Control System Plant Simulator: A Framework for Hardware-In-The-Loop Simulation

By

David Chandler

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Masters of Science in Computer Engineering

Supervised by

Dr. James Vallino
Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY
June 2007

Date:_____

Dr. James Vallino
Primary Advisor – R.I.T. Dept. of Software Engineering

Date:_____

Dr. Juan Carlos Cockburn
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Date:_____

Dr. Roy Czernikowski
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Control System Plant Simulator

I, DAVID CHANDLER, HEREBY GRANT PERMISSION TO THE WALLACE MEMORIAL LIBRARY TO REPRODUCE MY THESIS IN WHOLE OR PART.

_____
David Chandler

_____
Date

Dedication

For my Mother, who made me the person I am today, and for my Fiancée Cheri for the love and support that made this possible.

Acknowledgements

I am forever indebted to my thesis advisor, Dr. James Vallino of the Department of Software Engineering at Rochester Institute of Technology. His guidance, insight, advice, and encouragement kept this thesis on track through every obstacle. Dr. Vallino was always available to provide help, offer suggestions, and review progress. I could have asked for absolutely nothing more in an advisor.

I would also like to express my deep gratitude to Dr. Roy Czernikowski of the Department of Computer Engineering at Rochester Institute of Technology. Dr. Czernikowski made a personal dedication to my education from day one, and I cannot thank him enough.

Finally, I would like to thank Dr. Cockburn of the Department of Computer Engineering at Rochester Institute of Technology. His knowledge of control engineering is simply unmatched, and without his help this work could not have been completed.

My sincerest thanks to you all.

Abstract

Control systems are found in many aspects of modern life. From household appliances such as computer controlled ovens and refrigerators to complex missile defense systems, the popularity and importance of automated controllers has grown exponentially over the past few decades [28]. This thesis proposes to develop a simulation framework that can be used in the development of such digital control systems in academic environments.

Control Systems Design is a common subject for engineering students world-wide. Many tools exist to help students design and simulate digital controllers, such as MATLAB and SIMULINK, but actually implementing and testing a designed controller is important as well. Students learn far more from their studies when they have an actual laboratory experiment to help relate the abstract concepts of engineering to the real life design problems [32]. A number of simplified physical systems such as the inverted pendulum and digital servo are common in academic environments, but designing controllers for more practical systems is difficult due to the prohibitive costs associated with the equipment involved [21].

Most simulation frameworks readily available for students focus on the controller itself. They aid in design of the controller's mathematical model, but do not aid in physically testing the actual implementation of the controller. The Control System Plant Simulator will simulate the object a digital control system designer wishes to control – referred to as a 'Plant'. The Control System Plant Simulator follows the Hardware-in-the-loop concept [16] in that it takes the place of a physical plant. Designed and implemented controllers are attached to the Control System Plant Simulator, which will behave just as an actual plant will from the viewpoint of the controller. The simulator will read input from a digital controller external to the host computer of the simulator. It will evaluate the input in real-time, and provide output to the digital controller just as the actual plant will.

The Control System Plant Simulator can be used to aid in the development of control systems. Individuals can use the tool to prototype control systems without being forced to use expensive or limited physical resources. This can reduce production costs of control systems, and make possible more realistic control systems development in academic environments where resources

are more limited. This allows for the education of the next generation of control system designers and implementers in a more realistic setting.

Table of Contents

List of Figures

List of Tables

Glossary

Component: Basic element of the Framework. The components of the Framework are the User Interface, the Plant Simulation Unit (PSU), the File I/O Unit, the Physical ports and the Pseudo ports.

Controller: External to the framework, the controller is any system designed to produce input signals for the framework for the purpose of controlling the active simulated plant.

Engineering Units: See **Plant Engineering Data**.

File I/O Unit: Component responsible for writing output and status to files.

Framework: The system as a whole. Framework refers to all of the components and the interfaces between them.

Interface: The communication that is allowed to take place between any two components

I/O Specification: An I/O Specification is a set of parameters that divide the physical binary ports into smaller labeled 'virtual' ports, and scales analog port input to plant engineering values.

Physical Data Values: Data as it appears on the physical data registers.

Physical Port: Physical ports are the software representations of the physical interfaces of a connected data acquisition unit. Input physical ports read data from the hardware interface and store it as a physical data value. Output physical ports write physical data values to the hardware interface. Physical ports may be either input or output, and may provide or accept either digital or analog data.

Plant: Any physical object that is to be controlled. For the purposes of this program, 'Plant' will refer to the simulation of such an object.

Plant Definition Mode: Framework operation mode when the user is allowed to define and modify the parameters that specify the simulated plant.

Plant Definition Type: A specific methodology by which a plant has been defined. Examples include State Space defined plants or Transfer Function defined plants.

Plant Engineering Data: Data that has been formatted for the simulated plant.

Plant Simulation Unit (PSU): Responsible for simulating a defined plant. Takes plant engineering data values from the Pseudo input ports, uses them to evaluate the internal simulated plant, and provides output, as plant engineering data values, to the Pseudo output ports.

Pseudo Port: Pseudo ports provide the bridge between physical ports, and the inputs and outputs of a plant. Each Pseudo port is connected to an input or output of the plant, and one physical port. Input pseudo ports are responsible for converting the physical data values provided by physical ports into plant engineering data that is meaningful to the plant. Output pseudo ports are responsible for converting plant engineering data provided by the plant back into physical data values that can be sent to the hardware interface by the physical ports. Pseudo ports may be Digital Physical Ports may be divided into smaller ports.

Simulation Cycle: The Simulation Cycle consists of acquiring a set of inputs from the external data acquisition device. These inputs are formatted by pseudo ports into plant engineering data values and provided to the PSU. The PSU uses these inputs to define a new set of outputs based on executing the defined plant model for one cycle. These engineering units are returned to a pseudo port to be converted back into physical data. Physical output is forwarded to the physical port connected to the pseudo port that performed this conversion. This data is sent to the physical interface by the physical port. This cycle is repeated until the simulation is stopped.

Simulation Mode: Framework operation mode when simulation cycles are executed. The user can do nothing in this mode but stop the simulation.

User Interface (UI): This component provides the framework's external interface for control of the plant simulation and to monitor plant operation.

1. Introduction and Motivation

This thesis focuses upon the study of control systems, the plants they are designed to control, and real time simulation. It provides a low-cost flexible plant simulator that can be used for the development of digital controllers. Furthermore, this plant simulator provides benefits in academic teaching and research environments.

Digital Controllers are control systems that are implemented using digital computers to control a subject system, called a plant. Figure 1 shows a typical digital control system. An input signal r(t) is supplied, indicating to the control system what the desired output of the plant should be. This input is sampled through the use of an analog to digital converter. The difference of this signal and the discrete output of the plant is provided to the controller. The controller takes an input signal and generates a control command using a difference equation. This difference equation can be expressed in the frequency domain (after taking Z-transforms) as a transfer function provided that all initial conditions are zero. The digital output of the controller is converted to analog signals through the use of a digital to analog converter and provided to the plant. The plant will use these values to control its actuators. Sensors monitor the controlled variable of the output to close the loop. [24]

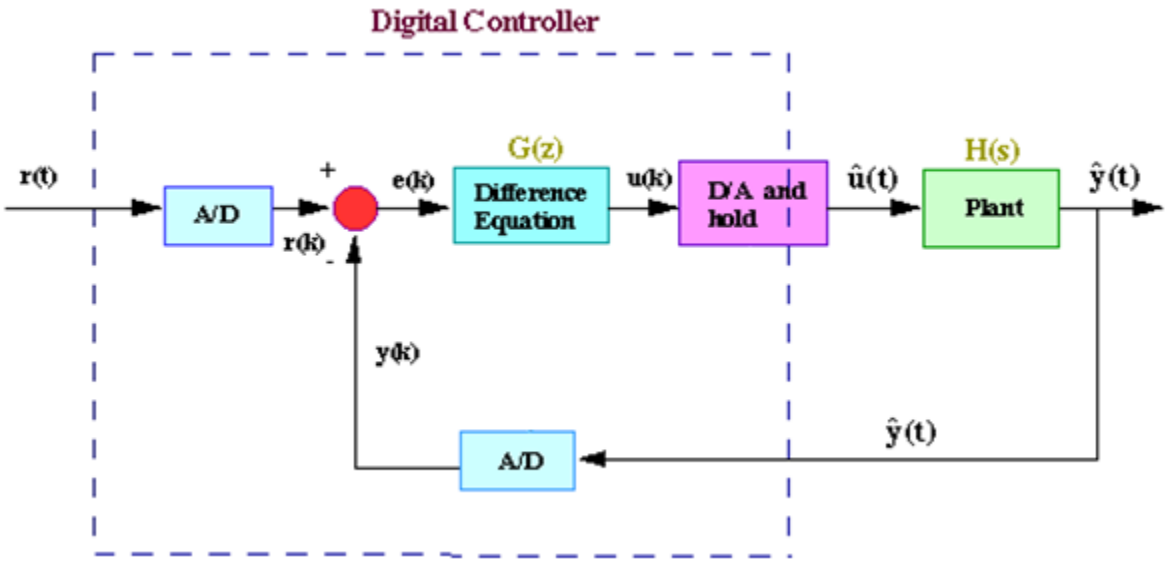

Figure 1. Typical Digital Control System [24]

Control System simulators have been developed in the past. The Shadow Plants Dynamic Simulation Testbed by Honeywell is an example of one such simulator [29]. However, these simulation systems are typically tailor-made for specific situations, products, or markets. Additionally, they can be prohibitively expensive, which reduces their ability to be widely adopted. They often lack the flexibility required for extendable interfaces (both user interfaces or data interfaces). Oftentimes specific information about the controller to be designed is required before these tools can even be used, and rarely are they designed to be connected to external controllers. The vast majority of these simulators are designed to enable the design of the theoretical controller, and not for testing the implemented design.

The Control System Plant Simulator developed here addresses the following issues:

- Current plant simulation frameworks are very expensive. The Control System Plant Simulator is an open source project, provided at no cost.
- Most simulators are designed to simulate the Control and the plant together. They simulate the mathematical models of each, and provide information about how these models interact. While extremely helpful during initial design phases, these simulators can not test the implementation of these designs. The Control System Plant Simulator runs a hardware-in-the-loop simulation that takes the place of a physical plant. It is designed to be connected to an implemented controller, and behave as a real plant would. This eliminates the expense or danger of testing actual equipment, while fully exercising the implemented controller.
- Hardware-in-the-loop test simulations have been made for specific situations, products, or markets. The Control System Plant Simulator provides a general framework upon which plants of different natures may be simulated by simply providing a model of the plant.
- The Control System Plant Simulator is extendable in that user interfaces, plant descriptions, and physical interfaces may be updated or customized with little difficulty.

The open source nature of the project ensures that deeper and more complex customizations are possible.

- Most hardware-in-the-loop simulators require specialized equipment or test boards to run. The Control System Plant Simulator is a Windows XP application. A port of the computation portion of the simulator utilizing Ardence RTX® Real-time Extension for Control of Windows® is provided for Real-Time simulation. While the initially designed plant simulator was tailored to a particular data acquisition board, the simulator is flexible enough that other data acquisition boards can be substituted with minimal modifications.

The Control System Plant Simulator is designed with academic environments in mind. It is of the utmost importance for students to be able to implement their designs as physical controllers, but it is often too expensive to test these controllers on physical targets. The targeted plant may be expensive, fragile, limited, or dangerous. The Control System Plant Simulator (CSPS) can be used in place of this equipment, and provide results for the students to monitor and determine the success of their controller design. Because of the low cost goal, the system may be used to prototype controllers developed by each student individually. This allows students the ability to work on controllers, correcting mistakes as they go, without the constant use of limited or unavailable lab equipment. The CSPS is designed to simulate many different kinds of plants, and as such has a flexible front end. The graphical user interface may be swapped out for any other designed by end users and implementers. These interfaces may be used to display animated versions of the plant (such as watching the level of water in a tank raise and lower) and are designed such that there is no impact to the system simulation. Section 3.1.1.1.2 describes UI Interchangeability in greater detail. The CSPS is designed to be portable. All operating system calls are abstracted in a separate OS Layer. Should an operating system change be necessary, only this layer will need to be altered.

The importance of simulating a designed digital controller is self evident. Simulation enables the designer to see what is going to happen before spending considerable effort implementing a design, or putting expensive equipment – and potentially human life – at risk with an untested design. However, one cannot ignore the physical experimentation phase of design altogether. In their paper "Theory, simulation, experimentation: an Integrated Approach

to Teaching Digital Control Systems", Harold Klee and Joe Dumas go to great pains to outline how important it is to both design and implement controllers [23]. They argue that "The combination of hands-on experience and computer simulation with the more traditional theoretical lecture material provides a well-rounded learning experience that better prepares the students to implement digital control systems in the real world" [23]. They describe a three step course for undergraduate students that begins with theory and how to design a digital controller mathematically. The controller the students design is then simulated to work out any problems with their theoretical design. Finally the controller is implemented and connected to real physical hardware. This 'start-to-finish' design and implementation is invaluable to students as it provides the whole picture.

This argument for making sure that students implement their controller designs and attempt to actually run their controllers on plants has been made a number of times. In their paper "Merging Physical Experiments Back Into The Learning Arena", Bjarne A Foss, Tor I Eikass, and Morten Hovd bemoan the recent trend "towards increased use of simulation in engineering education, coupled with a decline of the use of physical experiments."[12] They admit that the expense of physical equipment is prohibitive, but outline a number of reasons why it is important to implement the controllers they design. "The typical student therefore finds it motivating to work with laboratory experiments. A successful laboratory experiment is some proof that the student has been able to perform a task which is of relevance to the real world."[12] This argument is central to the purpose of this thesis. Simply simulating a theoretical controller is insufficient when learning control systems. One must attempt to build the controller, and make it actually control something. The CSPS is not a physical controller, but it behaves as one. To the student developing a controller, the CSPS must be treated exactly the same as a physical controller. The student must produce a controller that produces output and is insensitive to time delays, noise, and many other physical characteristics that are not present when simulating the control-plant interaction as a set of transfer functions.

The purpose of this thesis is to simplify the physical requirements for the experimentation phase of control systems education. Academic institutions often are forced to choose between asking students to design controllers for low-cost plants that are too simple to be realistic, or never attempting to control anything at all, relying entirely on simulations. This thesis provides

a framework upon which a plant may be created and run on a Windows XP environment, with the option of running in real time should a user decide to use Ardence RTX® Real-time Extension for Control of Windows®. This plant simulation may be used in place of a real physical plant, making physical equipment less important. Students may perfect their designs at their own workstations without the need of additional equipment.

2. Background

The following section provides background necessary to understand the concepts upon which the Control System Plant Simulator is based on. Basic information about Control Systems, Real Time Simulation, and Hardware-In-The-Loop simulation are provided.

2.1 Control Theory

2.1.1 Introduction

Control Theory is the study of how to automate the control of a dynamic system. The system under control, referred to as a plant, may have any number of variables that must be controlled. These controlled variables could be, for example, the speed of a car, the temperature of a room, or the physical level of water in a tank. Control designers develop controllers that are connected to the plant. These controllers set inputs and provide resources to the plants in such a way that the plant provides outputs at desired levels. For example, an automobile cruise control system regulates the throttle position in order to produce a desired speed. In any case, some input must be provided to the controller to indicate the desired plant output. This input is referred to as a reference, or set point value [28].

Sometimes, feedback is required to stabilize the plant. Feedback occurs when the output from the plant is provided to the controller. This allows the controller to monitor the response of the plant and determine how much the plant's output differs from the desired output [14]. See figure 2 for details.

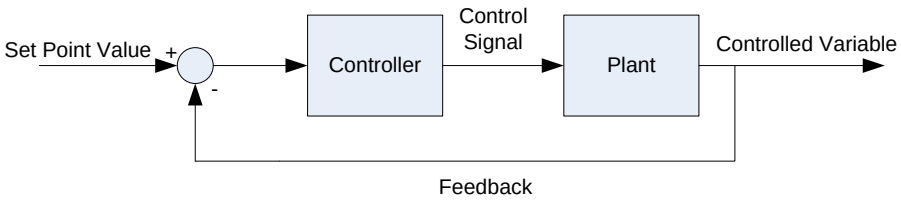

Figure 2: Control System Block Diagram

2.1.2 Transfer Function

In order to develop controllers, a designer must have some idea of how a plant will respond to the inputs provided. This means that the plant must be modeled mathematically. Equations must be developed that describe how the output of the plant is generated given an input. These are typically high order ordinary differential equations. If the differential equation is linear and time invariant, then it can be solved using Laplace transforms. The ratio of the Laplace transform of the output or response function to the Laplace transform of the input or driving function is referred to as a transfer function [14]. Transfer functions provide a compact and convenient description of linear time invariant systems, and they can be easily obtained from the dynamic equations that describe the system.

For example, Equation 1 provides a general differential equation mapping input x to output y that may define a plant to be controlled.

$$Equation\ 1 \qquad a_0\frac{d^n}{dy^n}y + a_1\frac{d^{n-1}}{dy^{n-1}}y + \cdots + a_{n-1}\frac{d}{dy}y + a_n y = b_0\frac{d^m}{dx^m}x + b_1\frac{d^{m-1}}{dx^{m-1}}x + \cdots + b_{m-1}\frac{d}{dx}x + b_m x \qquad (n \geq m)$$

The transfer function for this system is obtained by taking the Laplace Transforms of Equation 1 assuming zero initial conditions. This is shown in Equation 2, which simplifies to Equation 3.

$$Equation\ 2 \qquad G(s) = \frac{\mathcal{L}\left[b_0\frac{d^m}{dx^m}x + b_1\frac{d^{m-1}}{dx^{m-1}}x + \cdots + b_{m-1}\frac{d}{dx}x + b_m x\right]}{\mathcal{L}\left[a_0\frac{d^n}{dy^n}y + a_1\frac{d^{n-1}}{dy^{n-1}}y + \cdots + a_{n-1}\frac{d}{dy}y + a_n y\right]}$$

$$Equation\ 3 \qquad G(s) = \frac{Y(s)}{X(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_{m-1}s + b_m}{a_0 s^n + a_1 s^{n-1} + \cdots + a_{n-1}s + a_n}$$

This allows a plant to be described as a rational function of the complex variable *s* instead of a differential equation. Such a system is referred to as an nth order system, with n being the highest power of *s* in the denominator [14]. It is important to note that a transfer function can only map one input to one output. In order to describe a plant that has more than one input or output using transfer functions, a matrix of transfer functions must be used.

One may also factor the numerator and denominator polynomials that make up the transfer function, and express the transfer function as a product of factors, also called the zero-pole-gain form:

$$Equation\ 4 \qquad G(s) = K\frac{\prod_{i=1}^{m}(s - z_i)}{\prod_{i=1}^{n}(s - p_i)}$$

where K is a scalar constant referred to as the gain, the set of z_i are the zeros of the system, and the set of p_i are the poles of the system. Note that these poles and zeros may be complex numbers. However for the system to be implementable, all complex poles or zeros must come in complex conjugate pairs [31]. Poles and zeros are helpful when analyzing the response of the system the transfer function represents. They are used in the design of controllers, and to study the stability, causality, phase, and other factors of the system [14].

2.1.3 State Space Equations

As noted earlier, defining systems with multiple inputs and multiple outputs (MIMO) in terms of transfer functions is cumbersome, as the number of transfer functions required is equal to the number of inputs multiplied by the number of outputs. More common is the use of a set of state space matrices. Ogata defines the term state to be "the smallest set of variables (called state variables) such that the knowledge of these variables at $t = t_0$, together with the knowledge of the input for $t \geq t_0$, completely determines the behavior of the system for any time $t \geq t_0$" [28]. That is, the current output can be determined as a function of the current states and the current set of inputs. Additionally, the next state can be determined as a function of the current state and current set of inputs as well.

If *x* is a vector of *n* state variables, and *u* is a vector of *r* inputs, then the state equations are defined by

Equation 5

$$\dot{x}_1(t) = f_1(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$
$$\dot{x}_2(t) = f_2(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$
$$\vdots$$
$$\dot{x}_n(t) = f_n(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$

and the outputs $y_1(t)$, $y_2(t)$, … $y_m(t)$ are

Equation 6

$$y_1(t) = g_1(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$
$$y_2(t) = g_2(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$
$$\vdots$$
$$y_m(t) = y_m(x_1, x_2, \dots x_n;\ u_1, u_2, \dots, u_r; t)$$

If the system is linear and time invariant, the state space equations become

Equation 7 $\quad \dot{\boldsymbol{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$

Equation 8 $\quad \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)$

Here, **A** is the State Matrix, **B** is the Input Matrix, **C** is the Output Matrix, and **D** is the Feedthrough Matrix. The following diagram shows the relationship of these matrices and the inputs and outputs of the system:

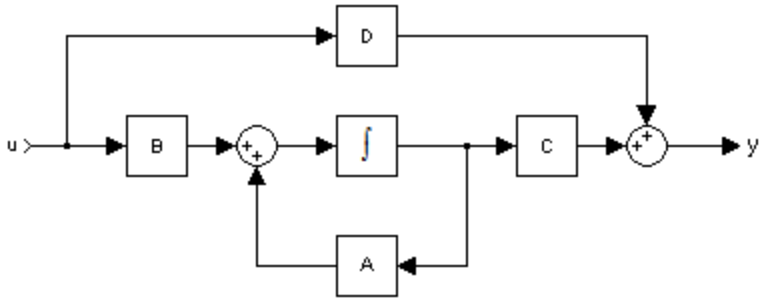

Figure 3: Block Diagram of State Space Matrices [28]

One notes that equation 9 relates the state space equations back to the transfer function of the system.

$Equation\ 9 \qquad H(s) = \mathrm{C}(s\mathrm{I}\text{-}\mathrm{A})^{-1}\mathrm{B}+\mathrm{D}$

2.1.4 Converting Transfer Functions to State Space

Transfer functions can be converted to state space equations in order to simplify their evaluation. A realization is a set of state space equations equivalent to a given transfer function. There are infinitely many state space realizations for a transfer function. Of the many possible realizations, the "Controller Canonical Form" is one of the simplest since it can be found directly from the coefficients of the transfer function. However, the resulting state-space matrices are not suitable for numerical computations. [14].

Consider the following n order system

$$Equation\ 10 \qquad G(s) = \frac{Y(s)}{X(s)} = \frac{b_0 s^n + b_1 s^{n-1} + \cdots + b_{n-1} s + b_n}{a_0 s^n + a_1 s^{n-1} + \cdots + a_{n-1} s + a_n}, \ a_0 \neq 0$$

Note that some of the b_i, but not all, may be zero as the numerator does not have to have the same order as the denominator.

The denominator can be normalized by dividing every element by a_0, as shown in equation 11.

$$Equation\ 11 \qquad G(s) = \frac{Y(s)}{X(s)} = \frac{\beta_0 s^n + \beta_1 s^{n-1} + \cdots + \beta_{n-1} s + \beta_n}{s^n + \alpha_1 s^{n-1} + \cdots + \alpha_{n-1} s + \alpha_n}$$

Note that the system has only one input and one output, and that β_0 will equal 0 unless the order of the numerator is equal to that of the denominator.

The State matrix is obtained by taking the negative of each coefficient in the denominator, ignoring the coefficient of the leading term, and placing them in the first row to form a companion matrix as follows:

$$Equation\ 12 \qquad \mathrm{A} = \begin{bmatrix} -\alpha_1 & -\alpha_2 & \cdots & -\alpha_n \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The Input matrix has a 1 as its very first element, all others are 0

$$Equation\ 13 \qquad \mathrm{B} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The output matrix values are equal to the original numerator values – the denominator values (skipping the first) scaled by β_0.

$$Equation\ 14 \qquad \mathrm{C} = [\{b_1 - (a_1\beta_0)\} \quad \{b_2 - (a_2\beta_0)\} \quad \cdots \quad \{b_n - (a_n\beta_0)\}]$$

Finally, the feedthrough matrix consists of only one element of value β_0.

[13]

$$Equation\ 15 \qquad \mathrm{D} = \beta_0$$

Equations 12 – 15 form the "controller' realization of the transfer function given.

2.1.5 Discrete-Time Systems

With the adoption of the microprocessor, more and more systems are controlled digitally. This means that the input of these systems is not read continuously, but rather at discrete time intervals. Likewise, new output values are calculated at discrete time intervals. Typical Digital to Analog Converters (DACs) operate as a zero order hold, meaning these outputs are held constant until they are recalculated.

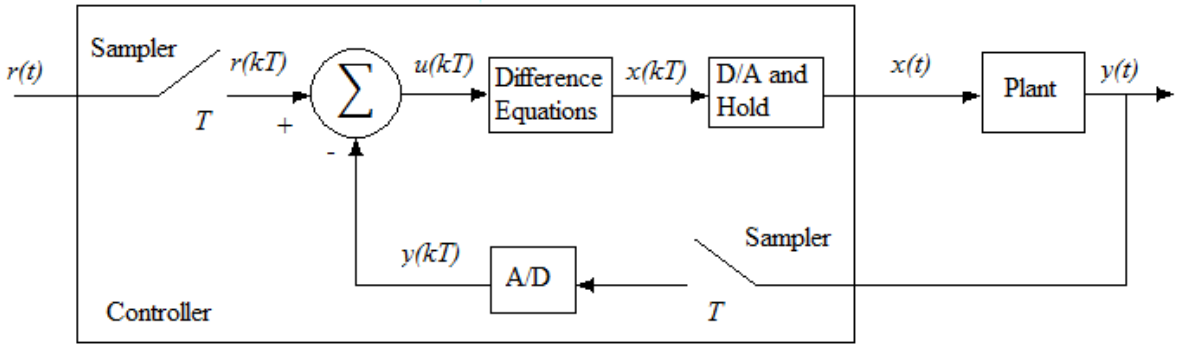

Figure 4: Digital Control System

To reflect these differences, instead of using the Laplace transform, the Z transform is used.

$$Equation\ 16 \qquad Z = \sum_{k=-\infty}^{\infty} u_k z^{-k}$$

Due to quantization in the analog to digital converter, u_k is an approximation of the value of u(t) at time kT where T is the sampling period. There are numerous methods by which a system may be descretized, including the forward rectangular, the backward rectangular, and bilinear methods [13].

The Z transformed transfer function is still expressed as a ratio of two polynomials, and can still be described in terms of poles and zeros. State space equations may be used to describe the transfer function (or functions) as well. Instead of the continuous values, discrete state space equations calculate the values at discrete time instants.

$$Equation\ 17 \qquad \mathrm{x}(k+1) = \mathrm{A}\mathrm{x}(k) + \mathrm{B}\mathrm{u}(k)$$

$Equation\ 18 \qquad \mathbf{y}(k) = \mathrm{C}\mathbf{x}(k) + \mathrm{D}\mathbf{u}(k)$

where *k* is the specific time index. [13]

2.1.6 Converting Continuous Time to Discrete Time Systems

Calculating discrete equivalents of continuous systems is commonly done by various approximations. The actual Z transform is simply the Laplace transform of the ideal sampled function. The forward rectangular, backward rectangular, and bilinear sampling approximation methods result in the following substitutions to convert a Laplace transformed continuous system to a Z transformed discrete system:

Table 1: Discrete Equivalence Equations

Discrete Equivalence Equations	
Forward Rectangular	$Equation\ 19 \qquad s = \left(\frac{z-1}{T}\right)$
Backward Rectangular	$Equation\ 20 \qquad s = \left(\frac{z-1}{zT}\right)$
Bilinear	$Equation\ 21 \qquad s = \left(\frac{2}{T}\frac{z-1}{z+1}\right)$

In order to discretize a set of state space equations, start with the original state space equations defined in Equations 7 and 8, and perform a Laplace transform on it, yielding

$Equation\ 22 \qquad s\mathbf{X} = \mathrm{A}\mathbf{X} + \mathrm{B}E$

$Equation\ 23 \qquad \mathbf{Y} = \mathrm{C}\mathbf{X} + \mathrm{D}U$

One may substitute for *s* as per equations 19, 20, and 21, convert to discrete equations, and solve for x(k+1) and y(k). Doing so results in the following discrete equivalent state space equations:

Forward Rectangular:

$Equation\ 24 \qquad \mathbf{x}(k+1) = (\mathrm{I}+\mathrm{A}T)\mathrm{x}(k) + \ T\mathrm{B}u(k)$

$Equation\ 25 \qquad \mathbf{y}(t) = \ \mathrm{C}\mathrm{x}(k) + \ \mathrm{D}\mathrm{u}(k)$

Backward Rectangular:

$Equation\ 26 \qquad \mathbf{x}(k+1) = (\mathrm{I}\text{-}\mathrm{A}T)^{-1}\mathrm{x}(k) + \ (\mathrm{I}\text{-}\mathrm{A}T)^{-1}\mathrm{B}\,Tu(k)$

$Equation\ 27 \qquad \mathbf{y}(t) = \ \mathrm{C}(\mathrm{I}\text{-}\mathrm{A}T)^{-1}\mathrm{x}(k) + \ \{\boldsymbol{D} + \boldsymbol{C}(\boldsymbol{I} - \boldsymbol{A}T)^{-1}\boldsymbol{B}\}\,Tu(k)$

Bilinear:

$$Equation\ 28 \qquad \mathbf{x}(k+1) = \left(\boldsymbol{I} + \frac{\boldsymbol{A}T}{2}\right)\left(\boldsymbol{I} - \frac{\boldsymbol{A}T}{2}\right)^{-1}\mathrm{x}(k) + \ \left(\boldsymbol{I} - \frac{\boldsymbol{A}T}{2}\right)^{-1}\mathrm{B}\sqrt{T}\,u(k)$$

$$Equation\ 29 \qquad \mathbf{y}(t) = \sqrt{T}\mathrm{C}\left(\boldsymbol{I} - \frac{\boldsymbol{A}T}{2}\right)^{-1}\mathrm{x}(k) + \ \left\{\boldsymbol{D} + \boldsymbol{C}\left(\boldsymbol{I} - \frac{\boldsymbol{A}T}{2}\right)^{-1}\frac{\boldsymbol{B}T}{2}\right\}u(k)$$

[13]

It is worth noting that the Forward Rectangular method is the least stable, and may require very high sampling rates to yield acceptable results.

