Application Note 1201-017

QD Software Development Kit

Introduction

The Quantum Design compressor Software Development Kit (SDK), is a set of tools that enable the creation of custom software applications that automate the control of the helium compressors in third party instruments. The compressor SDK was developed using C# in the .NET framework, utilizing a polymorphic design. It provides an easy interface for controlling the compressor functions, without the need of in-depth knowledge of low level controls. This SDK is provided to customers as an easy to use Dynamic Link Library (DLL). Also, a Graphical User Interface (GUI) application included in the kit, provides examples of using the SDK including how to interface of any of the Quantum Design compressor products to a PC.

SDK Components

The SDK consists of two public classes within the *QdCompressorSDK* namespace. The first class, called ControlComp provides the essentials for controlling the compressor. The second class AdvControlComp implements all of the features found in ControlComp and provides additional advanced features.

In order to not have to qualify the full class name, one can use the "using" directive to include the namespaces, as shown below:

```
using QdCompressorSdk;
```

Basic Interface

The "basic interface" is implemented by the *ControlComp* class which contains only the necessary commands for controlling the compressor in each of its run modes. There are a few custom data types used in *ControlComp* as input and return arguments. The AdvControlComp class includes basic operation and some diagnostic information.

Common Features: run modes

The *CompressorMode* is an enumeration of compressor operational modes. Table 1.1 shows the various modes and their applicability to a particular compressor type:

LowSpeed, NormalSpeed, and HighSpeed correspond to three power levels available in the compressors. The speeds at which the compressor capsule and cold head drive motor typically run in these situations are described in Table 1.2:

Table 1.1 Compressor Modes

Mode	HAC900S	HAC900	HAC4500	HLC4500
HighSpeed	YES	YES	YES	YES
NormSpeed	YES	YES	YES	YES
LowSpeed	YES	YES	YES	YES
CustomSpeed	YES	YES	YES	YES
PidMode	YES	NO	NO	NO
PdoSpeed	NO	NO	NO	YES

Table 1.2 Simplest Compressor Modes

Power Level	Scroll Compressor Speed	Cold Head Speed
HAC900/HAC900S		
High	30	70
Normal	18	50
Low	13	30
HAC4500/HLC4500		
High	60	70
Normal	40	60
Low	30	50

With *CustomSpeed*, the user specifies a head speed and compressor speed, in terms of frequency, and in units of Hz. The head speed describes the movement of the displacer in the cold head. The compressor speed refers to the frequency of the inverter. Using faster speeds yields faster cooling.

With *PidMode* (only available to the *HAC900S*), the firmware controls both the head and compressor capsule speeds using a PID control algorithm. The process variables used are the first-stage and second-stage temperatures. This mode depends upon set-points, easily set by an object command, as well as PDOs (CANOpen Process Data Objects) reporting the current first-stage and second stage temperatures. For example if the HAC900S is used with a Qauntum Design PPMS Versalab, than the Versalab cryostat controller sends the

required PDOs. If a Versalab is not present, an alternative means must be provided to report the required temperatures. Commands, discussed later, exist for specifying the required PDOs reporting the first-stage and and second stagetemperatures, as well as specifying the first-stage and second stage set-point temperatures. By default, the set-point temperatures are 40 K and 4 K respectively.

Note that *PdoSpeed* is only applicable to the HLC4500. It is similar to *PidMode* in that it adaptively adjusts speeds based upon PDO input.

