BeyondHMI

Dual Flow Computer Library

Developer's Guide: using BhiLibDualFc in an e!COCKPIT project

The following sections provide detailed instructions for creating a simple e!COCKPIT program which uses the dual flow computer library.

Contents

Developer's Guide: using BhiLibDualFc in an e!COCKPIT project 1
- Obtain the library file 3
- Install the library in e!COCKPIT 3
- Add the Library to Your Project 4
- Create persistent memory structures which will be used by the library 5
- Add necessary variables to the Program which calls the library function block 6
- Add supporting Code to the Program 8
- Link Program Visualizations to Library Visualizations 11
- Adjust e!COCKPIT project Task Interval 12

Licensing 13
- Trial Mode 13
- Steps to Obtain a Runtime License 13

Modifying your PLC program without corrupting library data 14

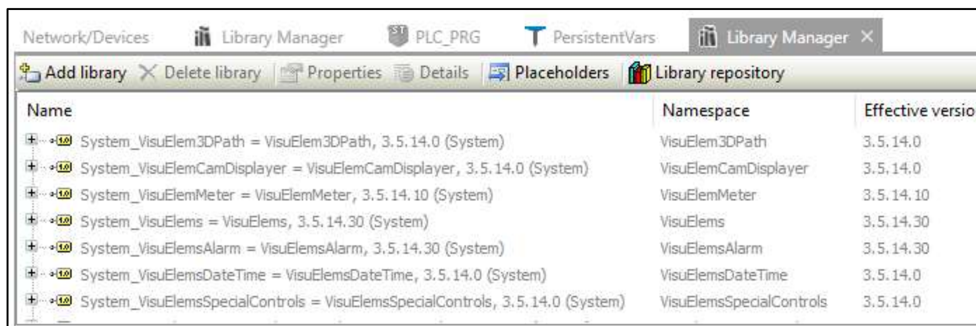
How your program can interact with the Library 16
- Reading Current Calculated Values from your program 16
  - Reading Gas Meter Values 16
  - Reading Oil Meter Values 16
- Reading Meter Configuration Values from your program 16
- Reading Station Configuration Values from your program 17
- Writing Meter Configuration Values from your program 17
- Writing Station Configuration Values from your program 17

Obtain the library file

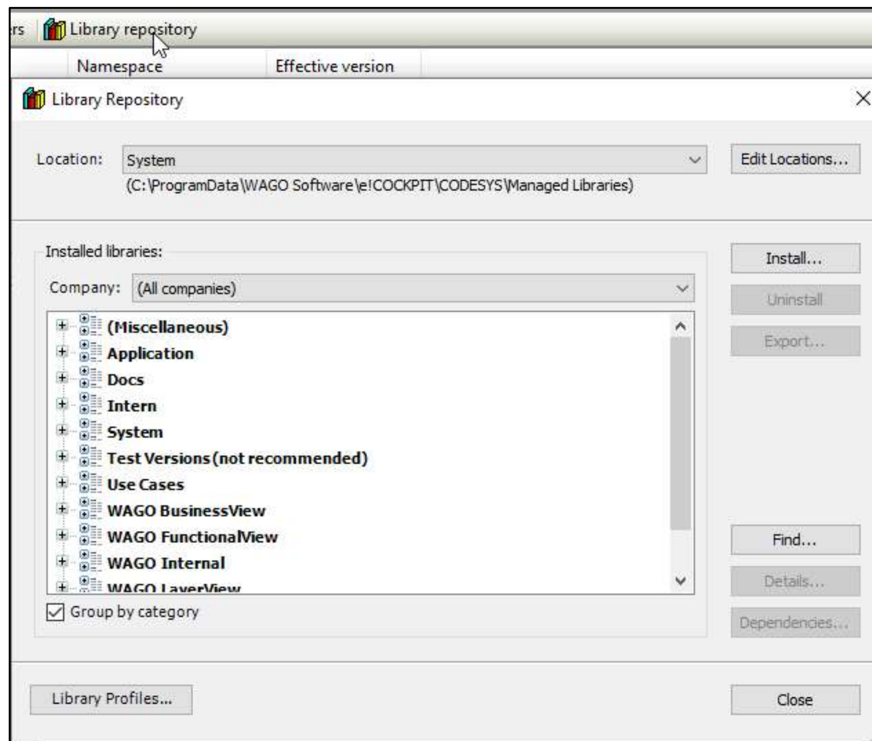
Request the BhiLibDualFc compiled-library from info@beyond-hmi.com. There is one version of the library. It supports a single (one) gas meter run and a single (one) liquid meter run. The meter run can be either an allocation meter or a custody-transfer meter.