2.1.7 Nonlinear Systems

All of the previous examples hold true for linear equations. However, there are many situations where the state equations described in Equations 7 and 8 are nonlinear. That is, systems for which the principle of superposition does not apply [28]. There is no simple equation for simulating nonlinear systems: the response to two inputs cannot be calculated by treating one input at a time and adding the results. To handle this problem, such systems can be approximated by linear equations for a particular operating region. In actuality, truly linear physical systems are rare. Many electromechanical, hydraulic, and pneumatic systems are only linear for limited operating ranges.

Consider a one dimensional system $\dot{x} = f(x)$. The function $y = f(x)$ is represented by a curve, and the tangent at a given point x^* represents a linear approximation of the function at

that point. Thus,

$Equation\ 30 \qquad \delta\dot{x} = \frac{dy}{dx}\delta x$ [3]

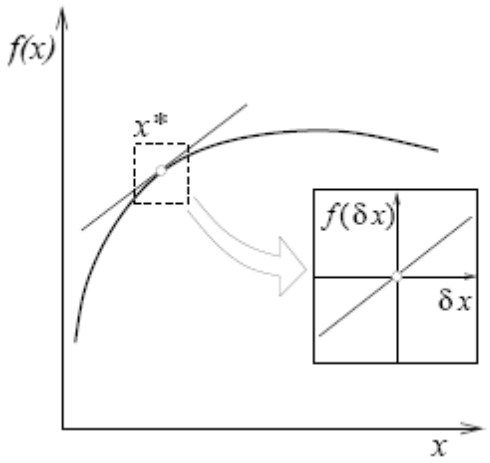

Figure 5: Linearization of Nonlinear Equation [3]

This process remains the same as the dimensions of the system increase. The state space equations of a linearized system then become

$$Equation\ 31 \qquad \begin{bmatrix} \delta\dot{x}_1 \\ \delta\dot{x}_2 \\ \vdots \\ \delta\dot{x}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_1 \\ \vdots \\ \delta x_n \end{bmatrix}$$

Where $\delta x = x - x^*$. [3]

This version of the simulator only simulates linear time-invariant systems.

2.2 Real Time Systems

A real-time system is a system that performs operations under specific timing constraints imposed on it by the real-time behavior of the 'outside' world [35]. Real time systems must produce results and perform actions that are both correct, and delivered within very specific time constraints. Reading an input line, or writing to an output line must be done while the data is still meaningful. These timing constraints are typically deterministic, meaning they refer to values and not statistics or averages [35].

Real time systems are divided into two main groups: Hard and Soft real time systems. A real time system that is 'Hard' is one that absolutely must meet the deadlines imposed on its tasks. If one of these deadlines is missed, even slightly, the system experiences a critical failure. Examples of hard real time systems include pace makers and nuclear reactor control systems. Soft real-time systems can withstand missing constraints. The effectiveness or quality of the data may be diminished, but failures are not critical. Examples of soft real-time systems include streaming video systems. If constraints are not met, the quality of the video may degrade, but the entire system is not lost.

Real-Time Operating Systems are a natural fit for digital control systems. They are designed to operate while interfacing with a real-world system, and provide certain guarantees as to what specific time intervals inputs will be read and outputs will be provided. Accurate sampling, and timely input updates are vital to the successful operation of a digital control system.

The CSPS provides a compiled version of the Computation Kernel that has an RTX® port of the OS abstraction layer. RTX® provides a set of libraries and a realtime subsystem that is installed as a Windows kernel device driver. The libraries provide access to this realtime subsystem through an API defined by RTX® to closely match that of the typical Win32 API. The RTX® architecture was designed such that it extends, not encapsulates, Windows to prevent interference with the Windows kernel. RTX® runs its own thread scheduler that preempts all Windows processing. This is how real time operation is realized. The typical RTX® application

consists of two programs: a program written for Win32, and another written for the RTSS Subsystem. Shared memory and IPC objects establish communication between these two processes. This allows a developer to use all of the tools available for windows development, and still have real time performance. [1]

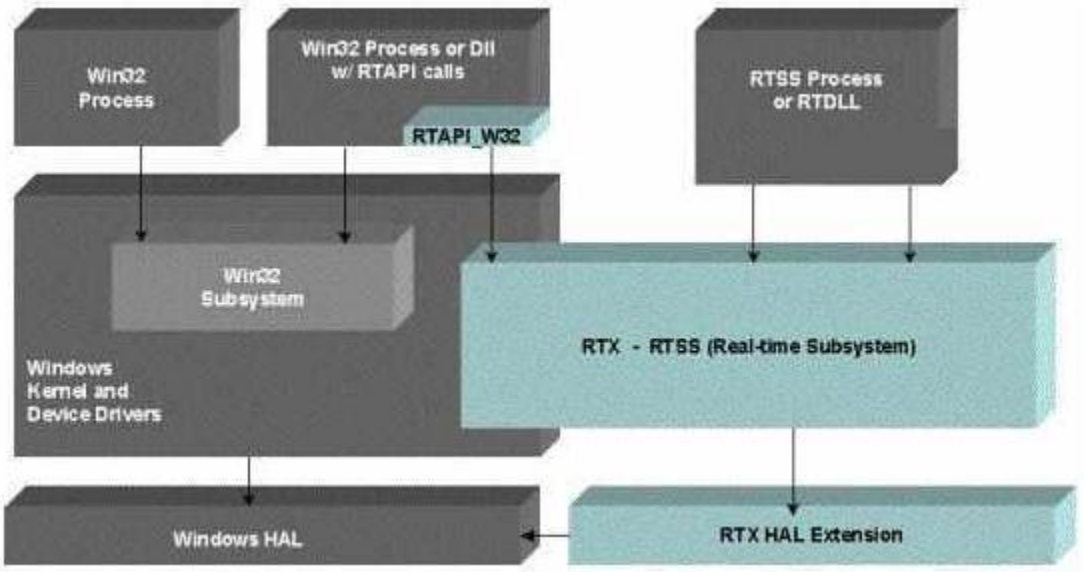

Figure 6: RTX® Architecture

2.3 Hardware-In-The-Loop Simulation

Hardware-In-The-Loop simulation is a technique that replaces real plants with simulated counterparts. These simulated plants accept inputs and produce outputs just as the real one does. From a 'black-box' perspective, the Hardware-In-The-Loop simulation and the physical plant appear exactly the same (figure 5).

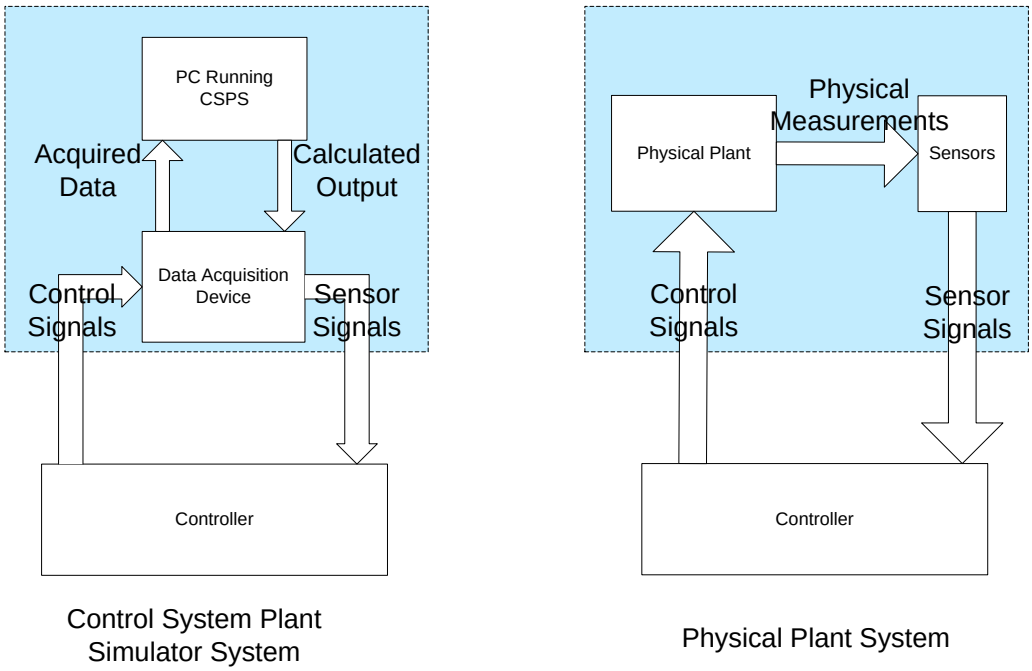

Figure 7: CSPS and Physical Plant as Black Box Systems

In his paper "Hardware-In-The-Loop Simulation and Its Application In Control Education" Wojciech Grega states that "The concept of 'hardware-in-the-loop (HiL) method is to use a simulation model of the process and the real target hardware."[16] He goes on to describe how a simulated model must process signals in real time and provide them to a controller as the real physical plant would. The controller provides control signals to the simulation model just as it would the actual plant it was designed to control. In this way, the simulation appears as a 'black-box' with inputs and outputs matching that of the target plant. The simulation takes the place of an actual physical plant. Figures 6 and 7 show the difference between the classical experimental setup (figure 6) and the hardware-in-the-loop setup (figure 7). In the classical

setup, a PC runs the implementation of the digital controller. The controller feeds output to a data acquisition device that will convert the digital commands to input for the plant. The plant physically responds to these input commands. Typically sensors monitor the state of the plant and provide information in the form of output signals. These signals are then fed back into the data acquisition device which provides them to the controller running on the PC. The hardware-in-the-loop method diagrammed by figure 7 shows the idea that the physical hardware that makes up a plant may be fully replaced by another computer. In figure 7 PC2 runs the implementation of the digital controller. The controller produces command signals that are converted to the exact same input as before by a data acquisition device. The difference is that the converted signals are given to PC1 through an I/O board, instead of a physical plant. PC1 runs a simulation of the plant. It takes these inputs, runs the simulation based upon them, and provides the output through the I/O board just as the plant did in figure 1. This is returned to PC 2 running the controller through the data acquisition device to close the loop.

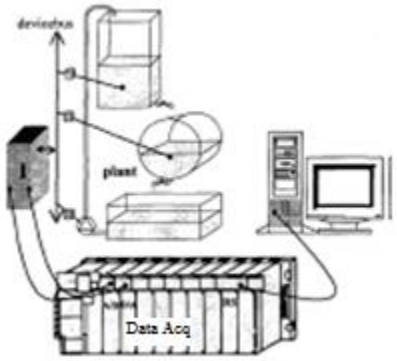

Figure 8: Classical Control Setup [16]

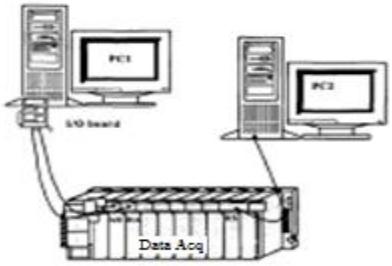

Figure 9: Hardware-in-the-loop method [16]

The Control System Plant Simulator behaves exactly the same way. The goal of the Control System Plant Simulator is to provide a framework that completely takes the place of almost any physical plant – with limitations imposed only by the physical constraints of the target computer and data acquisition board. Grega argues that the use of physical plants for experimental verification of controllers designed by students is extremely important for illustrating theory at work, but continues to explain that complex laboratory implementations of industrial plants are too expensive and dangerous for education purposes. He proposes Hardware-In-The-Loop simulations instead. This is the

motivation behind the Control System Plant Simulator. The CSPS provides a means by which academic environments can simulate a plant to be controlled by the students' controllers. The simulation may be simple, and used as a verification step before testing the controller on a physical plant, or it may be used entirely as a substitute for complex equipment like a jet engine. Grega explains that "The key to hardware-in-the-loop simulation method lies in the software."[16] He outlines specifically the need for such a tool as the Control System Plant Simulator.

Other hardware-in-the-loop simulators exist, such as those implemented by the dSPACE GmbH company, but these are targeted towards the TMS320 DSP processors and are highly specialized systems. Grega explains that the "drawback of this configuration is the price: the code generation software is very expensive due to its small market share."[16]. For his simulations, he had to generate a real time hardware-in-the-loop plant of his own as few academic institutions could afford the dSPACE tools. Grega concludes his paper arguing that without actually applying designed controllers to actual industrial situations, control engineering may as well be taught as an applied mathematics course. He states that "Often, the control projects are not complemented by practical activities due to the high cost of laboratory equipment."[16].

3. The Control System Plant Simulator

The Control System Plant Simulator is a solution to the problem of how to have a large number of students develop and test digital controllers on realistic plants. It has been shown that the benefits of actual in-lab work are many, but the use of real physical hardware is oftentimes impractical [16]. The Control System Plant Simulator is a Hardware-In-The-Loop suite of programs that provide a simulation framework for any plant that can be expressed as a set of State Space equations, as a traditional transfer function, or as a set of poles and zeros. The system is designed with flexibility in mind. It is fully expected that users of the Control System Plant Simulator will make simple, but powerful enhancements. As such, a complete description of the system design and implementation are required. This section provides this information. The overall system architecture is described using a top down approach. This section provides details of the implementation including class interaction diagrams and sequence diagrams. Finally, the delivered product is described, along with comments about the difficulties encountered and the compromises made to overcome them.

3.1 System Design

This section describes the design of the Control System Plant Simulator using a top down approach.

3.1.1 Process Space Overview

The Control System Plant Simulator is separated into three distinct process spaces as shown in figure 8. This modular design provides for a great deal of the flexibility the system provides for users. Should major adjustments be required to a portion of the system, such as the pluggable user interface, only one process space must be altered. The first process space is the User Interface. This is the process that users interact with. The User Interface design allows the framework user to create user interfaces customized for plant simulation. Individually developed User Interfaces do not even have to be developed using the same programming language as the remainder of the system. The

second process space is the Win32 process. This process deals with configuration and non-real-time interactions with the Windows operating system (such as file management). It stores, configures, saves, and loads all configuration information until the system is ready to begin simulation. The third and final process is the Computation process. The Computation process is responsible simulating the configured plant, and for reading and writing to the data acquisition device connected to the system. It is the lowest level process, and is given its own process space to allow it to be launched on a separate operating system (such as RTX®) or as a kernel level process. These processes send commands and data between each other through interfaces implemented through shared memory and named semaphores.

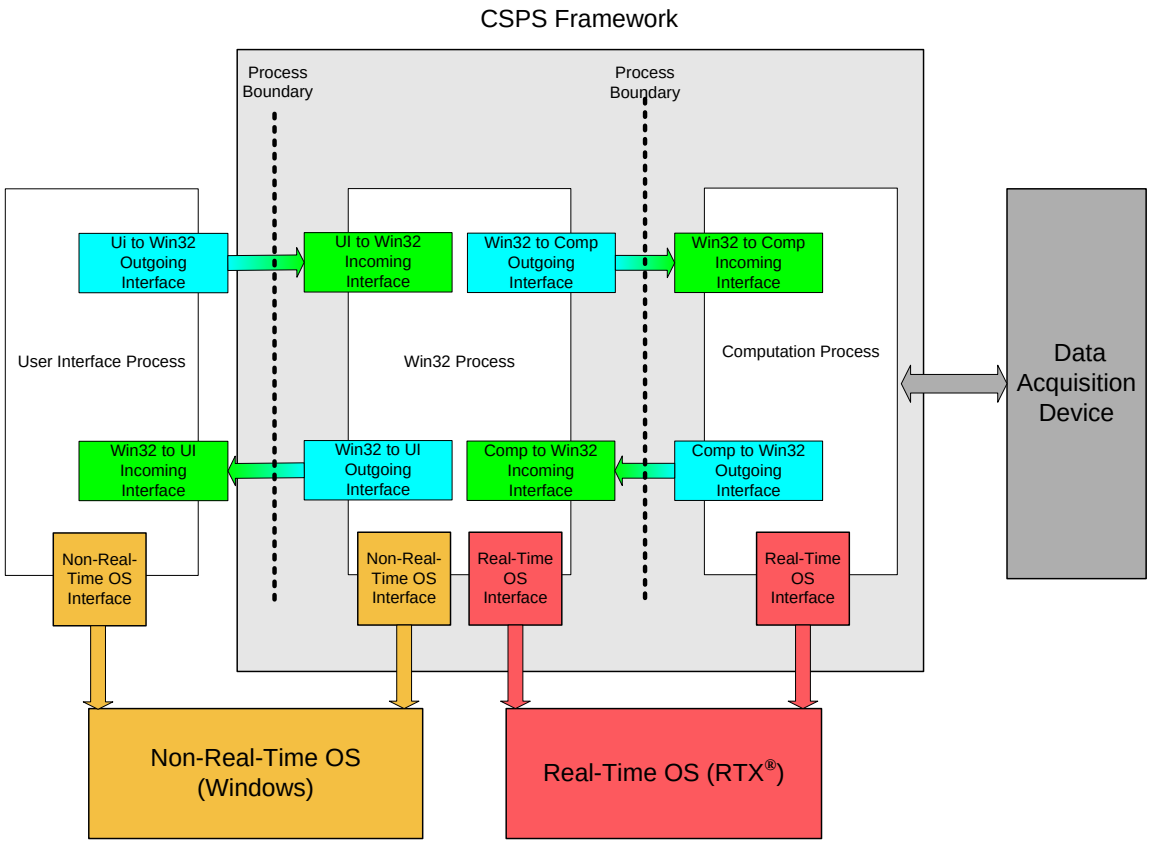

Figure 10: System Diagram

3.1.1.1 User Interface Process

The User Interface process is the top level process. It is what the individual users interact with directly. This process does not run in real time, and does not interfere with the processing of the Computation Process. This process allows the user to initialize, configure, start, and stop the simulations, and is designed for configurability. This is a component of the CSPS, and will be launched as part of the CSPS start-up procedure, but it is expected that individual framework users will design their own User Interface processes. This gives framework users the ability to tailor their interface to the plant they wish to simulate by adding plant specific details and animations to their interface. It is a vital part of the CSPS simulation, but exists outside of the CSPS framework as users will be making it themselves.

3.1.1.1.1 Responsibilities

The User Interface process is responsible for the following requirements. It is the responsibility of the user interface designer to make sure that each of these requirements is met in the user interfaces they design.

- User Interfaces must field all user commands and input.
- Format user input into structures meaningful to the Win32 process.
- Send properly formatted input and commands to the Win32 process.
- Inform the user of any relevant system information provided by the Win32 process. This includes but is not limited to error messages, informational messages, the current plant configuration, the current port configuration, and input or output values.
- May provide the user the ability to specify files that can be used for loading or saving configurations, or for logging data.
- Send the command to terminate the simulator when the user shuts the User Interface down.

3.1.1.1.2 Interchangeability

The User Interface process is designed to be interchangeable. Framework users typically will want a user interface tailored to the plant being simulated. One developer may wish to model the level of fluid in a tank and desires a user interface with a specialized GUI displaying an animated tank with visually changing water levels. Some developers may wish to allow users of their plant to modify the plant constants and will provide entry windows in their user interface for new values. Others still may wish to prevent any alterations and simply provide a 'Start' and 'Stop' button or text command. To allow for the infinite range of possible interfaces, the user interface is launched as a distinct application. Developers may write these user interface applications any way they wish, provided they adhere to the protocol for inter-process communication between the User Interface and the Win32 process. To facilitate this process, a user interface dynamically linked library (DLL) is provided as part of the Control System Plant Simulator suite of applications. The User Manual also provides specific instructions that describe how to develop user interface programs to work with the framework.

3.1.1.1.3 Inter-process Communication

The Win32 process must be informed which User Interface process to launch at boot time. User interfaces communicate with the Win32 process through an interface implemented with named semaphores and shared memory. This interface and the protocol for its use are described in detail in section 3.2 of the User Manual, The UiWinInterface API. Every user interface must make proper use of these defined semaphores and shared memory locations to operate with the system as a whole. A dynamically linked library is provided to help simplify this process. This library has been exported for use with Microsoft Visual Basic applications as well as traditional windows applications.

3.1.1.2 The Win32 Process

The Win32 process serves as the main entry point for the entire system. The Win32 process is the first process launched, and is responsible for launching the other

two processes. All commands from the user interface are fielded by the Win32 process, which then validates and executes them. It stores all configuration information before a simulation starts, and manages the Computation process.

3.1.1.2.1 Responsibilities

The Win32 process is responsible for:

- Launching the User Interface and Computation processes.
- Converting plants provided by the user to state space notation for the Computation Process.
- Storing plant and port configurations before a simulation is started.
- Validating plant and port configurations when the user attempts to start a simulation.
- Configuring the Computation process with validated plant and port configurations upon simulation start.
- All file input and output, including saving and loading configuration data.
- All logging, which must be done in a way that does not interfere with any other process in the system.
- Manages updates to the user interface, including any messages or port value updates.

3.1.1.2.2 Startup

Upon startup, the Win32 process launches and configures the Computation process and the User Interface process. It establishes connections to and from each of the processes, and initializes all system shared variables.

3.1.1.2.3 Logging

The Win32 process manages three different optional logs. These logs are written to files that the user may view upon simulation completion. Log messages may be informational messages describing how the simulation is running, critical messages

indicating major errors, and IO messages indicating the current input and output of the system. All log messages are forwarded to the UI process which can do with them what it wants, but critical logs are always written to standard output as a minimum. IO logs are written in comma delimited format, allowing them to be imported directly to database programs like Microsoft Excel with no difficulty.

3.1.1.3 Computation Process

The Computation process handles all calculations needed to simulate a configured plant. This process handles plant evaluation and input and output to physical hardware. It consists of two main parts: the Plant and the Port Manager. The plant is responsible for running the mathematical model of the plant. by repeatedly executing simulation cycles. Simulation cycles start by fetching data from the data acquisition device. This data is converted into plant engineering data values, and is used to calculate the next set of outputs. These outputs are converted back to physical data and are provided to the data acquisition device to be sent to the controller.

The Port Manager handles reading input for the Plant, and writing output from the plant to a connected Data Acquisition device. Both of these elements are configured by the Win32 process which obtained the configuration information from the User Interface process. It may also report changes in system state or data through the logging system.

3.1.1.3.1 Responsibilities

The Computation process is responsible for:

- Performing the plant simulation.
- Converting continuous plants to discrete
- Managing physical Input and Output.
- Mapping physical Input and Output to the inputs and outputs of the specified plant.
- Updating the Win32 process with status through update and log operations.

3.1.1.3.2 Physical Ports

The Computation process manages the physical I/O through a data acquisition device that is known at compile time. Should a data acquisition device be altered, a small section of the Computation process will need to be ported. Due to the modular design of the Computation process, the effort required to make this port is minimal. See User Manual Section 4: Porting a New Data Acquisition Device for specifics on how to develop for different data acquisition devices.

Physical ports are read and written to at periodic intervals. The user may configure these intervals individually for each physical port. This allows for data acquisition boards that take long periods of time to update their ports. When a port is read, the raw data from the acquisition board is read and stored to a cache. When the plant needs input data, the data is fetched from this cache. When the plant has calculated a new set of outputs, the output data is written back to the cache. When it is time for the output to be refreshed, these cached output values are written to the data acquisition device.

3.1.1.3.3 Pseudo Ports

While the physical ports available on a workstation may not often change, the plant simulated is expected to be changed frequently. Each plant will have different input and output requirements. Some will use simple integer input, others will need access to precise floating point input. Some will be Single Input Single Output (SISO) systems, and others will be Multiple Input Multiple Output (MIMO) systems. In addition, a plant defined for the Control System Plant Simulator should be able to be saved, and loaded on another system running the Control System Plant Simulator without altering the plant itself. To satisfy all of these requirements, the Control System Plant Simulator wraps Physical Ports in Pseudo Port objects. Pseudo Ports are named objects, whose names match perfectly with the inputs and outputs of a particular defined plant. Pseudo ports must reference real physical ports to ‘connect’ them to the inputs and outputs of the defined plant. Pseudo ports are also responsible for reading or writing physical input

from the physical data cache, and for converting the input and output of the current data acquisition device into engineering data meaningful to the plant as shown in figure 9.

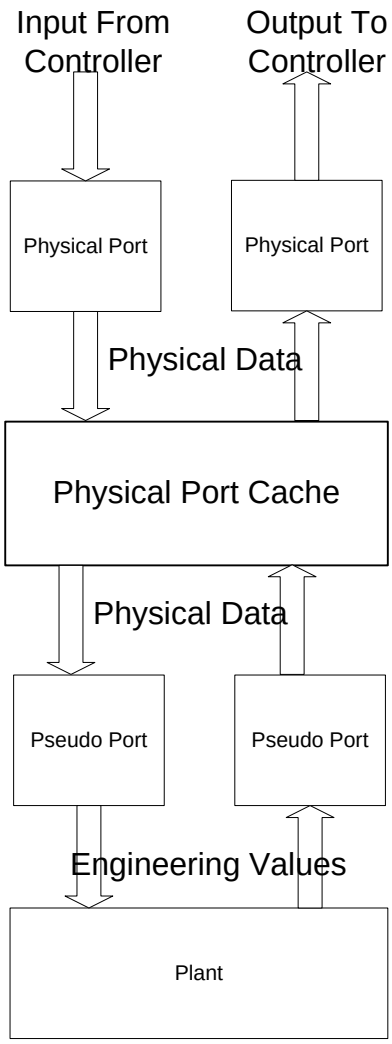

Figure 11: PhysicalPort and PseudoPort Interaction

Pseudo Ports come in two distinct types: binary and analog. Binary pseudo ports may be any size up to the maximum size of the physical port they wrap. Several binary pseudo ports may map to the same physical port provided the physical port is large enough (for example, a 32 bit physical port can be used as two 16 bit binary pseudo ports). Analog ports, on the other hand, have a one to one relationship with their physical ports. Analog pseudo ports convert physical data values to engineering data values through a simple linear scaling procedure that converts the largest possible engineering

value acceptable to the plant to the highest voltage that may be read from the physical port and the smallest possible engineering value to the lowest voltage. For example, a physical analog port may only be able to provide voltages between 0 and 5 volts, but this value represents a pressure measured in 50 to 200 PSI. Analog pseudo ports perform this conversion.

3.1.1.4 Interfaces

The three main system processes must communicate between each other to pass configuration, update, informational, and control messages. An incoming and outgoing interface is defined for each process. These interfaces are the means by which the processes communicate.

These interfaces are implemented through the use of shared memory and named semaphores. Figure 10 shows a generic interaction across process boundaries through these interfaces. Incoming interfaces run a separate thread of execution that block on a named semaphore (The Interface Semaphore). Action is initiated by sending commands to the connected outgoing interface that cause the semaphore to be unlocked. All outgoing values must be set in shared memory before unlocking this semaphore. These include values that indicate which operation has been called, as well as provide any parameters necessary for execution of the requested operation. The thread is then blocked on another named semaphore (The Return Semaphore).

When the Interface Semaphore is unlocked, the incoming interface thread unblocks and determines what operation was called by checking values in shared memory. Once the specific operation has been determined, the incoming interface thread copies any relevant parameters out of shared memory. All input or output parameters must be copied to or from memory. With all the necessary data, the incoming interface thread invokes the proper function on one of the objects in its process space. Once this function call returns, the incoming interface thread copies return data back to shared memory and unlocks the Return Semaphore indicating that the requested operation was completed. The thread that blocked when it made a call on the outgoing interface

unblocks and copies this data out of shared memory. The data is then provided to the calling process through a more traditional return value.

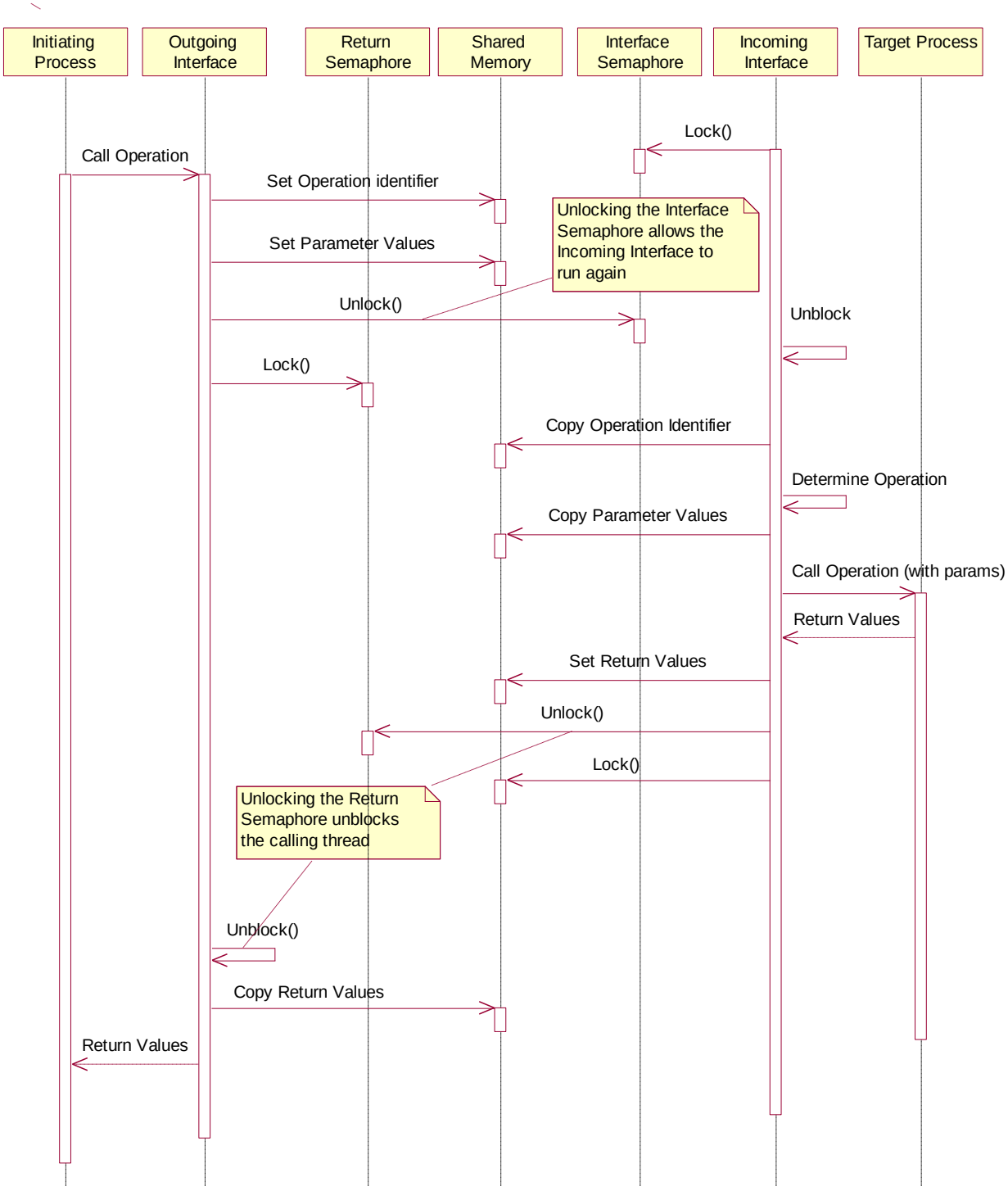

Figure 12: General Inter-Process Communication

For Example, the Computation Kernel has an incoming interface that allows the Win32 process to add pseudo ports to the configuration. The Win32 process calls an operation on the Outgoing Interface passing to it the pseudo port information as a parameter. This data is copied to shared memory, along with an identifier indicating that the 'AddPort' operation has been called. The interface semaphore is unlocked, and the caller blocks on the return semaphore. The Incoming interface thread unblocks and reads the identifier out of shared memory. It determines that the 'AddPort' operation has been requested and copies the parameters out of shared memory. It then calls the AddPort operation on the PortManager object, passing in the copied parameters. When the call returns, the return value is copied into shared memory and the return semaphore is unlocked. The incoming interface thread blocks on the interface semaphore again, while the calling thread in the original process space unblocks. It copies the return value out of shared memory and returns it to the caller in the Win32 process.