Basic Interface: data types

The *CompressorMode* enum describes the different compressor operating modes.

```
// Summary: An enumeration of the compressor run modes
enum CompressorMode
{
	LowSpeed    = 1,     // lowest speed setting
	MedSpeed    = 2,     // midway between low and highest speed
	HighSpeed   = 3,     // highest speed setting
	PdoSpeed    = 4,     // adaptive temperature control using PDOs
	CustomSpeed = 5,     // for specifying custom speeds
	PidMode     = 7      // PID temperature-control mode
}
```

Common Features: run status

Another public data type is an enumeration of the run status, shown below. It simply describes if the compressor is “Not Running”, “Running”, or “Only the Cold-head is running”.

```
// Summary: An enumeration of the compressor run status
enum RunStatus
{
	NotRunning      = 0, // Compressor is not running
	Running         = 1, // Compressor and cold-head are both running
	OnlyHeadRunning = 2  // Only the cold-head is running
}
```

In order to obtain the enumerated “Run Status”, the following command is used.

```
// Summary: Get the current run status as a RunStatus type
// Returns: The run status as an enum
RunStatus GetRunStatus();
```

Common Features: error codes

There is also a series of error codes that are constant and publicly available strings. Each code provides a description of the error, and is shown below.

```
// Summary: Indicates no error for last command
const string ERROR_NONE = "No Error";
// Summary: Indicates an error for last command
const string ERROR_REMOTE = "Remote mode must be enabled for this operation.";
// Summary: Indicates an error for last command
const string ERROR_CUSTOM_MODE = "The CustomSpeed mode must be selected.";

// Summary: Indicates an error for last command
const string ERROR_COBID = "The cobID is out of range [0x181 to 0x57F]";
// Summary: Indicates the requested speed is out of range
const string ERROR_SPEED = "The speed is out of range";
// Summary: Indicates the requested speed ratio is invalid
const string ERROR_SPEED_RATIO = "The head/comp speed ratio is invalid.";
// Summary: Indicates an error occurred during CAN i/o.
            An error code is appended to the error string.
const string ERROR_CAN = "CAN Error ";
```

For obtaining the current error code, the below function is used, which returns one of the error codes described above.

```
// Summary: Obtain the current command error code
string GetErrorCode();
```

Common Features: constructors

There are two constructors for the *ControlComp* class, one of which contains the argument “nodeId”, for specifying the CAN node ID of the HAC900S. By default, the CAN node ID is 4, and if this has not changed, the argument-free constructor can be used.

```
// Summary: Argument-free ControlComp constructor
ControlComp();

// Summary: ControlComp constructor with CAN node ID as an argument
ControlComp(byte nodeId = DefaultNodeId);
```

Common Features: basic operation

In order to control the compressor from anywhere other than the front panel, “Remote Mode” must be set. This is accomplished with the function below. If a command is attempted for which “Remote Mode” is required, and it is not enabled, the *ERROR_REMOTE* code will be set.

```
// Summary: Enable/Disable remote control mode
// Param "doEnable": true = enable, false = disable
// Returns: true = success, false = failure
bool SetRemoteModeEnable(bool doEnable);

// Summary: Check if "remote control mode" is enabled
// Returns: true = enabled, false = disabled
bool IsRemoteModeEnabled();
```

The *CompressorMode* described by the enumerated type discussed above is set with the following function.

```
// Summary: Set the compressor run mode type, if "remote mode enabled" is true
// Param "compMode": Mode selected from CompressorMode enum
// Returns: true = success, false = failure
bool SetRemoteModeType(CompressorMode compMode);

// Summary: Get the current compressor run mode as CompressorMode type
// Returns: The current compressor operational mode as an enum
CompressorMode GetRemoteModeType();


 // Summary: Get the current compressor run mode as a string
// Returns: The current compressor operational mode as a string
string GetRemoteModeString();
```

In order for the compressor to actually run, “Run Enable” must be set, and is accomplished by the following command. Note that “Remote Mode” must be enabled in order for this function to work.

```
// Summary: Enable/Disable the compressor to run if "remote control mode" is enabled

// Param "doEnable": true = enable, false = disable
// Returns: true = success, false = failure
bool SetRunEnable(bool doEnable);

// Summary: Check if "run" is enabled
// Returns: true = enabled, false = not enabled
bool IsRunEnabled();
```