Install the library in e!COCKPIT

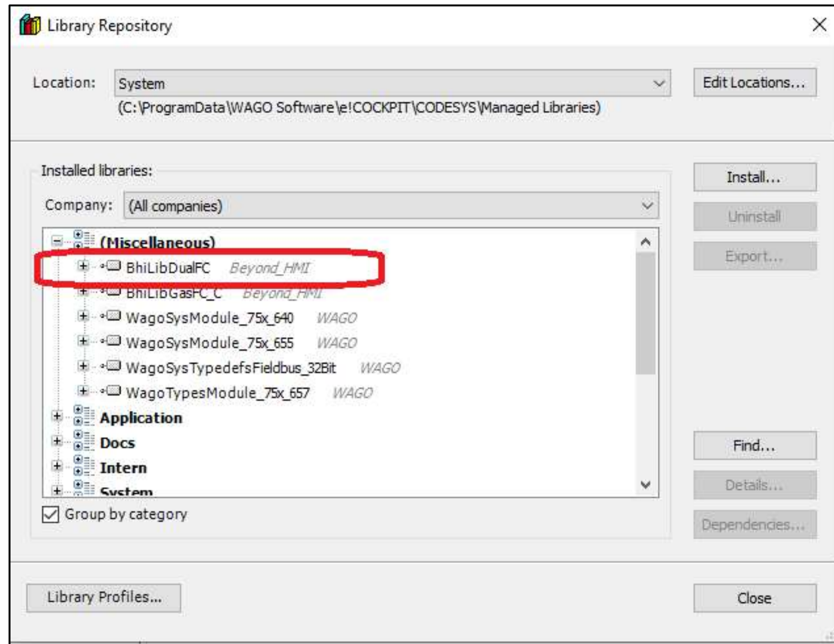
- Open any project in e!COCKPIT
- Navigate to a Library Manager



- Select **Library Repository**

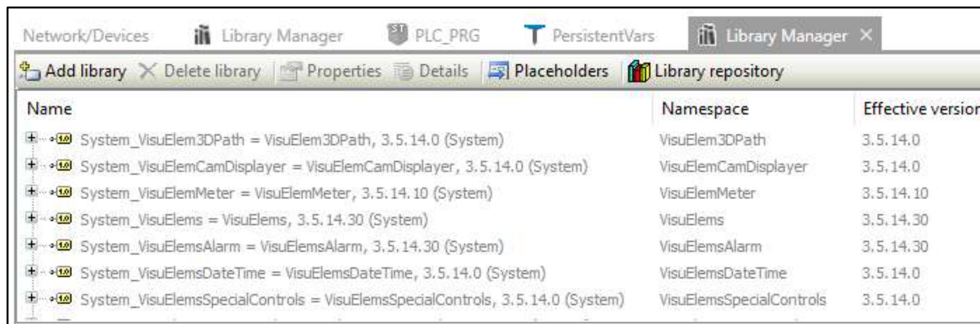


- Select **Install..**
- Navigate to the downloaded library file and click on **Open**.
- Verify that the library was installed in the **Miscellaneous** section

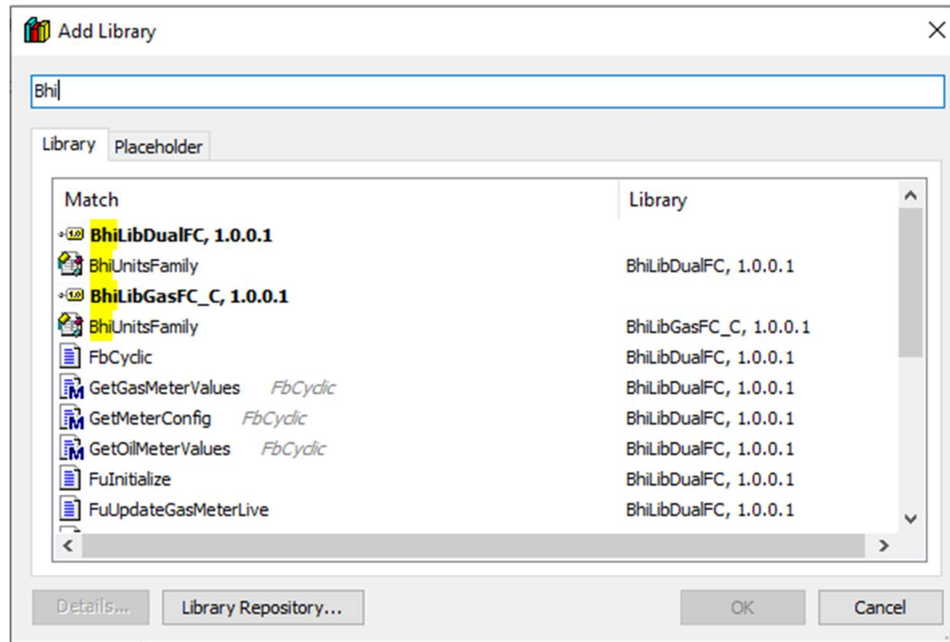


Add the Library to Your Project

- Create a project and designate the Device(s) in the project.
- Navigate to a Library Manager



Select **Add Library**



Start typing the library name until the library appears in bold text

Select the Library and select **OK**.