3.1.1.4.1 Multithreading Considerations

Incoming Interfaces themselves are thread safe in that only the thread created by the incoming interface's inheritance of class Thread accesses the private functions that access shared memory. This thread is blocked by the interface semaphore until an operation is ready. It is the responsibility of the outgoing interface connected to this incoming interface to make sure that the semaphore is not unblocked when memory is vulnerable.

While the incoming interface executes a requested operation, the initiating outgoing interface is blocked on the reply semaphore. It is the responsibility of the incoming interface to unlock the reply semaphore once execution is complete, even in the case of a failure. They access shared memory that must be protected against multiple access and modification.

There is one unit of shared memory that a pair of connected outgoing and incoming interfaces access. This shared memory holds parameters, return values, and data indicating which function has been called. Access to this shared memory MUST be limited to one and only one thread of execution. No operation may be called until the previous operation has completed. This means that it is the responsibility of the

Outgoing Interface to hold off any calling process until previous memory accesses have completed. This is done through the use of a protection semaphore. All Outgoing Interfaces are required to lock the protection semaphore upon function entry, and unlock it when the return value has been set and shared memory access is no longer needed.

In addition, all outgoing interfaces must wait for their connected incoming interface to complete their operations before unlocking the protection semaphore, even when there is no return value for the operation invoked. This is because the incoming interface relies on shared memory to receive input parameters and an indication of which function has been called. If an outgoing interface unlocked the protection semaphore, it is possible another operation could be called and would change shared memory to indicate a completely different operation. To prevent this issue, all outgoing interfaces are required to block on the reply semaphore until the connected incoming interface releases it.

3.1.1.5 Threading Structure

This section describes the threads that run within the CSPS.

3.1.1.5.1 User Interface

The threading structure of the user interface is completely up to the designer of the interface, but at least one additional interface thread will always be present. The User Interface has an additional thread of execution that handles incoming operation requests. It blocks until a request is made, unblocks to handle it, and blocks again once the request has been completed.

3.1.1.5.2 Win32 Process

The Win32 process one main thread of execution that is responsible for creating all of the objects within the system. It configures the objects and launches the process spaces during the initial boot. Once the system has been established, this thread blocks until the User Interface reports that the user has requested the system to shut down, at which time it handles tearing down all of the objects it created.

Beyond the main thread, the Win32 process consists of two incoming interface threads, one for handling requests from the User interface, and one for handling requests from the Computation Kernel. These threads block until an operation is requested, handle any requested operations, and then wait for more.

3.1.1.5.3 Computation Kernel

Like the Win32 process, the Computation Kernel has a main thread responsible for establishing all of the objects upon startup. It also has an incoming interface thread that handles operation requests from the Win32 process.

In addition to the startup and communication threads, the Computation Kernel has a thread dedicated to the Plant, the Plant Watchdog, and the Port Cache. The Plant thread handles the calculations required during simulation. It is manages the calculation portion of a Simulation Cycle. The Plant Watchdog thread monitors the plant and makes sure that all deadlines are met. The Port Cache thread handles reading from and writing to the physical data acquisition device.

3.1.2 Detailed Design

The CSPS Framework is a complex set of components designed for flexibility, alteration, and customization. To facilitate future development of the framework, this section has been written to provide the detailed design for the system as a whole. This includes the overall design for the operating system abstraction, each process space, and the classes that compose the processes.

3.1.2.1 Operating System Abstraction

Portability of the system is of prime importance to the CSPS Framework. A full port of the entire system is provided for RTX® in this distribution, and others are likely to be required in the future. The CSPS Framework requires the following from a target operating system:

- Support for named semaphores across process spaces. The target OS must provide semaphores that threads may block upon in one process space, and be signaled by another.
- Support for shared memory. The target OS must provide some shared memory or shared variable support that allows two process spaces to set, access, and alter the same variable.
- Support for threads.
- Sleep or pause commands. The target OS must provide the ability to stop the execution of a thread for a specified period of time.

In order to provide for a highly portable system, several classes were designed to provide an abstraction layer between the operating system and the Control System Plant Simulator. These classes may have to be ported should one or more of the system processes be moved to another operating system.

3.1.2.1.1 Duality of OS Abstraction Classes

Each of the Operating System Abstraction classes provides two different Operating System implementations. The first is a primary interface that is always expected to be implemented. The second is an optional interface for the Computation Kernel, should the Computation Kernel use an interface other than the primary one. The User Interface and Win32 process are expected to run on the same operating system, referred to as the 'Primary OS', but the Computation Kernel may run on a separate operating system extension, referred to as the 'Secondary OS', such as RTX®. By setting some precompiler definitions in the LocalDefinitions.h file, a developer may turn secondary code on or off. When porting from one system to another, the Operating

System Abstraction classes will have to change to reflect the new operating system. If only the operating system upon which the Computation Kernel launches is changing, only the secondary interface code will have to change.

3.1.2.1.2 Class Thread

The thread class abstracts operating system threads from the Control System Plant Simulator suite of applications. This class is responsible for creating, running, and managing a thread of execution.

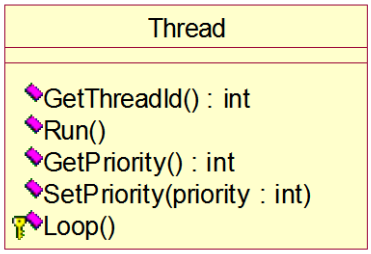

Figure 13: Class Thread

3.1.2.1.2.1 Responsibilities

The Thread class is responsible for managing all operating system specific operations required to start and run a thread of execution. It must manage the priority of the thread it created, and is responsible for releasing any resources dedicated to the thread upon destruction.

3.1.2.1.2.2 Usage

The Thread class is an abstract parent class. Other classes in the system that have operations that are required to execute within their own threads of execution inherit from class Thread and overwrite the virtual Loop() operation. Class Thread insures that the Loop() operation will be called by a new operating system thread of execution. Should Loop() return, the thread will complete and terminate.

3.1.2.1.3 Class InterfaceSemaphore

The Interface Semaphore class wraps traditional operating system semaphores. It must provide access to the semaphores in such a way that the same semaphore may be

accessed from separate process spaces at the same time. Typically this is handled as a named semaphore. When the semaphore is opened with a name that already exists in another process space, the existing semaphore is provided.

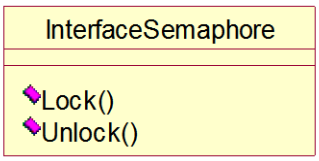

Figure 14: Class InterfaceSemaphore

3.1.2.1.3.1 Responsibilities

Interface Semaphore objects are required to be available across process space boundaries. If an Interface Semaphore is opened in one process with the same name as an Interface Semaphore in another, instead of creating a new Interface Semaphore, a handle to the original semaphore must be returned. Interface Semaphores are counting semaphores as opposed to binary. They decrement a count every time Lock() is called until that count reaches zero. Calling Lock() on an Interface Semaphore with count 0 causes the system to block.

3.1.2.1.3.2 Usage

The Interface Semaphore object is a simple named semaphore. It is constructed with a name and a maximum count. Currently, the CSPS framework utilizes semaphores that have a maximum count of 1, but the InterfaceSemaphore class allows for greater maximum counts for future extensions. Each time Lock is called on any of the interface semaphores constructed with the same name, the count is decremented. When lock is called on a semaphore that has reached zero, the calling process is blocked. Unlock adds resources to this count.

3.1.2.1.4 Class SharedMemoryInterface

The Control System Plant Simulator requires the use of Shared Memory in order to send information between process spaces. Shared Memory is memory that more than one application can access. The SharedMemoryInterface class manages the creation and cleanup of memory shared between process spaces.

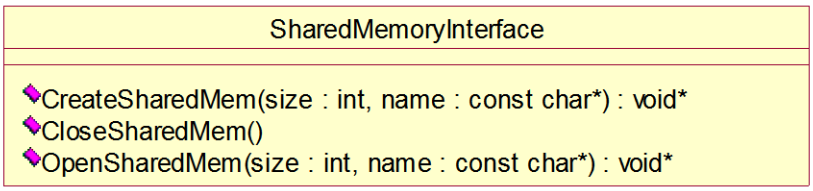

Figure 15: Class SharedMemoryInterface

3.1.2.1.4.1 Responsibilities

The SharedMemoryInterface is responsible for creating memory that can be accessed between different process spaces. When a SharedMemoryInterface is constructed in one process space with the same name as another SharedMemoryInterface in another process space, it is the responsibility of this class to make sure that the memory created is the same between the two. The class is also responsible for making sure that the memory is properly released when it is done with it. There is no process that is designated as one responsible for the creation of shared memory. All SharedMemoryInterface objects are constructed the same way. It is the responsibility of this class upon construction to look for other blocks of shared memory that were created with the same name. If shared memory exists, the newly constructed SharedMemoryInterface deals with the existing memory. If it does not, the SharedMemoryInterface creates a new block of shared memory.

3.1.2.1.4.2 Usage

Users of the SharedMemoryInterface create or open existing shared memory by indicating how much memory to allocate, and what name should be associated with that

memory. If shared memory exists with the given name, a pointer to the original memory will be returned, otherwise new memory will be allocated and tagged with the provided shared memory name.

3.1.2.1.5 Class FileInterface

The FileInterface class handles all file input and output operations. It is simplified from traditional Operating System file IO in that only a few select items can be written to or read from files, all values are written as text files, one complete line at a time. These files are input or output only.

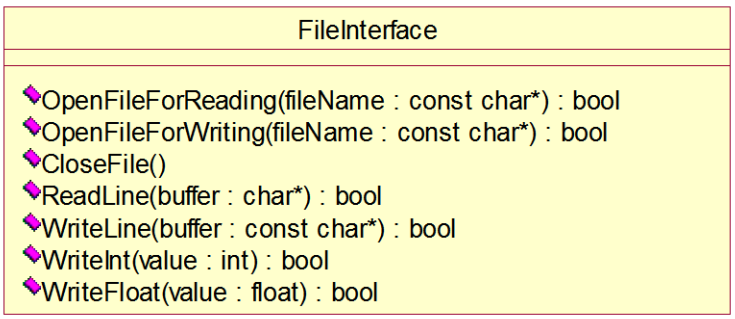

Figure 16: Class FileInterface

3.1.2.1.5.1 Responsibilities

FileInterface is responsible for opening files for read or write access, and for providing a common set of functions that can be called to read or write to these files. It wraps the file input/output API provided by the particular operating system it abstracts.

3.1.2.1.5.2 Usage

Files may be opened for read or write access only. Any time a file is opened for write access, the file is overwritten. Data is read or written one line at a time. When writing data, only the primitive types ‘integer’ and ‘float’ are supported. This is done to intentionally limit the capability of the Control System Plant Simulator to exactly what was needed.

3.1.2.2 *Interface Parent Classes*

The interface parent classes are abstract classes that provide the basic functionality required by the individual interfaces. . Interfaces are the unidirectional means of communication between process boundaries. On one side of the boundary, an OutgoingInterface allows a process to make a request on another process. On the other side, an IncomingInterface allows a process to field these requests, process them, and return any results. Each process boundary has a pair of these interfaces: one for each direction of communication.

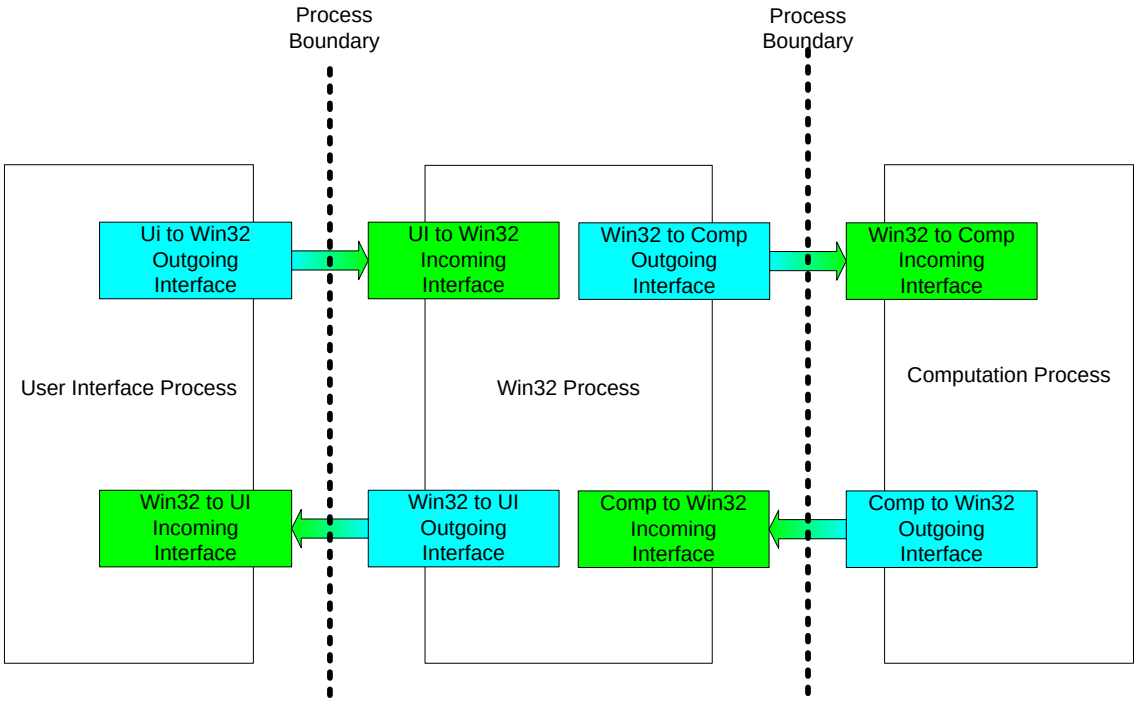

Figure 17: Interfaces between Process Boundaries

3.1.2.2.1 IncomingInterface

The IncomingInterface class is the parent class for all incoming interfaces. Incoming interfaces run their own thread of execution, which waits for operations to be

called by Outgoing Interfaces in other process spaces. It is an abstract class that must be overridden.

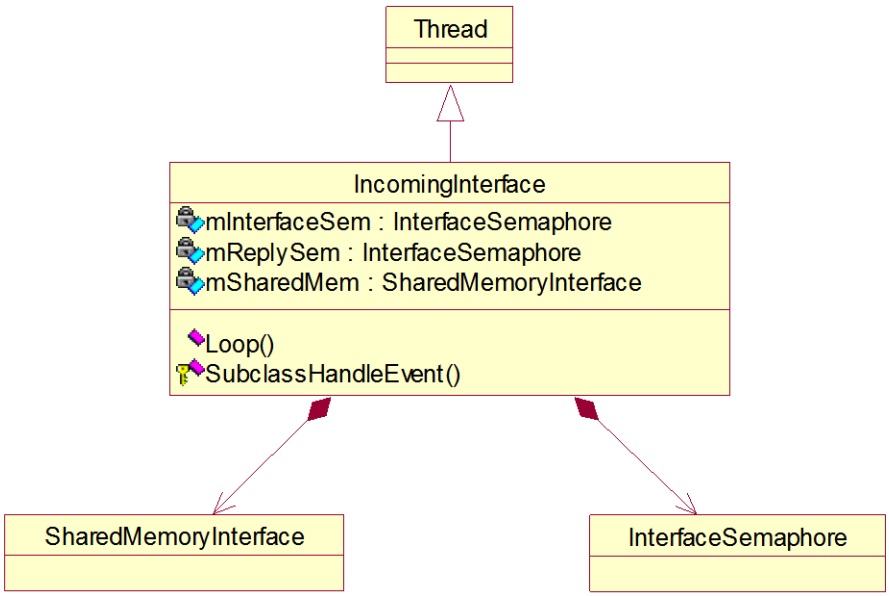

Figure 18: Class IncomingInterface

3.1.2.2.1.1 Responsibilities

The IncomingInterface class is responsible for managing the incoming side of an inter-process interface. It contains a semaphore for the interface, and another for return values, opening the local reference to them upon construction. It initializes an additional thread of execution that is always ready to field incoming operation requests. When this thread is unblocked by an Outgoing Interface, it forwards the request to child classes that are responsible for determining which operation was requested and how best to fulfill the operation request.

3.1.2.2.1.2 Usage

The IncomingInterface class inherits from class thread. It intiates a new thread of execution that blocks on the Interface Semaphore opened at construction time. When an Outgoing Interface unlocks, the thread is unblocked. The thread then calls SubclassHandleEvent which is a protected operation that must be overwritten by

inheriting child classes. In this operation, the child classes will determine what operation was requested by the outgoing interface, and how best to service that request.

3.1.2.2.2 OutgoingInterface

The OutgoingInterface class is the parent class for all interfaces that make operation calls on other processes. It is an abstract class that must be overridden and provides no implementation on its own.

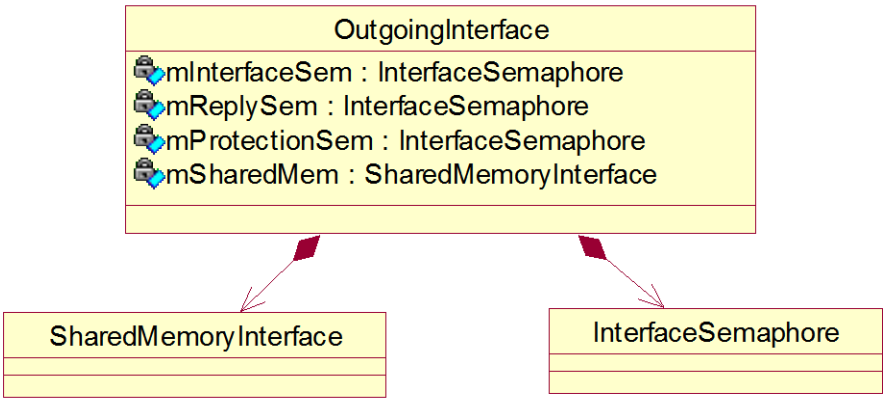

Figure 19: Class Outgoing Interface

3.1.2.2.2.1 Responsibilities

The OutgoingInterface class is responsible for managing the outgoing side of an inter-process interface. It handles the interface semaphore, and the return semaphore, opening the local reference to them upon construction, as well as shared memory between this Outgoing interface and a connected incoming interface.

3.1.2.2.2.2 Usage

The OutgoingInterface simply provides a guaranteed platform that inheriting OutgoingInterfaces can rely upon. Classes that inherit from OutgoingInterface are guaranteed to have interface events, semaphores, and shared memory.

3.1.2.3 User Interface

The User Interface process is unique in that little ‘official’ implementation is provided. Individual developers are expected to use the provided classes or dynamically linked library to build their own customized user interfaces. Two complete user interfaces are provided and detailed in The User Manual, but these are included as examples to help users understand how to use the provided classes.

The User Interface process is launched by the Win32 process, and is the front end with which users interact. There are no constraints on what the user interface provides, and it may be written in any programming language provided that language has an implementation for shared memory and named semaphores. It allows the user to command, control, and configure the Control System Plant Simulator. User interface developers may limit the functionality they provide to their users, but at minimum this interface must make a call to configure the plant and ports, and a start execution call.

Developers of a user interface may use the incoming and outgoing interface classes, or the dll that abstracts them.

3.1.2.3.1 WinToUiIncomingInterface

The WinToUiIncomingInterface handles all incoming function calls from the Win32 process. These incoming calls are informational in nature, and as such, their use is not required for operation.

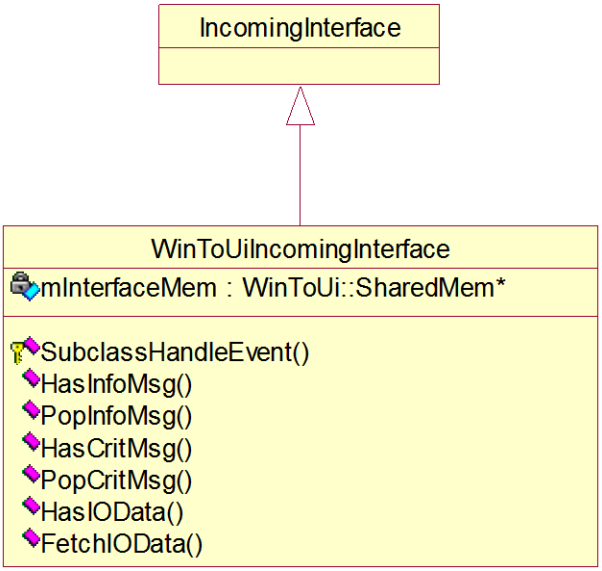

Figure 20: Class WinToUiIncomingInterface

3.1.2.3.1.1 Responsiblities

The WinToUiIncomingInterface is responsible for handling the incoming message and I/O update function calls from the Win32 process. It is also responsible for buffering incoming messages and updates. The User Interface itself may check these buffers as necessary.

3.1.2.3.1.2 Buffered Messages

Unlike other incoming interfaces, the WinToUiIncomingInterface cannot simply forward the requested function call to a specific object. No requirement is placed upon individually developed user interfaces to provide operations to handle these function calls. Callback operations are insufficient as well because some programming languages, such as Visual Basic 6, do not allow exterior threads to access their data members. The WinToUiIncomingInterface operates on its own Windows launched thread that cannot access Visual Basic callback operations.

As a result, an additional responsibility has been placed on the WinToUiIncomingInterface to buffer all incoming information. Informational messages and critical messages are stored in individual circular buffers. User Interfaces are

required to check to see that data exists in the buffer by calling the HasInfoMsg and HasCritMsg before popping messages out of the buffer. The oldest messages are always returned first. If more data is fed into the buffer before the User Interface removes it, the oldest messages are discarded completely. Input / Output data is not written to a buffer. Instead only the most recent Input / Output data is stored.

3.1.2.3.2 UiToWinOutgoingInterface

The UiToWinOutgoingInterface provides a set of operations that the user interface may call when it wants to invoke a function on the Control System Plant Simulator.

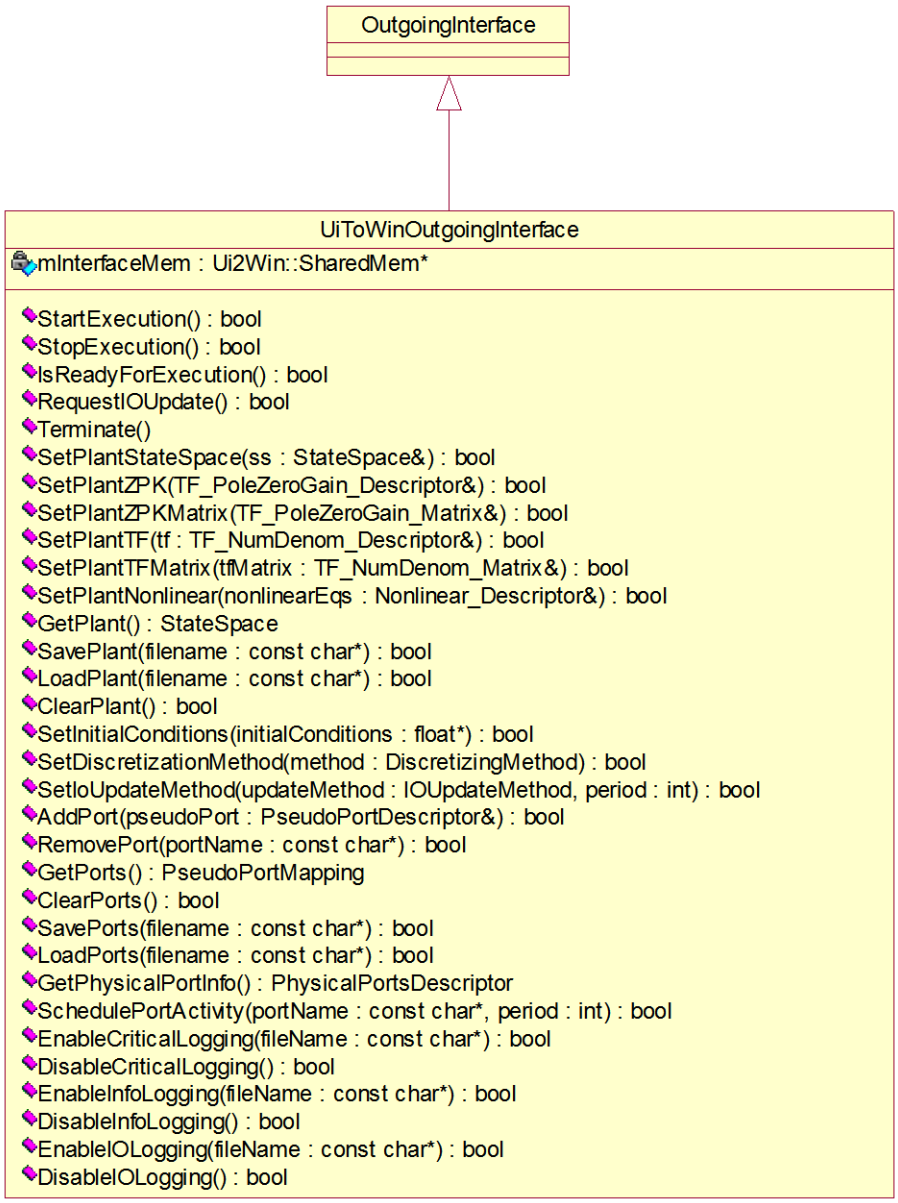

Figure 21: Class UiToWinOutgoingInterface

3.1.2.3.2.1 Responsiblities

The UiToWinOutgoingInterface is responsible for providing a complete set of functions for the user interface. Each of the provided operations block the calling thread from the user interface until the operation returns from the Win32 process. The UiToWInOutgoingInterface must also protect against multiple concurrent accesses to any of these functions. Only one may be called at a time.

3.1.2.3.3 UiWinInterface Dynamically Linked Library

The UiWinInterface Dynamically Linked Library instantiates a UiToWinOutgoingInterface a WinToUiIncomingInterface, and provides a simplified set of operations to access the functions provided by those interfaces. It is designed to be fully compatible with visual basic for quick and simple graphical user interface design. For detailed information about the UiWinInterface API and every function provided, see section 3.2 of the User's Manual: The UiWinInterfac API.

3.1.2.4 Win32 Process

The Win32 process is the main program in the Control System Plant Simulator. It is the first process launched by the user, and is responsible for launching each of the other processes in turn. Note that while the name implies complete reliance on Win32, it is possible to port the Win32 process to other operating systems by porting the OS Abstraction layer to the target OS. The Win32 process fields commands and configuration information from the user interface, and is responsible for saving and loading plant configurations and pseudo port mappings. Plant configurations and pseudo port mapping information are stored in data managers until the system is prompted to begin execution. At that time, the Win32 process verifies that the configurations are compatible, sends the configuration down to the Computation Kernel, and commands it to begin execution. The Win32 process consists of a number of managers, each of which is responsible for a different aspect of the system.

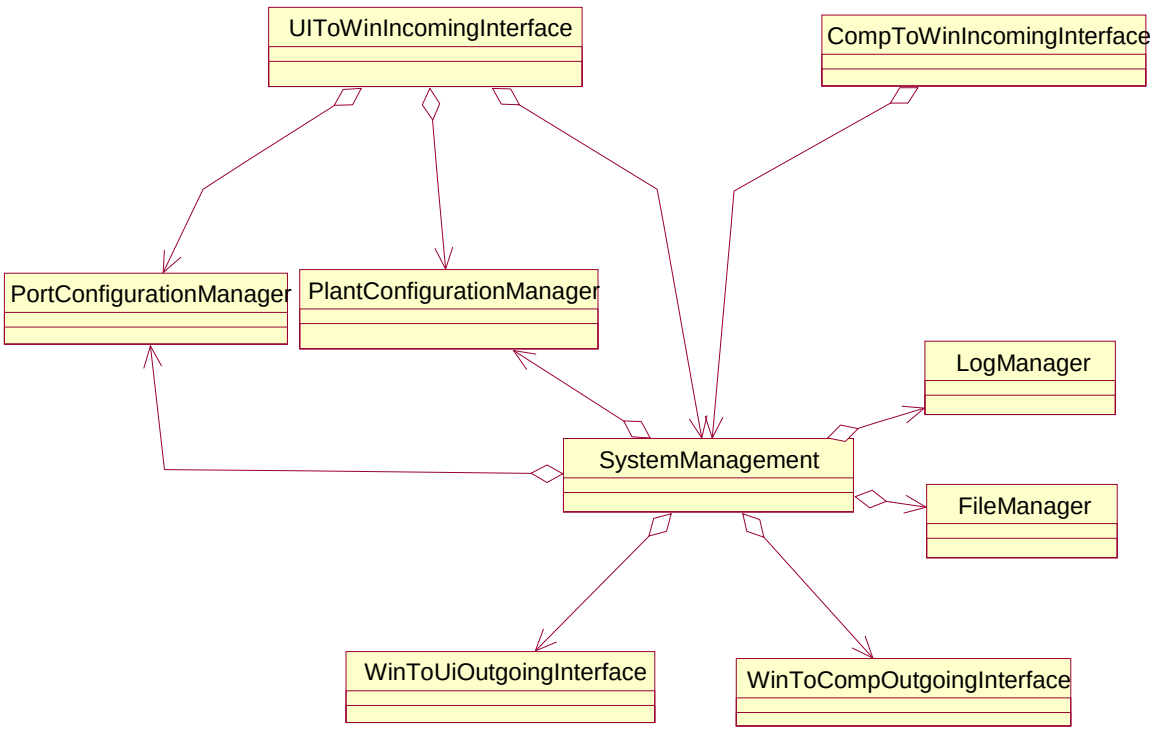

Figure 22: Win32 Process Diagram

The Win32 process has two interfaces: the interface to and from the User Interface, and the interface to and from the Computation process. The remainder of the process consists of four managers: The PortConfigurationManager, the PlantConfigurationManager, the LogManager, and the FileManager. The SystemManagement class ties all of the managers together.