The run time is the time in seconds that the compressor has been continuously running. If the compressor is rebooted, this timer resets to zero.

```
// Summary: Get the current run duration in seconds
// Returns: The time the compressor has been continuously running in seconds
int GetRunTime();
```

Common Features: commands for "custom speed" mode

In "Custom Speed" mode, the compressor and cold-head speeds need to be set to the desired custom values. The commands shown below are provided for this. The "Set" functions will return false if the provided arguments are incorrect, or the compressor is not in "Custom Speed" mode. Once both the custom head and custom compressor speeds have been selected, *ActivateCustomSpeeds()* must be called for the new speeds to take effect.

```
 // Summary: Set the custom cold-head speed (Hz) if "custom speed mode" is selected
// Param "freqHz": Frequency in Hz
// Returns: true = success, false = failure
bool SetCustomHeadSpeed(float freqHz);

// Summary: Get the current cold-head speed (Hz) setting
// Returns: The cold-head set-point speed/frequency in Hz
float GetCustomHeadSpeed();

// Summary: Set the custom compressor speed (Hz) if "custom speed mode" is selected
// Param "freqHz": Frequency in Hz
// Returns: true = success, false = failure
bool SetCustomCompSpeed(float freqHz);

// Summary: Get the current compressor speed (Hz) setting
// Returns: The current set-point compressor/inverter speed/frequency in Hz
float GetCustomCompSpeed();

// Summary: Activates the custom speeds set via SetCustomCompSpeed() and SetCustomHeadSpeed()
void ActivateCustomSpeeds();
```

For the *HAC900S* only, the ratio of the head speed to the compressor speed must be at least 2. The function below verifies if this requirement is met. If used with the *HLC4500* or *HAC4500*, *IsSpeedRatioValid()* always returns true, since these compressors are not limited by speed ratio.

```
// Summary: Indicates if the speed ratio, headSpeed/compSpeed, is valid
// Param "headSpeed": Cold-head speed in Hz
// Param "compSpeed": Compressor speed in Hz
// Returns: true = valid, false = invalid
bool IsSpeedRatioValid(float headSpeed, float compSpeed);
```

Common Features: speed and pressure

The following commands are provided for reporting the actual compressor and cold-head speeds.

```
        // Summary: Get the actual compressor speed (Hz)
        // Returns: The actual compressor speed/frequency in Hz
float GetActualCompSpeedHz();

        // Summary: Get the actual cold-head speed (Hz)
        // Returns: The actual cold-head speed/frequency in Hz
        float GetActualHeadSpeedHz();
```

Finally, functions are provided for obtaining the compressor supply and return pressures in units of mega-Pascal.

```
       // Summary: Get the current supply pressure (MPa)
       // Returns: The supply pressure in MPa
       float GetSupplyPressureMPa();
       /// <summary>Get the current return pressure (MPa)</summary>
       /// <returns>The return pressure in MPa</returns>
       float GetReturnPressureMPa();
```

Common Features: commands for PID mode (HAC900S)

In PID mode (*HAC900S* only), the compressor uses both the first-stage temperature and second-stage temperature as control variables. Two PDOs (process data objects – a CANOpen mechanism) are sent from an external module over the CAN bus in order to report the actual first-stage and second-stage temperature.

If the compressor is used with a Versalab, these PDOs, which associate the first-stage temperature to the internal gas switch temperature and the second-stage to the superconducting magnet temperature, are sent automatically by the Versalab. However, if PID control is desired for a compressor without a Versalab, then one must provide a module that obtains the first-stage and second-stage temperatures. If this data is available, the *TransmitTempPDOs* function can be used to send the data to the compressor, allowing it to run in PID mode. The section called “Providing Actual Temperatures to the SDK” discusses how to provide the required temperature measurements to the SDK.

```
        // Summary: Transmit PDOs reporting actual first-stage and second-stage temperatures to the
        compressor
        // Param "firstStageTempK": Actual first-stage temperature (K)>
        // Param "secStageTempK": Actual second-stage temperature (K)
        void TransmitTempPdos( float firstStageTempK, float secStageTempK);
```

