Control System Plant Simulator Users Guide

By Dave Chandler

Table of Contents

1. Introduction

The Control System Plant Simulator (CSPS) is a plant simulation framework designed to allow users to launch simulations of plants that respond to real digital and analog input. The plants defined by the user operate as part of a Hardware-In-The-Loop simulation. That is, a computer running the Control System Plant Simulator, connected to a data acquisition device, behaves to the outside world exactly as the plant it is simulating does. It has inputs that accept signals in the same format as the real physical plant, and produces outputs equivalent to those produced by the sensors that sample the state of the real physical plant. From a black box perspective, the controller can view the CSPS and the physical plant as equivalent.

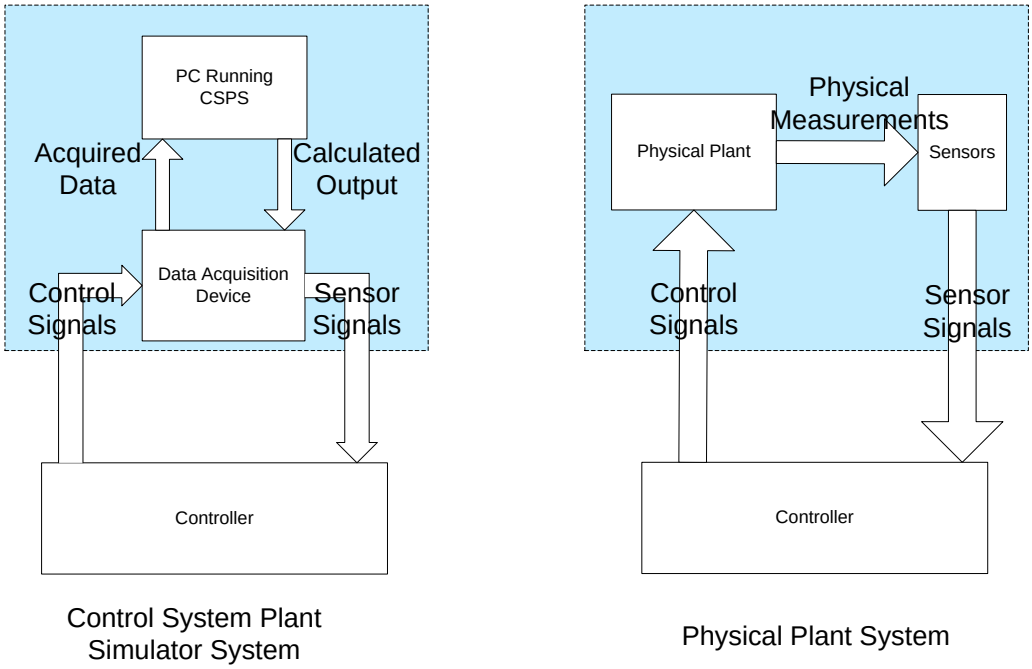

Figure 1: CSPS and Physical Plant as Black Box Systems

Figure 1 demonstrates this concept. Note that the sensor and control signals appear the same in both cases to the Controller. By eliminating the physical plant, the CSPS helps control designers debug their systems at their workstations through the use of a suite of windows applications. This is particularly useful in educational programs, as the availability of real plants or simulations of them may be somewhat limited.

1.1 System Overview

The CSPS is designed as a modular suite of programs to allow for the greatest level of flexibility. It is divided into three separate programs that interact with each other. These process spaces are the Kernel, the Computation Kernel, and the User Interface. The Kernel is the main entry point for the system, and launches the other two processes. It stores all configuration information, fields User Interface requests, and handles communicating with the Windows operating system for file IO. The Computation Kernel is where the actual simulation is run. It is launched in its own process space to allow for real time execution in a system such as RTX. It handles reading from, and writing to the connected data acquisition devices, and simulates the equation of the configured plant. The User Interface is the process with which end users interact. Since the CSPS may be configured to simulate a myriad of physical objects, it is not possible to develop a single user interface for all plants. Some may want a graphical interface that shows the level of fluid in a tank as it drains and is filled, others may want a command line based system. Developers may wish to restrict access to the features provided by the CSPS to just a handful of options, such as a simple start/stop interface with a plant hard coded in place. To allow this level of flexibility, users are expected to create their own user interface that interacts with the CSPS Kernel. This is made simple through the use of the UiWinInterface dynamically linked library.

1.2 Concepts and Definitions

Every plant has **inputs** and **outputs**. Inputs are the values that drive the plant. Controllers provide inputs to plants in order to control their behavior. Outputs are the values that a controller will monitor to determine the behavior of the plant. An input for a DC motor would be the voltage applied. An output would be the speed of the motor as read by a rotational sensor. The CSPS will read inputs from an outside source, and provide outputs back to that same source.

The Control System Plant Simulator manages two different kinds of **ports**. In this context, a port is an interface that allows the CSPS to read data from, or write data to a data acquisition device. The CSPS has what are called **physical ports** and **pseudo ports**. Physical ports are the ports that represent interfaces on a data acquisition device. If a data device provides two analog inputs and four analog outputs, it is said that the device has a total of six physical ports. Data read from or written to a physical port is referred to as **physical** or **raw data**, and is the actual data that will be transmitted back to the connected controller.

Pseudo ports are an abstract concept, internal to the CSPS. Pseudo ports map the physical ports to the inputs and outputs of a defined plant. They also convert the raw physical data into values that are meaningful to the plant. These values are referred to as **engineering values**. For example, suppose a plant had an input value measured in radians, and an output value measured in feet per second. A controlling signal enters the CSPS through a physical port as a voltage reading. The user must define a pseudo port that connects to this physical port, converts the data from voltage to radians, and provides it to the proper input of the plant. Another pseudo port must be defined that takes the speed output from the plant, converts that back into voltage, and provides that voltage to its connected physical port.

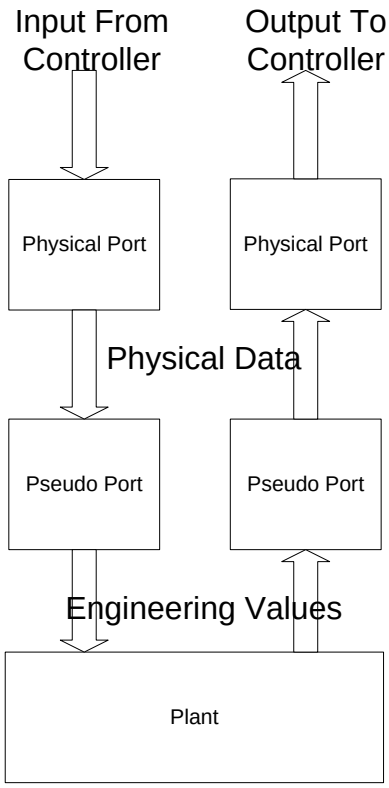

Figure 2: PhysicalPort and PseudoPort Interaction

2. CSPS Operation

The Control System Plant Simulator provides a number of features designed to help users define and configure their plants as completely as possible. The following section provides information on how to operate the Control System Plant Simulator. Note that as the project is provided open source, code will be referenced when necessary to provide a more complete picture of how the system works.