3.1.2.4.1 UiToWinIncomingInterface Class

The UiToWinIncomingInterface class fields commands from the User Interface process. It determines what operations have been requested, copies relevant data out of shared memory, and makes the appropriate calls on the Win32 manager classes.

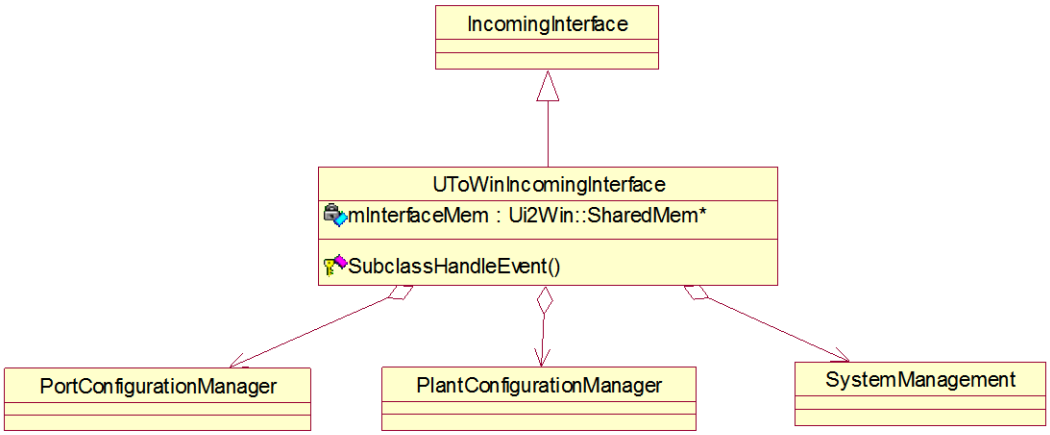

Figure 23: Class UiToWinIncomingInterface

3.1.2.4.1.1 Responsibilities

As a class that inherits from IncomingInterface, The UiToWinIncomingInterface is responsible for overriding the HandleEvent operation by parsing out which operation has been requested, and performing the appropriate actions to handle the operation. Once the appropriate manager has processed the operation request, the UiToWinIncomingInterface is responsible for making sure any return values are copied back into shared memory before releasing the semaphore that the outgoing interface that started this operation is waiting on.

3.1.2.4.1.2 Framework Start-Up

Upon framework start-up, the Win32 process spawns the user interface, but has no way to know when the user interface has actually started, established its shared interface, and is ready to receive function calls from the WinToUiOutgoingInterface. To handle this problem, in addition to all of the operations that a user may initiate through the interface, the UiToWinIncomingInterface must also be prepared to receive a function request from the user interface that simply indicates that the UI is active and ready to receive function calls.

3.1.2.4.2 CompToWinIncomingInterface Class

The CompToWinIncomingInterface class handles commands from the Computation process and makes the appropriate calls on the SystemManagement class to execute the requested operations.

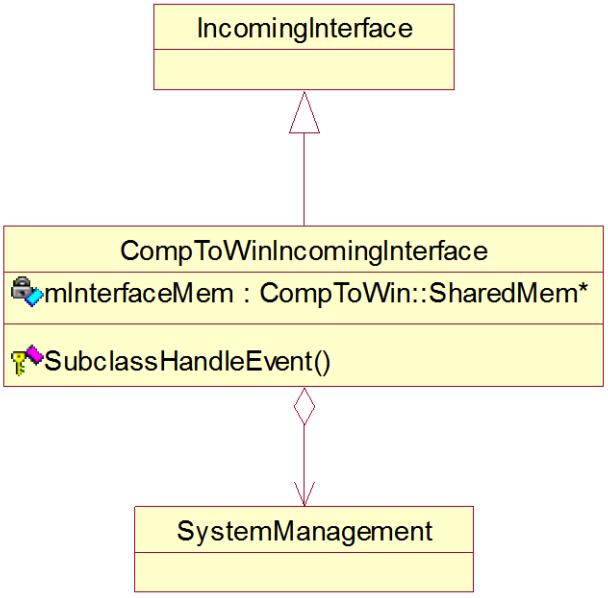

Figure 24: Class CompToWinIncomingInterface

3.1.2.4.2.1 Responsibilities

As a child class of IncomingInterface, the CompToWinIncomingInterface class is responsible for implementing the HandleEvent operation by determining which operation

has been requested, copying relevant data out of shared memory, and invoking the appropriate function call on the SystemManagement class. Once that call has returned, any return values are placed in Shared Memory. Once the call has returned, the CompToWinIncomingInterface is responsible for making sure any return values are copied back into shared memory before unlocking the semaphore associated with the incoming request.

3.1.2.4.3 WinToUiOutgoingInterface Class

The WinToUiOutgoingInterface class sends system updates up to the User Interface's incoming interface. The User Interface may choose to listen to or ignore these operation requests as their completion is not critical to system execution.

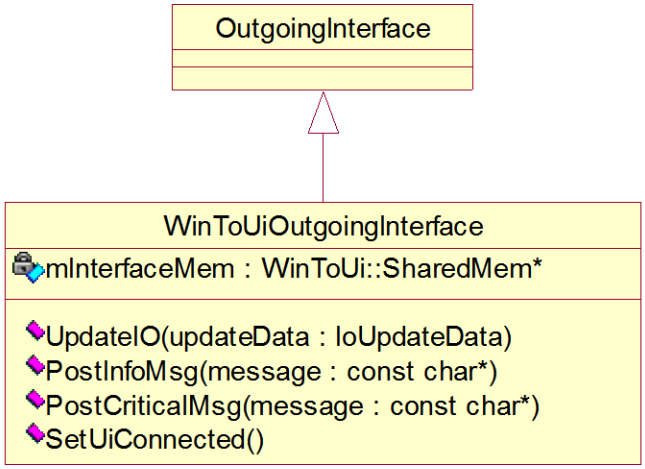

Figure 25: Class WinToUiOutgoingInterface

3.1.2.4.3.1 Responsibilities

The WinToUiOutgoingInterface is responsible for updating the user interface with the current status of the simulation. Informational and critical messages are passed to the user interface, which may decide what to do with them. During simulation, the status of the ports as seen by the plant is provided through the UpdateIO operation to allow the user interface to display the current state.

3.1.2.4.3.2 Framework Start-up

Upon framework start-up, the WinToUiOutgoingInterface cannot know if a user interface has been launched and is ready to receive incoming commands. For that reason, no outgoing operations may be enacted until the user interface reports that it is ready to receive function calls. The WinToUiOutgoingInterface is responsible for ignoring all function requests until SetUiConnected is called, indicating that a user interface is available. It is at this point that function calls can be made on the User Interface.

3.1.2.4.4 WinToCompOutgoingInterface Class

The WinToCompOutgoingInterface class provides an interface for the Win32 process to the Computation process.

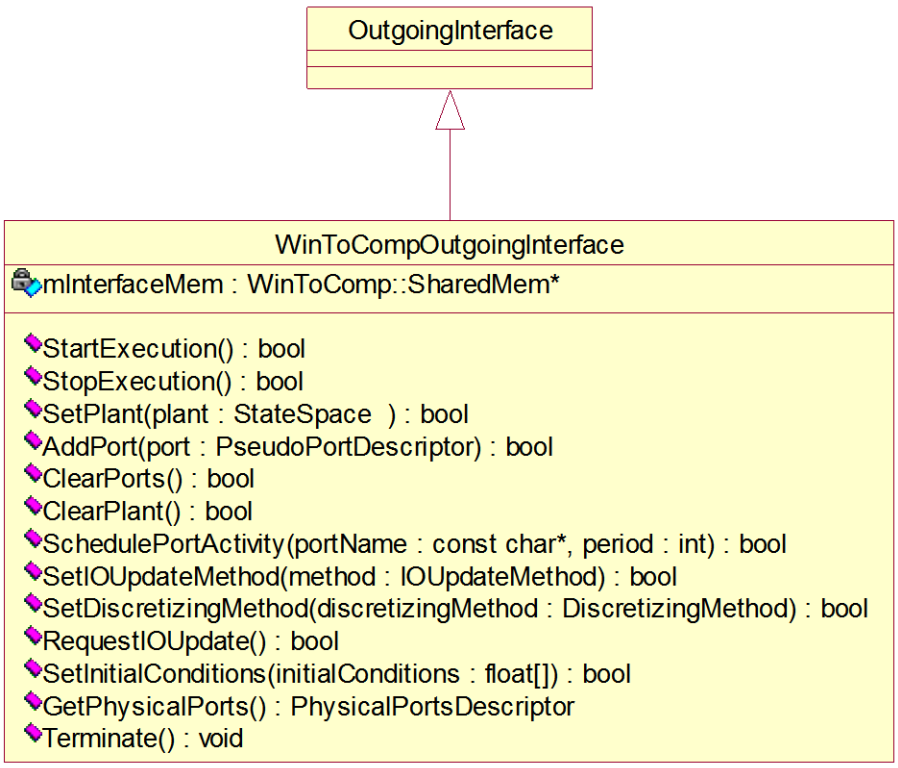

Figure 26: Class WinToCompOutgoingInterface

3.1.2.4.4.1 Responsibilities

The WinToCompOutgoingInterface is responsible for converting the input operations into function calls to be invoked on the Computation Process. It copies the input parameters to shared memory and blocks. It will remain blocked until the computation kernel incoming interface completes the operation and unblocks the interface semaphore. At that point, the WinToCompOutgoingInterface is responsible for copying the return value out of shared memory and returning it to the calling process. The operations provided by this interface allow the Win32 process to start and stop execution, configure the Computation Kernel with plant or port information, and shut down the program completely.

3.1.2.4.5 PortConfigurationManager Class

The PortConfigurationManager maintains the port configuration in the Win32 process. The port mapping stored by the port configuration manager does not need to be complete or valid until the start of simulation; at this stage, the user may alter and update the values. The UiToWinIncomingInterface interfaces directly with the PortConfigurationManager to set and review the current PseudoPort configuration. The SystemManager interfaces with the PortConfigurationManager to access the port mapping for save and load operations. The port mapping is fetched from the PortConfigurationManager by the SystemManager to configure the Computation Kernel when prompted to start simulation execution.

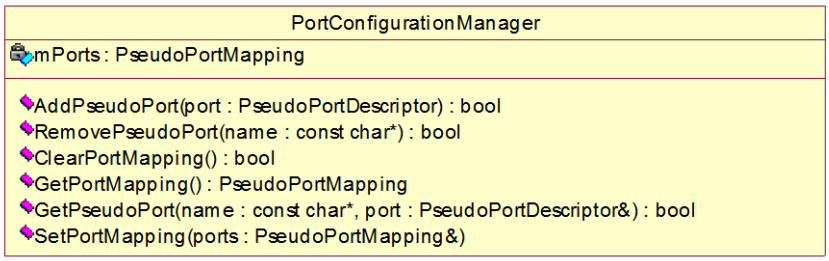

Figure 27: Class PortConfigurationManager

3.1.2.4.5.1 Responsibilities

The PortConfigurationManager is responsible for maintaining the Pseudo Port mapping. It provides a repository for all pseudo ports currently configured. Ports are added, removed, and retrieved by name.

3.1.2.4.6 PlantConfigurationManager Class

The PlantConfigurationManager maintains the plant configuration in the Win32 process. The UiToWinIncomingInterface interfaces directly with the PlantConfigurationManager to set and review the plant configuration. The SystemManager interfaces with the PlantConfigurationManager to access the plant for save and load operations. The plant is fetched from the PlantConfigurationManager by the SystemManagement class when it is needed to configure the Computation Kernel.

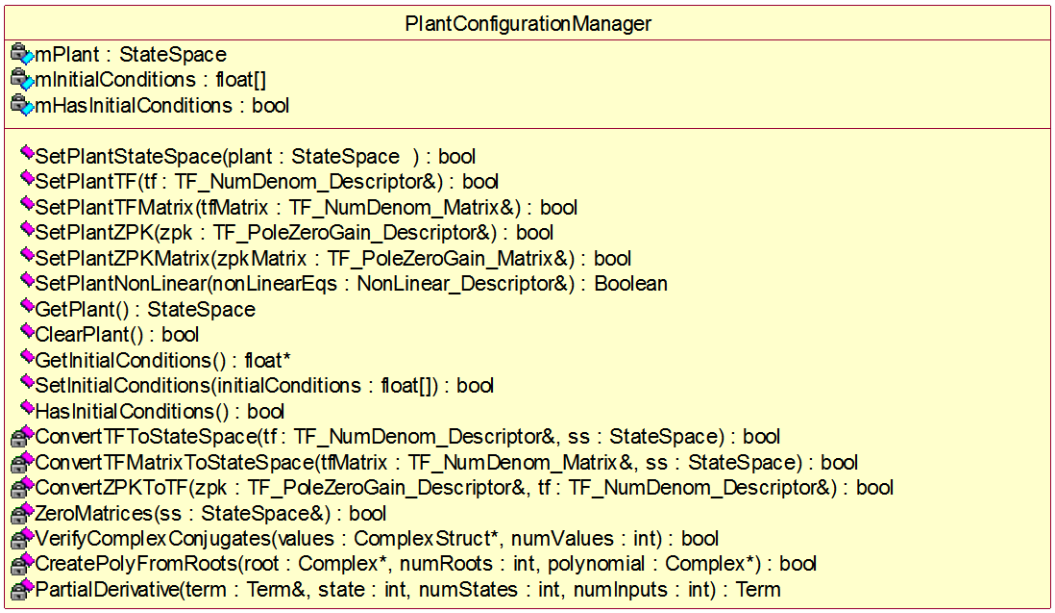

Figure 28: PlantConfigurationManager

3.1.2.4.6.1 Responsibilities

The PlantConfigurationManager is responsible for maintaining the plant in the Win32 process. Plants may be specified as a set of state space matrices, as transfer functions, a matrix of transfer functions, or as a set of nonlinear equations. The transfer

functions themselves may be specified in terms of the coefficients of the numerator and denominator, or as a set of zeros, poles, and the gain associated with them. The system only stores plants as systems of state space equations, and the Computation Kernel can only accept state space equations as well. For this reason, the PlantConfigurationManager has the additional responsibility of converting transfer functions to state space equations for storage. The PlantConfigurationManager must prevent invalid plants from being configured. This includes making sure that all matrices are formatted properly, that complex poles and zeros are delivered in conjugate pairs, and that input and output names are not misplaced in the plant specifications. It also must maintain any initial conditions associated with the plant

3.1.2.4.7 LogManager Class

The LogManager handles all logging operations. This operation is governed by the SystemManagement class that passes log information to the user interface before sending it down to the LogManager.

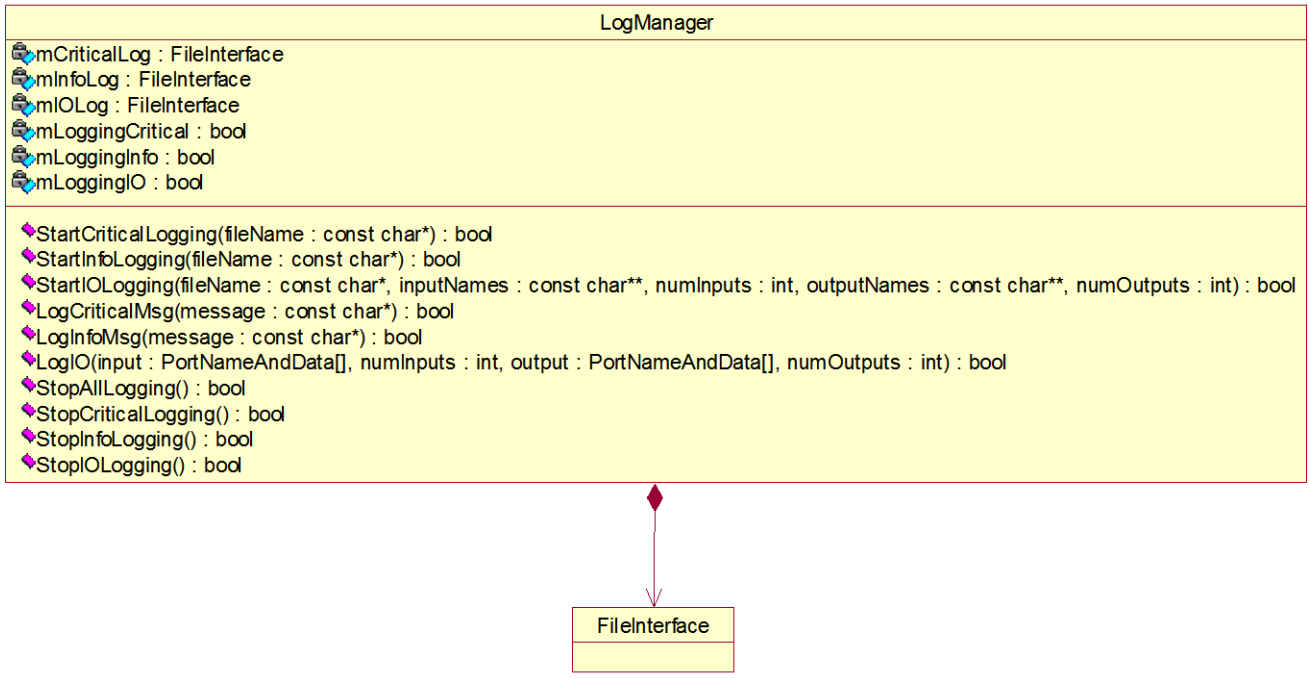

Figure 29: Class LogManager

3.1.2.4.7.1 Responsibilities

The LogManager is responsible for handling all logging requests. This involves managing each of the individual FileInterfaces involved for logging, and formatting the data received for logs. IO logs, for example, are written as comma delimited files, and require formatting to be performed on every entry.

3.1.2.4.8 FileManager Class

The FileManager class handles all file I/O with the exception of the logs. For the purposes of organization, the FileManager is split into two different file managers; one for the plant file I/O, and one for the port file I/O.

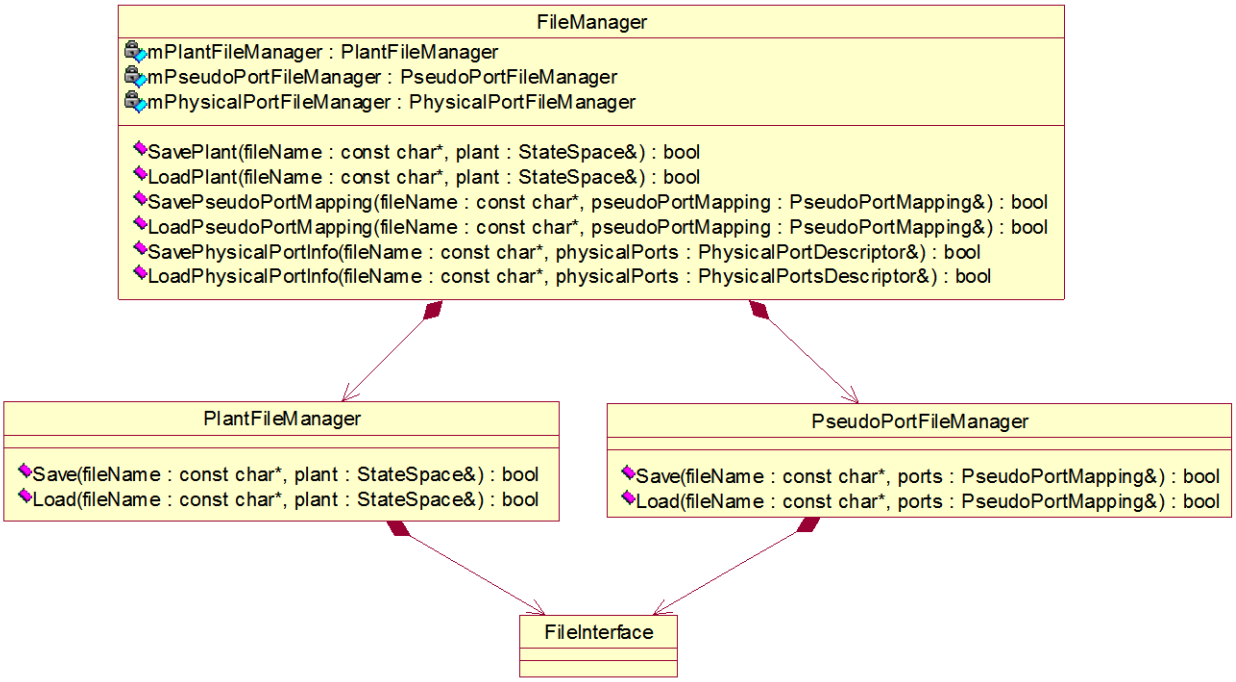

Figure 30: Class FileManager

3.1.2.4.8.1 Responsibilities

The FileManager is responsible for saving and loading the plant and the pseudo port mapping. This is done by forwarding the incoming call to one of two different sub-file managers. Each of the sub-file managers are responsible for saving and loading to files in predefined user readable format. See sections 2.2.5 and 2.3.4 of the User Manual for specifics on the format of Plant and Pseudo Port configuration files. This allows users to enter their own plants and port mappings through any basic text editor.

3.1.2.4.9 SystemManagement Class

The SystemManagement class handles all interactions between the various managers that make up the Win32 process. The UiToWinIncomingInterface and CompToWinIncomingInterface communicate directly with the SystemManagement to handle all external requests that involve manager interaction.

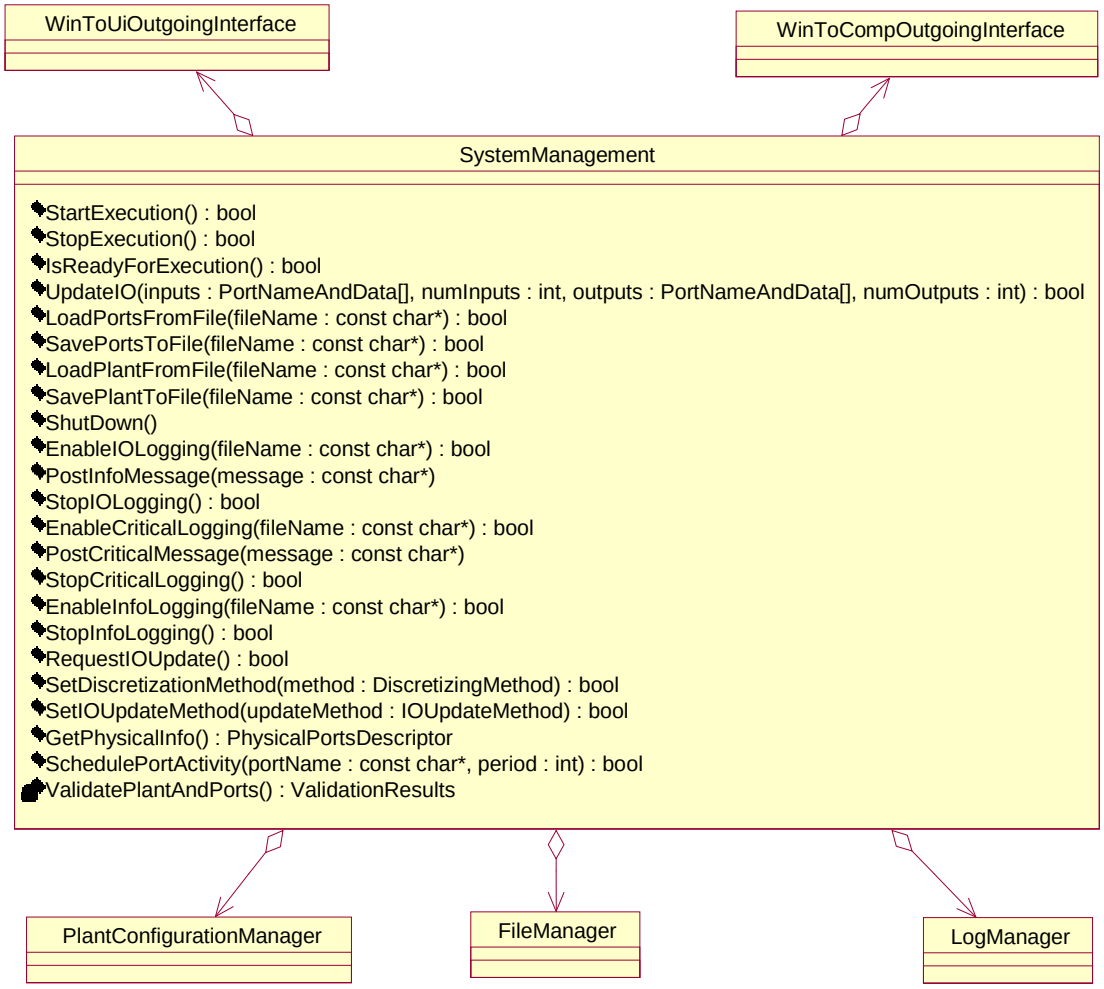

Figure 31: Class SystemManagement

3.1.2.4.9.1 Responsibilities

The SystemManagement class is the main class of the Win32 process. The majority of all incoming operations are initially sent through the SystemManagement class which utilizes the services provided by the individual managers to fulfill each request. It is responsible for verifying the current configuration, sending that configuration down to the Computation process, and for starting and stopping the actual simulation. During load operations it retrieves information from the File Manager to set the data in the PlantConfigurationManager and PortConfigurationManager. It also collects data from PlantConfigurationManager and PortConfigurationManager to send to the FileManager for saving information.

3.1.2.5 Computation Kernel Process

The Computation Kernel process is launched by the Win32 process, and is responsible for handling the actual simulation of the configured plant. It is responsible for managing the physical IO, and as such, some small porting effort may be required for different data acquisition boards. It accepts a plant configuration from the Win32 process as a set of state space equations, discretizes the plant if necessary, and executes simulation cycles. Discretization is performed in this process instead of in the Win32 process to move the discretization of plants one level further away from the end user. When a user provides a continuous plant, they should not see their plant modified in significant ways, especially when saving it to, or loading it from a file. It may become difficult for a user understand that the plant they loaded really is the discrete equivalent of what they intended to load. In addition, it is a requirement of the simulation itself that the plant be discretized. The plant will be kept in its initial state as much as possible until the component that needs it to be changed encounters it.

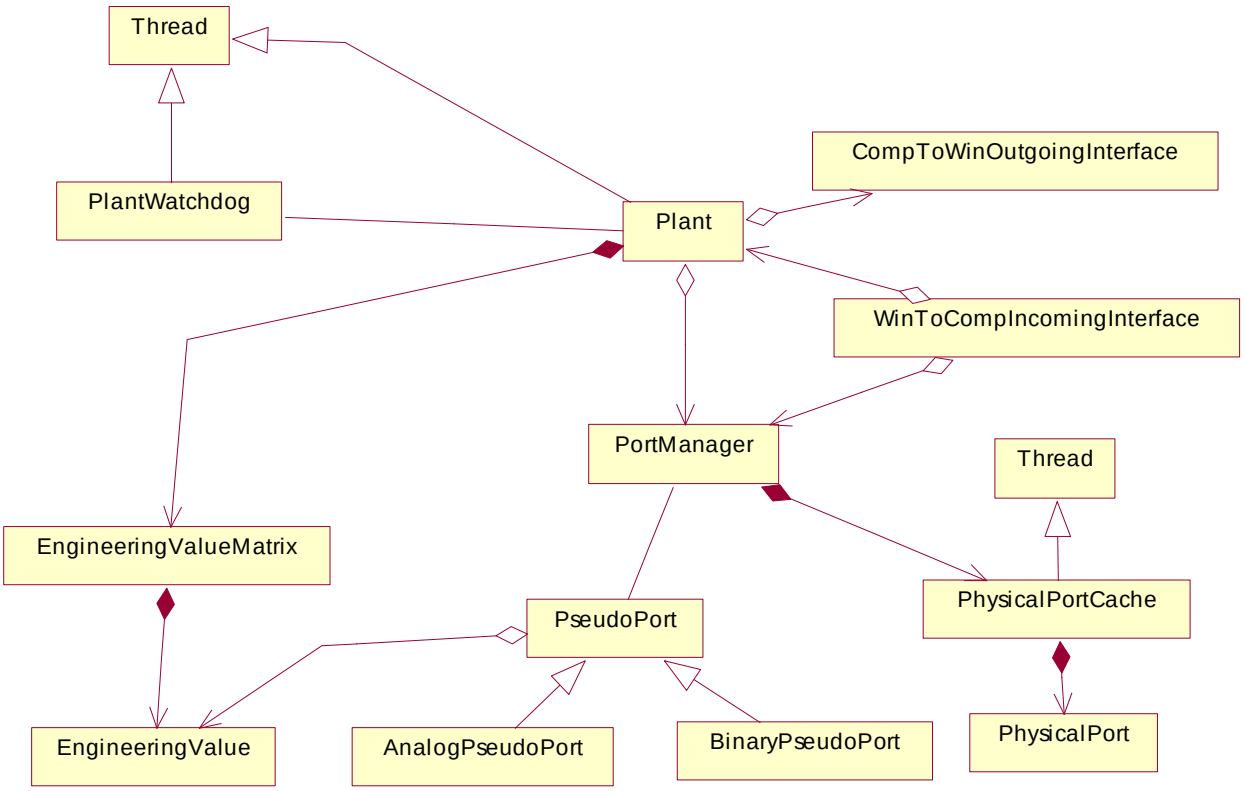

Figure 32: ComputationKernel Process

Access to physical data is provided by physical ports that are launched statically by the PhysicalPortCache. The PhysicalPortCache caches data that is to be written to, or has been read from physical ports. PseudoPorts map the physical ports to Plant inputs and outputs, and convert physical data to EngineeringValues that are meaningful to the plant. Configurations and commands from the Win32 process are fielded by the WinToCompIncomingInterface and are forwarded to one of the two major components of the Computation Kernel; The Plant, or the Port Manager. The Plant handles configuring and discretizing the plant provided. When it comes time to run the simulation, the Plant is responsible for taking inputs and using them to evaluate the plant at periodic intervals on its own thread of execution. At this stage, all values are stored as engineering values, or in engineering value matrices. The PlantWatchdog monitors the plant to make sure that updates are performed within a specified period of time. Should the updates miss their deadlines, the PlantWatchdog will send a critical message to the Win32 process.