Create persistent memory structures which will be used by the library

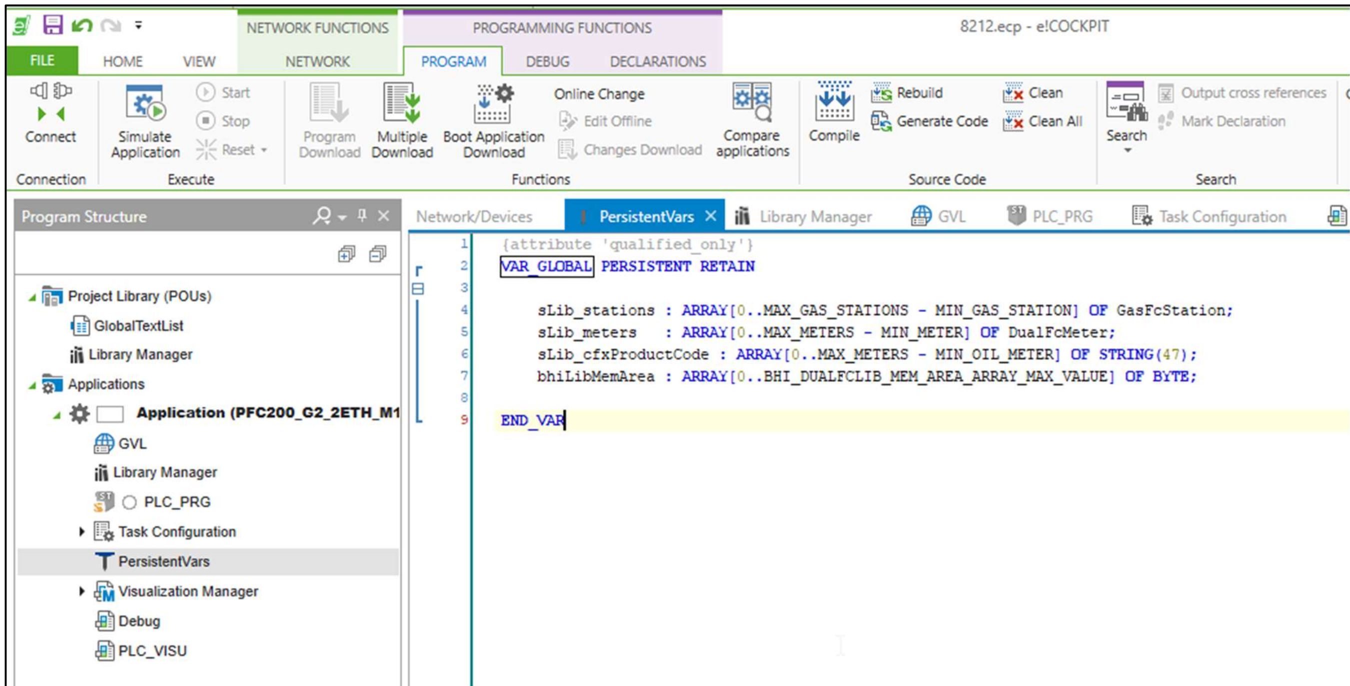
The library needs some of its data structures to persist – even when the PLC program is loaded or the power to the PLC is cycled. Your program needs to allocated these structures and pass them to the library.

If one has not already been created, add a **Persistent Variables** Object to the Project

Navigate the project's **Persistent Variables** object

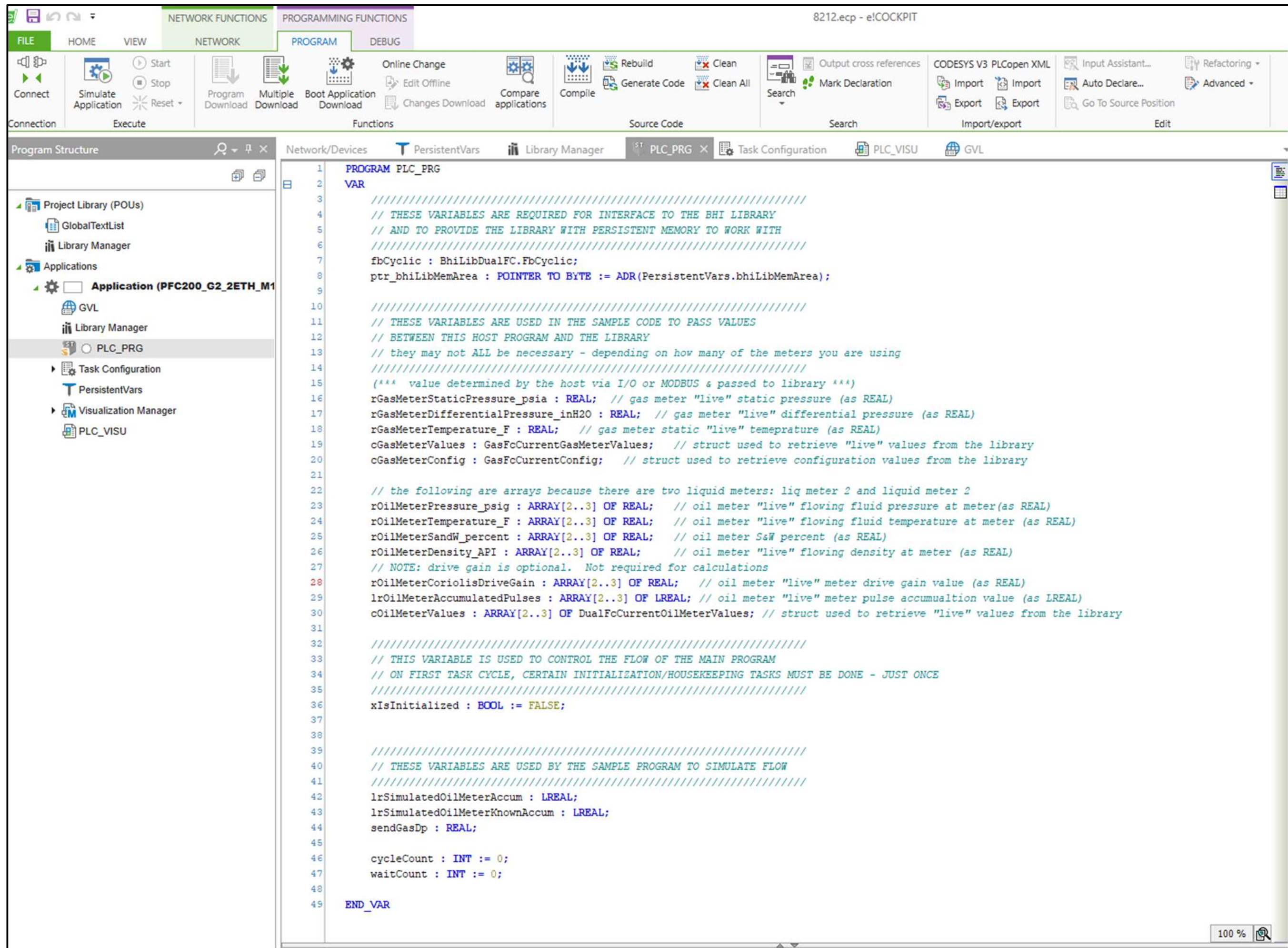
Add the following declarations to the persistent memory area (Copy these lines into the e!COCKPIT window):

```
VAR_GLOBAL PERSISTENT RETAIN

        sLib_stations : ARRAY[0..MAX_GAS_STATIONS - MIN_GAS_STATION] OF GasFcStation;
        sLib_meters   : ARRAY[0..MAX_METERS - MIN_METER] OF DualFcMeter;
        sLib_cfxProductCode : ARRAY[0..MAX_METERS - MIN_OIL_METER] OF STRING(47);
        bhiLibMemArea : ARRAY[0..BHI_DUALFCLIB_MEM_AREA_ARRAY_MAX_VALUE] OF BYTE;

END_VAR
```