2.1 Framework Startup

The Kernel is the main process for the Control System Plant Simulator. Launching the Windows Kernel starts the other processes and initiates the Control System Plant Simulator. Currently, the CSPS has implemented a Kernel that runs on a Windows XP operating system. To port the Kernel, a designer must only update the

Operating System Abstraction code to use a new operating system. Section 5 provides information on porting the OS Abstraction code. The Kernel is a console process that may be launched from a command prompt. It accepts two command line arguments to indicate which user interface and which computational kernel to launch. Specify the arguments as follows:

```
CSPS_W32_Kernel.exe ui <user interface process name> comp <computation kernel>
```

Note that both ‘ui’ and ‘comp’ are tags required to indicate that a User Interface or Computation Kernel are being specified. If ‘ui’ or ‘comp’ are not provided, the system defaults to the supplied Windows Computation Kernel, and the ExampleUI Visual Basic GUI. This command line can be captured in a windows short cut to allow users quick access to the CSPS using specialized user interfaces.

2.2 Plants

Users must define a plant before simulation may begin. There are a number of ways a plant may be manually entered into the system. Plants may be saved to or loaded from files as well.

2.2.1 Defining a plant manually

A plant may be defined by any of the following methods, though the preferred method is as a set of State Space Matrices.

- As a set of State Space matrices. State space matrices describe the plant according to the following equations:

  $\dot{\boldsymbol{x}}(t) = \mathrm{A}\mathrm{x}(t) + \mathrm{B}\mathrm{u}(t)$

  $\mathbf{y}(t) = \mathrm{C}\mathrm{x}(t) + \mathrm{D}\mathrm{u}(t)$

  Where **x** is the state vector, $\dot{\boldsymbol{x}}$**u** is the input vector, **y** is the output vector, **A** is the state matrix, **B** is the input matrix, **C** is the output matrix, and **D** is the feedthrough matrix.

- As a transfer function defined by numerator and denominator. A transfer function is the ratio of the Laplace transform of the output or response function to the Laplace transform of the input or driving function, defined as follows:

$$G(s) = \frac{Y(s)}{X(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_{m-1} s + b_m}{a_0 s^n + a_1 s^{n-1} + \cdots + a_{n-1} s + a_n}$$

- As a matrix of numerator/denominator defined transfer functions. Transfer functions are limited to describing the relationship between one input and one output. A matrix of transfer functions, where the columns represent the inputs and the rows represent the outputs may be used for Multiple Input Multiple Output (MIMO) systems.
- As a transfer function defined by a set of zeros and poles (with gain). Poles and zeros are values that would cause a transfer function to be equal to 0, or undefined for a particular point. In the following equation:

$$G(s) = K \frac{(s - b_0)(s - b_1) \ldots (s - b_m)}{(s - a_0)(s - a_1) \ldots (s - a_m)}$$

  K represents the gain, a represents the set of poles, and b represents the set of zeros.
- As a matrix of zero-pole-gain defined transfer functions. Like numerator/denominator defined transfer functions, zero-pole-gain transfer

functions can only associate one input with one output. A matrix of these transfer functions may be used instead.

- As a set of nonlinear equations. Nonlinear equations are equations for which the principle of superposition does not apply. These include equations such as the following:

$$y = xz^2$$

The user may define a number of these nonlinear equations to define a plant.

It is the responsibility of the user interface designer to determine exactly how to collect the data needed to define a plant by any of these methods. See section 3 on UI Development for information on the functions provided to define a plant manually.

2.2.2 Input and Output Names

The inputs and outputs of a plant are named. These names are used to connect pseudo ports (section 2.3) to the inputs and outputs of the plant. Pseudo ports connect the inputs and outputs of the plant to the physical ports provided by the connected data acquisition device. Pseudo ports are named, and these names must match the names of the inputs and outputs of the plant in order for the system to operate properly. For example, a plant that has input ‘gas’ and output ‘speed’ requires an input pseudo port named ‘gas’ and an output pseudo port named ‘speed’ that indicate which physical port is to be read for the ‘gas’ value and which port to write the ‘speed’ value to.

2.2.3 Discrete Systems and Sampling

When defining a plant, the user must indicate whether the plant is discrete or not, and the sampling rate in hertz. Even if the plant is not discrete, a sampling rate must be provided as the Control System Plant Simulator must evaluate the plant at specific intervals. The CSPS will discretize continuous plants automatically. Currently the CSPS provides three methods for discretizing the plant: forward rectangular, backward

rectangular, and bilinear. If the user does not specify a discretization method, bilinear is chosen by default. Forward rectangular is not recommended as it is the least stable method and often requires high sampling rates for successful transformation.

2.2.4 Initial Conditions

The CSPS provides operations to user interface designers that allow end users to define initial conditions for the states of a plant. Initial conditions are to be supplied as engineering value (values with meaning to the plant) floating point parameters.

2.2.5 Saving and Loading predefined plants

The CSPS may save or load plants to a text file. This file is ASCII and is human-readable to simplify plant definitions. One does not have to start or run the CSPS in order to write a plant definition file that may be used later. Plants are only saved in State Space notation. If the plant was defined by a different method, the CSPS will convert it to a state space form automatically. The CSPS only supports reading files in state space form as well. The expected format is as follows:

```
NUM_STATES
<The number of states>
NUM_INPUTS
<The number of inputs>
NUM_OUTPUTS
<The number of outputs>
INPUT_NAMES
<The names of the input ports – MUST match pseudo ports>
OUTPUT_NAMES
<The names of the output ports – MUST match pseudo ports>
STATE_MATRIX
<State matrix value at position 0,0>
<State matrix value at position 0,1>
```

<State matrix value at position 0,2>

…

<State matrix value at position 0,n>

<State matrix value at position 1,0>

<State matrix value at position 1,1>

…

<State matrix value at position n,n>

INPUT_MATRIX

<Input matrix value at position 0,0>

<Input matrix value at position 0,1>

<Input matrix value at position 0,2>

…

<Input matrix value at position 0,n>

<Input matrix value at position 1,0>

<Input matrix value at position 1,1>

…

<Input matrix value at position n,m>

OUTPUT_MATRIX

<Output matrix value at position 0,0>

<Output matrix value at position 0,1>

<Output matrix value at position 0,2>

…

<Output matrix value at position 0,n>

<Output matrix value at position 1,0>

<Output matrix value at position 1,1>

…

<Output matrix value at position m,n>

FEEDTHROUGH_MATRIX

<Feedthrough matrix value at position 0,0>

<Feedthrough matrix value at position 0,1>

<Feedthrough matrix value at position 0,2>

…

```
<Feedthrough matrix value at position 0,n>
<Feedthrough matrix value at position 1,0>
<Feedthrough matrix value at position 1,1>
…
<Feedthrough matrix value at position m,n>
IS_DISCRETE
<TRUE if the plant is discrete, FALSE otherwise>
SAMPLING_FREQ
<The sampling frequency in hertz>
END
```

2.3 Pseudo Ports

The user must define a complete pseudo port mapping before simulation can begin. Pseudo ports connect the inputs and the outputs of the plant to physical ports. This may be done at run time (if provided by the user interface) manually, or as part of a file that may be saved and loaded.