The Port Manager handles the physical ports and maintains the configured pseudo port mapping.

3.1.2.5.1 WinToCompIncomingInterface Class

The WinToCompIncomingInterface provides the means by which the Win32 process configures and commands the plant, and sends port related configuration information down to the port manager.

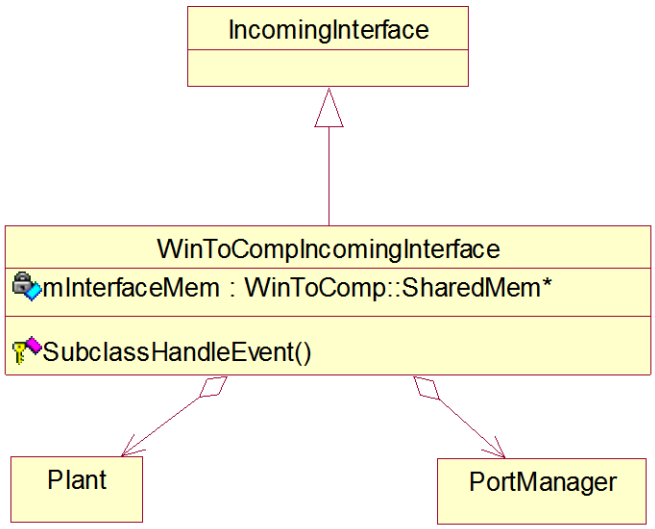

Figure 33: Class WinToCompInterface

3.1.2.5.1.1 Responsibilities

The WinToCompIncomingInterface is responsible for determining which operation has been requested by the Win32 process, and executing that operation on either the plant or the port manager. It is also responsible for unlocking the return semaphore that the Win32 process waits on to indicate that the operation is complete.

3.1.2.5.2 CompToWinOutogingInterface Class

The CompToWinOutgoingInterface provides the means by which the Computation Kernel reports status back to the Win32 process. These messages are not

vital to operation in that they do not affect the simulation themselves. They do, however report information that may be vital to the user, and are important to keep the user informed of the status of the simulation.

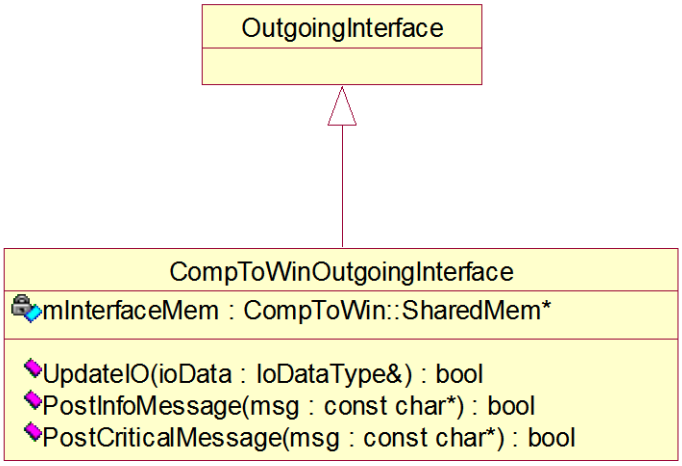

Figure 34: Class CompToWinOutgoingInterface

3.1.2.5.2.1 Responsibilities

The CompToWinOutgoingInterface is responsible for providing a means by which the Computation Kernel can report data updates and informational or critical messages back to the Win32 process. It must block the Computation Kernel until the Win32 process is done with the data. For that reason, only critical failures should be reported during simulation execution.

3.1.2.5.3 EngineeringValue Class

The EngineeringValue class represents values that are meaningful to the plant. All plant calculations are performed on EngineeringValue objects. This allows future implementations to change how the plant performs its calculations. Fast floating point arithmetic through the use of integers is a candidate for such improvements.

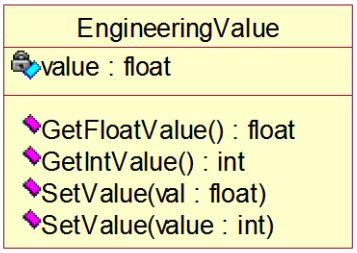

Figure 35: Class EngineeringValue

Note: Not shown: Full suite of operator functions.

3.1.2.5.3.1 Responsibilities

EngineeringValue is responsible for providing a wrapper for a floating point value, as all plant operations require floating point arithmetic. The EngineeringValue class provides an abstraction that allows arithmetic to be redefined in order to increase efficiency.

3.1.2.5.4 EngineeringValueMatrix Class

The majority of all computations required to evaluate a plant are performed as matrix operations. The EngineeringValueMatrix class is a matrix of engineering values, and provides functions to perform the required matrix operations on its data members.

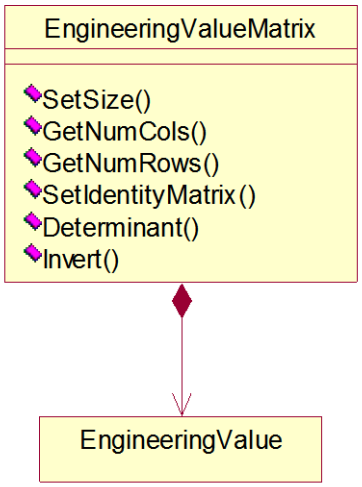

Figure 36: Class EngineeringValueMatrix

Note: Not shown: Full suite of operator functions.

3.1.2.5.4.1 Responsibilities

The EngineeringValueMatrix class provides a set of basic matrix operations, including matrix inversion, calculating the determinant of the matrix, and basic arithmetic operations. This simplifies the requirements imposed on the plant, and provides for a much cleaner implementation.

3.1.2.5.5 PhysicalPort Class

The PhysicalPort class represents an interface available on the connected Data Acquisition device. Raw physical data is read from or written to physical ports.

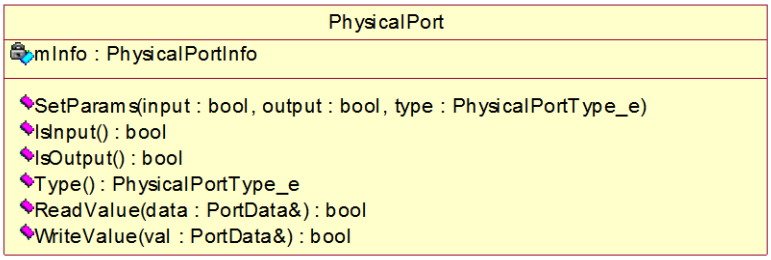

Figure 37: Class PhysicalPort

3.1.2.5.5.1 Responsibilities

The PhysicalPort class is an abstract class from which specific physical ports must inherit. Inheriting classes are required to provide an implementation for the Read and Write value operations that handle physical IO for a specific interface on the attached data acquisition device. New inheriting classes will have to be written for different data acquisition devices. See Section 4 of the User Manual for information about how to update the Control System Plant Simulator for new data acquisition devices.

3.1.2.5.6 PhysicalPortCache Class

The PhysicalPortCache owns all of the PhysicalPorts in the system, constructing them at launch. It manages a cache of data that has either been read from a physical port, or will be written to a physical port shortly. The values provided to the plant come from this cache, and not the ports themselves.

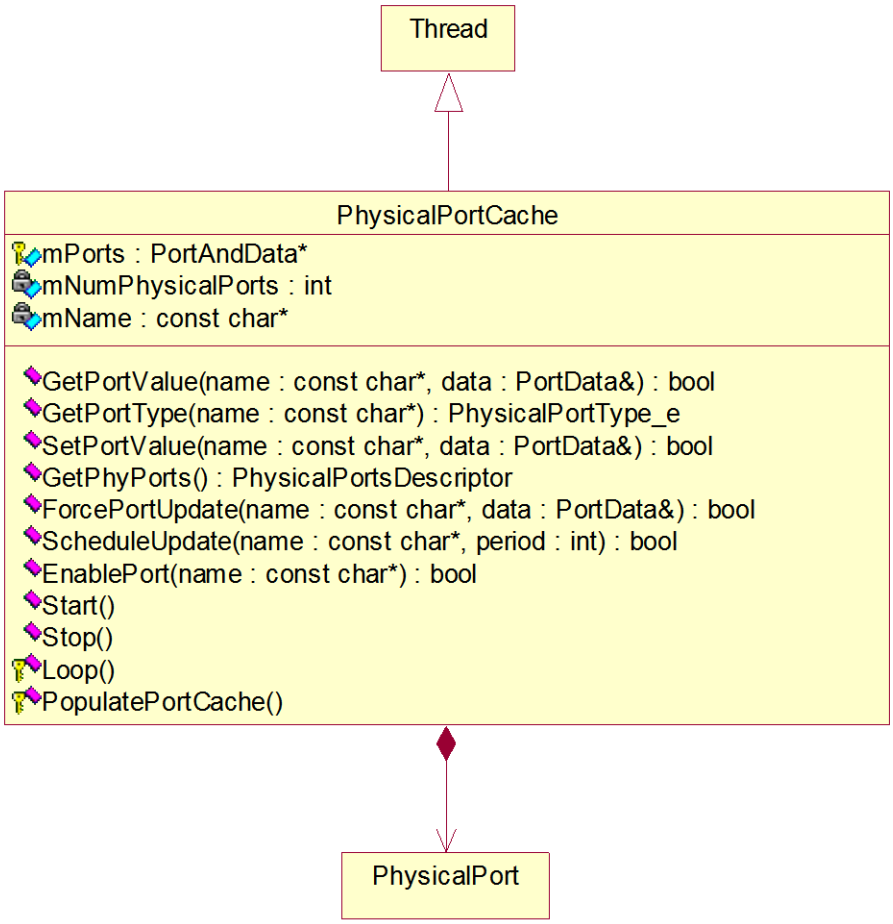

Figure 38: Class PhysicalPortCache

3.1.2.5.6.1 Responsibilities

The PhysicalPortCache is responsible for maintaining a cache of all physical values, and for updating the PhysicalPorts independently from the execution of the plant. When the simulation is running and the plant is being evaluated, reading from or writing to physical ports must not block the thread of execution that is evaluating the plant. Instead, the PhysicalPortCache has its own thread of execution. This thread will read data from the plants and store it in a cache. Accessing data from this cache is both fast and deterministic. When the plant has computed data to be output to the physical ports that data is stored in the cache instead. These cached values will be written to the physical ports at periodic schedulable intervals.

3.1.2.5.6.2 Update Schedule

Not all physical IO is performed at the same rate. Some physical interfaces may update much faster than others. Some may tie up considerable system resources, preventing or limiting access to other physical ports. For this reason, the Control System Plant Simulator allows the user to determine exactly how much time will pass between updates for each individual port. For example, an analog port that is known to take some time may be updated every 100 milliseconds, while a simple digital port may be updated every 20. Note that no update schedule should be faster than the plant cycle interval. These update schedules are treated the same for input or output ports. When it comes time to update a particular port, either a read or a write will be performed on the physical port depending on the port's configured direction.

3.1.2.5.6.3 Inheriting Implementations

The PhysicalPortCache is an abstract class that must be implemented by inheriting subclasses. This is because it manages all of the physical ports in the system, and is responsible for populating itself with these physical ports at framework start-up. Physical ports themselves are data acquisition device specific, as is the PhysicalPortCache. A new physical port cache must be written for each data acquisition device, but the only operation that needs to be implemented is the PopulatePortCache operation. All major functionality is handled by the parent.

3.1.2.5.7 PseudoPort Class

The PseudoPort class provides the connection between the user defined plant and the physical data acquisition unit. Inputs and outputs of the plant must be mapped to data that enters the system through the data acquisition unit. PseudoPorts indicate which plant inputs are provided data from which physical inputs, and which plant outputs provide data to which physical outputs. They also determine how that data must be formatted into meaningful engineering value data.

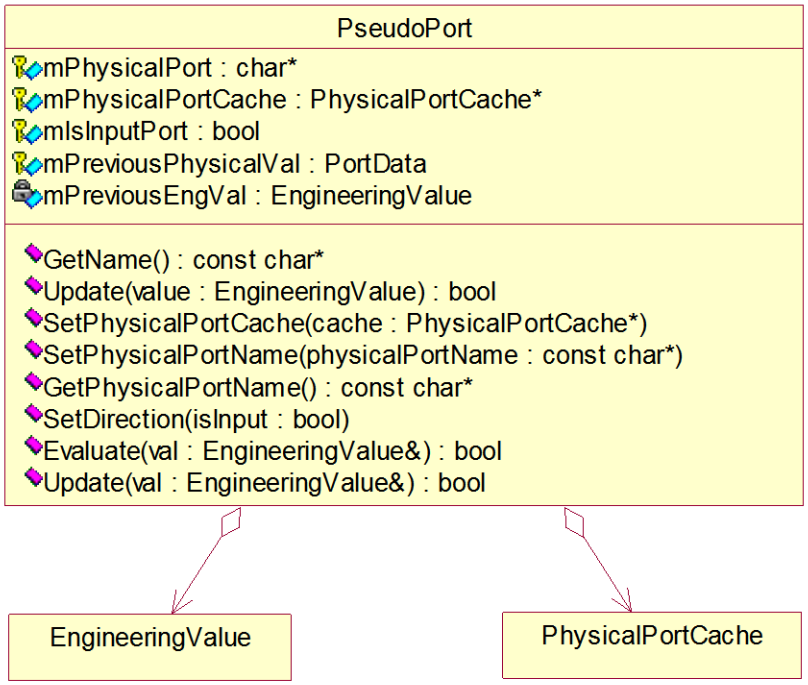

Figure 39: Class PseudoPort

3.1.2.5.7.1 Responsibilities

The PseudoPort is responsible for converting raw physical data read from a data acquisition device into Engineering values that can be used by the plant, and for changing engineering values provided by the plant back into raw physical data to be sent to the data acquisition device output ports. This conversion is done intelligently, in that it is only performed when a value has been changed. Neither incoming physical data values nor outgoing engineering value data are converted when the value has not changed. When the plant needs data, it requests an evaluation from all input pseudo ports. Each PseudoPort will then retrieve data from the physical port cache. They will convert the data if the value has changed from the previously converted one, or use a saved conversion of the previous value if it has not. These converted values are then provided to the plant as a set of inputs. When the plant has calculated a set of outputs, these are sent to the output PseudoPorts as EngineeringValues. If the EngineeringValues are different from previously provided EngineeringValues the PseudoPorts will convert them to physical data values and update the PhysicalPortCache. If not, they will provide the result of the previous conversion without performing the conversion calculation.

3.1.2.5.7.2 Plant and Physical Port Connection

Every input and output in the plant is named. A corresponding pseudo port must be provided for each and every one of these inputs and outputs. The pseudo port then has knowledge of which physical port it must retrieve data from, and send data to. When providing a set of pseudo ports for the system, users must take care to map every input and output to a valid physical input or output port.

3.1.2.5.7.3 Engineering Value Cache

PseudoPort operations are an unavoidable part of the evaluation of a plant. Data must be fetched and converted to values the plant can understand. To help improve compute time, each PseudoPort provides an engineering value cache. The plant could request data at a pace much faster than the PhysicalPorts can provide it. In these instances, the actual data retrieved from the PhysicalPortCache will have not changed. Each PseudoPort maintains a copy of the previous Engineering Value it provided for either an update or an evaluation, and the physical data it received that established the engineering value. Should the new physical data match the previous physical data exactly, the pseudo port will bypass the mapping calculation and simply provide the previous engineering value.

3.1.2.5.8 AnalogPseudoPort Class

Analog PseudoPorts are PseudoPorts that map to physical ports that provide analog data. Physical data is provided to the Analog Pseudo Port as a digital value that represents an analog measurement made by the data acquisition device. This class converts that measurement to a value meaningful to the plant.

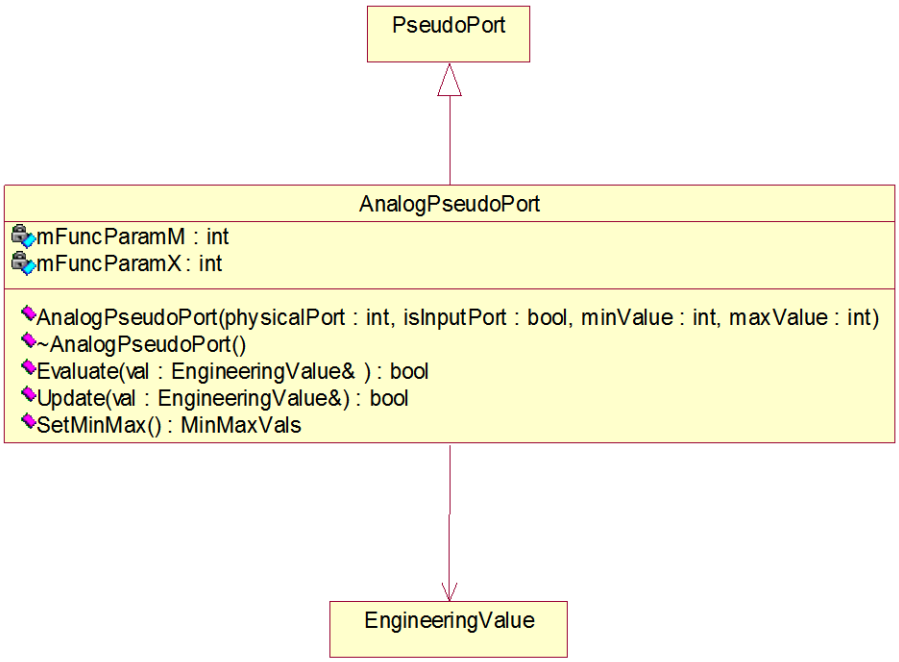

Figure 40: Class AnalogPseudoPort

3.1.2.5.8.1 Responsibilities

The AnalogPseudoPort directly wraps one analog physical interface. Its main function is to modify the data read from physical IO by scaling it to a range that is meaningful to the plant. This is accomplished through simple linear scaling, though more complex scaling systems could be added in the future. A physical data acquisition device may be able to only provide inputs in the range of -10 to 10 volts, but these values could correspond to any number of possible values and ranges. For example, consider a pitch controller for an airplane that has the angle of the pitch as an input in degrees from horizontal, from -0.6 to 0.6 radians. An AnalogPseudoPort can be configured to scale the measured value provided by the data acquisition device to this range of values.

3.1.2.5.8.2 Engineering Value Conversion

AnalogPseudoPort converts physical data values to analog data values by scaling the values linearly according to a traditional $y = mx + b$ equation where m and b are defined as:

$$Equation\ 32 \qquad m = \frac{\text{min plant value- min physical value}}{\text{max plant value- max physical value}}$$

$$Equation\ 33 \qquad b = \text{min plant value-(min physical value*m)}$$

Note that while this is the method currently used by the CSPS framework, it is not the only method that can be used to scale this data. Future developers may alter this method or add new ones as they see fit.

3.1.2.5.9 BinaryPseudoPort Class

The BinaryPseudoPort class connects physical digital ports to the inputs and outputs of the plant. BinaryPseudoPorts may divide larger physical ports by mapping to a subset of the physical port's bits.

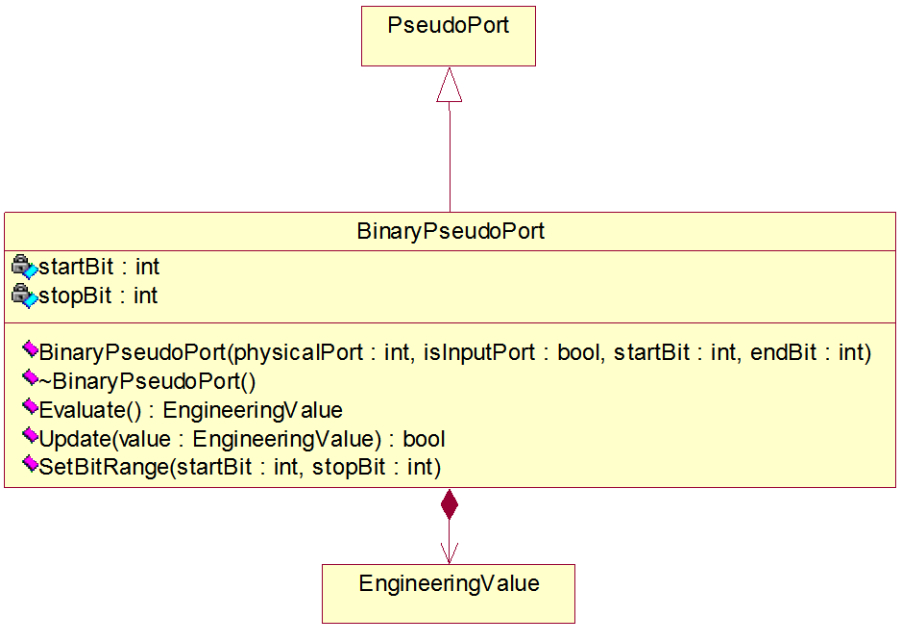

Figure 41: Class BinaryPseudoPort

3.1.2.5.9.1 Responsibilities

The BinaryPseudoPort directly wraps the bits of a physical digital interface. It does not scale values like the AnalogPseudoPort does, but it can be used to divide a physical digital port into numerous smaller ports to be provided as separate inputs and

outputs to the plant. For example, one 32 bit physical port may have two different named 16 bit BinaryPseuoPorts mapped to it. In this way, a single physical input can be used as two completely different plant inputs, which can be helpful when physical interface resources are scarce.

3.1.2.5.10 Port Manager Class

The PortManager contains the PhysicalPortCache and any number of Pseudo Ports that access the cache for updates. It converts user specifications for pseudo ports into actual port objects that can carry out the functions needed to read and convert data for the plant.

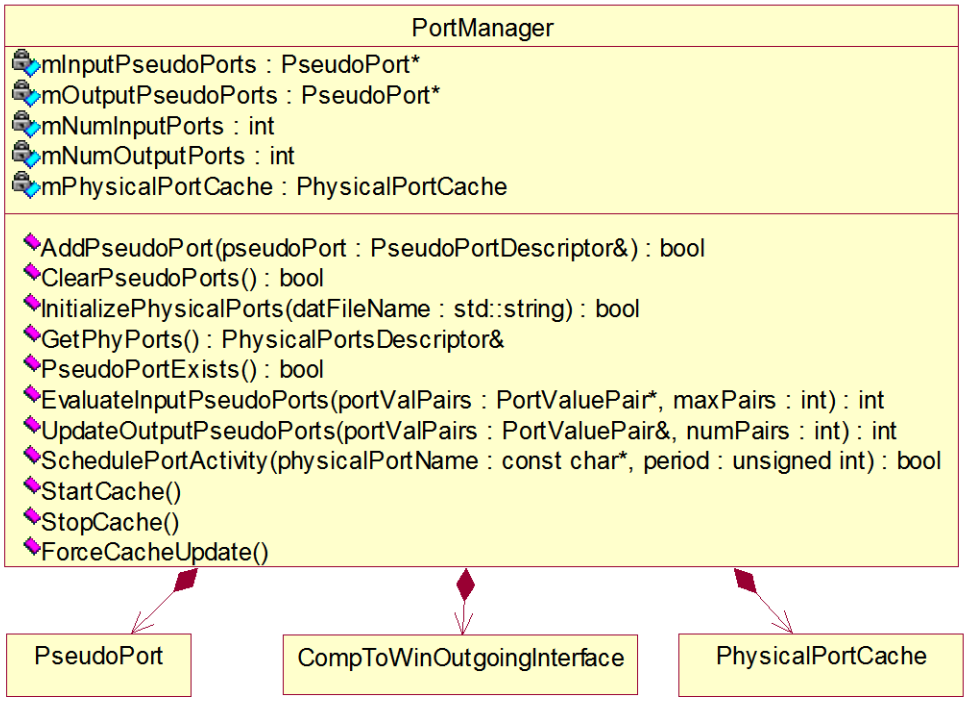

Figure 42: Class PortManager

3.1.2.5.10.1 Responsiblities

The PortManager class is responsible for maintaining the physical port cache and all of the PseudoPorts in the system. Any time a port is to be accessed, added to the system, or updated, the port manager fields the request and sends it to the intended port.

It is responsible for instantiating the physical port, and as such contains code that may need to be altered should the data acquisition device be changed. It is also responsible for retrieving data from the ports when requested. The plant will request all input values at once, or will provide values to be sent to a set of output ports. The PortManager retrieves the data from the ports and provides it to the plant as sets of name-value pairs. It must also determine which ports are being updated from a set of name-value pairs and update the pseudo ports accordingly.

3.1.2.5.11 Plant Class

The Plant class represents the plant evaluator. It handles the simulation of the configured plant by fetching data from the port manager and applying it against the plant configured by the user.

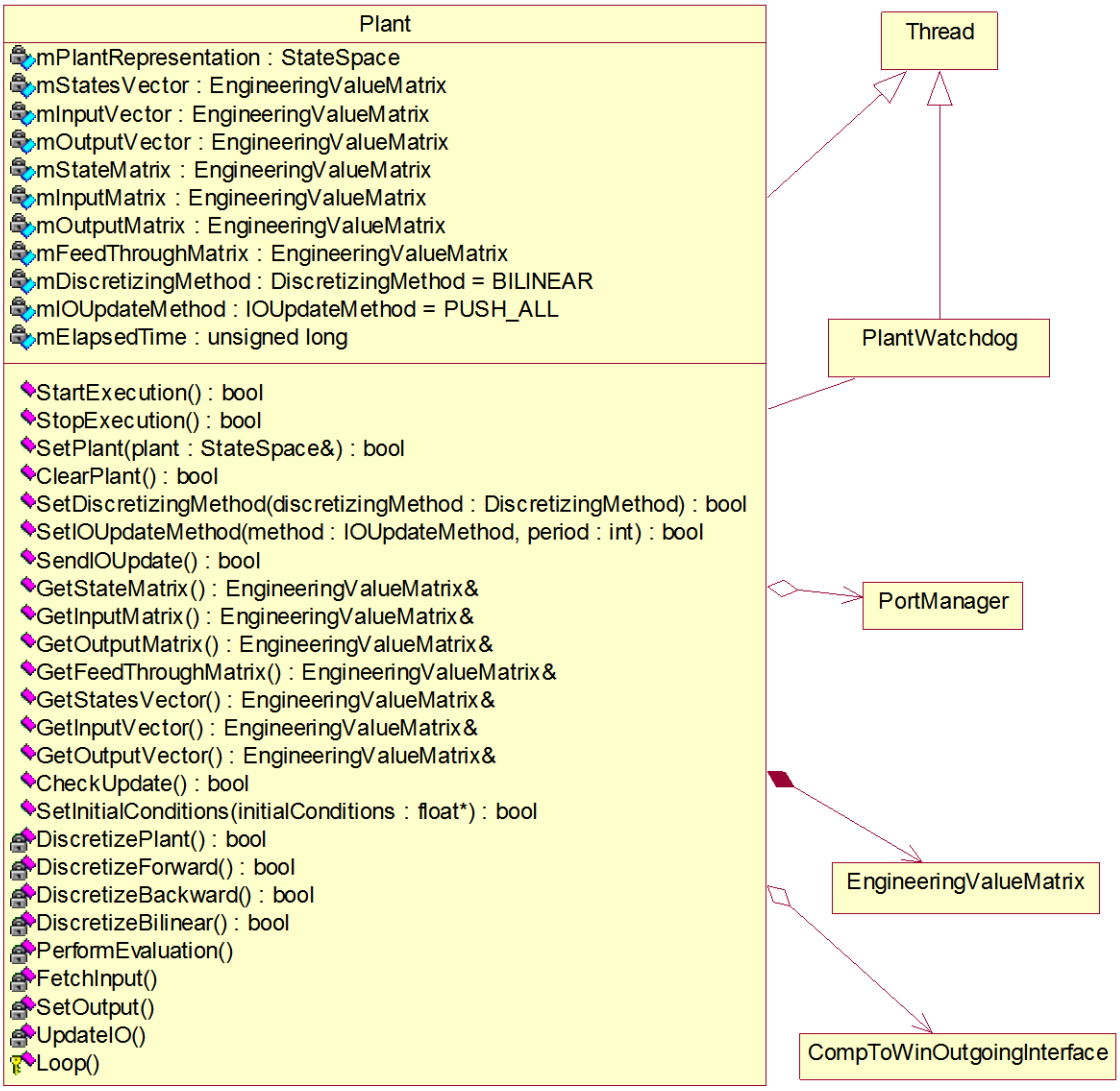

Figure 43: Class Plant

3.1.2.5.11.1 Responsibilities

The Plant Class is responsible for simulating the configured plant. It accepts a configured plant as a set of state space matrices, discretizes it using a configurable method, and simulates it against data it retrieves from the PortManager. It periodically calculates a new set of states and outputs, the latter of which are provided back to the PortManager for physical output. It also sends update messages back to the Win32 process containing the EngineeringValue information for the simulation cycle. This

output update may be configured to be pushed every simulation cycle, after a certain number of simulation cycles, or only when requested. As all plants are evaluated at discrete intervals, continuous plants must be discretized by the Plant class. For this reason, the user must always provide a sampling frequency when defining plants.

3.1.2.5.12 PlantWatchdog Class

The PlantWatchdog class is initiated by the plant when simulation is begun. It monitors the plant to make sure that the plant performs simulation cycles frequently enough.