For setting and viewing the set-point temperatures for PID mode, the following functions are provided. The default set-points, set in the compressor firmware, are 4 K for the second-stage and 40 K for the first stage.

```
// Summary: Set the desired temperature set-points (K)
// Param "ssTempKSetpoint": Temperature set-point for second-stage
// Param "fsTempKSetpoint": Temperature set-point for first-stage
void ConfigTempSetpoints(float ssTempKSetpoint, float fsTempKSetpoint);

// Summary: Get the current temperature set-points (K)
// Param "ssTempKSetpoint": Set-point temperature for second-stage
// Param “fsTempKSetpoint": Set-point temperature for first-stage
void GetTempSetpoints(ref float ssTempKSetpoint, ref float fsTempKSetpoint);
```

The actual first and second stage temperatures are obtained with the following function. The actual temperatures are only relevant in PID mode, when PDOs that report the actual temperatures are available. If there are no PDOs available, this function will return 0 K for both the first-stage and second-stage.

```
// Summary: Get the actual first-stage and second-stage temperature (K)
// Param "ssTempK": Actual temperature for second-stage
// Param "fsTempK": Actual temperature for first-stage
void GetActualTemps(ref float ssTempK, ref float fsTempK);
```

Advanced Interface

The Advanced Interface inherits all of the features of the basic Interface, and provides some additional diagnostic tools. These tools are unlikely to be desired by a typical customer.

Common Features: additional diagnostics

The firmware version can be obtained via *GetCompressorVersion()*. The *HAC900S* is unique in that it has two sets of firmware. Therefore, the SDK provides *GetHeadVersion()* as well, which provides the version of the cold-head firmware.

```
// Summary: Obtain the compressor firmware version
string GetCompressorVersion();

// Summary: (HAC900S only) Obtain the cold-head firmware version
string GetHeadVersion();
```

An enumeration is provided which describes the system status, and is shown below.

```
// Summary: Enumeration of system status for the compressor
enum SystemStatus
{
    Unknown                 = 0,   // status unknown
    NotRunning              = 1,   // compressor is not running
    AdaptivePower           = 5,   // adaptive power mode selected
```

```
        CustomizedSpeed         = 6,   // custom speed selected
        Startup                 = 7,   // in startup
        OverTemperatureFault    = 10,  // temperature fault occurred
        GasPressureFault        = 11, // gas pressure fault occurred
        OilSystemFault          = 12, // oil system fault occurred
        InverterFault           = 13, // inverter fault occurred
        AdsorberLifeExceeded    = 14, // charcoal adsorber needs to be replaced
        GeneralFailure          = 15  // miscellaneous failure
    }
```

Advanced Interface: constructors

There are two constructors for the *AdvControlComp* class, one of which contains the argument "nodeId", for specifying the CAN node ID of the HAC900S. By default, the CAN node ID is 4, and if this has not changed, the argument-free constructor can be used.

```
// Summary: Argument-free AdvControlComp constructor
AdvControlComp();

// Summary: AdvControlComp constructor, with option to specify CAN Node ID
AdvControlComp(byte nodeId = DefaultNodeId);
```

The system status, as described the *SystemStatus* enum, is obtained with the following function.

```
// Summary: Obtain the system status as a SystemStatus enum type
// Returns: The System Status as an enum
SystemStatus GetSystemStatus();
```

