_OBJ	Delete_OBJ	Edit_CBK	Create_OBJ
Admin	Administrai...	%	%	%	%	☑	☑	☑	☑	☑	☑
ChinaEngi...	testing and...	REMOTESI...	CHINA	%	%	☑	☑	☑	☐	☑	☑
Engineer	testing and...	%	%	%	%	☑	☑	☑	☐	☑	☑
Observer	reading only	OMAOFFICE	RTU%	DRTU|PERF	%	☑	☐	☐	☐	☐	☐
Operator	reading an...	%	%	%	%	☑	☑	☐	☐	☐	☐
Tester	read and c...	%	%	%	%	☑	☑	☑	☐	☐	☐

Figure 94 - User Groups window

10.1.1 Group Name

The user group is identified by Group Name. The name is an alpha-numeric string of up to 32 characters.

The group name can be created or edited in the Group Definition window as shown in Figure 95.

To modify the group name, click the View/Edit Group command in the pop-up menu (Figure 94).

To create a new group, click either Create Similar Group or Create New Group in the pop-up menu (Figure 94).

When using Create Similar Group, a new group that has similar attributes to the highlighted existing group can be created. This function provides an easy way to create similar groups but the attributes of the new group can still be modified. When using Create New Group, the empty Group Definition window appears, and all necessary information defining the new group is input from scratch. Figure 95 indicates the new Group Definition window.

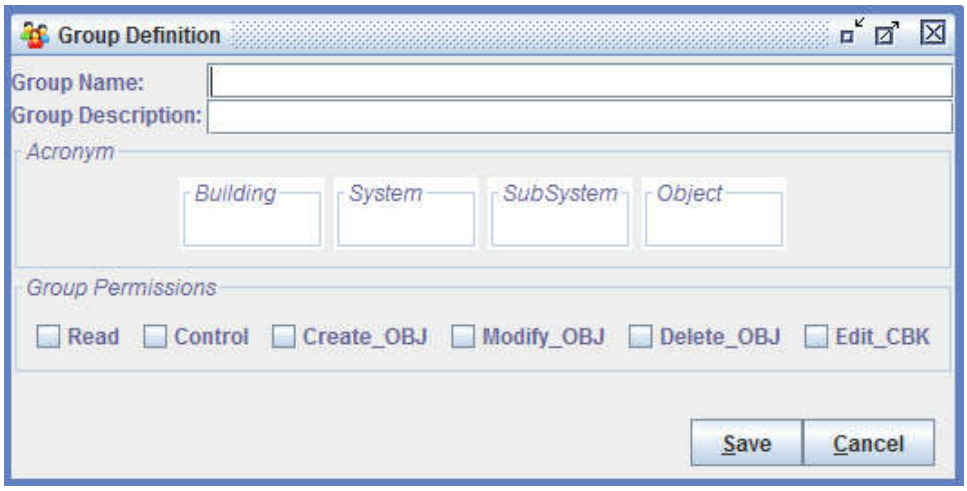


Figure 95 - New Group Definition window

10.1.2 Group Description

Group Description is an optional field to describe the user group. Up to 200 characters can be used in the description.

10.1.3 Acronym

The acronym here is a filter to control which objects are accessible for group members. Two signs have special meaning here:

1. %: this stands for 0 or more occurrences of any letter. e.g., a single "%" in Building means any objects, whereas "RTU%" for System means all objects having System starting with "RTU", such as "RTU", "RTU1".
2. |: this stands for OR relationship between its left and right part. e.g., "RTU%|VAV%" in System means objects that having RTU or VAV as starting characters in System field.

Note: the four parts of an acronym combine in AND relationship, meaning an object will have to meet all four fields to be selected as accessible to group member.

10.1.4 Group Permission

The privilege options for a group include Read, Modify, Edit CB, Control, Create, and Delete.

Check the box in front of an option to give the permission to the group.

10.2 Administrator

Only an administrator has the privilege to access, create, edit, and delete the user groups. A non-administrator user cannot view the Group menu and toolbar.

To classify a user to an administrator use the User Object Definition window (Figure 32). An existing administrator level user can define a user as an administrator. Non-administrator level users cannot change themselves into administrator level users.

11 Check Dependencies

A control block may represent a dependency on other objects if they are referenced in the control block code. Digi-SFT will not allow deletion of any referenced object where such dependencies exist. When an object is highlighted, check dependencies will list the objects upon which the selected object has a dependency.

To check the dependency of an EMCS object:

1. Highlight the object in the Grid display
2. Right click the object to open the pop-up menu (Figure 96)
3. Click Check Dependencies to open the Dependency List window (Figure 97)

Therefore, before an EMCS object is deleted, all the objects displayed in Dependency List should be deleted first.