2.3.1 Name

Every pseudo port must be given a name. This name indicates which plant input or output is connected to the pseudo port. In order for a simulation to be initiated, every plant input and output must be supplied with a pseudo port with a matching name. These names have a maximum length of 8 characters.

2.3.2 Physical Port Name

Pseudo ports connect plant inputs and outputs to physical ports. Not only does the name of the pseudo port have to match an input or output of the plant, but the pseudo port must also name the physical port it is connected to.

2.3.3 Pseudo port types

The user may define two different pseudo ports: Binary and Analog. Binary pseudo ports may be connected to digital physical ports. Analog pseudo ports may be connected to analog physical ports. Each provides different features for converting the values provided by the port to data relevant to the plant.

2.3.3.1 Binary Pseudo Ports

Binary pseudo ports map digital physical ports to binary plant inputs and outputs. Several binary pseudo ports may be connected to the same physical port, dividing it into smaller sub-ports. For example, a 32 bit digital physical port may be divided into two 16 bit pseudo ports. This is accomplished by indicating the high and low bits of the physical port a pseudo port maps to.

2.3.3.2 Analog Pseudo Ports

Analog pseudo ports map analog physical ports to plant inputs and outputs. Only one analog pseudo port may map to a physical port. Users may configure analog pseudo ports to scale the data read to engineering values meaningful to the plant. For example, a plant may have an input that accepts values between 100 and 500 PSI, with physical data provided between -10 and +10 volts. The Analog Pseudo Port will scale -10 to 100 and +10 to 500. All values in between are scaled linearly.

2.3.4 Defining, Saving, and Loading Pseudo Ports

It is the responsibility of the User Interface designer to determine how to collect the above information needed to define a pseudo port, but it is suggested that the UI provide information about the connected physical ports to help users determine which ones to connect their pseudo ports to and how to define those connections. Additionally, pseudo ports may be saved to or read from a file defined as follows:

NUM_PSEUDOPORTS

<Number of Pseudo Ports>

PORT_NAME

<The name of port 1>

PHY_PORT_NAME

<The name of the physical port connected to port 1>

IS_BINARY

<**TRUE** if port 1 is binary, **FALSE** otherwise>

IS_INPUT

<**TRUE** if port 1 is an input port, **FALSE** otherwise>

BINARY_MAX *(only for binary ports)*

<The highest bit of the physical port that port 1 maps to>

BINARY_MIN *(only for binary ports)*

<The lowest bit of the physical port that port 1 maps to>

ANALOG_MAX *(only for analog ports)*

<The engineering value maximum voltage represents>

ANALOG_MIN *(only for analog ports)*

<The engineering value minimum voltage represents>

PORT_NAME

<The name of port 2>

…

END

2.4 Logging

The CSPS provides three different log systems: Informational logs, Critical logs, and IO logs. Informational log messages provide basic system information such as "Simulation Started", "Simulation Stopped", and "Plant Configured". Critical logs provide error messages, and should not be ignored. Examples of critical log messages include "Invalid Plant Configuration" "Missed Deadline" and "Failure to start simulation". IO log messages contain the current values provided as inputs and calculated outputs of the plant in *engineering values*. The current elapsed time is included as well.

2.4.1 Log Behavior

All log messages are passed back to the user interface. It is up to the user interface designer to determine what to do with them. It is highly recommended, at a minimum, to display all critical log messages. Critical log messages will always be displayed in the console that launched the Control System Plant Simulator, but should be prominently displayed whenever possible.

The user may configure how often IO log messages are generated. The three options are push all IO log messages, push IO log messages periodically, and pull IO log messages. When the push all option is selected (as it is by default) every time the plant is evaluated an IO log message is generated and sent to the Kernel. This may be dangerous as this happens during simulation time. It may impact simulation performance. Push periodic sends IO messages after a specified number of plant evaluations. Pull only sends IO messages when requested by the user.

2.4.2 Capturing logs to files

The CSPS has the ability to capture any of the three logs to individual data files. Informational and critical log messages get appended to the end of their specified files, one line at a time. IO logging is provided as a comma delimited file with each line consisting of the current elapsed time in seconds, the input port values, and the output port values, with an empty column in between the inputs and outputs. It is recommended that the IO logs are saved as .csv files, as this allows the file to be imported directly to Windows Excel.

2.5 Configuring Physical Port Updates

Physical ports are not read or written to as part of the plant evaluation process. There is no guarantee of how long such a read or write will take, especially considering the fact that the CSPS supports and encourages the addition of support for other data acquisition devices. Instead all values are read or written to a cache. Periodically this

cache is updated with values from physical ports for input parameters, and is used to update physical ports for output parameters.

Users may schedule update periods on a port by port basis. The CSPS provides operations to the user interface designer that allow physical port updates to be scheduled by simply providing the period in milliseconds and the name of the physical port to update. Update periods do not need to match or overlap, but can if the user desires.

2.6 Running a Simulation (Quick start guide)

The following are the steps necessary to run a complete simulation:

- Make all necessary hardware connections. The data acquisition device should be connected to the system and to the target controller before launching the CSPS.
- Launch the CSPS by executing the Kernel executable on the command line with arguments indicating which user interface and Computation Kernel to launch as well.
- Load or define a plant.
- Load or define a set of pseudo ports that map every input and output of the plant to physical ports.
- Set initial conditions (optional)
- Schedule physical port updates (optional).
- Start any desired logging (optional)
- Start the simulation.
- When finished, stop the simulation.

  Send the terminate command as described in 3.2.6 to the CSPS – Note that this command MUST be sent to properly shut down the CSPS. User Interface Designers should provide it as part of any shut down code in the UI.

3. UI Development

A significant portion of the work associated with setting up the Control System Plant Simulator is developing a User Interface for your plant. This section outlines how to build your own user interface that can interface directly with the CSPS.

3.1 Introduction

Users are expected to develop their own user interface processes specifically for their plants. Two simple user interfaces are provided, but these merely provide access to all of the functions provided by the Control System Plant Simulator. They are bulky and provide for the most general of interactions. Plant specific details (such as graphic representation of the state of the plant) simply do not exist, and access to every possible option creates for a cluttered and unwieldy interface. Still, they serve their purpose: To provide an example of how to create a User Interface and how to properly call the operations in the CSPS API. Users should employ these interfaces as a starting point for developing their own interfaces with features designed specifically for the target plant.

3.2 The UiWinInterface API

The CSPS suite includes an API that any user interface can use in order to send commands to the kernel. This API is provided by the UiWinInterface.dll dynamically linked library. Note that while all file names are assumed to be relative to the directory the CSPS executables are located in, the user may provide a fully qualified pathname such as C:\CSPS\Scripts\filename.txt as well. The API allows the user access to the following function calls:

3.2.1 InitDll

Signature: `void InitDll()`

Description: The InitDll operation must be called after the library has been linked into the user interface, but before any other operations may be

called. It initializes the classes in the dll and starts the threads that the dll uses to handle the interface communications.