Add necessary variables to the Program which calls the library function block

The following illustrates how the library can be used with a Structured Text Program. The code is reproduced below the screen capture. The following example code assumes that the PLC_PRG program will call the library.




```
VAR
        ////////////////////////////////////////////////////////////////////
        // THESE VARIABLES ARE REQUIRED FOR INTERFACE TO THE BHI LIBRARY
        // AND TO PROVIDE THE LIBRARY WITH PERSISTENT MEMORY TO WORK WITH
        ////////////////////////////////////////////////////////////////////
        fbCyclic : BhiLibDualFC.FbCyclic;
        ptr_bhiLibMemArea : POINTER TO BYTE := ADR(PersistentVars.bhiLibMemArea);

        ////////////////////////////////////////////////////////////////////
        // THESE VARIABLES ARE USED IN THE SAMPLE CODE TO PASS VALUES
        // BETWEEN THIS HOST PROGRAM AND THE LIBRARY
        // they may not ALL be necessary - depending on how many of the meters you are using
        ////////////////////////////////////////////////////////////////////
        (***  value determined by the host via I/O or MODBUS & passed to library ***)
        rGasMeterStaticPressure_psia : REAL;  // gas meter "live" static pressure (as REAL)
        rGasMeterDifferentialPressure_inH2O : REAL;  // gas meter "live" differential pressure (as REAL)
        rGasMeterTemperature_F : REAL;   // gas meter static "live" temeprature (as REAL)
        cGasMeterValues : GasFcCurrentGasMeterValues;       // struct used to retrieve "live" values
from the library
        cGasMeterConfig : GasFcCurrentConfig; // struct used to retrieve configuration values from the
library

        // the following are arrays because there are two liquid meters: liq meter 2 and liquid meter 2
        rOilMeterPressure_psig : ARRAY[2..3] OF REAL;       // oil meter "live" flowing fluid pressure
at meter(as REAL)
        rOilMeterTemperature_F : ARRAY[2..3] OF REAL;       // oil meter "live" flowing fluid
temperature at meter (as REAL)
```

```
        rOilMeterSandW_percent : ARRAY[2..3] OF REAL;        // oil meter S&W percent (as REAL)
        rOilMeterDensity_API : ARRAY[2..3] OF REAL;          // oil meter "live" flowing density at meter
(as REAL)
        // NOTE: drive gain is optional.  Not required for calculations
        rOilMeterCoriolisDriveGain : ARRAY[2..3] OF REAL;    // oil meter "live" meter drive gain value
(as REAL)
        lrOilMeterAccumulatedPulses : ARRAY[2..3] OF LREAL;  // oil meter "live" meter pulse accumualtion
value (as LREAL)
        cOilMeterValues : ARRAY[2..3] OF DualFcCurrentOilMeterValues; // struct used to retrieve "live"
values from the library

        /////////////////////////////////////////////////////////////////////
        // THIS VARIABLE IS USED TO CONTROL THE FLOW OF THE MAIN PROGRAM
        // ON FIRST TASK CYCLE, CERTAIN INITIALIZATION/HOUSEKEEPING TASKS MUST BE DONE - JUST ONCE
        /////////////////////////////////////////////////////////////////////
        xIsInitialized : BOOL := FALSE;


        /////////////////////////////////////////////////////////////////////
        // THESE VARIABLES ARE USED BY THE SAMPLE PROGRAM TO SIMULATE FLOW
        /////////////////////////////////////////////////////////////////////
        lrSimulatedOilMeterAccum : LREAL;
        lrSimulatedOilMeterKnownAccum : LREAL;
        sendGasDp : REAL;

        cycleCount : INT := 0;
        waitCount : INT := 0;

END_VAR
```