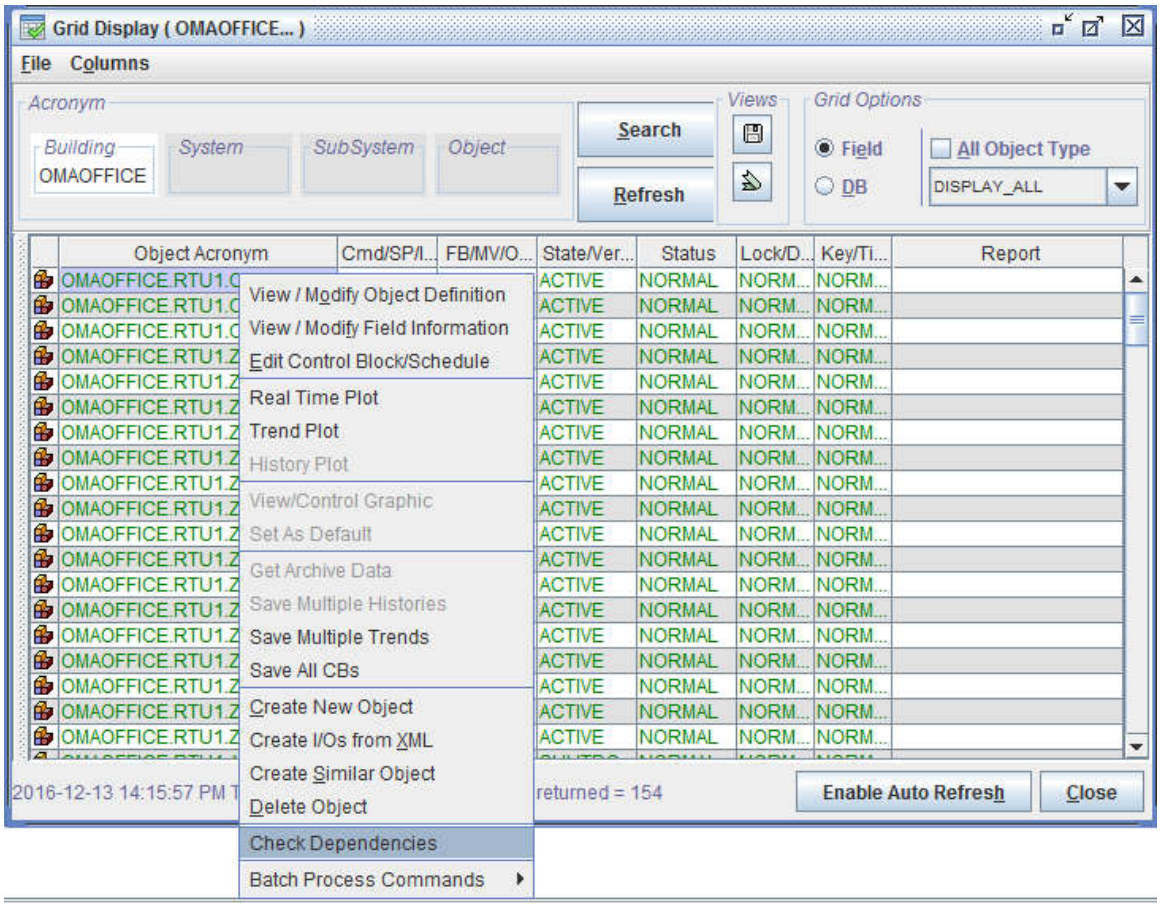


Figure 96 - Check Dependencies

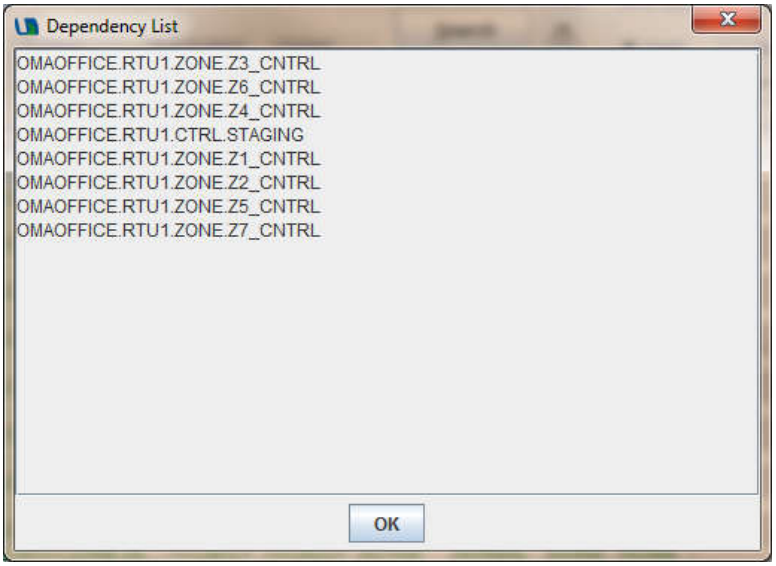


Figure 97 - Dependency List window

12 Batch Process Commands

This section introduces how to use Batch Process Commands in the Grid Display window.

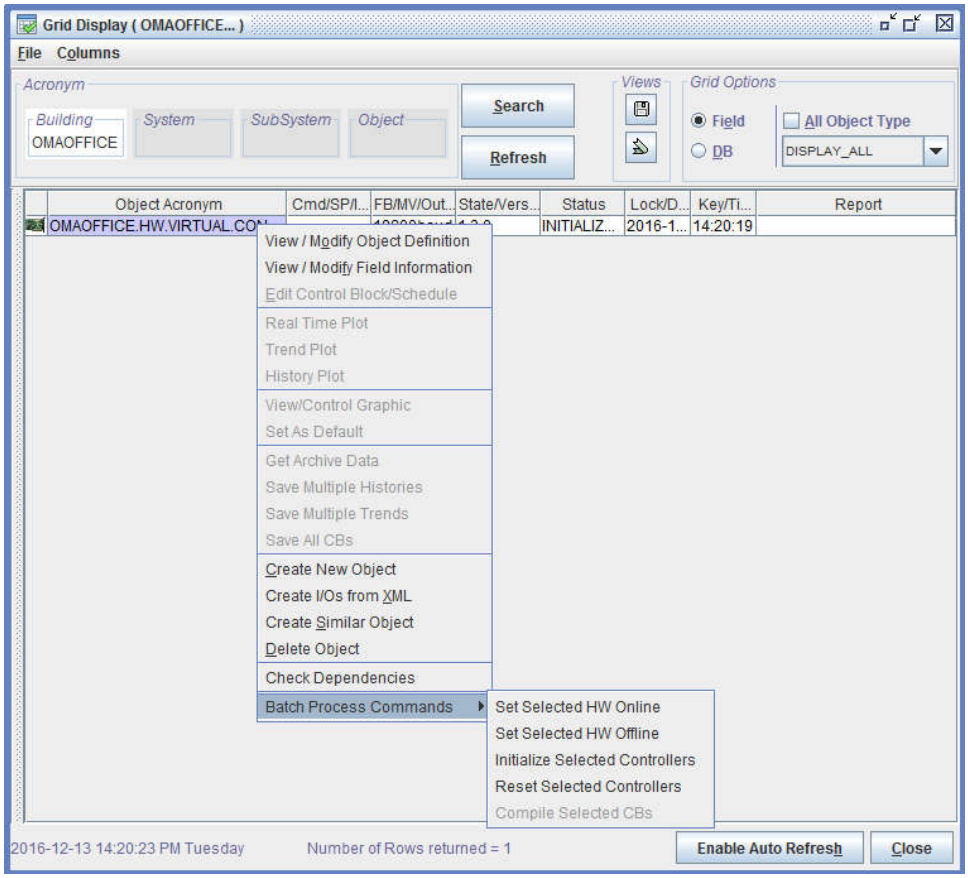


Figure 98 - Batch Process Commands

Batch Process Commands groups batch commands to manage controller, hardware, and control blocks. Those commands include (Figure 98):

1. Set Selected HW online (section 12.1)
2. Set Selected HW offline (section 12.2)
3. Initialize Selected Controllers (section 12.3)
4. Reset Selected Controllers (section 12.4)
5. Compile Selected CBs (section 12.5)

12.1 Set Selected HW Online

Deprecated. This action will take no effect. The controller is always online.