There are several other diagnostic functions described next, pertaining to the several key temperatures, head displacer-motor current, inverter current, inverter power consumption, the oil level, and the oil-flow ratio.

```
// Summary: Obtain the capsule temperature

// Returns: The temperature in Celsius
 float GetCapsuleTempC();

 // Summary: Obtain the helium temperature
 // Returns: The temperature in Celsius

float GetHeliumTempC();

 // Summary: Obtain the oil temperature
 // Returns: The temperature in Celsius
 float GetOilTempC();
```

```
 // Summary: Obtain the current used by the head inverter
 // Returns: The current in Amps
 float GetHeadCurrentA();

 // Summary: Obtain the current used by the compressor inverter
 // Returns: The current in Amps
 float GetCompCurrentA();

// Summary: Obtain the inverter power consumption (kW)
// Returns: The power consumed by the inverter in kW
float GetInverterPowerkW();

// Summary: Obtain the current cold-head oil level (%)
// Returns: The oil level percentage
float GetOilLevelPct();

// Summary: Obtain the current cold-head oil flow ratio
// Returns: The oil-flow ratio
float GetOilFlowRatio();
```

Advanced Interface: timed operation

The SDK provides a set of commands to configure the compressor to run for a specified duration in units of hours.

```
// Summary: Configure the compressor to run for the specified duration in hours
// Param "hours": duration for which to run the compressor
// Returns: true = command was successful, false = command failed due to timer already running,
or "hours" being invalid
bool RunCompForSpecifiedDuration(float hours);
// Summary: Stop the compressor timer. The compressor will continue running if stopped
void StopTimer();

// Summary: Returns the number of hours remaining for the compressor timer
// Returns: The time remaining in hours
float GetTimerRemainingTime();

// Summary: Indicates if the timer is currently running
// Returns: true = running, false = not running
bool IsTimerRunning();
```

Common Features: log file

The SDK also provides commands for logging diagnostic data. The following functions are used for logging data to a CSV file called <compressorType>.csv. The file is logged to the directory in which the .exe using this SDK is located. The default log interval is 3000 milliseconds, but this can be changed using *SetLogInterval*. The information that is logged depends on the compressor type. The *HAC900S* has the most diagnostics and therefore logs more data then *HAC4500* or *HLC4500*.

```
// Summary: Creates a log of important diagnostic values; updates once per second. Log name is
"qdCompLog.csv"
void StartLogging();

// Summary: This stops and closes the diagnostic log. Log name is "qdCompLog.csv”
void StopLogging();

// Summary: Returns the path of the log file
// Returns: Path of the log file location
string GetLogFilePath();

// Summary: Sets the time interval between logged data points
// Param “intervalMs": the time interval (milliseconds)
// Returns: true = command was successful, false = command failed due to invalid "intervalMs"
bool SetLogInterval(ushort intervalMs);
```

HAC900S-Specific Diagnostics

The *HAC900S* contains more diagnostic features than the *HAC4500* or *HLC4500*. A summary of these diagnostics is provided here. Shown below are functions for obtaining the PCB temperature, spool-valve current, and phase delay in degrees between the displacer and spool-valve motion.

```
// Summary: Obtain the cold-head PCB temperature (C)
// Returns: The temperature of the 5-phase board (cold-head controller) in degrees C
float GetBoardTempC();

// Summary: Obtain the cold-head valve current (A)

 // Returns: The current used by the spool valve
float GetValveCurrentA();

// Summary: Obtain the current cold-head valve phase delay setting (degrees
// Returns: The phase delay between the valve and displacer in degrees
float GetPhaseDelayDegrees();
```

There are further diagnostics for viewing the count of motor-synchronization, motor, or valve faults. These are listed below. These should always return 0, unless there is a problem.

```
// Summary: Obtain the number of motor-synchronization faults
// Returns: The count of motor-synchronization faults
uint GetNumMotorSynchFaults();

// Summary: Obtain the number of motor faults
// Returns: The count of motor faults
uint GetNumMotorFaults();
// Summary: Obtain the number of valve faults
// Returns: The count of valve faults
uint GetNumValveFaults();
```