Add supporting Code to the Program

The following illustrates how the library can be used with a Structured Text Program. The code is reproduced below the screen capture.

```
IF xIsInitialized THEN

        /////////////////////////////////////////////////////////////////////////////////////////
        // YOUR CODE TO ACCESS I/O OR COMMUNICATIONS NETWORK TO GET "LIVE" METER VALUES
        /////////////////////////////////////////////////////////////////////////////////////////

        // the example below uses values entered on the main visualization

        IF (waitCount > 35) THEN
                // update the value that we send to the library
                // simulates scanning of a meter with Modbus every 35/20ths of a second
                lrSimulatedOilMeterKnownAccum := lrSimulatedOilMeterAccum;
                waitCount := 0;
        ELSE
                waitCount := waitCount + 1;
        END_IF;

        cycleCount := cycleCount + 1;
        IF cycleCount > 1000 THEN
                cycleCount := 0;
        END_IF;

        // this code uses the input values from the visualization screen to populate the variables
        // which will be used to pass live values to the library
        // there is a little twist here because this sample program uses a checkbox on the visualization
        // to either have flow or to stop flow to all meters at the same time
```

```
        rGasMeterStaticPressure_psia := g_gasSp;

        IF (g_bFlowing) THEN
                rGasMeterDifferentialPressure_inH2O := g_gasDp;
        ELSE
                 rGasMeterDifferentialPressure_inH2O := 0.0; // simulate no DP
        END_IF;

        rGasMeterTemperature_F := g_gasT;


        IF (g_bFlowing) THEN
                // simulation of a moving accumulated quantity
                // this sample program uses the same accumulator for both oil meters
                lrSimulatedOilMeterAccum := lrSimulatedOilMeterAccum + g_accumIncr;
        ELSE
                 ;
                // and don't increment the totalizer
        END_IF;

        rOilMeterPressure_psig[2] := g_oilSp[2];
        rOilMeterTemperature_F[2] := g_oilT[2];
        rOilMeterSandW_percent[2] := g_oilSwPercent[2];
        rOilMeterDensity_API[2]       := g_oilDensity_API[2];
        rOilMeterCoriolisDriveGain[2]:= g_oilDriveGain[2];
        lrOilMeterAccumulatedPulses[2] := lrSimulatedOilMeterAccum;

        rOilMeterPressure_psig[3] := g_oilSp[3];
        rOilMeterTemperature_F[3] := g_oilT[3];
        rOilMeterSandW_percent[3] := g_oilSwPercent[3];
        rOilMeterDensity_API[3]       := g_oilDensity_API[3];
        rOilMeterCoriolisDriveGain[3]:= g_oilDriveGain[3];
        lrOilMeterAccumulatedPulses[3] := lrSimulatedOilMeterAccum;


        ////// end of example code used to simulate inputs //////

        /////////////////////////////////////////////////////////////////////////////////////
        // YOUR MAIN PROGRAM MUST CALL THE "...Update<>MeterLive" METHODS ON EACH PASS
        // FOR EACH METER THAT YOU ARE USING (up to 1 gas meter and up to 2 oil/liquid meters)
        // USING THE LATEST-AVAILABLE LIVE METER VALUES
        /////////////////////////////////////////////////////////////////////////////////////

        BhiLibDualFc.FuUpdateGasMeterLive(1, // gas meter number is 1
                                        rGasMeterStaticPressure_psia, // "live" value - most recently
acquired value
                                        rGasMeterTemperature_F, // "live" value - most recently acquired
value
                                        rGasMeterDifferentialPressure_inH2O, // "live" value - most
recently acquired value
                                        0); // pulse count is always zero for AGA-3 meters.  This is for
future functionality


        BhiLibDualFc.FuUpdateOilMeterLive(2, //meter number
                                             rOilMeterPressure_psig[2], // pressure
                                             rOilMeterTemperature_F[2], // temperature
                                             rOilMeterSandW_percent[2], // S&W percent
                                             lrOilMeterAccumulatedPulses[2], // LREAL: accum (or flow
rate - depending on meter configuration)
                                             rOilMeterDensity_API[2], // density API
                                             rOilMeterCoriolisDriveGain[2]); // coriolis drive gain
(pass zero if N/A)
        BhiLibDualFc.FuUpdateOilMeterLive(3, //meter number
                                             rOilMeterPressure_psig[3], // pressure
                                             rOilMeterTemperature_F[3], // temperature
                                             rOilMeterSandW_percent[3], // S&W percent
```

```
                                          lrOilMeterAccumulatedPulses[3], // LREAL: accum (or flow
rate - depending on meter configuration)
                                          rOilMeterDensity_API[3], // density API
                                          rOilMeterCoriolisDriveGain[3]); // coriolis drive gain
(pass zero if N/A)

        ///////////////////////////////////////////////////////////////////////////////
        // YOUR MAIN PROGRAM MUST CALL THE "fbCyclic METHOD ON EACH PASS
        // THIS CALL IS MADE ONCE AND ONLY ONCE PER TASK CYCLE
        // THIS CALL ALLOWS THE LIBRARY TO HAVE CPU CYCLES TO EXECUTE ITS LOGIC
        ///////////////////////////////////////////////////////////////////////////////

        fbCyclic();

        ///////////////////////////////////////////////////////////////////////////////
        // THE "Get<>" calls are optional
        // THESE CALLS ALLOW YOU TO RETIREVE "LIVE" VALUES FROM THE LIBRARY FOR A SPECIFIC METER
        // WHICH YOU CAN USE IN YOUR PLC PROGRAM