12.2 Set Selected HW Offline

Deprecated. This action will take no effect. The controller is always online.

12.3 Initialize Selected Controllers

You may initialize controllers (hardware objects) using this command by first selecting one or more controllers then choose Initialize Selected Controllers from the pop-up menu.

When a controller is initialized, all EMCS objects will be discarded from the controller. This is immediately followed by a download of all controller resident objects currently defined in the database to the controller. Once all objects are downloaded, the controller resumes operation.

The controllers will be initialized one at a time and their initialization status will appear in a batch process results window.

An error window will appear if no controllers are selected.

Follow the steps below to initialize a controller:

1. Click a cell for a controller in the Grid Display window to select that controller
2. Right click to open a pop-up menu
3. Click Batch Process Commands (Figure 98)
4. Click Initialize Selected Controllers to open a message window (Figure 99)
5. Click Yes to initialize the controller; or No to cancel the operation
6. If click Yes, the Batch Process results window similar to shows the results of the execution.

Figure 99 - Initialize Selected Controller message window

12.4 Reset Selected Controllers

Use Reset Selected Controllers command to reboot the controllers.

Follow the steps below to reset a controller:

1. Click a cell for a controller in the Grid Display window to select that controller
2. Right click to open a pop-up menu
3. Click Batch Process Commands (Figure 98)
4. Click Reset Selected Controllers to open a message window similar to Figure 99
5. Click Yes to Reset controllers; or No to cancel the operation
6. If click Yes, the Batch Process results window similar to shows the results of the execution.

12.5 Compile Control Blocks (Admin Only)

You may compile control blocks using this command by first selecting one or more control blocks then choose Compile Selected CBs from the pop-up menu. If successful, it will be downloaded to the appropriate hardware platform and run.

The control blocks will be compiled one at a time and their compilation status will appear in a batch process results window.

An error window will appear if no control blocks are selected.

Follow the steps below to reset a controller:

1. Click a cell for a control block in the Grid Display window to select the CB
2. Right click to open a pop-up menu
3. Click Batch Process Commands (Figure 98)
4. Click Compile Selected CBs to open a message window
5. Click Yes to compile the CB; or No to cancel the operation
6. If click Yes, the Batch Process results window similar to shows the results of the execution.

13 Settings

The Settings option is used to change specific Digi-SFT settings.

Each time one or more of the available settings gets changed the application needs to be restarted in order to apply the new settings. This will be performed automatically when the user clicks the Save button.

Click Session menu; and then click Settings (Figure 1) to open the Settings window as shown in Figure 100. Or use the Settings toolbar (Figure 4) to open the same window.

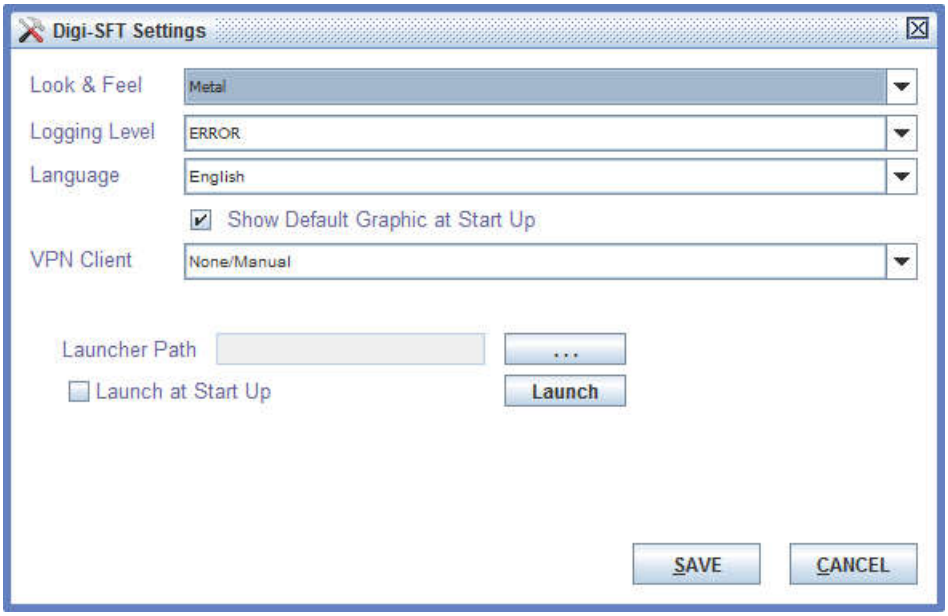


Figure 100 - Settings window

13.1 Style

The "Look and Feel" of the Java user interface can be changed. Available Look and Feels are: Windows, Windows Classic, CDE/Motif, Nimbus and Metal as shown in Figure 101.

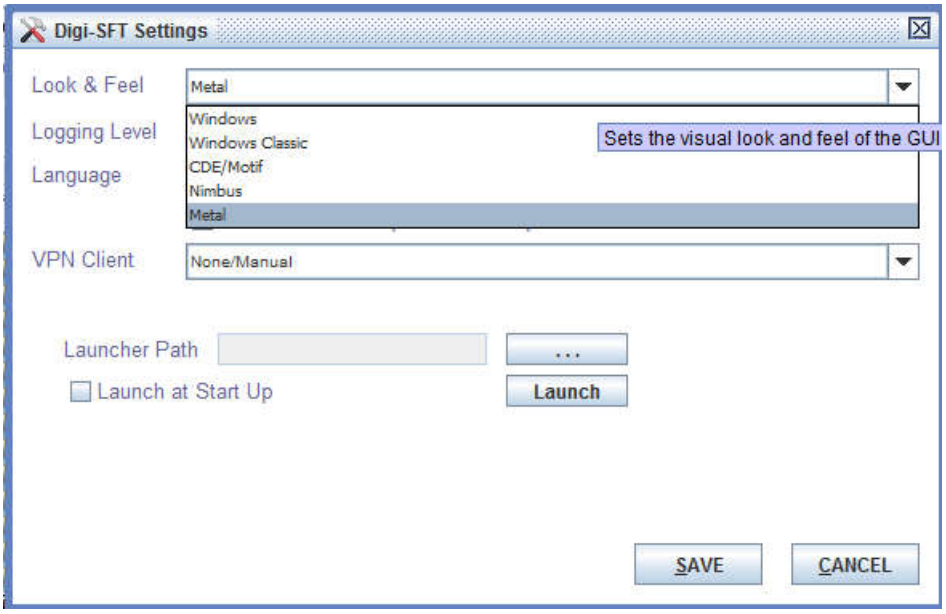


Figure 101 - Look and Feel

13.2 Logging Level

The logging level of the information logged in the log file can be set by selecting a value from the available ones in the Logging Level drop down (Figure 102).