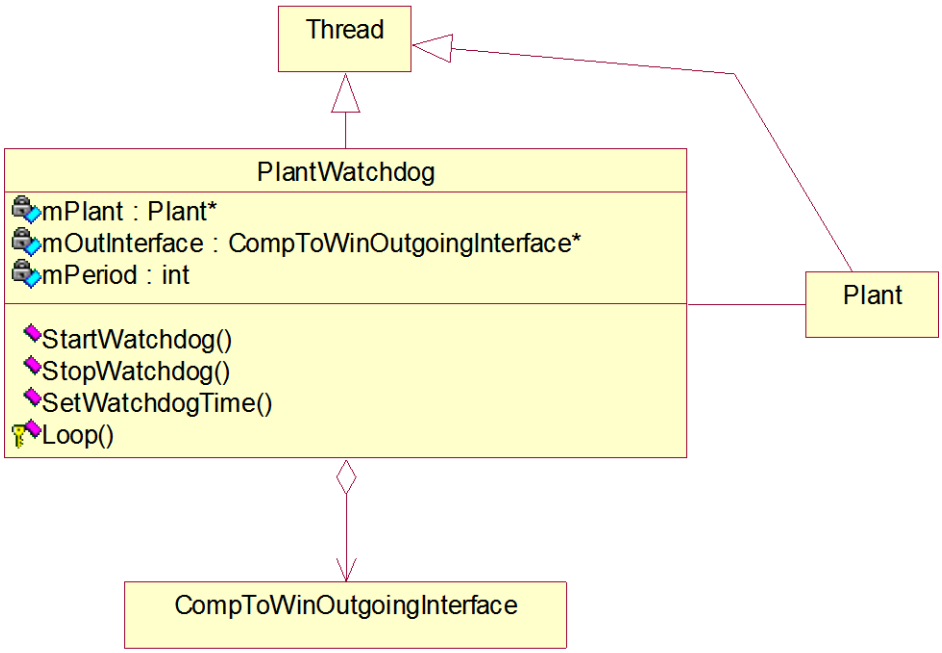

Figure 44: Class PlantWatchdog

3.1.2.5.12.1 Responsibilities

The PlantWatchdog is responsible for making sure that the plant operates fast enough so as not to miss deadlines. After a configurable period of time, the PlantWatchDog checks to see if the Plant has calculated a new set of outputs. If a new set of outputs has not been calculated since the last time the watchdog checked, the

PlantWatchdog sends a critical error message back to the Win32 Process. This is accomplished in a separate thread of execution so as not to affect the simulation itself.

3.1.3 System Interactions

Individual users may command the Control System Plant Simulator to perform numerous actions, each of which involve interactions between numerous objects and processes. The following section describes, in detail, some of the more important interactions performed to execute these commands.

3.1.3.1 System Launch

The Control System Plant Simulator is launched by initiating the Win32 process, which is responsible for launching the other processes. It must also wait until the other processes have registered themselves before allowing user controlled operations.

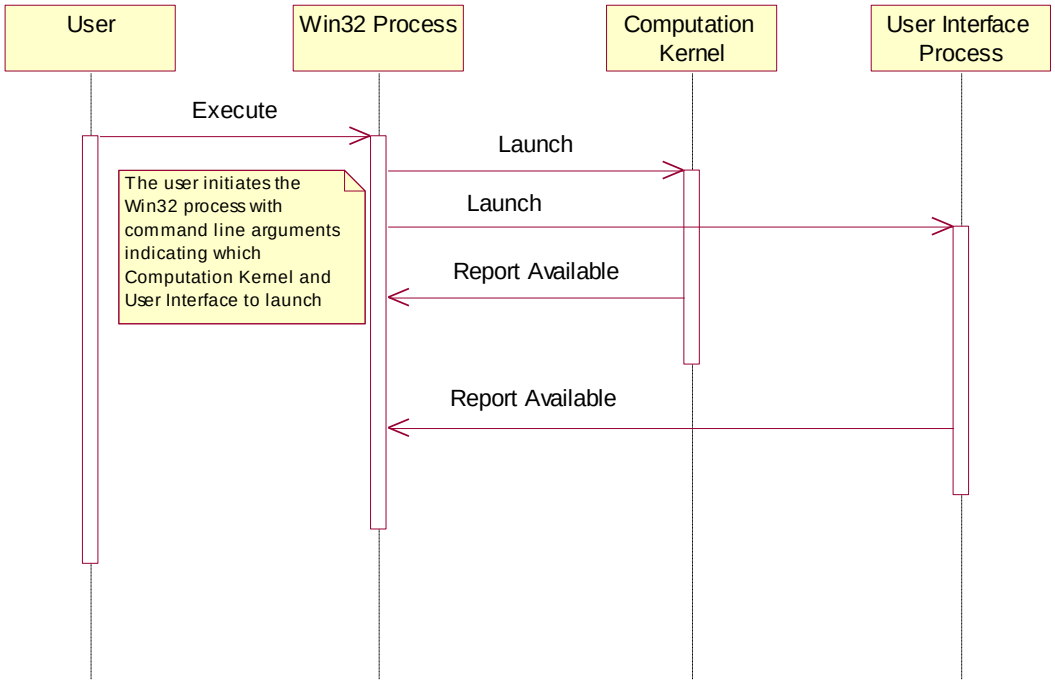

Figure 45: System Launch Sequence Diagram

Any commands that are to be sent to either to Computation Kernel or the User Interface will be dropped until those processes have reported that they are available.

3.1.3.2 Saving and Loading Operations

The Control System Plant Simulator allows users to save the configurations they have defined. Both plant and port configurations may be written to files that can be loaded in the future.

3.1.3.2.1 Saving and Loading Ports

The UiToWinOutgoingInterface provides operations that allow user interfaces to command the Win32 process to save or load pseudo port mapping to or from a file.

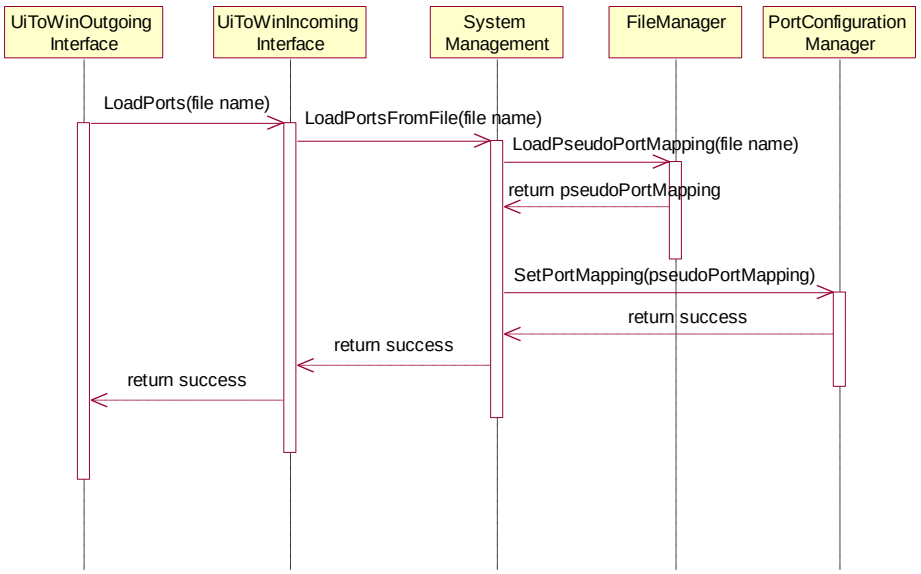

Figure 46: Load Port Mapping Sequence Diagram

To load a port mapping from a file, the file name is passed to SystemManagement. SystemManagement commands the FileManager to read the indicated file and sends the loaded pseudo PortMapping to the PortConfigurationManager.

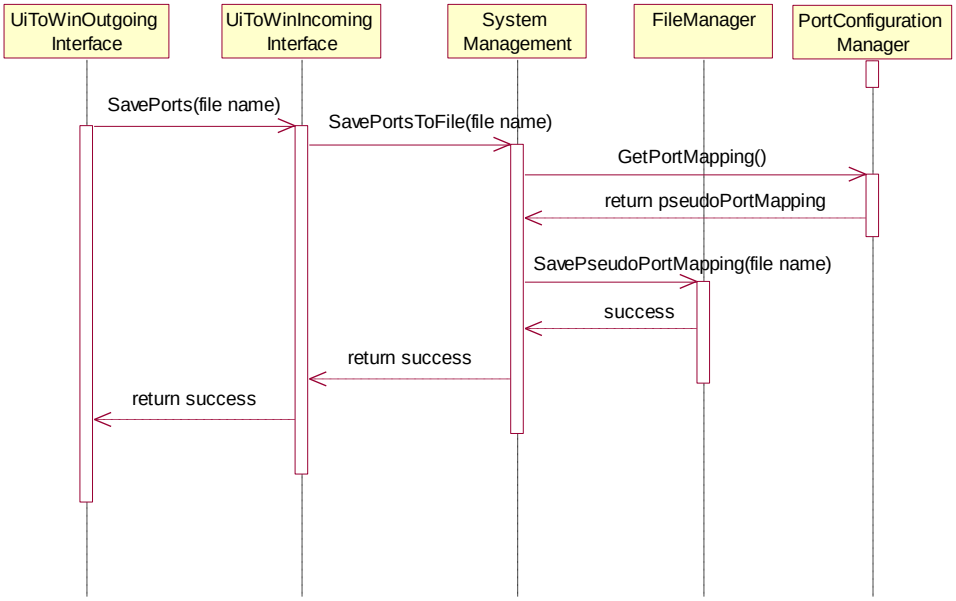

Figure 47: Save Port Mapping Sequence Diagram

To save a port mapping to a file, the file name is passed to SystemManagement. SystemManagement requests the pseudo port mapping from the PortConfigurationManager, and commands the file manager to save the retrieved mapping to the indicated file.

3.1.3.2.2 Saving and Loading Plants

The UiToWinOutgoingInterface provides operations that allow user interfaces to command the Win32 process to save or load plants to or from a file. Plants may only be saved and loaded as state space systems. A plant initially defined by the user by any other means, such as a transfer function defined by a set of zeros and poles, will be converted by the PlantConfigurationManager as soon as it is set.

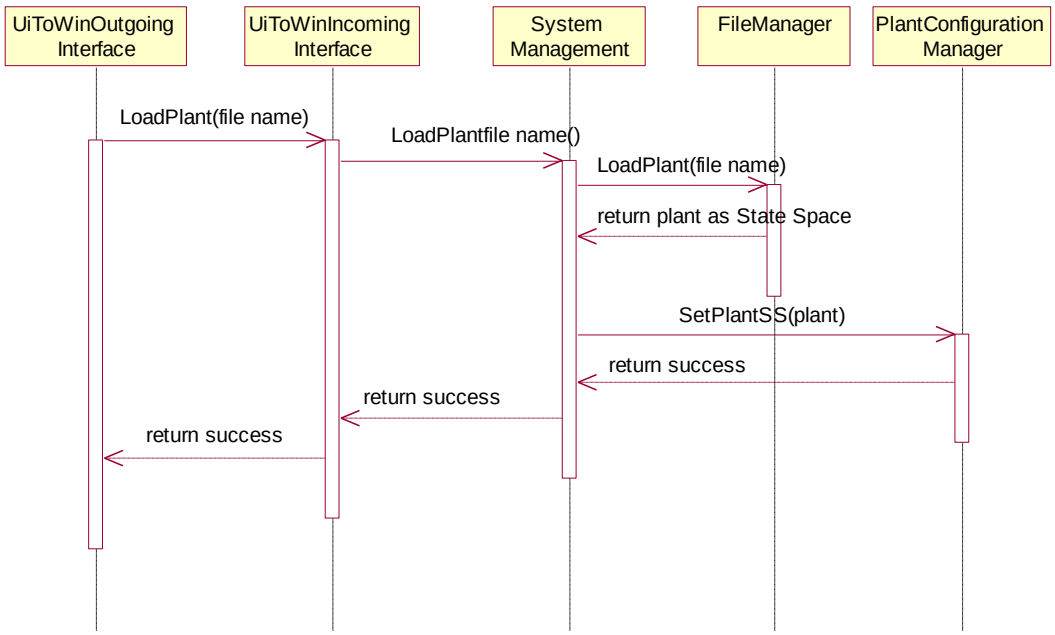

Figure 48: Load Plant Sequence Diagram

To load a plant from a file, the file name is passed to SystemManagement. SystemManagement commands the file manager to read the indicated file and sends the loaded state space plant to the PlantConfigurationManager.

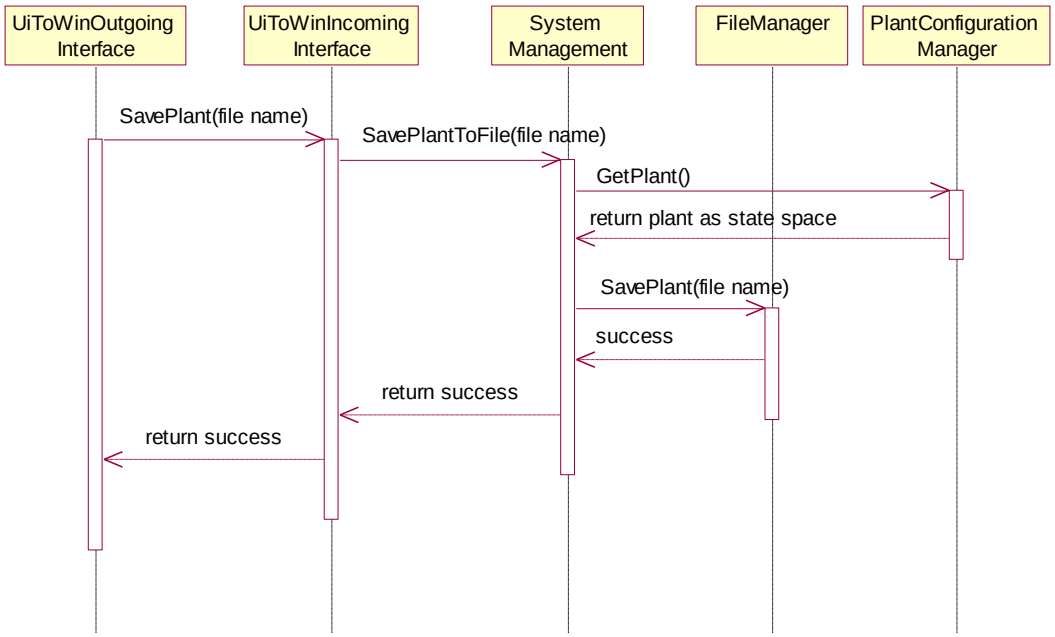

Figure 49: Save Plant Sequence Diagram

To save a plant configuration to a file, the file name is passed to SystemManagement. SystemManagement requests the plant, as a set of state space matrices, from the PlantConfigurationManager, and commands the file manager to save the retrieved configuration to the indicated file.

3.1.3.3 Physical Data Retrieval and Caching

Data is acquired from the physical data acquisition device at periodic intervals and stored for use by the other elements of the system. When data is ready for output, it is written back to this same cache, where it will be used to periodically update the physical IO as well.

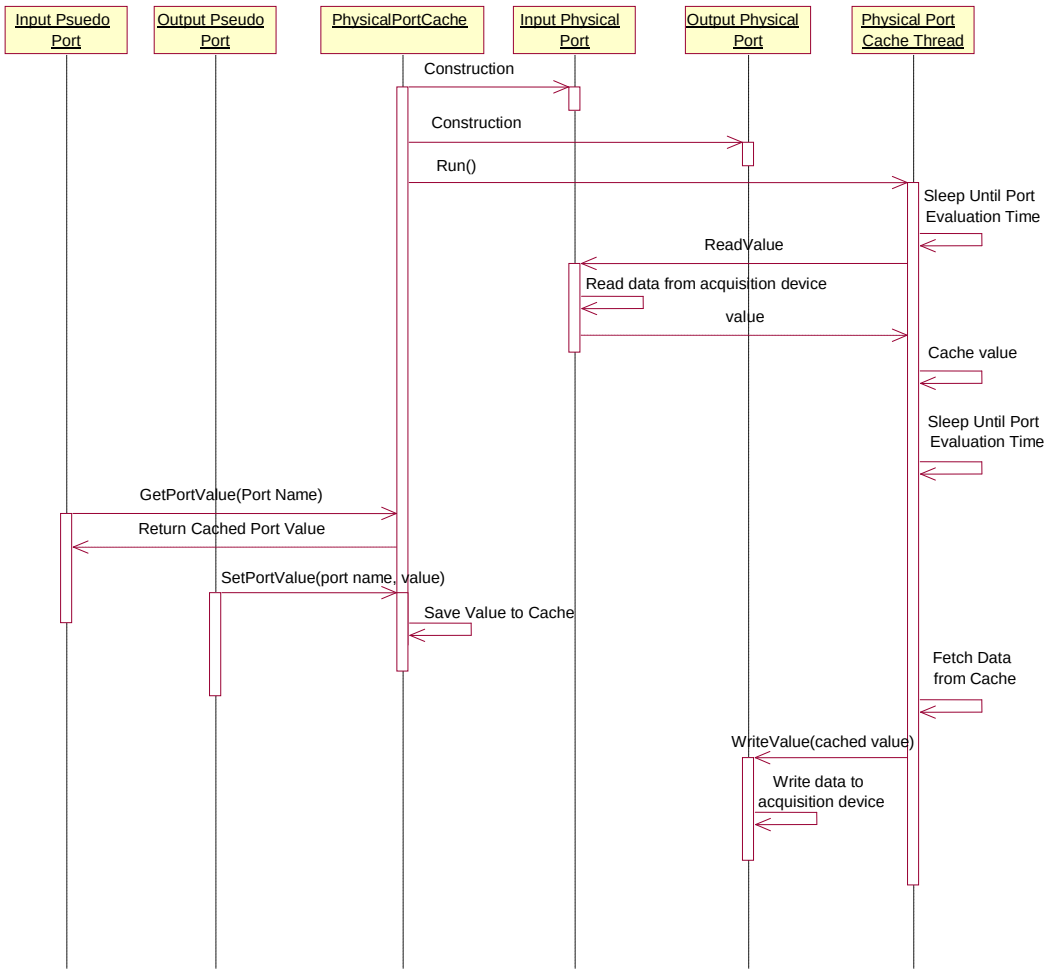

Figure 50: Physical Data Retrieval and Caching Sequence Diagram

As the Computation Kernel is launched, the PhysicalPortCache populates itself with PhysicalPort objects representing all of the physical interfaces offered by the data acquisition device connected to the system. Once the PhysicalPortCache has been populated with ports, it starts a separate thread of execution that will be responsible for reading from and writing to the physical ports. This thread determines which port is the next to be updated and how long it must wait before the update is to occur. This is a relative time measurement that does not account for computation time or the delay in activation out of sleep. The period of time between updates may be configured by the user for each port individually.

When the waiting period is up, the thread determines whether to read from, or write to the physical port. The first port update in figure 48 is an input port. The PhysicalPortCache commands the PhysicalPort to read from its interface and stores the returned value into a cache. Once the read is complete, the PhysicalPortCache determines how long to wait before the next update and sleeps again. When a PseudoPort needs to access physical data, it requests data from the PhysicalPortCache which provides the value from its internal cache. When a PseudoPort needs to write data to physical output, it sends the data to the PhysicalPortCache, which stores it in its internal cache for future use. Eventually an output PhysicalPort will be scheduled for update. When the PhysicalPortCache recognizes that it must update a PhysicalPort designated for output, it retrieves data from its cache and sends it to the PhysicalPort. The PhysicalPort then writes the data to its physical interface.

3.1.3.4 Simulation Start

When the user requests the start of simulation, the Win32 process must verify that the configurations are valid, and send the necessary configuration information to the Computation Kernel.

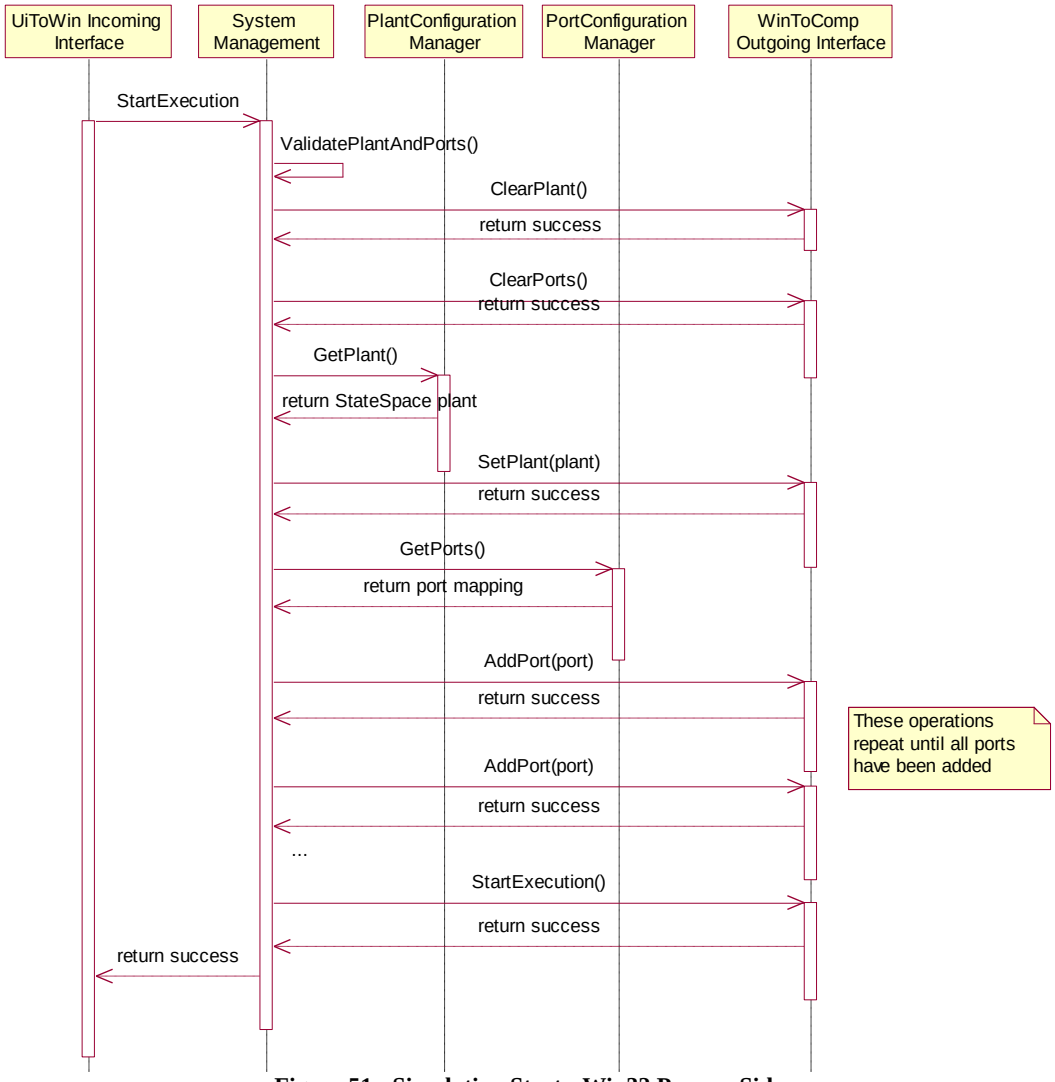

Figure 51: Simulation Start - Win32 Process Side

The process starts when the request from the User Interface reaches the Win32 process through the UiToWinIncomingInterface. The start execution call is forwarded to SystemManagement. Initially, the SystemManagement makes sure that the plant and port configuration is valid. This check makes sure that no input or output names are reused for either the plant or the ports, and that every input or output of the plant has a corresponding pseudo port.

Once the configuration has been validated, the SystemManagement sends commands to the Win32 process through the WinToCompOutgoingInterface to clear any current plant and port configurations. Once those are clear, the plant is fetched from the

PlantConfigurationManager and sent to the Computation Kernel. If the Computation Kernel accepts the plant, SystemManagement fetches the port mapping from the PortConfigurationManager. It walks through the port mapping and configures the Computation Kernel one port at a time. If every port is sent successfully, SystemManagement commands the Computation Kernel to begin simulation.

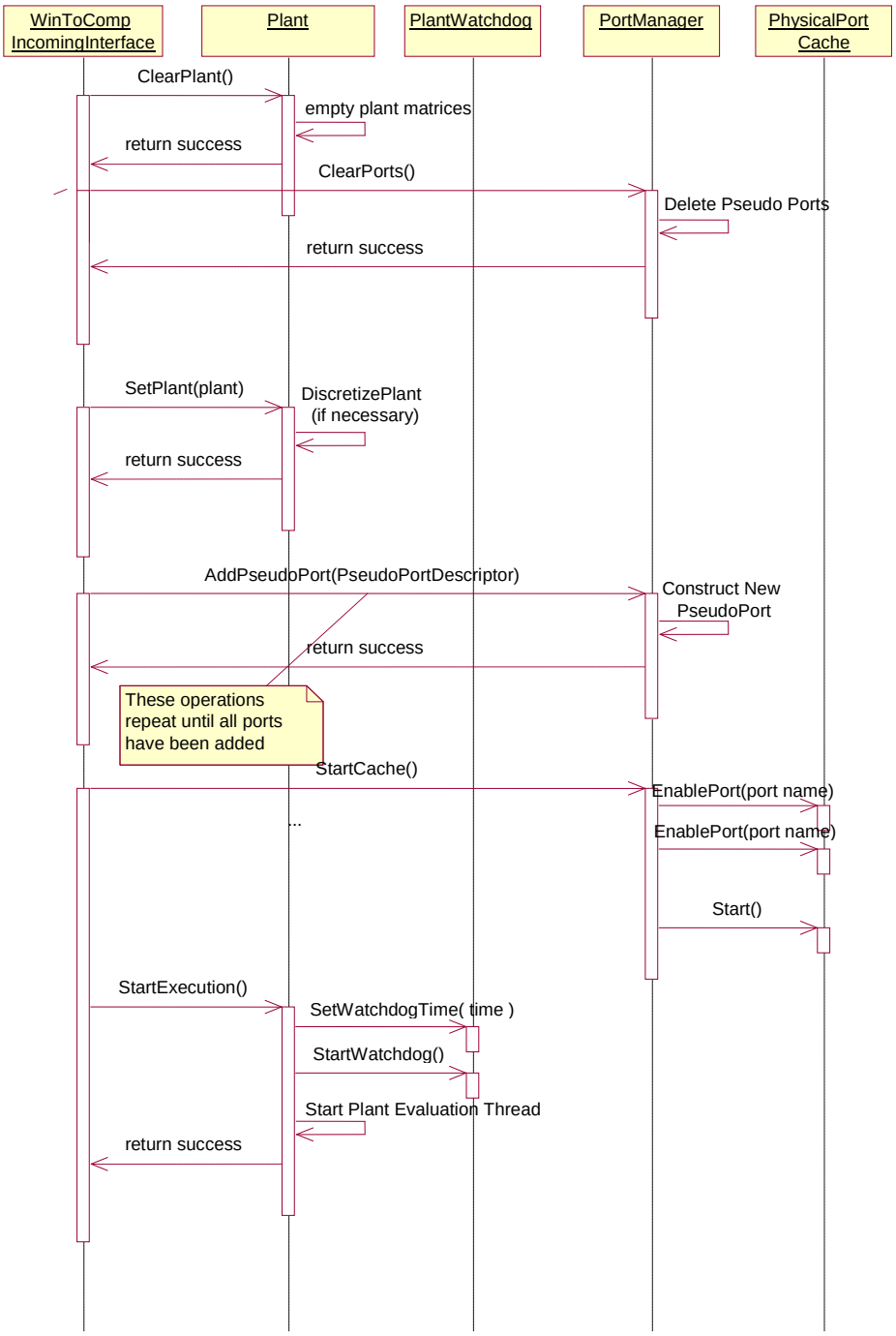

Figure 52: Simulation Start - Computation Kernel Side

The Computation Kernel must provide a response to each of the commands sent by the Win32 process. ClearPlant and ClearPort commands simply empty the current configurations. When a plant is established through the SetPlant command, the Plant will have to discretize any continuous plants before any operations may proceed. The AddPseudoPort operation provides the information needed to instantiate a PseudoPort object. The PortManager creates these PseudoPort objects according to the descriptors passed in, adding them one by one to its configuration. The actual start command results in two separate calls: StartCache called upon the PortManager, and StartExecution called upon the Plant object. When StartCache is called on the PortManager, it enables all of the physical ports that will be used for the simulation. The PhysicalPortCache has an object for every physical interface, but only a select few may actually need to be written to or read from. Calling EnablePort on the PhysicalPortCache indicates that the port enabled must be added to the list of ports that must be updated periodically. The StartExecution operation causes the plant to initiate the PlantWatchdog and begin to evaluate the plant at discrete intervals.

3.1.3.5 Simulation Cycle

A simulation cycle consists of all of the operations required to fully evaluate the plant at a particular instant in time. This involves retrieving inputs, evaluating outputs and states, and sending calculated output values down to PhysicalPortCache.