Arguments: none

Return Value: none

3.2.2 CallStartExecution

Signature: `bool CallStartExecution()`

Description: The CallStartExecution sends a command to the CSPS to start execution of the simulation. A valid plant and pseudo port mapping must have been established for this call to be successful.

Arguments: none

Return Value: Returns true if the system was able to start execution, false otherwise.

3.2.3 CallStopExecution

Signature: `bool CallStopExecution()`

Description: The CallStopExecution sends a command to the CSPS to stop execution of the simulation.

Arguments: none

Return Value: Returns true if the system was able to stop execution, false otherwise.

3.2.4 CallIsReadyForExecution

Signature: `bool CallIsReadyForExecution()`

Description: The CallIsReadyForExecution operation sends a command to the CSPS that will cause it to determine if all of the prerequisites for system execution have been met.

Arguments: none

Return Value: Returns true if the CSPS is able to initiate simulation, and false if it cannot.

3.2.5 CallRequestIOUpdate

Signature: `bool CallRequestIOUpdate()`

Description: The CallRequestIOUpdate operation sends a command to the CSPS that will cause it to generate an IO update log message.

Arguments: none

Return Value: Returns true if an IO update log message was generated, false if it was not.

3.2.6 CallTerminate

Signature: `void CallTerminate()`

Description: The CallTerminate operation sends a command to the CSPS that will cause it to terminate all operation and shut down.

Arguments: none

Return Value: none

3.2.7 CallSetPlantStateSpace

Signature: `bool CallSetPlantStateSpace(StateSpace& ss)`

Description: The CallSetPlantStateSpace operation sets the plant configuration using state space notation. The UI designer must fill a StateSpace structure that contains two dimensional arrays to store the state space matrices themselves (The State, Input, Output, and Feedthrough matrices), information indicating the number of inputs, outputs, and states, the names of the inputs and outputs, whether the plant is discrete or not, and the frequency at which to sample the plant.

Once the UI has collected this data it sends it to the CSPS framework by packaging it in a StateSpace structure and passes it as the ss parameter of this function.

For example, if the user wishes to configure a plant with the following state space matrices

$$A = \begin{bmatrix} 0 & 1 \\ -1.5 & -2.5 \end{bmatrix}$$
$$B = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix}$$
$$C = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$D = \begin{bmatrix} 0 \end{bmatrix}$$

then the User Interface must provide a state space structure with the matrix values set as follows:

```
ss.stateMatrix[0][0] = 0;
ss.stateMatrix[0][1] = 1;
ss.stateMatrix[1][0] = -1.5;
ss.stateMatrix[1][1] = -2.5;
ss.inputMatrix[0][0] = 0;
ss.inputMatrix[1][0] = 0.5;
ss.outputMatrix[0][0] = 0;
ss.outputMatrix[0][1] = 1;
ss.feedthroughMatrix[0][0] = 0;
```

See the provided ExampleUI and SimpleUI source code for more examples of how to fill the state space structure.

Arguments:

- **ss –** a reference to a StateSpace structure that contains information needed to define a plant. This includes the state space equations, sampling rate, and the discrete or continuous nature of the plant.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.8 CallSetPlantZPK

Signature: `bool CallSetPlantZPK(TF_PoleZeroGain_Descriptor& zpk)`

Description: The CallSetPlantZPK operation sets the plant configuration using a transfer function defined by a set of zeros and poles. The UI Designer must fill a TF_PoleZeroGain_Descriptor with information about each pole and zero, the input and output names, whether the plant is discrete or not, and the frequency at which to sample the plant. Once the UI has collected this data it sends it to the CSPS framework by packaging it in a TF_PoleZeroGain_Descriptor structure and passes it as the zpk parameter of this function.

For example, if a user wishes to provide a transfer function defined as follows

$$H(s) = 7.34\frac{(s-4)(s+5.5)}{\{s-(1+3j)\}\{s-(1-3j)\}}$$

Then they are defining a system with zeros at 4 and -5.5, a pole at -1-3j, and a pole at -1+3j. These values must be entered into the gain variable, the poles array, and zeros array of the TF_PoleZeroGain_descriptor structure as follows:

```
zpk.gain = 7.34;
zpk.poles[0].x = 4;
zpk.poles[0].j = 0;  // no complex part
zpk.poles[1].x = -5.5;
zpk.poles[1].j = 0;  // no complex part
zpk.zeros[0].x = -1;
zpk.zeros[0].j = 3;
```

```
zpk.zeros[1].x = -1;
zpk.zeros[1].j = -3;
```

See the provided ExampleUI and SimpleUI code for more examples of how to fill the zero-pole-gain structure.

Arguments:

- zpk – is a reference to a TF_PoleZeroGain_Descriptor structure that contains information needed to define a plant. This includes the poles and zeros, the gain, sampling rate, and the discrete or continuous nature of the plant.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.9 CallSetPlantZPKMatrix

Signature: `bool CallSetPlantZPKMatrix( TF_PoleZeroGain_Matrix& zpkMatrix)`

Description: The CallSetPlantZPKMatrix operation sets the plant configuration using a matrix of transfer functions defined by zeros and poles. The matrix is defined by sets of TF_PoleZeroGain_Descriptor structures organized in a matrix. The columns of this matrix make up the inputs, and the rows make up the outputs of the plant.

For example, if a plant has 3 inputs and 2 outputs the matrix is defined as follows:

	INPUT 1	INPUT 2	INPUT 3
OUTPUT 1	H11(s)	H12(s)	H13(s)
OUTPUT 2	H21(s)	H22(s)	H23(s)

where H11 through H23 are transfer functions defined as sets of zero-pole-gains. H11 relates INPUT 1 to OUTPUT1, H12 relates INPUT 2 to OUTPUT1, and so on.

See the provided ExampleUI and SimpleUI code for more examples of how to fill the zpk matrix structure.

Arguments:

- zpkMatrix – A reference to a TF_PoleZeroGain_Matrix structure that contains information needed to define a plant. It includes a matrix of transfer functions, with the row of the matrix indicating an output, and the column of the matrix indicating an input. For example, position 2,3 of the matrix defines a transfer function that involves input 3 and output 2.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.10 CallSetPlantTF

Signature: `bool CallSetPlantTF(`
`TF_NumDenom_Descriptor& tf)`

Description: The CallSetPlantTF operation sets the plant configuration using a transfer function defined by two polynomials: a numerator and a denominator. The UI Designer must fill a TF_NumDenom_Descriptor with the polynomials that make up the numerator and denominator of the transfer function, the input and output names, whether the plant is discrete or not, and the frequency at which to sample the plant. The polynomials are defined by coefficients, similar to Matlab. Once the UI has collected this data it sends it to the CSPS framework by packaging it in a TF_NumDenom_Descriptor structure and passes it as the tf parameter of this function.