The higher the selected level is the fewer details get logged in. In order to not use significant processing time for logging at run-time, the user is recommended to use ERROR setting. In case

a defect is submitted that cannot be easily recreated on a test system, the user can clear the log file, set the Logging Level to DEBUG, run the scenario that recreates the defect and provide the log file to Bes-Tech engineers in order to help finding the cause of the problem.

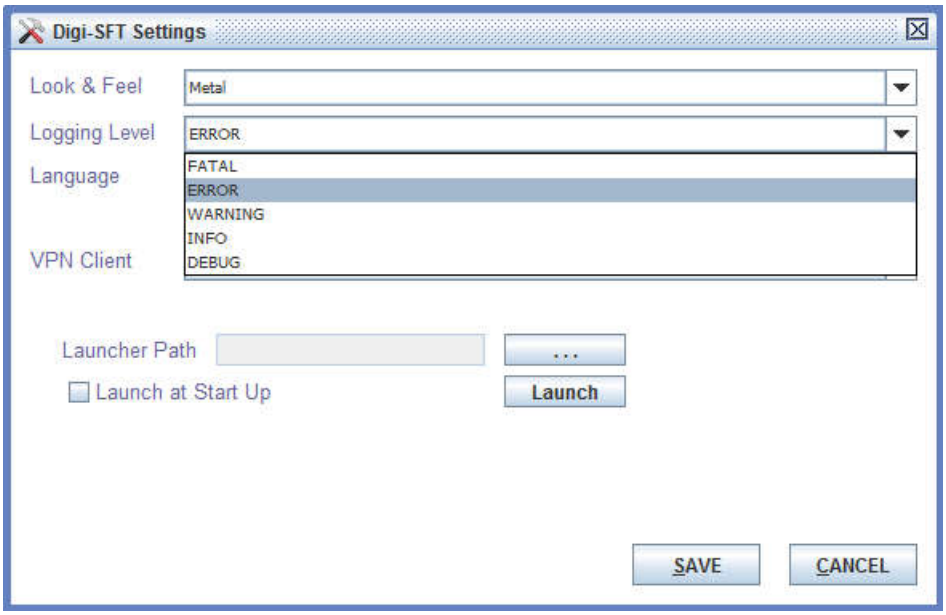


Figure 102 - Logging Level

13.3 Language

Digi-SFT supports the following languages: English and Chinese (simplified). The user interface language can be changed by updating this setting (Figure **103**).

User's Guide document is also provided in the above listed languages.

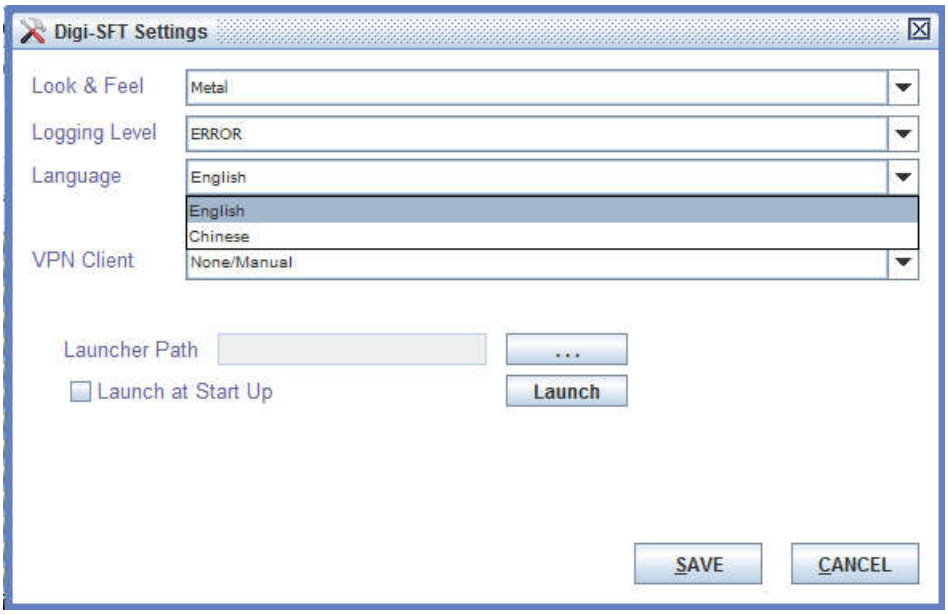


Figure 103 - Language

13.4 Default Graphic

User can set up a graphic object to be opened as default behavior after a successful login. To enable this feature, check this box, and in Grid View, right click on any Graphic object and select “Set as Default”.