        ///////////////////////////////////////////////////////////////////////////////

        // OPTIONAL: get live meter values from the gas meter (meter 1)
        // cGasMeterValues is a struct
        cGasMeterValues := fbCyclic.GetGasMeterValues(1);

        // OPTIONAL: get live meter values from the first oil/liquid meter (meter 2)
        cOilMeterValues[2] := fbCyclic.GetOilMeterValues(2);

        // OPTIONAL: get live meter values from the second oil/liquid meter (meter 3)
        cOilMeterValues[3] := fbCyclic.GetOilMeterValues(3);

        // OPTIONAL: IF YOUR PROGRAM NEEDS TO READ OR MODIFY THE CONFIGURATION
        // OF A METER, YOU CAN DO THIS BY ACCESSING THE
        // PersistentVars.sLib_meters STRUCT ARRAY OF STRUCTS (FOR METERS)
        // ... but you have to use an index of 1 less than what is intuitive..
        // ... (that is: you have to treat the meter array as being zero-indexed)
        // gas meter is at [0], first oil meter is at [1], second oil meter is at [2]
        // EXAMPLES:
        // If you want to read or change the orifice plate size for the GAS meter,
        //              read or assign to PersistentVars.sLib_meters[0].plate_coneSize_in
        // If you want to read or change the meter factor for the FIRST oil meter
        //              read or assign to PersistentVars.sLib_meters[1].meterFactor
        // If you want to read or change the meter contract hour for the SECOND oil meter
        //              read or assign to PersistentVars.sLib_meters[2].oilContractHour

        // OPTIONAL: IF YOUR PROGRAM NEEDS TO <<<READ>> THE CONFIGURATION OF THE GAS STATION
        // (where gas quality, standard conditions, and contract hour are stored)
        // read the PersistentVars.sLib_stations ARRAY OF STRUCTS (<<< at index zero>>>)
        // EXAMPLE:
        // If you want to READ the Methane Percent for the GAS meter,
        // read PersistentVars.sLib_stations[0].Methane_pcnt

        // OPTIONAL: IF YOUR PROGRAM NEEDS TO <<<WRITE>> THE CONFIGURATION OF THE GAS STATION
        // (where gas quality, standard conditions, and contract hour are stored),
        // THE APPROACH DEPENDS ON WHAT YOU WANT TO CHANGE.  FOR GAS QUALITY INFO, A SPECIAL
        // PROCEDURE MUST BE FOLLOWED.  FOR OTHER INFO, YOU CAN JUST ASSIGN A VALUE.
        // EXAMPLE:
        // If you want to WRITE the gas meter atmospheric pressure,
        // simply assign a value to   PersistentVars.sLib_stations[0].atmosphericPressure_psia
        //
        // but if you want to WRITE <<gas quality info>>,
        // you must use the GetGasQuality method to populate a struct containing all gas quality info
        // then you modify the struct
        // then you pass the modified struct to the SetGasQuality method
        // EXAMPLE:
        // declare a variable of the proper type:
        // gasQualityInfo : UIGasQuality;
        // call the "get" (using a zero-indexed meter value of zero for the first gas meter):
        // ex: g_bOperationSuccess := fbCyclic.GetGasQuality(0,ADR(gasQualityInfo));
```

```
        // modify the struct:
        // ex: gasQualityInfo.Methane_pcnt := 27.6;
        // call the "set"(using a zero-indexed meter value of zero for the first gas meter):
        // ex: g_bOperationSuccess := fbCyclic.SetGasQuality(0,ADR(gasQualityInfo));
        // Note: when calling SetGasQuality, you are responsible for ensuring that total composition =
100.0%
        //              If the gas composition does not toal 100% or the heating value is < 500,
        //              the operation will fail and no changes will be made in the library

        //example code below...
        IF (g_bGetQuality) THEN
                g_bOperationSuccess := fbCyclic.GetGasQuality(0,ADR(g_gasQualityInfo));
                g_bGetQuality := FALSE;
        END_IF

        // make changes to the gasQualityInfo struct on the visualization

        IF (g_bSetQuality) THEN
                g_bOperationSuccess := fbCyclic.SetGasQuality(0,ADR(g_gasQualityInfo));
                g_bSetQuality := FALSE;
        END_IF
        // end example code for get quality, set quality

//!!! DO NOT ATTEMPT TO WRITE DIRECTLY TO THE PersistentVars.sLib_stations ARRAY OF STRUCTS !!!!
        // IF YOU DO WRITE TO THIS STRUCT, YOUR VALUES WILL BE OVER-WRITTEN BY THE LIBRARY
        //
        // TO CHANGE THE GAS STATION INFO, USE THE DEDICATED FUNCTIONS:



ELSE
        // one-time initialization activities


        /////////////////////////////////////////////////////////////////////////////////////
        // THE HOST PROGRAM !!! MUST !!! CALL AGA8_SetupGERG AND FuInitialize BEFORE CALLING
        // ANY OTHER FUNCTIONS OR METHODS IN THE LIBRARY
        /////////////////////////////////////////////////////////////////////////////////////

        // Intitialize variables for gas property calculations
        BhiLibDualFC.AGA8_SetupGERG();

        // pass in pointers to persistent memory areas that will be used by the library
        // the library does not allocate its own persistent memory.  Your host program
        // must allocate the persistent memory and then alloe the library to utilize that memory

        BhiLibDualFC.FuInitialize(ADR(PersistentVars.sLib_stations),
                                  ADR(PersistentVars.sLib_meters),
                                  ADR(PersistentVars.sLib_cfxProductCode),
                                  ptr_bhiLibMemArea,
                                  SIZEOF(PersistentVars.bhiLibMemArea));

        xIsInitialized := TRUE; // set the flag so the execution does not go into this branch again

        // the next two lines are initializations of sample code variables.  Not required in your program
        g_accumIncr := 0.00417; //1.0/100.0;
        lrSimulatedOilMeterAccum := 0.0;


END_IF;
```