HAC900S Specific: warm-up or cool-down

By design, the HAC900S is capable of cooling or warming up the Quantum Design GA-1 cold head and therefore a cryogenic environment coupled to its first and second stages. By default the compressor is set for cooling. The following functions provides a means of specifying warming or cooling. Currently, the warming feature is disabled, but will be enabled in a future release.

```
// Summary: Configure the cold-head for "warm up" mode
void ConfigForWarmup();

// Summary: Configure the cold-head for "cool down" mode (default)
void ConfigForCooldown();
```

HAC900S Specific: spool-valve diagnostics

Finally, there are several diagnostic functions pertaining to spool-valve movement. Understanding these functions requires low-level knowledge of how the compressor works, and will not be described here.

```
// Summary: Obtain the maximum valve encoder count per valve cycle
// Returns: The maximum position of the valve per cycle in encoder counts
short GetValvePosMax();
// Summary: Obtain the minimum valve encoder count per valve cycle
// Returns: The minimum position of the valve per cycle in encoder counts
short GetValvePosMin();

// Summary: Obtain the duration at which the valve is in the maximum position per cycle (ms)
// Returns: The time (ms) that the valve was in the max position per cycle
ushort GetValveMaxDurationMs();
```

```
        // Summary: Obtain the duration at which the valve is in the minimum position per cycle (ms)

         // Returns: The time (ms) that the valve was in the min position per cycle
        ushort GetValveMinDurationMs();

        // Summary: Obtain the duration at which the valve is in the rising position per cycle (ms)
        // Returns: The time (ms) taken for the valve to transition from the min to max position
        ushort GetValveRiseDurationMs();

        // Summary: Obtain the duration at which the valve is in the falling position per cycle (ms)
        // Returns: The time (ms) taken for the valve to transition from the max to min position
        ushort GetValveFallDurationMs();

        // Summary: Get the ratio, (max valve duration) / (min valve duration). Useful for determining
        if the valve movement is symmetric
        // Returns: The ratio, (max valve duration) / (min valve duration)
        float GetValveMaxMinRatio();

        // Summary: Get the ratio, (rising valve duration) / (falling valve duration). Useful for
        determining if the valve movement is symmetric
        // Returns: The ratio, (rise valve duration) / (fall valve duration)
        float GetValveRiseFallRatio();
```

Basic Interface: Example

Provided below is an example of using the SDK for the HAC900S to set custom compressor speeds, and make the compressor run.

```
        // Create a ControlComp object
        ControlComp comp = new ControlComp();
        // Enable remote control
        comp.SetRemoteModeEnable(true);
        // Set the speed mode to "CustomSpeed"
        comp.SetRemoteModeType(CompressorMode.CustomSpeed);
        // Set the head speed to 60 Hz
        comp.SetCustomHeadSpeed(60);
        // Set the compressor speed to 25 Hz
        comp.SetCustomCompSpeed(25);
        // Start the compressor running

        comp.SetRunEnable(true);
```

Advanced Interface: Example

This example shows how to use the advanced compressor control object to set the compressor to warmup mode, set it to low speed, and make it automatically shut itself off after a specified duration of 24.3 hours.