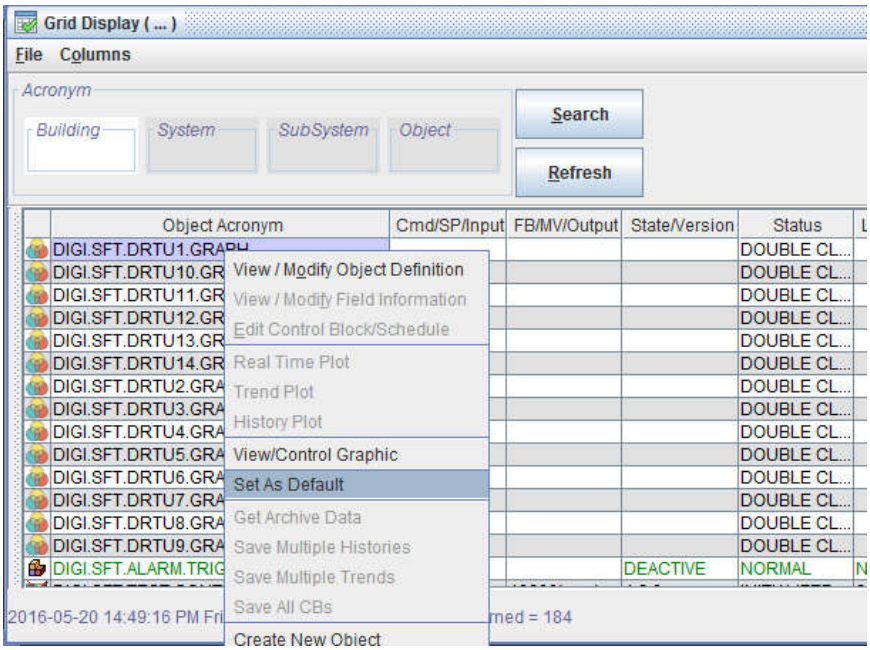


Figure 104 - Default Graphic

13.5 VPN Client

If VPN is needed, user can store VPN authentication information in settings. Digi-SFT Client will use this to automatically connect to VPN before connecting to a Digi-SFT Server.
If our pre-defined VPN login strategy doesn’t cover In situ VPN method. User can choose the “None/Manual” option. This requires users select a path for VPN software exactable file, and user can click Launch to immediately run that program, or check “Launch at Start up” to let Digi-SFT Client invoke that program after automatically after start up.

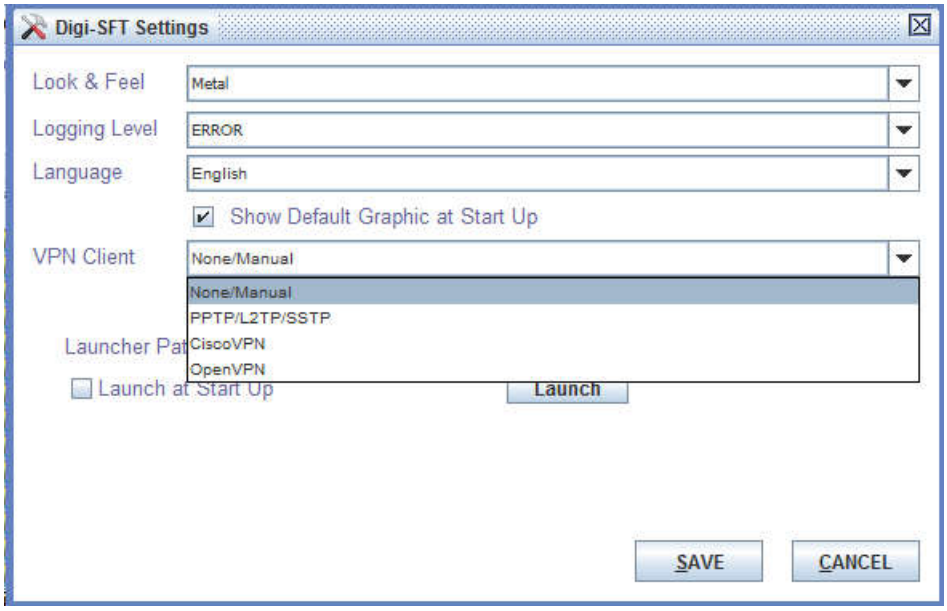


Figure 105 – Setting VPN support

14 Appendix

14.1 DCL Keywords

Language Keywords	Reference Section Name	Reference Section Number
AI	Object Variable Types	4.2.4
and	Logical Operators	4.4.3
AO	Object Variable Types	4.2.4
bool	Primitive Variable Types	4.2.3
break	break Statement	4.6.8
case	switch-case Statement	4.6.6
CB	Object Variable Types	4.2.4
continue	continue Statement	4.6.9
Date	Time and Data Functions	4.5.4
default	switch-case Statement	4.6.6
DI	Object Variable Types	4.2.4
DO	Object Variable Types	4.2.4
do	while and do-while Statement	4.6.5
else	If-else Statement	4.6.3
float	Primitive Variable Types	4.2.3
extern	External Variables	4.2.7
for	for Statement	4.6.4
HIST	HIST Objects	4.2.9.7
HW	HW Objects	4.2.9.9
if	If-else Statement	4.6.3
input	Input/Output Variables	4.2.8
int	Primitive Variable Types	4.2.3
LOOP	CB and LOOP Objects	4.2.9.6
or	Logical Operators	4.4.3
output	Input/Output Variables	4.2.8
return	return Statement	4.6.10
stop	stop Statement	4.6.7
time	Time Expressions	4.5.3
while	while and do-while Statement	4.6.5

14.2 DCL Variable Usage Summary

The following table summarizes where and how the different variable types may be declared and used in DCL.

Variable Type	Math Op's	Logical Op's	Global Scope	Function Scope	External Variable (1)	Pass-by-Value to Function	Pass-by-Reference to Function	Assigned a Return Value from Function
float	Y	Y	Y	Y	Y	Y	Y	Y
int	Y	Y	Y	Y	Y	Y	Y	Y
bool	N	Y	Y	Y	Y	Y	Y	Y
time	N	Y	Y	Y	Y	Y	Y	Y
date	N	Y	Y	Y	Y	Y	Y	Y
object	N	N	Y	N	N	N	Y	N
float array	Y	N(2)	Y	Y	N	N	Y	N
int array	Y	N(2)	Y	Y	N	N	Y	N
bool array	N	N(2)	Y	Y	N	N	Y	N
time array	N	N(2)	Y	Y	N	N	Y	N
date array	N	N(2)	Y	Y	N	N	Y	N
object array	N	N	Y	N	N	N	Y	N

Table Notes:

1. Local scope external variables are not allowed. All external variables must be declared with global scope (i.e., they must be declared at the beginning of a DCL file outside of any functions).
2. Logical operations can be performed on individual array *elements* but not entire arrays themselves.