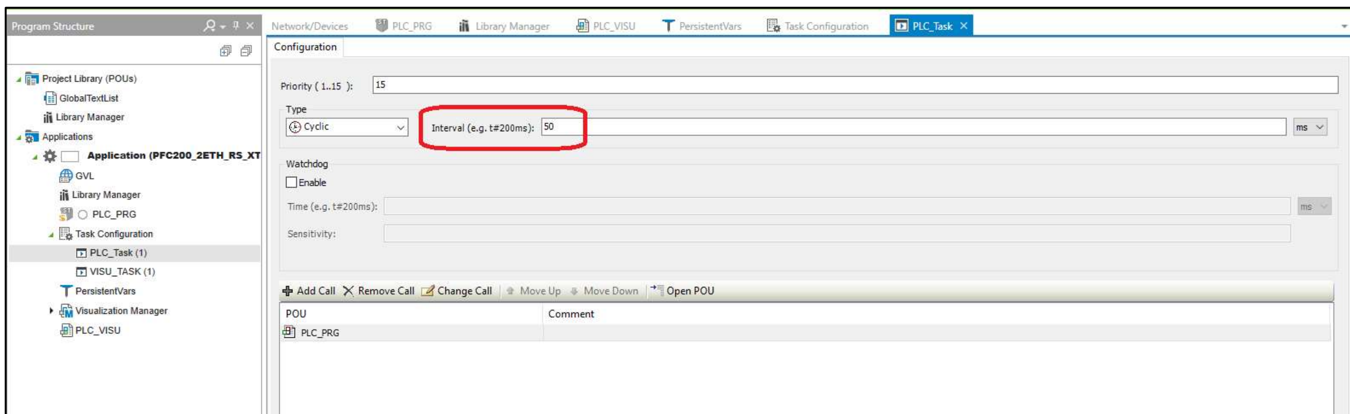
Link Program Visualizations to Library Visualizations

The library includes a number of visualizations for interacting with the flow computer functions. The library visualizations can all be accessed from BhiLibDualFC.visFcDualMain.

The screen capture below shows an example of a simple visualization containing a single button linking to the flow computer menu Visu.

Adjust e!COCKPIT project Task Interval

The library must be called by your program by a task with execution interval of ***50 msec***.



.

Licensing

The BhiLibDualFc library utilizes runtime licensing. Each PLC upon which it runs must have a license. Licenses ae obtained from Beyond HMI, Inc.

Trial Mode

Upon startup, the library will run in trial mode for approximately 4 days. While in trial mode, the library is fully functional. After the 4 day period passes – and if no license is installed - the library will stop calculating flow.

If the PLC program is stopped and restarted, the 4 day trail period begins again. Therefore, PLC program developers should be able to develop and test programs without needing a license for their development PLCs.

Steps to Obtain a Runtime License

To fully license the BhiLibDualFc library on a PLC, the following steps must be executed:

- Include library features in a PLC program (*reference other instructions for PLC program developers within in this document*)
- Install the PLC program on the target PLC specimen
- Open the library's Admin screen and capture the Site Code
- Transmit the site code to Beyond HMI, Inc. and provide payment information
    - Please use info@beyond-hmi.com to initiate contact with us.
- Wait for Beyond HMI, Inc. to return a license file
- Install the license file in the PLC's /home/user/ directory
- Open the library's Admin screen and confirm that the license check result is green

Licenses are perpetual. No maintenance fee is required. Licenses are keyed to a site code and are not portable between PLCs. Please contact Beyond HMI if you need to move a license to another PLC.

Modifying your PLC program without corrupting library data

Your PLC program will inevitably need to be modified – possibly after physical flow has already been accumulated by the flow computer library. Certain changes to your PLC program (adding persistent variables, for example) can cause data in the library's memory to be cleared or corrupted. Beyond HMI has developed tools to support changing your PLC program without losing accumulated volume in the BHI library. The following section describes the procedure you should follow to maintain the integrity of your flow data while making PLC programming changes:

Note: The following steps must be executed in order. Please read and study the entire procedure list before beginning PLC program maintenance.

Stop physical flow

In order to prevent loss of accumulated flow, all processes for all meters must be shut-in to prevent flow while the PLC program is being maintained.

Failure to follow this step may result in lost flow accumulation.

Save the library configuration to file

Using Admin features, save the configuration to file. Make note of the file name you use when saving.