For example, if the user wishes to provide a plant defined by the following transfer function:

$$H(s) = \frac{s^3 + 2s^2 + 4}{s^4 + 3.3s^3 - 4s^2 + 7s + 12.4}$$

the user interface would have to fill a TF_NumDenom_Descriptor as follows

```
tf.numerator[0] = 1;      // S^3
tf.numerator[1] = 2;      // 2s^2
tf.numerator[2] = 0;      // 0s
tf.numerator[3] = 4;      // 4
tf.denominator[0] = 1     // s^4
tf.denominator[1] = 3.3   // 3.3s^3
tf.denominator[2] = 4     // 4s^2
tf.denominator[3] = 7     // 7s
tf.denominator[4] = 12.4 // 12.4
```

See the provided ExampleUI and SimpleUI code for more examples of how to fill the TF_NumDenom_Descriptor structure.

Arguments:

- tf – A reference to a TF_NumDenom_Descriptor structure that contains information needed to define a plant specified by a transfer function. This includes the numerator and denominator of the transfer function, the sampling rate, and the discrete or continuous nature of the plant.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.11 CallSetPlantTFMatrix

Signature: `bool CallSetPlantTFMatrix(`
`TF_NumDenom_Matrix& tfMatrix)`

Description: The CallSetPlantTFMatrix operation sets the plant configuration using a matrix of transfer functions defined by transfer functions. The matrix is defined by sets of TF_NumDenom_Descriptor structures organized in a matrix. The columns of this matrix make up the inputs, and the rows make up the outputs of the plant.

For example, if a plant has 3 inputs and 2 outputs the matrix is defined as follows:

	INPUT 1	INPUT 2	INPUT 3
OUTPUT 1	H11(s)	H12(s)	H13(s)
OUTPUT 2	H21(s)	H22(s)	H23(s)

where H11 through H23 are transfer functions defined as numerators and denominators of polynomials. H11 relates INPUT 1 to OUTPUT1, H12 relates INPUT 2 to OUTPUT1, and so on.

See the provided ExampleUI and SimpleUI code for more examples of how to fill the transfer function matrix structure.

Arguments:

- tfMatrix – A reference to a TF_NumDenom_Matrix structure that contains information needed to define a plant. It includes a matrix of transfer functions, with the row of the matrix indicating an output, and the column of the matrix indicating an input. For example, position 2,3 of the matrix defines a transfer function that involves input 3 and output 2.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.12 CallSetPlantNonlinear

Signature: `bool CallSetPlantNonlinear(`
`NonLinear_Descriptor& nonlinearEqs)`

Description: The CallSetPlantNonlinear operation sets the plant configuration using a set of nonlinear equations. These nonlinear equations are defined by a NonLinear_Descriptor structure. This structure is defined by sets of Terms. Each term consists of a coefficient and an array of powers for the variables available in the nonlinear equation. For example, the nonlinear equation $y = x + 2z + 3x^2z^3$ would be defined by 3 terms. The first term has coefficient 1, and state powers 1 and 0. The second term has coefficient 2 and state powers 0 and 1. The final term has coefficient 3, and state powers 2 and 3. Once the UI has collected this data it sends it to the CSPS framework by packaging it in a NonLinear_Descriptor structure and passes it as the nonlinearEqs parameter of this function.

See the provided ExampleUI and SimpleUI code for more examples of how to fill the NonLinear_Descriptor structure.

Arguments:

- nonlinearEqs – A reference to a TF_NonLinear_Descriptor structure that contains information needed to define a plant. It defines nonlinear equations as a set of terms and coefficients.

Return Value: Returns true if the plant is established properly, and false if the call fails.

3.2.13 CallGetPlant

Signature: `bool CallGetPlant(StateSpace& ss)`

Description: The CallGetPlant operation retrieves the plant that is currently configured in the CSPS.

Arguments:

- ss – A reference to a StateSpace structure. It is an output parameter and will be populated with the current plant in the CSPS.

Return Value: Returns true if the plant was retrieved successfully, false otherwise.

3.2.14 CallSavePlant

Signature: `bool CallSavePlant(const char* fileName)`

Description: The CallSavePlant operation saves the currently specified plant to a file.

Arguments:

- fileName – The name of the file to save the plant to.

Return Value: Returns true if the plant is saved successfully, false otherwise.

3.2.15 CallLoadPlant

Signature: `bool CallLoadPlant(const char* fileName)`

Description: The CallSavePlant operation loads a plant from the specified file.

Arguments:

- fileName – The name of the file to load the plant from.

Return Value: Returns true if the plant is loaded successfully, false otherwise.

3.2.16 CallClearPlant

Signature: `bool CallClearPlant()`

Description: The CallClearPlant operation clears out the currently configured plant.

Arguments: none.

Return Value: Returns true if the plant is cleared successfully, false otherwise.

3.2.17 CallSetInitialConditions

Signature:
```
bool CallSetInitialConditions(
        InitialConditionsWrapper& wrapper)
```

Description: The CallSetInitialConditions operation establishes the initial conditions for the plant. If this is never called, initial conditions are assumed to be 0.

Arguments:

- wrapper – A reference to an InitialConditionsWrapper structure. The InitialConditionsWrapper contains an array of floating point values that hold the initial conditions of the system.

Return Value: Returns true if the initial conditions were set properly, false otherwise.

3.2.18 CallSetDiscretizationMethod

Signature:
```
bool CallSetDiscretizationMethod(
                long method)
```

Description: The CallSetDiscretizationMethod operation sets the method by which continuous plants are turned into discrete ones. Plants are discretized when simulation is started, so users may alter the discretization method before or after a plant has been provided to the system, but NOT during simulation.

Arguments:

- method – a long enumerated value that indicates the method. The current possible values are:
  0 to indicate the Forward Rectangular method.
  1 to indicate the Backward Rectangular method.
  2 to indicate the Bilinear method.

Return Value: Returns true if the discretization method is set properly. False otherwise.

3.2.19 CallSetIOUpdateMethod

Signature:
```
bool CallSetIOUpdateMethod(long method,
                                    int period)
```

Description: The CallSetIOUpdateMethod operation determines how often IO update messages will be generated.

Arguments:

- method – a long value that indicates the method. The current possible values are:

  0 to indicate a method where an IO Update is sent every simulation cycle.

  1 to indicate a method where an IO Update is sent every **period** cycles.

  2 to indicate a method where no IO Updates are sent. They must be requested.
- period – the number of system cycles that must be executed before generating an IO update. Valid only for PUSH_PERIODIC mode.

Return Value: Returns true if the IO Update method is set properly. False otherwise.

3.2.20 CallAddPort

Signature: `bool CallAddPort(`
`PseudoPortDescriptor& pseudoPort)`

Description: The CallAddPort operation adds a pseudo port to the current pseudo port mapping.

Arguments:

- pseudoPort – A reference to a PseudoPortDescriptor structure containing all information needed to establish a pseudo port.

Return Value: Returns true if the port is added successfully, false if the port addition fails.

3.2.21 CallRemovePort

Signature: `bool CallRemovePort(const char* portName)`

Description: The CallRemovePort operation removes a pseudo port from the current port mapping.

Arguments:

- portName – the name of the port to remove.

Return Value: Returns true if the port was removed, false otherwise.

3.2.22 CallGetPorts

Signature: `bool CallGetPorts(PseudoPortMapping& ports)`

Description: The CallGetPorts operation retrieves information about the complete pseudo port mapping.