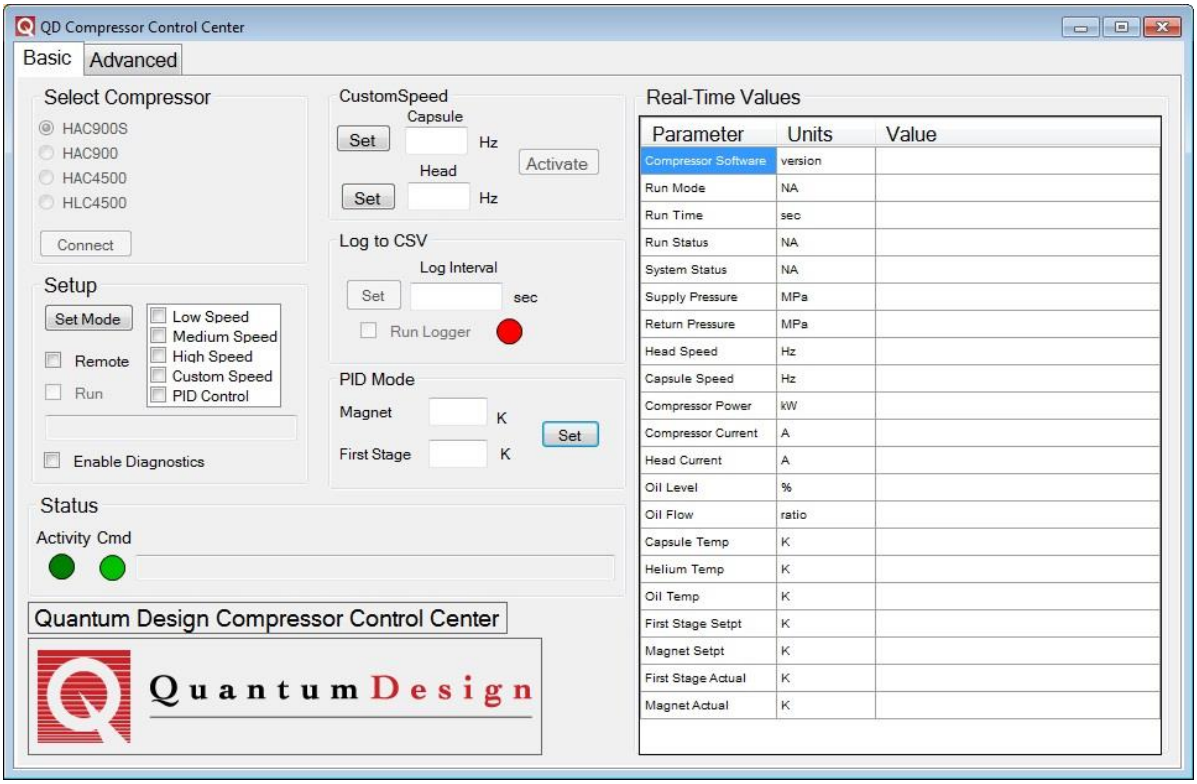
```
// Create a AdvControlComp object
AdvControlComp advComp = new AdvControlComp();
// Configure for warmup mode
advComp.ConfigForWarmup();
// Set the mode to low speed
advComp.SetRemoteModeType(ControlComp.CompressorMode.LowSpeed);
// Tell the compressor to run for 24.3 hours
advComp.RunCompForSpecifiedDuration(24.3);
```

Graphical User Interface (GUI)

A GUI is available that allows for PC-based control of the compressor, and provides a thorough example of using the SDK. The GUI contains two tabs: one for the basic operations, and one for the advanced operations.