Save a maintenance file

Check the **Save Maintenance File** checkbox on the Advanced Admin screen. Wait for the checkbox to be unchecked. This indicates that a maintenance file has been saved to the PLC file system.

Perform PLC program maintenance

At this point, you are free to make changes to the PLC program and load those changes onto the PLC.

Restore Library configuration

Using Admin features, load the configuration file that you previously saved.

Force Maintenance Recovery

Check the **Force Maintenance Recovery** checkbox on the Advanced Admin screen. Wait for the checkbox to be unchecked. This indicates that meter accumulators have been recovered from the maintenance file.

Resume physical flow

At this point, process flow can be resumed without loss of accumulated volume.

How your program can interact with the Library

In addition to the requirements of initializing the library and passing live meter readings to the library, your program code can interact with the BhiLibDualFc library to:

- Read current values of calculated parameters for each meter
- Read meter and station configuration information
- Writing meter and station configuration Information

The following sections provide further detail about how to execute these interactions from your program code.

Reading Current Calculated Values from your program

Reading Gas Meter Values

Use the *fbCyclic.GetGasMeterValues* method to read current values from the meter.

This method takes a single input parameter:

- meter number (always 1 for the gas meter in the BhiLibDualFc library)

The method returns a structure of type *GasFcCurrentGasMeterValues* . This structure includes members which can be read to determine current values from the calculations, such a mass, volume and energy flow rates, and mass volume and energy accumulations for hourly, daily, and monthly periods.

Reading Oil Meter Values

Use the *fbCyclic.GetOilMeterValues* method to read current values from the meter.

This method takes a single input parameter:

- meter number (use 2 for the first oil/liquid meter and 3 for the second oil/liquid meter)

The method returns a structure of type *DualFcCurrentOilMeterValues*. This structure includes members which can be read to determine current values from the calculations, such a gross/net volume, etc. flow rates and accumulations for hourly, daily, and monthly periods.

Reading Meter Configuration Values from your program

Your program can directly read meter configuration parameters (such as gas meter orifice plate size or oil meter meter factor). You do this by reading from the members of the *sLib_meters* array that you declared in persistent memory.

You must use an index of 1 less than what is intuitive. That is: you have to treat the meter array as being zero-indexed. The gas meter is at [0], the first oil meter is at [1], the second oil meter is at [2].

The following examples illustrate the approach…

- If you want to read the orifice plate size for the GAS meter, read PersistentVars.sLib_meters[0].plate_coneSize_in.

- If you want to read the meter factor for the FIRST oil meter, read PersistentVars.sLib_meters[1].meterFactor.
- If you want to read the meter contract hour for the SECOND oil meter, read PersistentVars.sLib_meters[2].oilContractHour

Reading Station Configuration Values from your program

Your program can directly read station configuration parameters (such as gas meter contract hour or gas quality information). You do this by reading from the members of the *sLib_stations* array that you declared in persistent memory.

In the BhiLibDualFc library, there is only one gas meter station, and it is accessed at index zero. For example, If you want to READ the Methane Percent for the GAS meter, read PersistentVars.sLib_stations[0].Methane_pcnt.

Writing Meter Configuration Values from your program

Your program can directly assign values to meter configuration parameters (such as gas meter orifice plate size or oil meter meter factor). You do this by reading from or assigning to the members of the *sLib_meters* array that you declared in persistent memory.

You must use an index of 1 less than what is intuitive. That is: you have to treat the meter array as being zero-indexed. The gas meter is at [0], the first oil meter is at [1], the second oil meter is at [2].

The following examples illustrate the approach…

- If you want to modify the orifice plate size for the GAS meter, assign the new value to PersistentVars.sLib_meters[0].plate_coneSize_in.
- If you want to modify the meter factor for the FIRST oil meter, assign the new value to PersistentVars.sLib_meters[1].meterFactor.
- If you want to modify the meter contract hour for the SECOND oil meter, assign the new value to PersistentVars.sLib_meters[2].oilContractHour

Writing Station Configuration Values from your program

If your program needs to modify the configuration of the gas station (where gas quality, standard conditions, and contract hour are stored), the approach depends upon what you need to modify. For gas quality information, a special procedure must be followed. For other station parameters, you can just assign values to the parameter. For example, If you want to modify the gas station atmospheric pressure, simply assign a value to PersistentVars.sLib_stations[0].atmosphericPressure_psia

But if you want to modify gas quality information (gas composition, heating value, etc.) you must use the GetGasQuality method to populate a struct containing all gas quality info. Then you modify the struct. Finally, you pass the modified struct to the SetGasQuality method

For example:

You declare a variable of the proper type:

Example code: gasQualityInfo : UIGasQuality;

Then you call the "get" (using a zero-indexed station number value of zero for the first gas station):

Example code: g_bOperationSuccess := fbCyclic.GetGasQuality(0,ADR(gasQualityInfo));

Then you modify the struct:

Example code: gasQualityInfo.Methane_pcnt := 27.6;

Finally, you call the "set" (using a zero-indexed station number value of zero for the first gas station):

Example code: g_bOperationSuccess := fbCyclic.SetGasQuality(0,ADR(gasQualityInfo));

> *Note: When calling SetGasQuality, you are responsible for ensuring that total composition = 100.0%. If the gas composition does not total 100% or the heating value is < 500, the operation will fail and no changes will be made in the library.*