Arguments:

- ports – a reference to a PseudoPortMapping structure. This is an output parameter that will be populated with the necessary information.

Return Value: Returns true if the port mapping was successfully retrieved, false otherwise.

3.2.23 CallClearPorts

Signature: `bool CallClearPorts()`

Description: The CallClearPorts operation

Arguments: none.

Return Value: Returns true if the port mapping was successfully cleared, false otherwise.

3.2.24 CallSavePorts

Signature: `bool CallSavePorts(const char* fileName)`

Description: The CallSavePorts operation saves the current port mapping to an ASCII based file.

Arguments:

- fileName – The port mapping will be saved to the file indicated by this file name. If such a file does not yet exist, a new one will be created. If the file does exist, it will be overwritten.

Return Value: Returns true if the port mapping was successfully saved, false otherwise.

3.2.25 CallLoadPorts

Signature: `bool CallLoadPorts(const char* fileName)`

Description: The CallLoadPorts operation loads the port mapping from an ASCII based file.

Arguments:

- fileName – The port mapping will be loaded from the file indicated by this file name.

Return Value: Returns true if the port mapping was successfully loaded, false otherwise.

3.2.26 CallGetPhysicalPortInfo

Signature:
```
bool CallGetPhysicalPortInfo(
                PhysicalPortsDescriptor& phyPorts)
```

Description: The CallGetPhysicalPortInfo operation retrieves information about the physical ports as established by the PhysicalPortCache.

Arguments:

- phyPorts – A reference to a PhysicalPortsDecriptor. This is an output parameter and will be populated with the necessary data.

Return Value: Returns true if the physical port information was successfully retrieved, false otherwise.

3.2.27 CallSchedulePortActivity

Signature:
```
bool CallSchedulePortActivity(
                const char* portName,
                int period)
```

Description: The CallSchedulePortActivity operation establishes the update period for a particular physical port. The indicated physical port will be read from periodically according to the specified length of time.

Arguments:

- portName – The name of the physical port to schedule.
- period – the length of time in milliseconds between updates. Note that no minimum period is enforced. The user interface designer must take care not to over burden the system.

Return Value: Returns true if the physical port schedule is set properly, false otherwise.

3.2.28 CallEnableCriticalLogging

Signature:
```
bool CallEnableCriticalLogging(
                              const char* fileName)
```

Description: The CallEnableCriticalLogging operation starts logging critical messages to a file.

Arguments:

- fileName – The name of the file to save critical messages to.

Return Value: Returns true if the critical log is properly initiated.

3.2.29 CallDisableCriticalLogging

Signature: `bool CallDisableCriticalLogging()`

Description: The CallEnableCriticalLogging operation stops logging critical messages to a file.

Arguments: none.

Return Value: Returns true if the critical log is stopped.

3.2.30 CallHasCriticalMsgs

Signature: `bool CallHasCriticalMsgs()`

Description: The CallHasCriticalMsgs operation determines whether any critical messages have been sent to the user interface or not.

Arguments: none.

Return Value: Returns true if the dll has buffered any critical messages. The operation returns false if the critical message buffer is empty.

3.2.31 CallFetchCriticalMsg

Signature: `bool CallFetchCriticalMsg(MsgType& msg)`

Description: The CallFetchCriticalMsg operation retrieves the oldest critical message in the critical message buffer

Arguments:

- msg – A MsgType struct containing the critical message as it was stored in the critical message buffer. This is an output parameter and will be populated by calling the operation.

Return Value: Returns true if the operation successfully retrieves a message. False otherwise.

3.2.32 CallEnableInfoLogging

Signature:
```
bool CallEnableInfoLogging(
                          const char* fileName)
```

Description: The CallEnableInfoLogging operation starts logging informational messages to a file.

Arguments:

- fileName – The name of the file to save informational messages to.

Return Value: Returns true if the informational log is properly initiated.

3.2.33 CallDisableInfoLogging

Signature: `bool CallDisableInfoLogging()`

Description: The CallEnableInfoLogging operation stops logging informational messages to a file.

Arguments: none.

Return Value: Returns true if the informational log is stopped.

3.2.34 CallHasInfoMsgs

Signature: `bool CallHasInfoMsgs()`

Description: The CallHasInfoMsgs operation determines whether any informational messages have been sent to the user interface or not.

Arguments: none.

Return Value: Returns true if the dll has buffered any informational messages. The operation returns false if the critical message buffer is empty.

3.2.35 CallFetchInfoMsg

Signature: `bool CallFetchInfoMsg(MsgType& msg)`

Description: The CallFetchInfoMsg operation retrieves the oldest informational message in the informational message buffer

Arguments:

- msg – A MsgType struct containing the informational message as it was stored in the critical message buffer. This is an output parameter and will be populated by calling the operation.

Return Value: Returns true if the operation successfully retrieves a message. False otherwise.

3.2.36 CallEnableIOLogging

Signature:
```
bool CallEnableIOLogging(
                          const char* fileName)
```

Description: The CallEnableIOLogging operation starts logging IO messages to a file.

Arguments:

- fileName – The name of the file to save IO messages to.

Return Value: Returns true if the IO log is properly initiated.

3.2.37 CallDisableIOLogging

Signature: `bool CallDisableIOLogging()`

Description: The CallEnableIOLogging operation stops logging IO messages to a file.

Arguments: none.

Return Value: Returns true if the IO log is stopped.

3.2.38 CallHasIOMsgs

Signature: `bool CallHasIOMsgs()`

Description: The CallHasIOMsgs operation determines whether a new IO update has been provided to the User Interface since the last time one was read.

Arguments: none.

Return Value: Returns true if a new IO Update has been posted since the last time CallHasIOMsgs was called. The operation returns false otherwise.

3.2.39 CallFetchIOMsg

Signature: `bool CallFetchIOMsg(IoUpdateData& data)`

Description: The CallFetchIOMsg operation retrieves the most recent IO update from the IO Update buffer.

Arguments:

- data – An IoUpdateData structure containing the values of all inputs and outputs.

Return Value: Returns true if the operation successfully retrieves the last IO Update value. The operation returns false otherwise.

3.3 Initializing and using the dll

The UiWinInterface dynamically linked library provides simplifies the process of developing a user interface by performing all of the work needed to send commands to, and receive commands from the CSPS.

3.3.1 C++ User Interfaces

C++ user interfaces must load the library at run time, declare function pointers, and map all of the functions provided by the dll to the pointers. To simplify this process, UiWinInterface.h has been provided that performs the majority of these operations itself. It declares all of the function pointer types, declares instances of those types, and provides the InitializeOps function. InitializeOps retrieves the process address of each operation from the dll and maps them to the function pointers it declared. The developed user interface must still load the library (by using the Windows LoadLibrary operation for example), and pass this loaded library to the InitializeOps operation. Once InitializeOps has successfully performed the function mapping, one needs only to make the call on the functions defined.

The first function called *MUST* be InitDll. This operation starts the threads that must run in the dll and establishes all of the classes needed. Only after InitDll has been called on the API can any other operations be successfully performed.

See the SimpleUi.exe project for an example of how to load and use the provided UiWinInterface library.