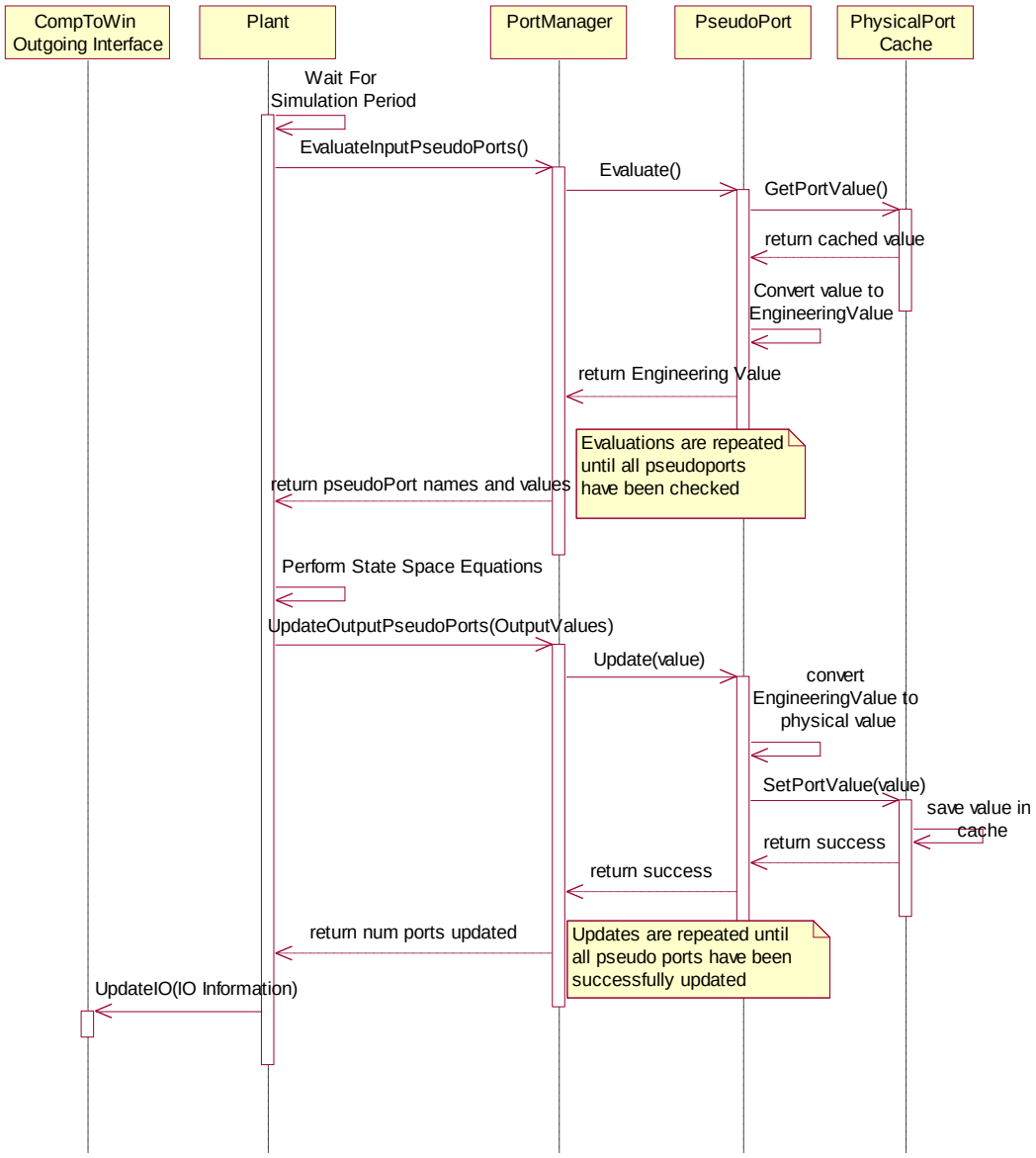

Figure 53: Simulation Cycle

Every simulation cycle begins by fetching Engineering values from the PortManager to be used as inputs. The EvaluateInputPseudoPorts call causes the PortManager to walk through all of the configured input PseudoPorts, calling Evaluate on each one of them. These PseudoPorts fetch the most recently cached physical value from the PhysicalPortCache, and converts it to an EngineeringValue if it is different from the last physical value retrieved from the PhysicalPortCache. If it was not, the previously calculated EngineeringValue is used instead. Either way, an EngineeringValue is returned to the PortManager. The PortManager compiles these results into a set of

port/value pairs and returns them to the Plant. The Plant uses these values as inputs and calculates the next set of outputs and states using the configured plant matrices according to Equations 7 and 8.

The calculated states are saved for the next simulation cycle. The outputs are sent to the PortManager through the UpdateOutputPseudoPorts call. Again, the PortManager walks through its list of output PseudoPorts and calls Update on each of them with the calculated output value. These PseudoPorts convert the new EngineeringValue to a physical output value. if necessary because it changed, and sends the data to the PhysicalPortCache through the SetPortValue operation. The PhysicalPortCache stores the output value in its cache until the next time the physical port is updated, at which point the cached value will be sent directly to physical output.

Finally, the plant updates the Win32 process with the current input/output data. This is performed according to the configurable IO Update Method. IO Information can be provided every simulation cycle, periodically every few simulation cycles, or only when requested by the Win32 process.

3.2 Implementation Details

The preceding section detailed the design of the Control System Plant Simulator. This section provides information about how the design is implemented in the package provided. Users are expected to customize, add to, and update the Control system Plant Simulator to meet their own needs. The programs provided by this thesis are but one of many possible versions.

3.2.1 Code and Programming Environment

The Control System Plant Simulator is an open source project written with the intent of allowing end users to change and rebuilt code. As such, some information regarding the build environment and code style is provided to simplify end user development.

3.2.1.1 File Organization

A complete listing of each file and where they are located is included in Appendix A. This section provides basic information as to how the files are organized, and where to look for the provided executable Control System Plant Simulator files.

The Control System Plant Simulator is provided as a package that includes precompiled executable files, and modifiable source code. All files are contained in a directory named **CSPS**. This will be referred to as the root directory.

There are four folders in the root directory: **CODE**, **DOCUMENTS,** **RTX_PKG,** and **WIN_PKG**.

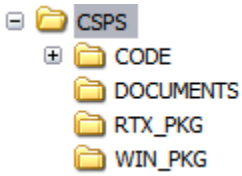

Figure 54: Directory Structure

The **CODE** directory contains all of the Visual Studio and Visual Basic projects and source code. The **DOCUMENTS** directory contains all of the documents provided for the CSPS Framework, including the User Manual. The **RTX_PKG** directory contains the executables required to launch the provided RTX® port of the Control System Plant Simulator. Finally, the **WIN_PKG** contains all of the executables needed to launch the Win32 version of the CSPS. Note that the Windows and RTX® versions are similar, but must be compiled separately.

The **CODE** directory contains a separate folder for each project in the Visual Studio 6.0 project, the project itself, and a folder for the Visual Basic user interface. It also contains a **Shared** folder used to house all of the code that does not change from version to version.

See Appendix A for a complete listing of the provided directories.

3.2.1.2 Required Tools

The Control System Plant Simulator was developed using Microsoft Visual Studio 6.0, and is intended to be run primarily as a 32 bit Windows console application. Two user interfaces are provided. One is a console application using Microsoft Visual Studio 6.0, and another is a simple GUI built with Microsoft Visual Basic 6.0. Two Computation Kernels are provided, both of which are built using the Microsoft Visual Studio programming environment. One of these is a Win32 process that does not run in real time, but can utilize USB interfaces, and the other is an RTX® process that DOES run in real time. The RTX® process was built using RTX® version 6.0 libraries**.** Two Win32 processes are provided as well. While neither runs in real time, one of them must be built using the RTX® libraries in order to communicate with the RTX® Computation Kernel. The Windows based Computation Kernel was written to communicate with the Data Translations 9812 data acquisition board. To build this process the Data Translations DT-OpenLayers libraries must be included. To successfully run this application, a Data Translations 9812 data acquisition device must be attached to the host PC via USB.

RTX® runs in Kernel mode. This is problematic because as a kernel mode application, RTX® processes may only be launched by a process with administrative privileges. This can be circumvented by creating a service that is responsible for launching RTX® processes because services run with administrator privileges. ServiceExample.zip in the **DOCUMENTS** directory is an example produced by Ardence® Technical support team as an example of how to create a service that can launch RTX processes.

3.2.1.3 Workspace and Projects

The code for the Control System Plant Simulator is provided as part of a Visual Studio 6.0 workspace titled CSPS. This workspace consists of the following projects:

- **CSPS_W32_CompKernel** – The CSPS_W32_CompKernel project contains the code and configurations necessary to build the Win32 port of the Computation Kernel. To build the CSPS_W32_CompKernel, the Active Configuration must be set to **Win32 Release**. Building this project will create the CSPS_W32_CompKernel.exe executable, a copy of which will be placed in the directory **/CSPS/WIN_PKG.**
- **CSPS_RTX_CompKernel –** The CSPS_RTX_CompKernel project contains the code and configurations necessary to build the RTX® Computation Kernel. To build the CSPS_RTX_CompKernel, the Active Configuration must be set to **Win32 RTSS Release** which will ensure that the final executable is a true RTX® process. Building this project will create the CSPS_RTS_CompKernel.rtss RTX® executable, a copy of which will be placed in the directory **/CSPS/RTX_PKG.**
- **CSPS_W32_Kernel** – The CSPS_W32_Kernel project contains the code and configurations necessary to build a version of the Win32 process that is compatible with the CSPS_W32_CompKernel Computation Kernel process. The Operating System Abstraction files are configured such that Windows operating system calls are used for inter-process communication. To build the CSPS_W32_Kernel, the Active Configuration must be set to **Win32 Release.**

Building this project will create the CSPS_W32_Kernel.exe executable, a copy of which will be placed in the directory **/CSPS/WIN_PKG.**

- **CSPS_W32_Kernel_RTX_PORT –** The CSPS_W32_Kernel_RTX_Port contains the code and configurations necessary to build a version of the Win32process that is compatible with the CSPS_RTX_CompKernel Computation Kernel process. The Operating System Abstraction files are configured such that RTX® operating system calls are used for the inter-process communication between the W32 process and the Computation Kernel. To build the CSPS_W32_Kernel_RTX_PORT, the Active Configuration must be set to **Win32 Release.** Building this project will create the CSPS_W32_Kernel_RTX_PORT.exe executable. Note that this is NOT an RTX® application. It does NOT run in real-time. It does need to be linked against the RTX® runtime library to access RTX® operating system calls in order to communicate with the CSPS_RTX_CompKernel.rss executable. Building this project places a copy of the CSPS_W32_Kernel_RTX_Port.exe file in **/CSPS/RTX_PKG.**
- **SimInterface –** The SimInterface project provides a 'helper' application outside of the scope of the Control System Plant Simulator. It is not needed to run the CSPS successfully. The RTX® version of the Control System Plant Simulator provided does not provide USB support. This means that the primary data acquisition device (Data Translations 9812) can not be used. Instead a simulated physical interface was created for the RTX® port. The CSPS_RTX_CompKernel is provided with a PhysicalPortCache designed to interact with this simulated interface. It will read and write data to shared memory instead of to a physical interface. The SimInterface program will allow the user to set input values, review output values, or run a script setting input values on the fly. See Section 6.1 of the User Manual to review the operational instructions for the SimInterface executable. To build the SimInterface project, the Active Configuration must be set to **Win32 Release.** Building this project will create the SimInterface.exe executable, a copy of which will be placed in **/CSPS/RTX_PKG.**

- **SimpleUI** – SimpleUI is a project providing an example of a simple text based Win32 that can interface with any version of the Control System Plant Simulator. Note that while it can be used as a fully functional interface, it is intended to serve as an example end users may review to help them create their own User Interface processes. For more information on how to construct a User Interface process, see Section 3 of the User Manual**.** It demonstrates how to make function calls from the User Interface to the Win32 process utilizing the UiWinInterface.dll. To build the SimpleUI project, the Active Configuration must be set to **Win32 Release.** Building this project will create the SimpleUI.exe executable, a copy of which will be placed in **/CSPS/WIN_PKG** and **/CSPS/RTX_PKG**. See Section 3.5 of the User Manual for more information about the commands accepted by Simple UI.
- **UiWinInterface** – The UiWinInterface project provides the files and configurations needed to construct the UiWinInterface dynamically linked library that can be used to aid User Interface development. To build the UiWinInterface project, the Active Configuration must be set to **Win32 Release**. Building this project will create the UiWinInterface.dll library, a copy of which will be placed in **/CSPS/WIN_PKG** and **/CSPS/RTX_PKG.**

In addition, a single Visual Studio project is provided. **ExampleUI.vbp** is a visual basic project that constructs a fully operational GUI that can interface with the Win32 process through the use of the UiWinInterface library. Like SimpleUI this is not intended to be used as an interface, but rather to serve as an example of how such a user interface may be created, but it is a fully functional working interface. See section 3.6 of the User Manual for more information about the ExampleUI interface and the functionality provided.

3.2.1.4 Conventions

As users are expected to modify the code provided by the Control System Plant Simulator, some information must be provided as to how the code is organized. This will help facilitate searching through the code when modifying or porting it.

3.2.1.4.1 LocalDefinitions.h

The vast majority of the code used in each project goes unchanged. Typically only the Operating System Abstraction objects must actually be changed. Different implementations of the PhysicalPortCache and PhysicalPort classes will also be needed for different physical port interfaces. The two ports of the Control System Plant Simulator use different physical data acquisition systems; the RTX® port interfaces with the SimInterface and the Win32 port interfaces with the Data Translations 9812 data acquisition device. As noted in Section 3.1.2.1.1 operating system objects provide two implementations, a primary and a secondary. For the Win32 port, all operations are carried out as Windows operating system calls, so the secondary implementation is ignored. For the RTX® port, the secondary implementation contains RTX® calls. This duality is achieved without modifying the Operating System Abstraction code through the use of pre-compiler directives.

Every project contains a file titled LocalDefinitions.h. LocalDefinitions.h defines a set of pre-compiler definitions that indicate whether or not a secondary interface is needed. If the secondary interface is required, it is up to the users to modify the Operating System Abstraction files to make the proper secondary interface operating system calls. LocalDefinitions.h also contains definitions that indicate which set of physical ports to include. Two sets are provided – SimPorts and DTPorts for the SimInterface and Data Translations 9812 respectively – but more may be added. This limits the amount of code that must be changed in the PortManager and prevents the creation of separate versions of the PortManager for each physical interface.

3.2.1.4.2 Extensions and Header Files

The Control System Plant Simulator is written in C++. All source files are written with .cpp extensions. Header files are provided with .h extensions. Every class is declared in its own header file. Detailed information about the signature of each function provided by the class is stored in the header file, but is left out of the source. This information includes what the function does, what each of its parameters are, and what the return value represents. Should a user be confused as to what a particular function does, or what the parameters represent, the .h file has the answers.

3.2.1.4.3 Naming Conventions

Naming conventions do not follow standard Windows MFC naming conventions. No type information is included in the name of a particular variable, and all names are provided in 'bumpy' notation. Local data members are prefaced by 'm', as in mDataMember. Variables are lowercase. Functions and class names are both uppercase.

3.2.2 Data Acquisition Device

The Control System Plant Simulator executables provided are designed to interoperate with the Data Translations 9812 data acquisition device. This is a USB driven data acquisition device that provides the following interfaces:

- One 8 bit Digital Input
- One 8 bit Digital Output
- 8 Analog Inputs accepting voltages from -10 volts to +10 volts
- 2 Analog Outputs, providing voltages from -10 volts to +10 volts.

The DTPhysicalPortCache and DTPhysicalPort classes have been provided to manage this data acquisition device.

3.2.3 Plant Definition Types Supported

The Control System Plant Simulator package provided offers six different ways to define a plant, each of which, with the exception of the state space method, must be converted to a set of state space equations. The plant may be defined as a set of State Space matrices, as a transfer function defined by a numerator and denominator, as a transfer function defined by a set of poles and zeros, as a matrix of transfer functions defined either way, or as a set of nonlinear equations. Transfer functions are converted to state space equations as per the controller algorithm defined in Section 2.1.4**.** Matrices of transfer functions are converted to state space equations by converting each transfer function and placing the resultant state space matrix as a submatrix within the overall system matrix. Conversion of a set of nonlinear equations is performed as described in Section 2.1.7**.** New plant definition types may be added by modifying the PlantConfigurationManager class, and adding to the options available to the user when setting the plant.

3.3 Evaluation

In order to gage the effectiveness of the Control System Plant Simulator, a number of plants were simulated using both versions (Win32 and RTX®) and compared against a Matlab simulation of the same plant. Note that the Win32 version can fully interface with the Data Translations DT 9812 device, and therefore can behave as a full Hardware-In-The-Loop system. Each plant was simulated using the Win32 version along with the Data Translations DT-9812 data acquisition device, allowing for full physical IO. All outputs provided for the Win32 version of the CSPS are oscilloscope captures. The RTX® port, however, cannot send data to the Data Translations DT-9812 device. Instead, the RTX system was interfaced with a simulated port interface. All RTX® results were taken from the log file produced while running under RTX®.

Finally, the CSPS was provided to a class of Real-time Systems students as a tool to enable them to develop their own real-time plants. The following plants were used to test the effectiveness of the Control System Plant Simulator.

3.3.1 Results Comparison

The following table outlines the results for each plant against the expected values.

Table 2: Results Comparison

Plant	CSPS Version	Expected Value	Simulated value	% difference
Spring-Mass	Win32	0.15 at 1.26 sec	0.14 at 1.26 sec	-6.6%
		0.251 at 2.26 sec	0.25 at 2.26 sec	-0.4%
		0.329 at 5.38 sec	0.33 at 5.38 sec	+0.3%
	RTX®	0.15 at 1.26 sec	0.148 at 1.3 sec	-1.3%
		0.251 at 2.26 sec	0.251 at 2.3 sec	0%
		0.329 at 5.38 sec	0.33 at 5.4 sec	+0.3%
Airplane Pitch	Win32	0.44 at 4.57 sec	0.44 at 4.56 sec	0%
		0.468 at 6.83 sec	0.46 at 6.80 sec	-1.7%
		0.838 at 16.1 sec	0.812 at 16.1 sec	-3.1%
	RTX®	0.44 at 4.57 sec	0.44 at 4.6 sec	0%
		0.468 at 6.83 sec	0.47 at 6.8 sec	+0.4%
		0.838 at 16.1 sec	0.84 at 16.1 sec	+2.4%
Bus Suspension	Win32	2.23e-5 at 0.633 sec	2.04e-5 at 0.64 sec	-8.5%
		3.55e-6 at 1.25 sec	3.30e-5 at 1.28 sec	-7.0%

		1.29e-5 at 28.2 sec	1.02e-5 at 28.2 sec	-20.9%
	RTX®	2.23e-5 at 0.633 sec	2.2e-5 at 0.6 sec	-1.34%
		3.55e-6 at 1.25 sec	4.0e-6 at 1.2 sec *	+15%
		1.29e-5 at 28.2 sec	1.3e-5 at 28.2 sec	+0.8%
Car Shock System	Win32 Car Pos	1.42 at 1.34 sec	1.52 at 1.38 sec	+7.0%
		0.988 at 6.14 sec	1.02 at 6.14 sec	+3.2%
	Win32 Wheel Pos	1.26 at 1.24 sec	1.22 at 1.24 sec	-3.2%
		0.955 at 5.76 sec	0.98 at 5.76 sec	+2.6%
	RTX® Car Pos	1.42 at 1.34 sec	1.42 at 1.3 sec	0%
		0.988 at 6.14 sec	0.988 at 6.1 sec	0%
	RTX® Wheel Pos	1.26 at 1.24 sec	1.26 at 1.2 sec	0%
		0.955 at 5.76 sec	1.00 at 5.8 sec	4.7%

* the value at 1.3 sec = 3.0e-6, so 3.55 for 1.25 is reasonable.

The sections that follow contain specifics about each of the simulated plants and the values obtained from them.

3.3.2 Spring – Mass System

One of the simplest plants evaluated is that of a spring and weight system. This system models a mass connected to a spring. A force, the input, is applied pushing the mass, which stretches the spring. The spring and a damping frictional force work against the mass.

3.3.2.1 State Space Equations

The state space equations are defined as follows:

State Matrix: $\begin{bmatrix} 0 & 1 \\ -1.5 & -2.5 \end{bmatrix}$

Input Matrix: $\begin{bmatrix} 0 \\ 0.5 \end{bmatrix}$

Output Matrix: $[0 \quad 1]$

Feedthrough Matrix: $[0]$

[25]

3.3.2.2 *Matlab Simulation*

Figure 53 contains the Matlab simulated step response to the Spring – Mass system.

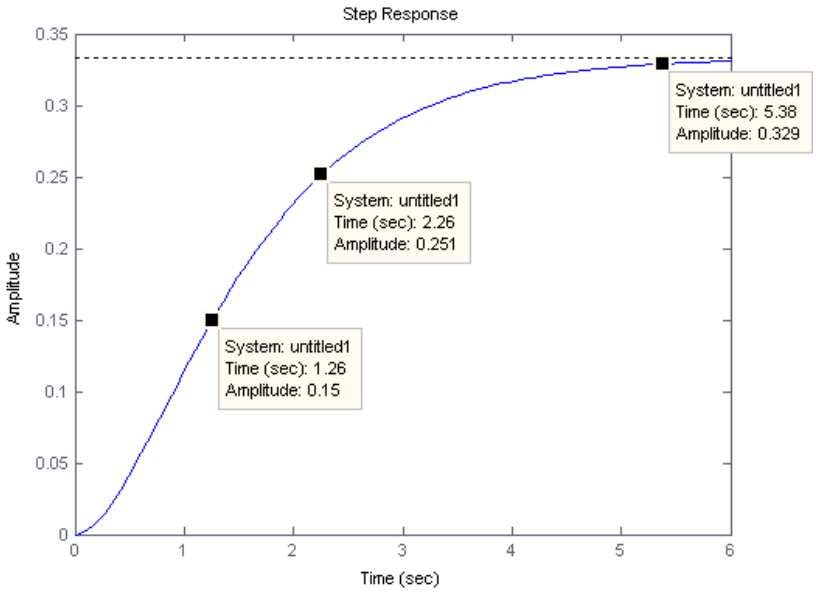

Figure 55: Matlab Simulation of the Step Response of the Spring-Mass System

Note the indicated data points along the curve. These three data points are compared directly to data points taken at similar points in time during physical simulation.

3.3.2.3 *Windows CSPS Simulation*

The following figures contain captures from an oscilloscope during the step-response simulation of the Spring – Mass plant.

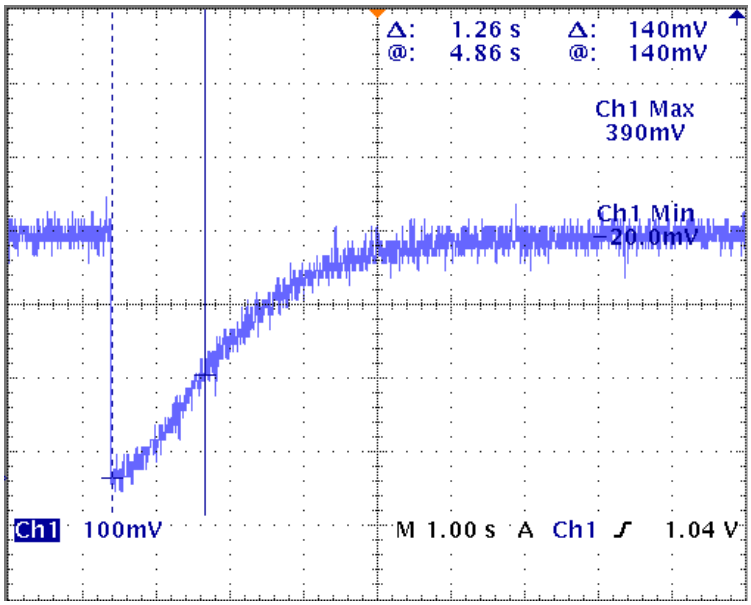

Figure 56: Oscilloscope Capture of Simulation of Spring Mass System, With Datapoint at 1.26 Seconds

Figure 54 displays the oscilloscope capture of the step response of the Spring-Mass plant. The cursors measure the difference between time and voltage from the initial state. After 1.26 seconds the voltage has increased by 0.14 volts. Compare this to the Matlab simulation, where after 1.26 seconds, the amplitude had reached a value of 0.15.

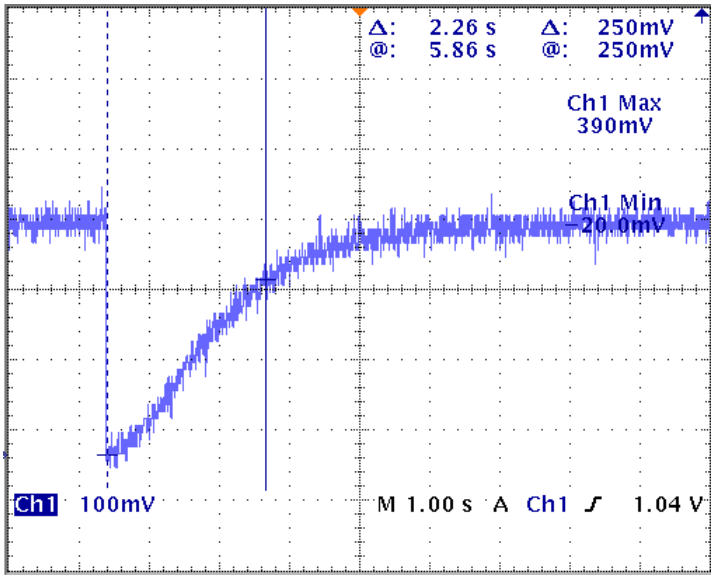

Figure 57: Oscilloscope Capture of Simulation of Spring Mass System, With Datapoint at 2.26 Seconds

Figure 55 shows that after 2.26 seconds, the voltage has increased by 0.25 volts. Compare this to the Matlab simulation where, after 2.26 seconds, the amplitude had reached a value of 0.251.

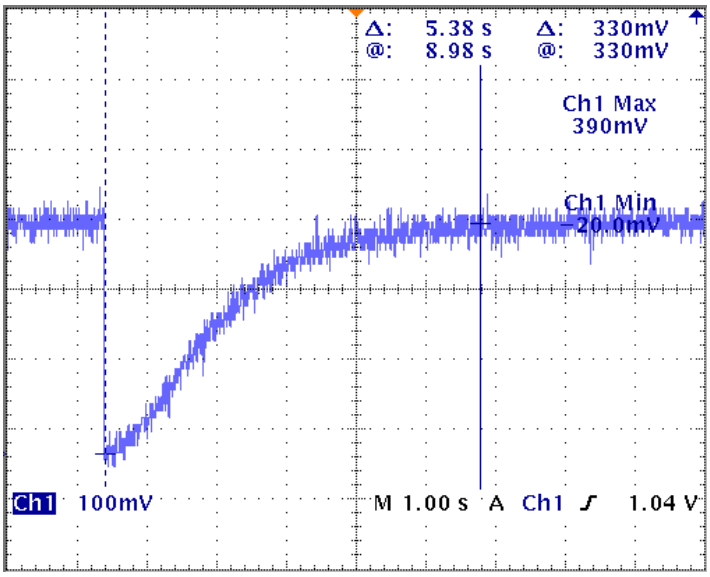

Figure 58: Oscilloscope Capture of Simulation of Spring Mass System, With Datapoint at 5.38 Seconds

Finally, figure 56 shows that after 5.38 seconds, the voltage has reached 0.33 volts. Compare this with the Matlab simulation where, after 5.38 seconds, the amplitude reached a value of 3.29.

Note that the signal appears very noisy. This happens primarily because the curve is very small; it reaches steady state at approximately 300 millivolts. The captures were obtained after zooming in a great deal to capture the distinct features of the curve causing small variations to appear quite large.

3.3.2.4 *RTX® CSPS Simulation*

The following are diagrams created from the log files produced during the simulation of plant using the RTX® CSPS framework.

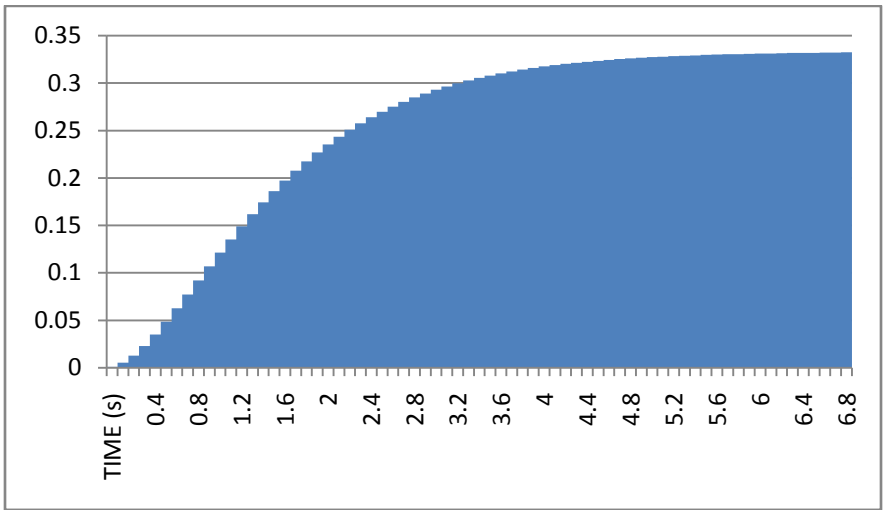

Figure 59: RTX Simulation of Spring-Mass System

One can see through inspection that the output produced is similar to that of the actual curve as calculated by Matlab. Specific values produced by the simulation to compare against those of Matlab are:

- 0.148 volts at 1.3 seconds
- 0.251 volts at 2.3 seconds
- 0.33 volts at 5.4 seconds

3.3.3 Airplane Pitch Plant

To demonstrate the ability of the Control System Plant Simulator to represent an unstable plant, the following model of how an aircraft will pitch given the elevator deflection angle was simulated.

3.3.3.1 State Space Equations

The model was defined through the following state space equations:

State Matrix: $\begin{bmatrix} -0.739 & -.921 & 0 \\ 1 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$

Input Matrix: $\begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix}$

Output Matrix: $[0 \quad 0.4604 \quad 0.1419]$

Feedthrough Matrix: $[0]$

[24]

3.3.3.2 Matlab Simulation

Figure 58 contains the Matlab simulated step response to the plant

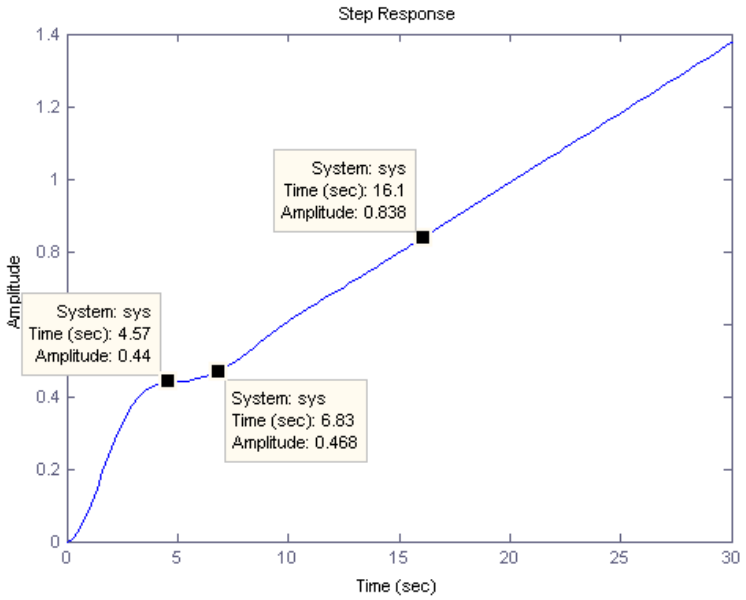

Figure 60: Matlab Simulation of Airplane Pitch Plant

Note the indicated data points along the curve. These data points will be compared against the physical values obtained during simulation. It is also worth noting that the response increases without bounds; the system is unstable.

3.3.3.3 *Windows CSPS Simulation*

The following figures contain captures from an oscilloscope during the step-response simulation of the Airplane Pitch plant.