14.3 DCL EMCS Constants

14.3.1 Control Block and LOOP State

Constant	Description
STATE_ACTIVE	This constant defines the active state for a control block or loop. See section 4.1.1 for more information about control block states.
STATE_DEACTIVE	This constant defines the deactivate state for a control block or loop. See section 4.1.1 for more information about control block states.
STATE_RESTARTED	This constant defines the restarted state for a control block. See section 4.1.1 for more information about control block states.
STATE_RESUMED	This constant defines the resumed state for a control block or loop. See section 4.1.1 for more information about control block states.
STATE_SHUTDOWN	This constant defines the shutdown state for a control block or loop. See section 4.1.1 for more information about control block states.
STATE_STOPPED	This constant defines the stopped state for a control block. See section 4.1.1 for more information about control block states.

14.3.2 Lock/Key

Constant	Description
LK_HIGH	This constant defines the second highest level of privilege for the lock and key function. See section 3.12 for more about lock and key.
LK_LOCKOUT	This constant defines the highest level of privilege for the lock and key function. See section 3.12 for more about lock and key.
LK_MED	This constant defines the second lowest level of privilege for the lock and key function. See section 3.12 for more about lock and key.
LK_NORMAL	This constant defines the lowest level of privilege for the lock and key function. See section 3.12 for more about lock and key.

14.3.3 Report Severity

Constant	Description
SEVERITY_HIGH	This constant defines the second highest level of report severity.
SEVERITY_LIFE_SAFETY	This constant defines the highest level of report severity.
SEVERITY_LOW	This constant defines the lowest level of report severity.
SEVERITY_MEDIUM	This constant defines the second lowest level of report severity.

14.3.4 Status

Constant	Description
STATUS_ALLOW_ACCESS	
STATUS_AMBIENT_ENTHALPY	
STATUS_AMBIENT_REL_HUM	
STATUS_AMBIENT_TEMP	
STATUS_CHILLED_WATER_PRESSURE	
STATUS_CHILLED_WATER_TEMP	
STATUS_COLDEST_TEMP	
STATUS_DAY_TYPE	
STATUS_DECON	Decontamination.
STATUS_DECON_ABORT	Decontamination is aborted.
STATUS_DECON_DEHUM	Decontamination with dehumidification.
STATUS_DENY_ACCESS	
STATUS_DISABLE_NOTIFICATION	
STATUS_EMERGENCY_ALARM	User hit “emergency” flow button on fume hood monitor.
STATUS_ENABLE_NOTIFICATION	
STATUS_EQUIPMENT_FAILURE	
STATUS_FAILED_TO_CLOSE	Object failed to close.
STATUS_FAILED_TO_OPEN	Object failed to open.
STATUS_FAILED_TO_START	
STATUS_FILLING	
STATUS_FLOW_ALARM	Fume hood flow is out of tolerance (i.e., too high or too low)
STATUS_FORCED_OPEN	
STATUS_HARDWARE_LOCKED	
STATUS_HARDWARE_UNLOCKED	
STATUS_HELD_OPEN	
STATUS_HICLAMP	
STATUS_INITIALIZED	
STATUS_INVALID_COMMAND_VALUE	

Constant	Description
STATUS_INVALID_CV	
STATUS_ INVALID_MV	
STATUS_ INVALID_READING	
STATUS_ INVALID_STATE	
STATUS_LINE_OPEN	
STATUS_LINE_SHORT	
STATUS_LOCALLY_MODIFIED	Controller or other resident objects were locally modified by technician.
STATUS_LOCLAMP	
STATUS_NOT_INITIALIZED	
STATUS_NO_DATA	Scribe has not received any new values for a period of time.
STATUS_NO_SCODE	
STATUS_OLD_PROTOCOL	
STATUS_OPERATION_MODE	
STATUS_POWERUP	
STATUS_PROGRAMMING	
STATUS_ PROGRAMMING_FAILURE	
STATUS_READY	
STATUS_RESET	
STATUS_STARTED	
STATUS_STEAM_PRESSURE	
STATUS_VALIDATING_CREDENTIAL	
STATUS_WAITING	
STATUS_WAITING_ON_KEYPAD	
STATUS_WAITING_ON_READER	

14.3.5 Undefined Primitives

Constant	Description
UNDEFINED_DATE	The value of this constant represents an undefined date.

Constant	Description

UNDEFINED_FLOAT	The value of this constant represents an undefined floating point number. For example, to ignore an undefined floating point number in an array:
	`for(i = 0; i < arraySize(x[]); i = i + 1)`
	`{`
	`  if(x[i]==UNDEFINED_FLOAT) {continue;}`
	`  xSum=xSum+x[i];`
	`  countX++;`
	`}`

UNDEFINED_INT	The value of this constant represents an undefined integer number.
UNDEFINED_BOOL	The value of this constant represents an undefined Boolean.
UNDEFINED_TIME	The value of this constant represents an undefined time.

14.3.6 User Settable Status

Constant	Description
STATUS_CYCLE_PUMPS	
STATUS_DAY	
STATUS_DEADBAND	
STATUS_DEADBAND_NO_REHEAT	
STATUS_DEFROST	
STATUS_EXTERNAL_RESET	
STATUS_HIGH_SPEED	
STATUS_INTERNAL_RESET	
STATUS_LOW_SPEED	
STATUS_MANUAL	
STATUS_MANUAL_OVERRIDE	
STATUS_NIGHT	
STATUS_NORMAL	
STATUS_NORMAL_NO_REHEAT	
STATUS_NULL_POINT	
STATUS_OCCSENS	
STATUS_OFF	
STATUS_ON	
STATUS_RUN_BOTH_PUMPS	

Constant	Description
STATUS_RUN_PUMP1	
STATUS_RUN_PUMP2	
STATUS_STAFF_HOLIDAY	
STATUS_STUDENT_HOLIDAY	
STATUS_SUMMER_DEADBAND	
STATUS_SUMMER_NORMAL	
STATUS_TEMP_OCCUPIED	
STATUS_UNOCC	
STATUS_WINTER_DEADBAND	
STATUS_WINTER_NORMAL	
STATUS_WINTER_SHUTDOWN	

Example code for using DCL EMCS status constants:

```
activate()
{
   // Get default values and set status to normal
   defaultValues();
   setStatus(STATUS_NORMAL);
   report(0, "");
}
deactivate()
{  setStatus(STATUS_OFF);  }

shutdown()
{  setStatus(STATUS_OFF);  }
```

14.4 DCL Coding Standards

14.4.1 Introduction

Coding standards and conventions are necessary for the following reasons:

- Portability and Reuse. Code that is written once is more likely to be used again.
- Consistency and Neatness.
- Maintainability.
- Clarity.
- Productivity.