3.3.2 Visual Basic

Visual basic user interfaces may still make use of the dll, but cannot use the UiWinInterface.h InitializeOps function. Visual basic requires dll functions to be declared using the Lib command. Each structure must also be defined in terms that visual basic can use. This has been provided for future UI developers in the included GlobalModule.bas module file under the ExampleUI.vbp project. This file declares all necessary structures and functions. By including this module, any visual basic project has full access to the API.

As with C++ User Interfaces, the first function called *MUST* be InitDll. Only after InitDll has been called on the API can any other operations be successfully performed.

3.4 Logging

When enabled, log messages and updates are sent to the user interface API by the CSPS. These messages cannot simply be forced down to the User Interface as some systems such as Visual Basic do not allow outside threads to access their code. Instead the API will buffer or store these messages. It is the responsibility of the UI designer to determine when to fetch buffered messages from the API. The UI Designer should consider using a separate thread to process log messages and updates.

3.5 SimpleUI

SimpleUI is a fully functional example command line User Interface. It provides access to every feature provided by the CSPS Framework. The command interface of SimpleUI is as follows:

- quit – Exits Program
- start – Starts Execution
- stop – Stops execution
- isready – Determines if the system is ready to start
- requestio – Requests a single IO update
- setplantss – Sets a plant using state space equations
- setplantzpk – Sets a plant using zero-pole-gain notation
- setplantzpkmatrix – Sets a plant using a matrix of zpk defined transfer functions.
- setplanttf – Sets a plant using a transfer function.
- setplanttfmatrix – Sets a plant using a matrix of transfer functions.
- setplantnonlinear – Sets a plant using nonlinear equations
- getplant – Retrieves a plant in state space notation.
- saveplant – Saves the plant to a file
- loadplant – Loads a plant
- clearplant – Clears the current plant.
- setinitcond – Sets the initial conditions
- setdmethod – Sets the Discretization method
- setumethod – Sets the IO update method

- addort – Adds a pseudo port to the system
- removeport – Removes a pseudo port from the system by name.
- getports – Retrieves the current port mapping.
- clearports – Clears the port mapping
- saveports – Saves the port mapping to a file
- loadports – Loads a port mapping
- getphyports – Retrieves physical port information
- schedphyport – Schedules the update period for a physical port
- logio – Logs IO data to a file
- stopiolog – Stops logging IO data
- chkiodata – Checks to see if IO Data is available
- fetchiodata – Fetches IO data
- logcrit – Logs critical data to a file
- stopcritlog – Stops critical logging
- chkcritmsg – Checks to see if any critical messages have been posted.
- fetchcritmsg – Fetches the oldest critical message
- loginfo – Starts informational message logging
- stopinfolog – Stops informational message logging
- chkinfomsg – Checks to see if any informational messages have been posted.
- fetchinfomsg – fetches the oldest informational message.

3.6 ExampleUI

Like SimpleUi.exe, ExampleUI is an example of a User Interface. ExampleUI is a visual basic GUI that uses the UiWinInterface.dll library to communicate with the CSPS Framework. It provides access to each of the operations provided by the framework to demonstrate how to use each one.

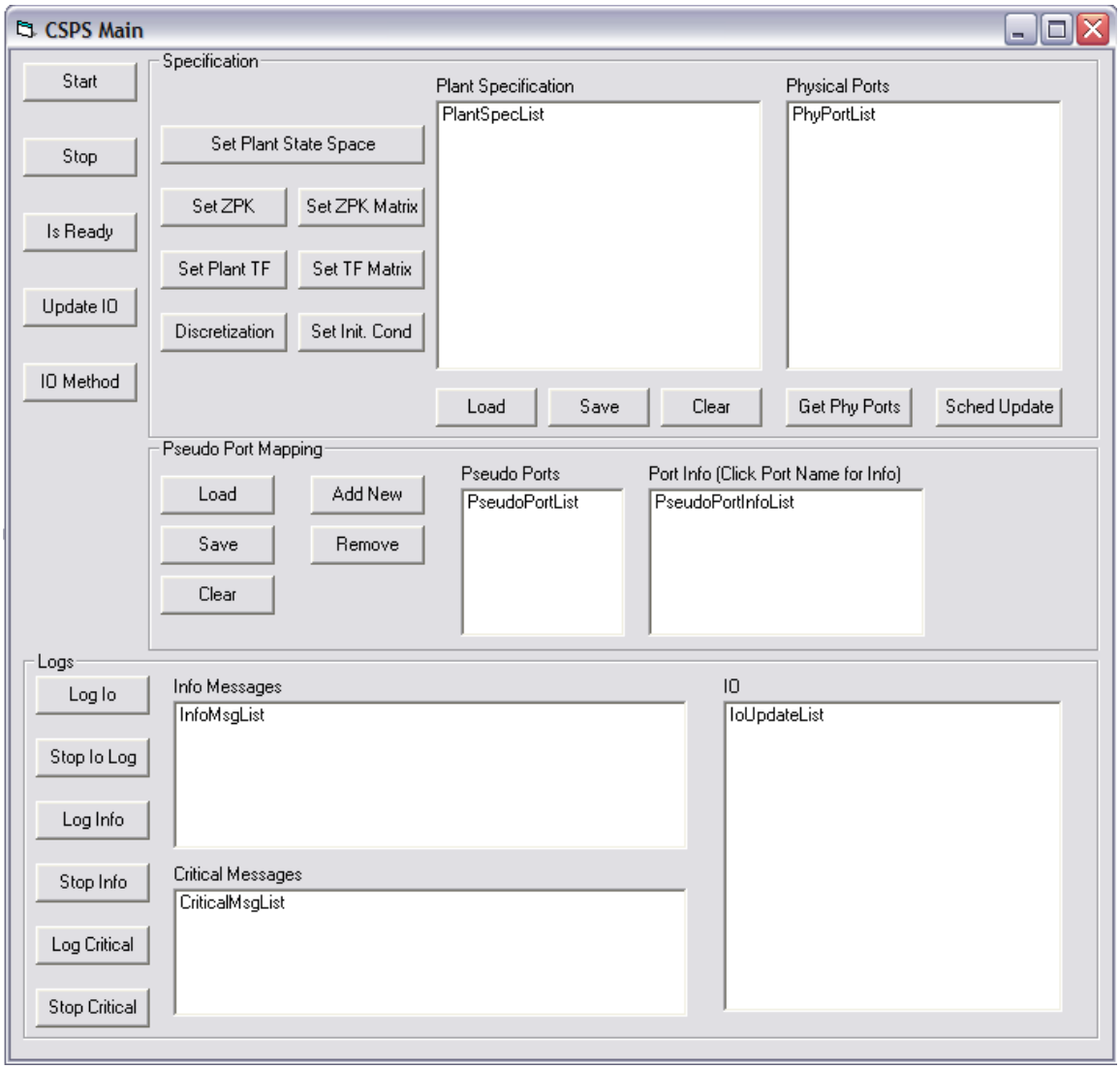

Figure 3: ExampleUI Operational Screen

The ExampleUI main window is broken into four main sections. At the far left are the system control buttons. These allow the user to start, stop, and otherwise configure the system. At the top right is the Specification frame. This section allows the user to set the plant and review the physical ports available. The Pseudo Port Mapping frame allows users to add, remove, load, save, or clear the current pseudo port mapping. Pseudo ports appear by name in the Pseudo Port list. Clicking on a name brings up information about the particular pseudo port in the Port Info window. The bottom of the window consists of the log information. All log messages are displayed in each of the specified windows. The buttons allow the user to start and stop logging this data to text files.