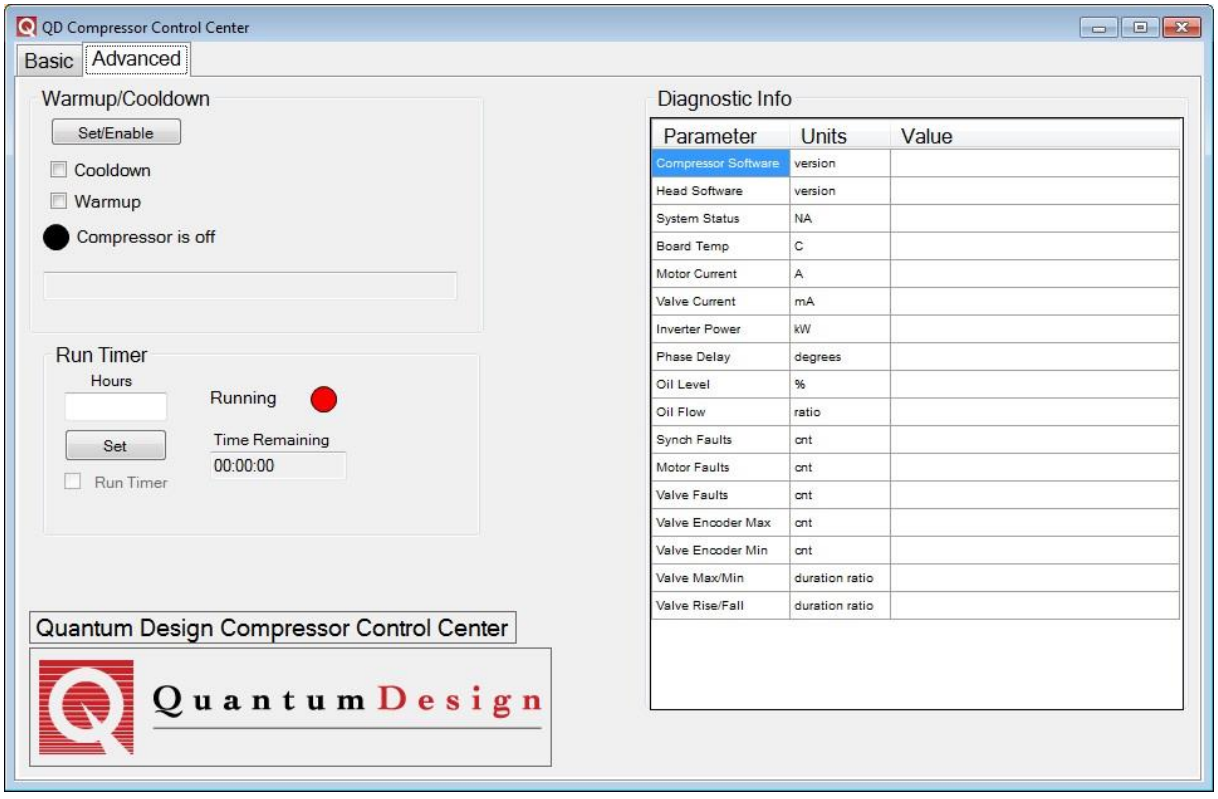
Basic Tab

The desired compressor is selected within the “Basic” tab. Once a compressor has been selected, the grayed-out sections become available. Then, the run mode of the compressor can be selected, as well as enabling the diagnostic information. The GUI code shows how to utilize a data-grid widget to create a table like the diagnostic display of “Real-Time Values”. The Log to CSV contains the real-time variables in comma-separated format, and uses the “Log Interval” to determine how frequently to log data. The default log interval is 1000 milliseconds. The timer causes the compressor to run for the specified duration in hours. The diagnostic info is generated using a data-grid widget.



Advanced Tab

The advanced tab appears after selecting a compressor. It contains a section for “Warmup/Cooldown”, “Run Timer” and “Diagnostic Info”.



Providing Actual Temperatures to the SDK (HAC900S only)

In order for the compressor PID control mode to work, the first-stage and second-stage temperatures must be measured and provided to the compressor firmware. When a compressor is used with a Quantum Design Versalab, the Versalab provides the required temperatures. However, if the compressor is used without a Versalab, another means of measuring and reporting the required temperatures is necessary, if PID control mode is desired.

The compressor SDK provides the command *TransmitTempPDOs*, which can be used to send the required temperatures via PDOs (process data objects) over the CAN bus. Some means of providing the temperatures to the PC running the SDK is required. This could be accomplished with external temperature sensors, used in conjunction with LabVIEW or something similar. The following diagram illustrates this configuration.

Alternatively the PDOs can be sent by any CANopen hardware connected to the CAN network. The PDOs for the first stage and magnet temperatures are COB-ID 0x1C6 and 0x486 respectively. The PDO length is 6 bytes and the content is a 4 byte floating point temperature followed by a 2 byte integer temperature status. Alternate PDOs can be configured using the compressor (node 4) 0x1401 and 0x1402 CAN dictionary entries but the length and content must remain unchanged.