14.4.2 Basic Principals

- **Keep the code simple.** It is better to have a number of CB's and functions that do one thing clearly and well, rather than one "super" CB or function that attempts to do everything.
- **Be explicit.** Say what you mean.

- **Be consistent.** Use the same rules as much as possible.
- **Keep the *spirit* of the standards.** Where you have a coding decision to make and there is no direct standard, then you should always keep within the spirit of the standards.
- **Avoid complicated statements.** Statements comprising many decision points are hard to follow.
- **Updating old code.** Whenever existing code is modified try to update the document to abide with the conventions outlined in this document. This will ensure that old code will be upgraded over time.

14.4.3 Source Files

Use of TAB character

The standard indent level should be 4 spaces. This insures that any DCL source file can be consistently displayed in any kind of text editor.

CB File Documentation

At the beginning of each file, include a comment block which is formatted as the following example.

```
/*
************************************************************** CB
FULL NAME: [BUILDING.SYSTEM.SUBSYSTEM.NAME]
* PROGRAMMER(s): Joe Programmer (JP)
* CREATED DATE: MM/DD/YYYY
* DESCRIPTION: This CB accomplishes the following tasks: ...
*
* INPUT: cooling set point
* OUTPUT: heating set point
*
* ACTIVATED BY: temp changes
* CONTROLS: heating
*
* CHANGE LOG:
* DATE    WHO   DESCRIPTION
* ---------------------------------------------------------------
* 7/02/2012 K.C.      Added super feature
* 1/15/2015 L.A.      Fixed Bug
**************************************************************/
```

DCL File Layout

DCL CB source files should be organized in the following manner:

- File documentation (previous section)
- Global variable declarations
- Any non-reserved user defined functions
- activate() function
- resume() function
- main() function
- deactivate() function
- shutdown() function

Note that not all of the above elements will necessarily be present in every DCL file.

14.4.4 Commenting

Keep code and comments visually separate.

Comments should be inserted using either of the following styles:

1. Comments Along Side Code.

```
float fTempValue;     // Current EU value of temp sensor
int iCount;           // Current counts
bool bFlag;           // Calculation complete flag
```

As can be seen, this is most often most appropriate when commenting variable declarations.

2. Comments Above Code

```
// Loop through array and perform CRC calculation
for(i = 0; i <= MAX - 1; i++)
update(a[i]);

// Put array into data buffer
bufferWrite(a[], value);
```

Be sure to include a blank line between non-related comments as in the example.

Multi-line comments.

Avoid the following comment style:

```
/* This type of comment can lead to confusion especially when
describing a function like clkUpdateTime(). The function looks
like actual code! */
```

Instead, use the either of the following methods

```
// This is much better because
// there is a comment symbol
// which starts each line.

/* This is much better because
* even though there is not a
* comment symbol on each line,
* there is an asterisk which
* starts each line.
*/
```

User Defined Function Comments

Every user defined function should have a comment section preceding the function definition. The suggested layout of this section is shown by the following example.

```
/*
*****************************************************************
FUNCTION NAME: pidLoop
*
* DESCRIPTION: Implements PID control algorithm.
*
* ARGUMENTS: msrdVar = AI input
* cntrdVar = AO output
```

```
* k[] = Array of P, I, and D constants
*
* RETURNS: Current output between 0 and 100
*
* NOTES: Moves controlled var value to appropriate
* position.
*****************************************************************/
float pidLoop(float msrdVar, float &cntrdVar, float k[])
{
     // Function definition
}   // End pidLoop
```

14.4.5 Variable and Function Naming Conventions

Variable and function names should be descriptive so that their purpose is clear.

Capitalization

Use a lower case letter for the first letter of a non-constant variable or function. Then, capitalize each successive word. Examples include:

```
ductPressure
maxInternalTemp
roomTemp
damperPosition
```

Use standard acronyms, abbreviations and mnemonics consistently.

The following are from the “Acronyms, Abbreviation and Mnemonics (AAM) Dictionary.”

Argument	Arg
Buffer	Buf
Clear	Clr
Clock	Clk
Compare	Cmp
Configuration	Cfg
Context	Ctx
Delay	Dly
Device	Dev
Disable	Dis
Display	Disp
Enable	En
Error	Err
Function	Fnct
Hexadecimal	Hex
Initialize	Init
Manager	Mgr
Manual	Man
Maximum	Max
Message	Msg
Minimum	Min
Multiplex	Mux
Overflow	Ovf
Parameter	Param

Previous	Prev
Priority	Prio
Read	Rd
Ready	Rdy
Register	Reg
Schedule	Sched
Synchronize	Sync
Timer	Tmr
Trigger	Trig
Write	Wr

14.4.6 Code Layout

Only have one action per line of code

```
pressure = 12.5;
doneFlag = TRUE;
```

Instead of:

```
pressure = 12.5; doneFlag = TRUE;
```

Spacing

Write array subscript operators without spaces around them.

```
a[i]
movingAverage[12]
```

Parenthesis after function names have no spaces before them.

```
startup();
```

At least one space is needed after each comma to separate function parameters in function definitions and arguments in function calls.

```
// Function definition
int startup(float maxValue, int count)
{
     // startup code
}

// Function call
nightSetBack(LOOP L[], float setBackTemp);
```

At least one space is needed after each semicolon in a for loop.

```
for (i = 1; i <= n; i++)
```

Binary operators are written with at least one space between them and their operands.

```
c1 = c2;
x + y
i += 2;
n > 0 ? n : -n;
a < b
c >= 2
```

Unary operators are written with no space between them and their operand.

```
++i
!ready
j–
```

The keywords if, while, for, and return are followed by one space.

```
if (a > b)
while (x > 0)
for (i = 0; i < 10; i++)
return (y);
```

Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis.

```
x = (a + b) * c;
```

Bracing Style

The suggested DCL bracing style is easier to illustrate by examples than to describe.

```
if (x > 0)
{
     y = 10;
     z = 5;
} // End if
else
{
     if (z < LIM)
     {
           x = y + z;
           z = 10;
     } // End if
     else
     {
           x = y – z;
     }
} // End else
```

Note that this approach makes it easy to ensure that every opening brace has a corresponding closing brace. Also, indentation clarifies what is within a particular statement and what is not.