4. Porting to a new Data Acquisition Device

The provided Win32 version of the Control System Plant Simulator's Computational Kernel has been built for use with the Data Translations DT-9810 data acquisition device. The RTX version has been built for use with the simulated interface. Future users will have to port some of the code in the Computational Kernel in order to accommodate other data acquisition devices.

Individual data acquisition devices are represented in the code by the Physical Port Cache. PhysicalPortCache.h and PhysicalPortCache.cpp define a parent class from which individually developed PhysicalPortCache objects must inherit.

Individual interfaces on the data acquisition device are represented in the code by Physical Ports. PhysicalPort.h and PhysicalPort.cpp define a parent class from which individually developed PhysicalPort objects must inherit.

Porting to a new data acquisition device involves creating child classes for both the PhysicalPortCache and the PhysicalPort that properly represent the device, and instantiating the new PhysicalPortCache object in the port manager. Each PhysicalPort child class must implement a specific ReadValue and WriteValue operation that handles physically reading from or writing to the interface on the data acquisition device. The PhysicalPortCache child object must implement specific PopulatePortCache operations that will instantiate the new PhysicalPort child class objects that represent the interfaces on the data acquisition device. The constructor of the PhysicalPortCache child object must call this operation.

The PortManager owns the PhysicalPortCache and must instantiate the specific child class upon construction. This code will have to be modified to instantiate the new PhysicalPortCache once it has been developed.

5. Porting Operating System Abstraction Layer

As noted previously, the Control System Plant Simulator is composed of three completely separate process spaces. The Computation Kernel is the lowest level process that handles simulation of the plant and physical IO. The Win32 process (or Kernel) is the main process that manages all data and launches the other two processes. Finally, the User Interface is a user developed process that collects input from the user to drive the system.

These process spaces do not need to be launched using the same operating system. A typical RTX program is usually split into two processes: a Win32 process and an RTX process. The provided Control System Plant Simulator has been designed for use in both Windows and RTX. This was accomplished by abstracting away a number of operating system objects including the semaphore, files, shared memory, and threads. Wrapper classes were provided for each of these elements that can be ported to any operating system desired. If the operating system changes, only the wrapper classes would need to be altered to make the new OS level calls.

There is, however, an additional level of complexity. Some processes in the CSPS may require the use of both operating systems at once. For example, the Kernel must communicate with both the Computational Kernel and the User Interface process. Two versions of the CSPS have been provided: one where all processes operate under Windows, and one where the Computation Kernel operates in RTX. This means that the Kernel must use RTX operating system calls to communicate with the Computation Kernel, and Windows operating system calls to communicate with the User Interface.

To manage such a problem, each operating system abstraction class features two implementations for each function call. Which implementations are actually compiled is handled by a number of precompiler definitions defined in the LocalDefinitions.h header for each project. If both operations are available, the CSPS selects which one to use by setting the compInterface flag at compile time. If only one implementation is available, the flag is ignored.

To port these operations to another OS, replace one or both of the operating system interfaces with code that performs the same function in the new operating system. Once that is complete, a new LocalDefinitions.h file will have to be created that sets the appropriate flags for the processes affected.

6. Additional Tools

Any additional tools or materials are described in the sections that follow.

6.1 The SimInterface Tool

One of the provided versions of the CSPS runs on Ardence RTX real-time extensions for windows. This RTX port of the CSPS cannot interoperate with the Data Translations DT-9812 data acquisition device because the device connects to the computer through a USB interface. An RTX Computation Kernel was written, but could only be simulated against a set of simulated physical ports. The SimInterface program is a command line program that provides these simulated physical ports. It is a console application that allows a user to define physical ports for the system, set values on those ports at runtime, and monitor output values from the CSPS system. Note that if this program is to be used, it *MUST* be launched and initialized before the CSPS is initiated. CSPS systems compiled with SimInterface as their physical port system look for the shared memory provided by SimInterface at boot time.

6.1.1 Configuring SimInterface

SimInterface must be provided a set of physical ports to simulate at run time. This may be accomplished manually, or through a script. Users who decide to configure the physical ports manually will be provided a series of questions by the application including how many ports the interface will have, and information on the properties of each of those ports. Users may also create a script configuration file to use when configuring the SimInterface. The format of this file is as follows:

```
IN_PORT
<Name of input port 1>
<Port 1's type, either ANALOG, BIT, BYTE, WORD, or LONG>
<Port 1's initial value>
IN_PORT
<Name of input port 2>
```

…

OUT_PORT

<Name of output port 1>

< Port 1's type, either **ANALOG, BIT, BYTE, WORD, or LONG**>

< Output Port 1's initial value>

…

PERIOD

Default update period for all physical ports

END

6.1.2 Running SimInterface

Once configured, SimInterface provides the full port listing to allow the user to review the configuration.

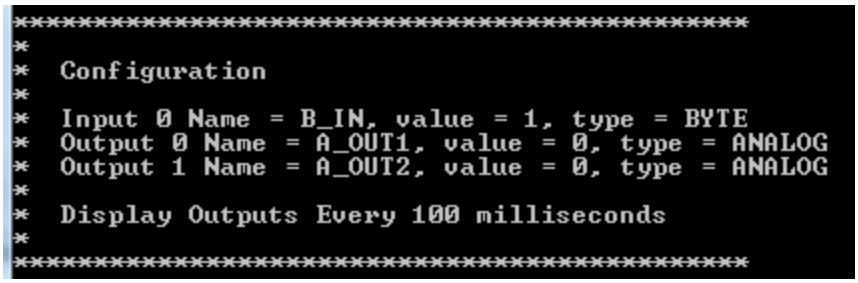

Figure 4: SimInterface Initialization

SimInterface may be run in two modes: manually or by script.

6.1.2.1 Manual Mode

If the user selects Manual mode execution, SimInterface will do nothing and ignore all incoming values (Plant outputs) until the user provides a prompt. The SimInterface tool responds to the following prompts:

- s - Start monitoring plant output.
- p - Pause plant output monitoring
- v - Set the value of a Physical Port
- q - Quit

6.1.2.2 Script Mode

The SimInterface can also set and clear data values according to a script. The script will cause the SimInterface to update physical port values automatically. When running a script, the user will be prompted to indicate how many iterations through the script are desired. The script will be run the specified number of times and the program will close. The script is defined as a text file formatted as follows:

WAIT
<Wait time in milliseconds>
SET_PORT
<port name>
<value to set>
SET_PORT
<second port name>
<second value to set>
…
END_WAIT *Indicates that all port settings for this 'wait' have completed*
WAIT
<wait time in milliseconds>
…
END_WAIT
END_SCRIPT