14.5 Example DCL Code

14.5.1 Time Scheduler

This DCL example schedules two different air handling systems.

```
/*
*****************************************************************
* CB FULL NAME: [CLASSROOM.AH.UNIT.SCHEDULER]
* PROGRAMMER: Joe Programmer (JP)
* CREATED: 10/10/2014
* DESCRIPTION: This CB schedules the classroom air handling
* units. It uses external boolean variables to indicate whether * a
particular AHU should run. Other CB’s monitor these
```

```
* externals and act accordingly.
*
* INPUT: schedule intervals
* OUTPUT: air handling units run state
*
* ACTIVATED BY: time changes
* CONTROLS: air handling units
*
* CHANGE LOG:
* DATE    WHO   DESCRIPTION
*-----------------------------------------------------------------
*
*****************************************************************/

extern bool ah1Run, ah2Run;             // External
time BEGIN = 7:0:0;               // Constant
time NOON = 12:0:0;               // Constant
time QUIT = 19:0:0;               // Constant

main()
{
     while(TRUE)
     {
           switch
           {
                 // Note that break statements are not necessary
                 // since case statements are mutually exclusive
                 case mon()
                 {
                       ah1Run = BEGIN -> QUIT;
                       ah2Run = BEGIN -> NOON;
                 }
                 case tue()
                 {
                       ah1Run = BEGIN -> QUIT;
                       ah2Run = BEGIN -> NOON;
                 }
                 case wed()
                 {
                       // Evening Classes!
                       ah1Run = BEGIN -> QUIT;
                       ah2Run = (BEGIN -> NOON) or
                                   (19:0:0 -> 22:0:0);
                 }
                 case thu()
                 {
                       // Evening Classes!
                       ah1Run = BEGIN -> QUIT;
                       ah2Run = (BEGIN -> NOON) or
                                   (19:0:0 -> 22:0:0);
                 }
                 case fri()
                 {
                       ah1Run = BEGIN -> QUIT;
                       ah2Run = BEGIN -> NOON;
                 }
```

```
                    case sat()
                    {
                          ah1Run = NOON -> QUIT;
                          ah2Run = FALSE;
                    }
                    case sun()
                    {
                          ah1Run = 10:0:0 -> 14:0:0;
                          ah2Run = NOON -> QUIT;
                    }
               } // End switch
               // Delay 1 minute between day / time check
               delay(60);
          } // End while
     } // End main
```

15 Glossary

A **Alarm**

An object that provides an audible or visual warning of a problem or condition.

Analog

Data that has continuous values in both time and amplitude to represent information.

Analog Input

An object that provides a continuous signal from a sensor.

Analog Input Module

A device that receives an analog signal, usually from a sensor, and converts and sends a Digital signal, usually to a controller.

Analog Output

An object that provides a continuous signal sent to an actuator.

Analog Output Module

A device that receives a Digital signal, usually from a controller, and converts and sends an analog signal, usually to an actuator.

B **BAS**

Building automation system.

C **Client**

A computer that accesses a remote computer system, known as a server.

Control block

An object that is a computer program written in a domain specific programming language.

D **Digital**

Data that uses discrete or discontinuous values to represent information.

Digital Input

An object that provides a discrete signal from a switch or limit device.

Digital Output

An object that provides a discrete signal sent to an on/off device.

Distributed Control System

A control system in which the controller elements are distributed throughout the system with each component controlled by one or more controllers.

E

EMCS

An Energy Management Control System provides the comfort control of a traditional HVAC system with the additional constraint of minimizing energy consumption.

Ethernet

Computer networking technology for local area networks.

F

G

Graphic

An object that is an interactive illustration that can be used to display real-time information of any part of the system and can control output devices in the system.

H

Hardware

An object that represents a piece of hardware, such as

- Field computer
- Multiplexor
- PMC/PMC2
- Server

History

An object that offers a data acquisition mechanism that requires programming a control block and is used to save user defined variables at a programmable data acquisition rate.

HVAC

Heating, ventilation, and air conditioning.

I

J

K

Key

The key value is one of four values: normal, medium, high, and lock-out that can be assigned to an object to control other objects, such as control blocks and users. An object can control the controllable object when the key value is greater than or equal to the lock value.

L

Lock

The lock value is one of four values: normal, medium, high, and lock-out. It can be assigned to controllable objects, such as analog outputs, control blocks, Digital outputs, and loops.

Loop

An object that provides proportional, integral, derivative (PID) control.

M

N

O **Object**

An object is an individual element of Digi-SFT system, including:

- Alarm
- Analog input
- Analog output
- Control block
- Digital input
- Digital output
- Graphic
- Hardware
- History
- Loop
- Schedule
- User

Object Identifier

Three part numbering scheme that uniquely identifies objects with a top, middle, and bottom that is assigned by the system based on the type of object being created and where the object resides. The following convention is used for the object identifier:

- Top – represents a particular field computer or user.
- Middle – represents a controller or field computer process.
- Bottom – represents a specific object on the controller or a specific object managed by a field computer process.

P **Points**

Input/output objects, such as

- Analog input
- Analog output
- Digital input
- Digital output

Q

R **RS-232**

Standard for serial binary data signals.

RS-485 (EIA-485)

A standard that specifies a two-wire, half-duplex, multipoint serial connection.

S **Schedule**

An object that allows you to define what time events occur and on what days, such as

weekdays, weekends, and holidays.

SQL

Structured Query Language is a language designed for relational database management systems.

Server

A computer dedicated to running software that services requests over a network connection.

Setpoint

The desired value for an automatic control system.

Smart Actuator

An actuator that receives a Digital command signal, usually from a controller.

Smart Sensor

A sensor that sends a Digital signal, usually to a controller.

T **Trend**

A term used to describe data acquired and saved. There are two types of trending, automatic and history. The automatic trending will save the standard predefined output variable for any object at a periodic acquisition rate. The history object requires programming a control block and is used to save user defined variables at a programmable data acquisition rate.

U **User**

An object that defines an individual's account settings.

V **VAV**

Variable Air Volume is a method of controlling an HVAC system that uses a varying amount of air volume.

W **Workstation**

Any terminal or personal computer connected to a network.

X

Y

Z

Index