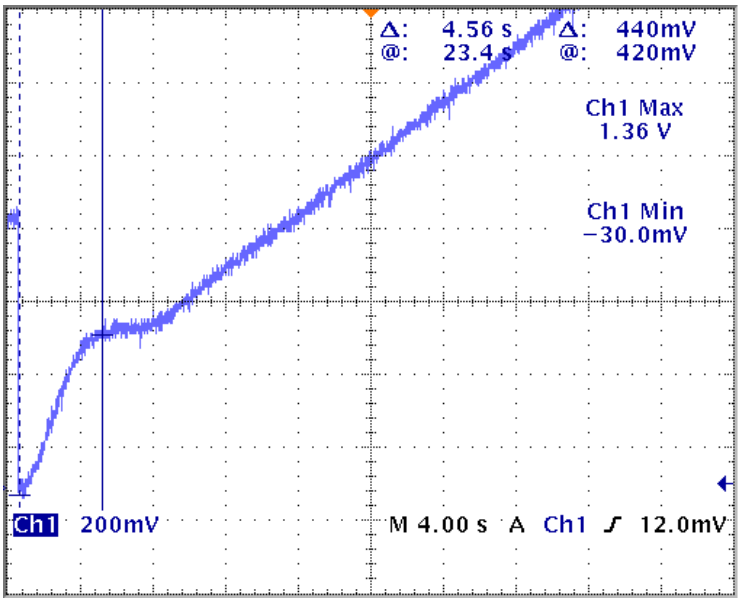

Figure 61: Oscilloscope Capture of Simulation of Airplane Pitch Plant, With Data Point at 4.56 Seconds

Figure 59 displays the oscilloscope capture of the step response of the Airplane Pitch plant. The cursors measure the difference between time and voltage from the initial state. After 4.56 seconds the voltage has increased by 0.44 volts. Compare this with the Matlab simulation where after 4.57 seconds, the amplitude reached a value of 0.44.

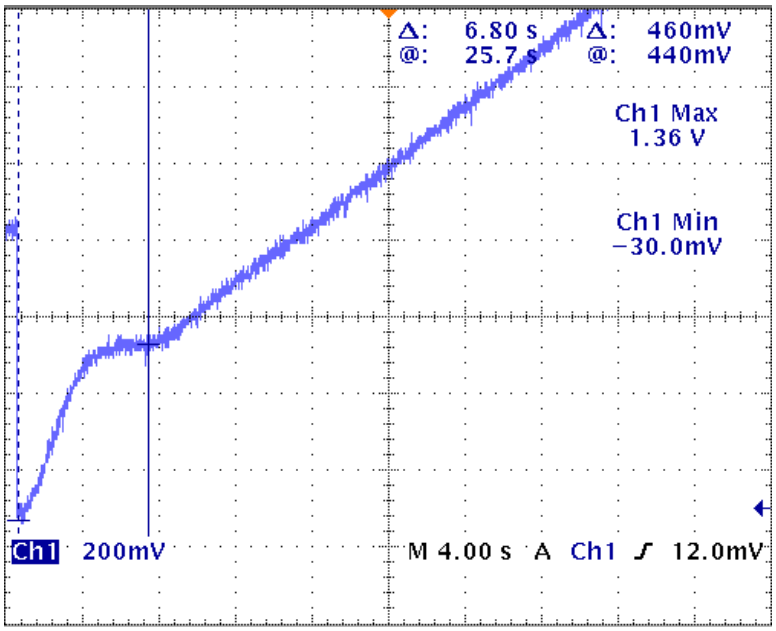

Figure 62: Oscilloscope Capture of Simulation of Airplane Pitch Plant, With Data Point at 6.80 Seconds

Figure 60 shows that after 6.80 seconds, the output has increased by 0.460 volts. Compare his with the Matlab simulation where, after 6.83 seconds, the amplitude reached a value of 0.468.

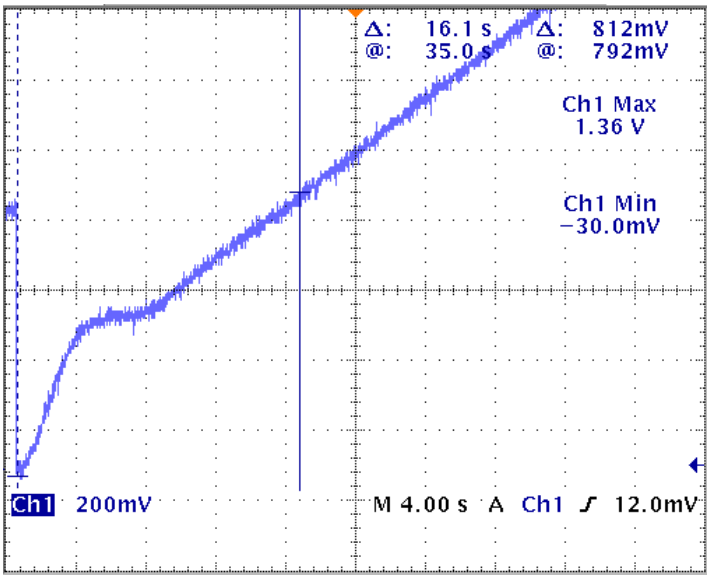

Figure 63: Oscilloscope Capture of Simulation of Airplane Pitch Plant, With Data Point at 16.1 Seconds

Finally, figure 61 shows that, after 16.1 seconds, the voltage has increased by 0.812 volts. Compare this with the Matlab simulation that shows that after 16.1 seconds, the amplitude had reached a value of 0.838.

3.3.3.4 *RTX® CSPS Simulation*

The following are diagrams created from the log files produced during the simulation of plant using the RTX® CSPS framework.

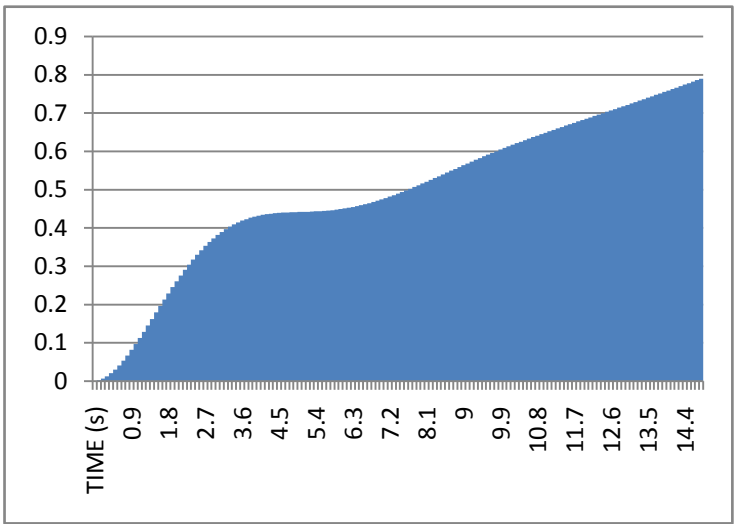

Figure 64: RTX Simulation of Airplane Pitch Plant

Again one can clearly see that the simulation closely follows the curve produced by Matlab. Actual data points to be compared with those read from Matlab are:

- 0.44 volts at 4.6 sec
- 0.47 volts at 6.8 sec
- 0.84 volts at 16.1 sec

3.3.4 Bus Wheel and Suspension System

Bus suspension systems can be simplified as spring – damper systems. The following system was chosen to be simulated because its output is extremely small. This will not only test the Control System Plant Simulator's ability to simulate a plant, but also the ability of the Analog Pseudo Ports to scale values up into output ranges. If the pseudo port is configured to scale all engineering values from 0 to 0.00003 to the physical range of -10 volts to 10 volts, than any engineering value of 0 will be scaled down to -10 volts of physical output, where a value of 0.00003 will be amplified to 10 volts. Thus all

values will be scaled by approximately 666,666.67. Any calculation errors should be magnified and noticeable.

3.3.4.1 State Space Equations

The State Space equations are defined as follows:

State Matrix: $\begin{bmatrix} -48.17 & -28.92 & -3.361 & -12.21 \\ 64 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 8 & 0 \end{bmatrix}$

Input Matrix: $\begin{bmatrix} 0.01563 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Output Matrix: $[0 \quad 0.003525 \quad 0.002347 \quad 0.009766]$

Feedthrough Matrix: $[0]$

[24]

3.3.4.2 Matlab Simulation

Figure 63 contains the Matlab simulated step response to the plant

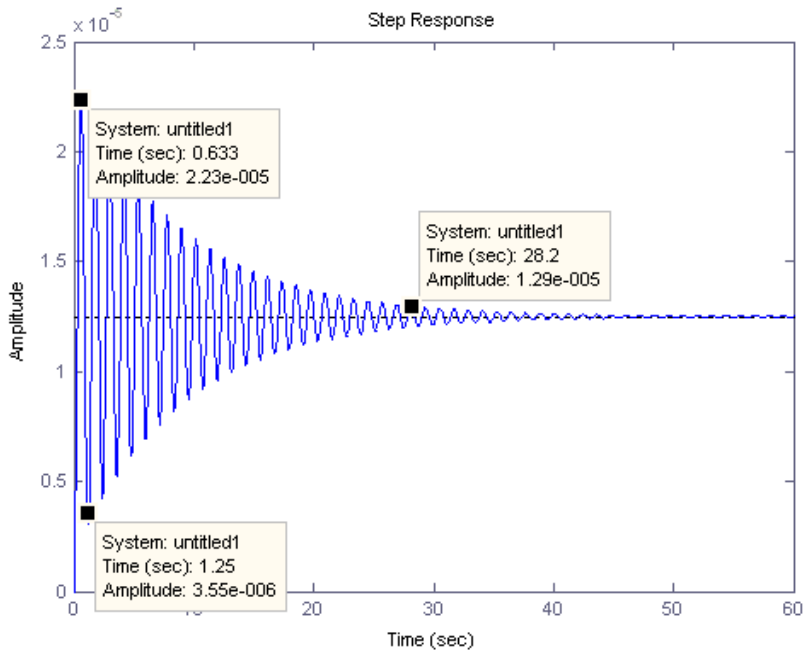

Figure 65: Matlab Simulation of Bus Wheel and Suspension System

The bus suspension system oscillates before reaching a steady state final value. Note however, how very small the indicated values are. These data points will be compared against the scaled up versions provided by the physical output during simulation.

3.3.4.3 *Windows CSPS Simulation*

The following figures contain captures from an oscilloscope during the step-response simulation of the Bus Suspension plant.

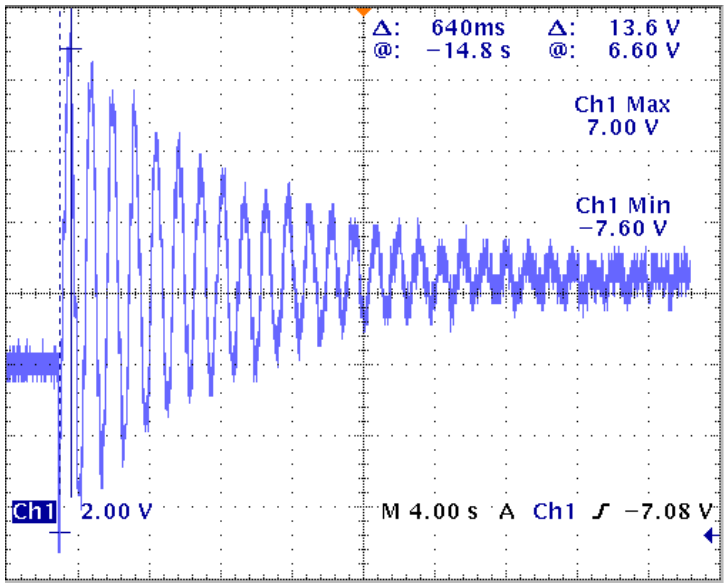

Figure 66: Oscilloscope Capture of Simulation of Bus Wheel and Suspension System, With Data Point at 0.64 Seconds

Figure 64 is an oscilloscope capture of the step response of the plant, as simulated by the Control System Plant Simulator. Here one sees that after 640 milliseconds the system has a voltage difference of 13.6 volts from startup. The pseudo port for this particular output was established with a minimum value of 0 and a maximum value of 0.00003, meaning that the engineering value 0 is scaled to -10 volts for physical output, and the engineering value of 0.00003 is scaled up to +10 volts. This is the reason the output has been recentered around 0 volts, and why the difference between the initial position and the measured peak is so great. A difference of 13.6 volts is approximately

the equivalent of an amplitude of 0.0000204. Compare this to the Matlab simulation where, after 633 milliseconds the system reached an amplitude of .0000223.

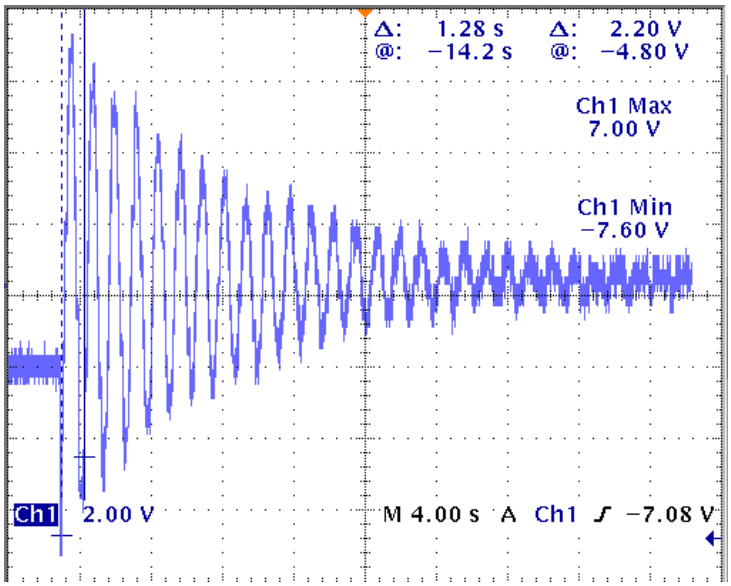

Figure 67: Oscilloscope Capture of Simulation of Bus Wheel and Suspension System, With Data Point at 1.28 Seconds

Figure 65 indicates a value of 2.2 volts after 1.28 seconds. After conversion, this is approximately equal to an amplitude of 0.0000033. Compare this to the Matlab simulation where, after 1.25 seconds the amplitude reached a value of .00000355.

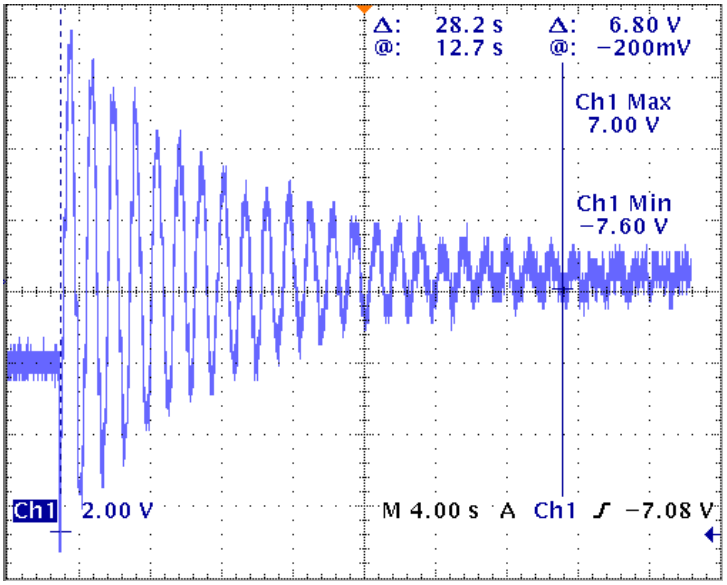

Figure 68: Oscilloscope Capture of Simulation of Bus Wheel and Suspension System, With Data Point at 28.2 Seconds

Finally, figure 66 indicates a value of 6.8 volts after 28.2 seconds of execution. After conversion, this is approximately equal to an amplitude of 0.0000102. Compare this with the Matlab simulation where, after 28.2 seconds the system reached an amplitude of 0.0000129.

3.3.4.4 *RTX® CSPS Simulation*

As described previously, it was only possible to simulate the RTX® port. The following is a chart of the data collected during that simulation.

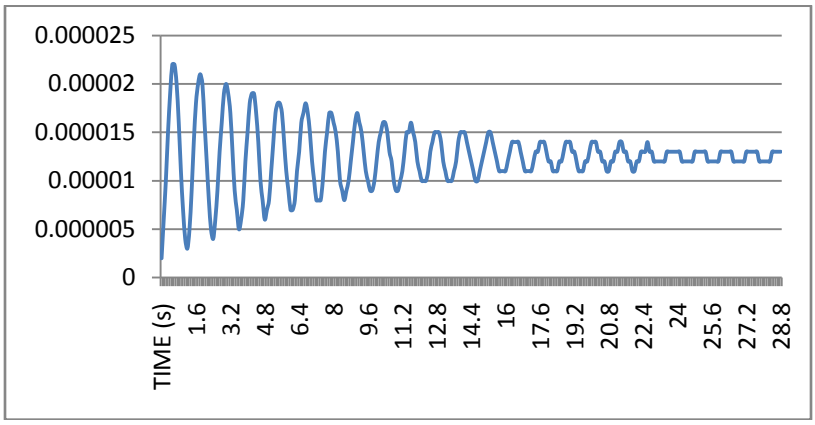

Figure 69: RTX Simulation of Bus and Wheel Suspension System

The curve produced by the Control System Plant Simulator is remarkably similar to the curve produced by Matlab. Actual data points to be compared with those read from Matlab are:

- 0.000022 at 0.6 sec
- 0.000004 at 1.2 sec
- 0.000013 at 28.2 sec

3.3.5 Car and Wheel Shock Absorber System

The final plant used for evaluation is a simple automobile shock absorber system. The system has one input (Road position) and two outputs (Car position and Wheel Position), testing the multiple output capability of the Control System Plant Simulator.

3.3.5.1 State Space Equations

The State space equations for the Car and Wheel Shock Absorber system are defined as follows:

State Matrix: $\begin{bmatrix} 0 & 1 & 0 & 0 \\ -160 & -60 & 80 & 40 \\ 0 & 0 & 0 & 1 \\ 8 & 4 & -8 & -4 \end{bmatrix}$

Input Matrix: $\begin{bmatrix} 20 \\ -1120 \\ 0 \\ 80 \end{bmatrix}$

Output Matrix: $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

Feedthrough Matrix: $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$

[25]

3.3.5.2 Matlab Simulation

Figure 68 contains the step response of the Car Suspension system plant.

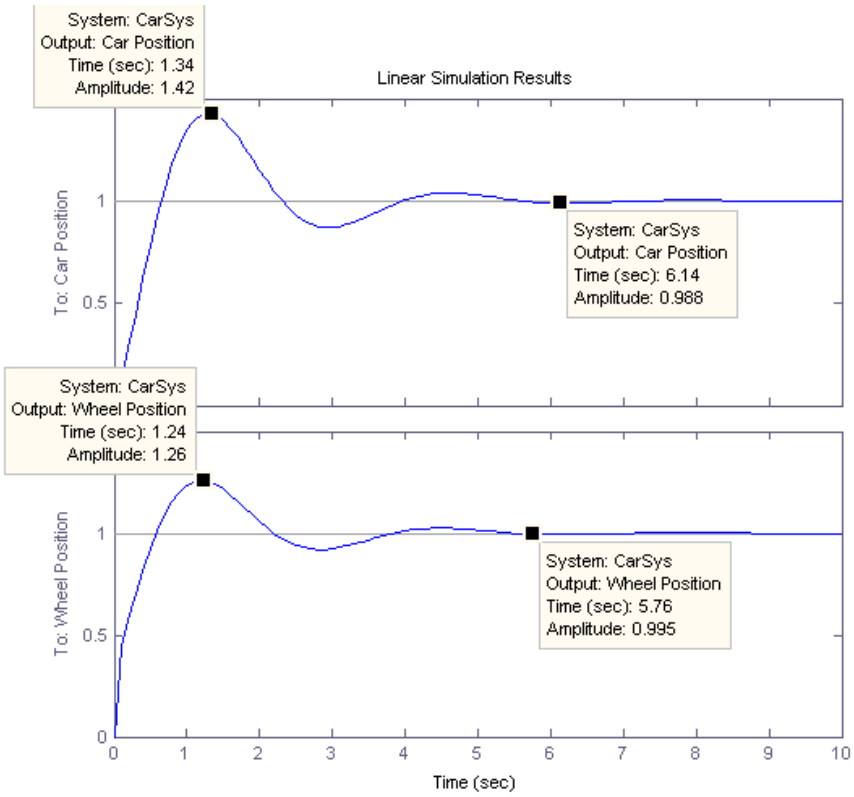

Figure 70: Matlab Simulation of Car Shock Absorber System

Note the data points taken at the peak of both outputs and at the point near the steady state. These data points will be compared against the values obtained during simulation.

3.3.5.3 Windows CSPS Simulation

The following figures contain captures from an oscilloscope during the simulation of the Car Suspension system plant.

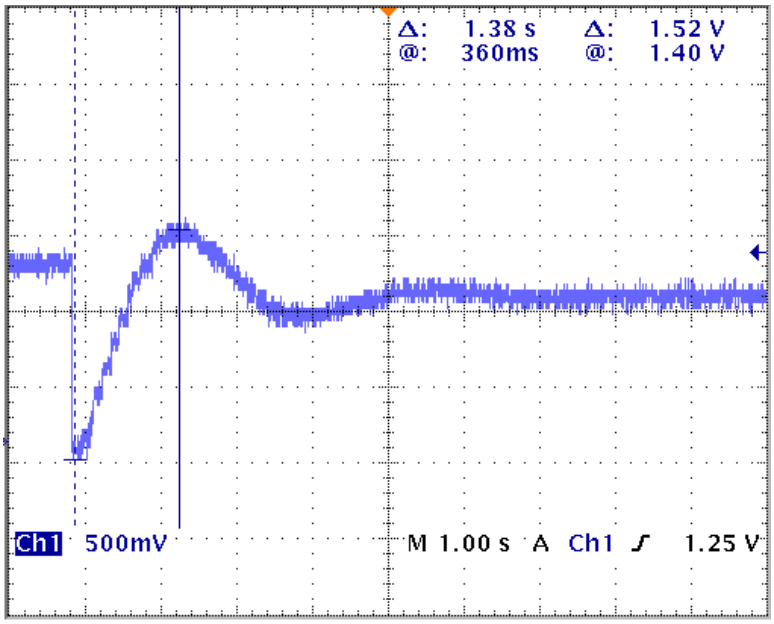

Figure 71: Oscilloscope Capture of Simulation of Car Position Step Response with Data Point at 1.38 Seconds

Figure 69 is a capture of the car position step response. Note the difference in time and voltage are measured from the beginning of simulation. At 1.38 seconds the output has peaked at 1.52 volts. Compare this with the Matlab simulation where after 1.34 seconds the amplitude reached a value of 1.42.

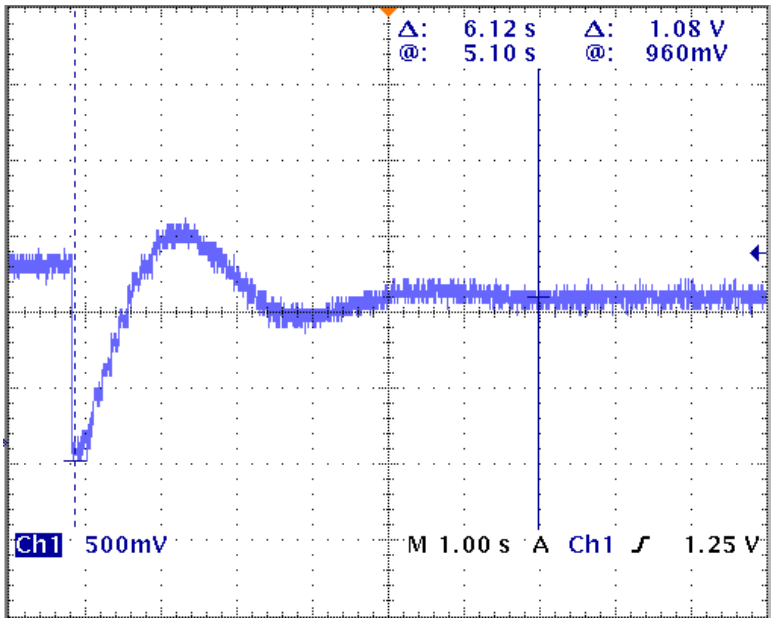

Figure 72: Oscilloscope Capture of Simulation of Car Position Step Response with Data Point at 6.12 Seconds

Figure 70 is a capture of the car position step response as well, with time and voltage measurements from zero to near steady state. Note that the system steadies out after approximately 6.12 seconds at 1.08 volts. Compare with the Matlab simulation where after 6.14 seconds the amplitude reached a value of 0.988.

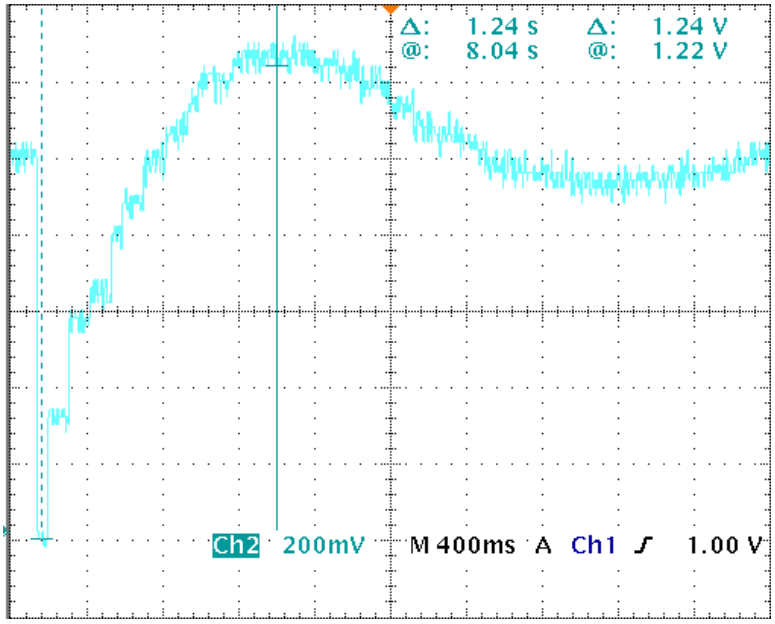

Figure 73: Oscilloscope Capture of Simulation of Wheel Position Step Response with Data Point at 1.24 Seconds

Figure 71 is a capture of the wheel position step response. The change in time and in voltage are measured from 0, thus the voltage peaks at 1.22 volts after 1.24 seconds. Compare with the Matlab simulation where after 1.24 seconds the amplitude reached a value of 1.26.

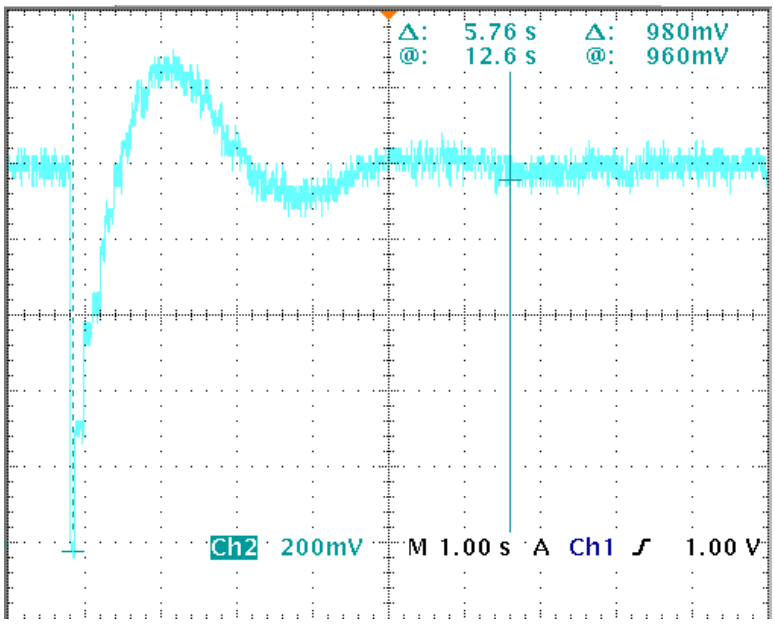

Figure 74: Oscilloscope Capture of Simulation of Wheel Position Step Response with Data Point at 5.76 Seconds

Figure 72 is a capture of the wheel position step response as well, with time and voltage measurements from zero to near steady state. Note that the system steadies out after approximately 5.76 seconds at 0.98 volts. Compare with the Matlab simulation where, after 5.76 seconds, the amplitude reached a value of 0.995.

3.3.5.4 RTX® CSPS Simulation

The following charts were produced from data captured during simulation under RTX®.

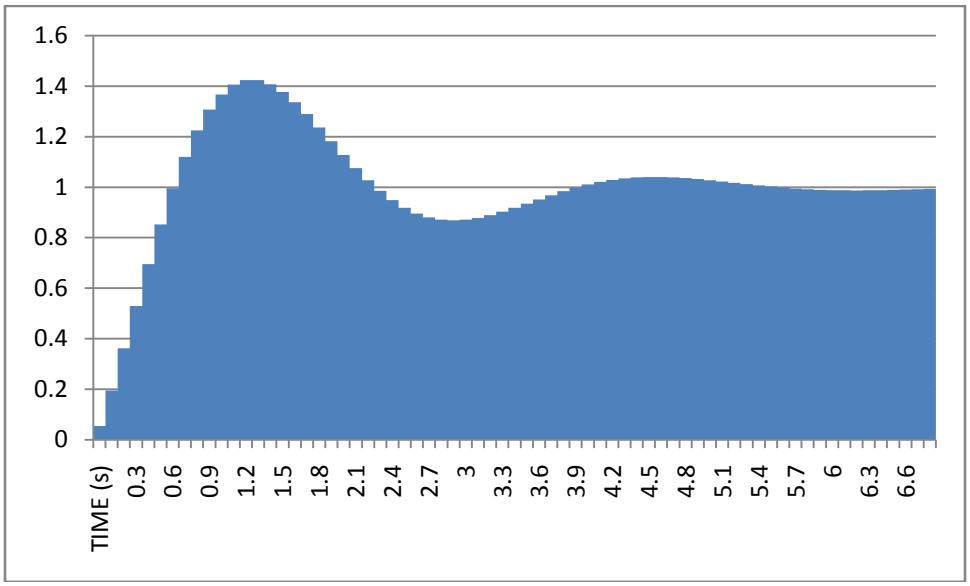

Figure 75: RTX Simulation of Car Position Step Response

Figure 73 provides Car Position data from the RTX® simulation of the plant. Relevant data points for comparison against the Matlab plot are as follows:

- 1.42 volts at 1.3 seconds
- 0.988 volts at 6.1 seconds

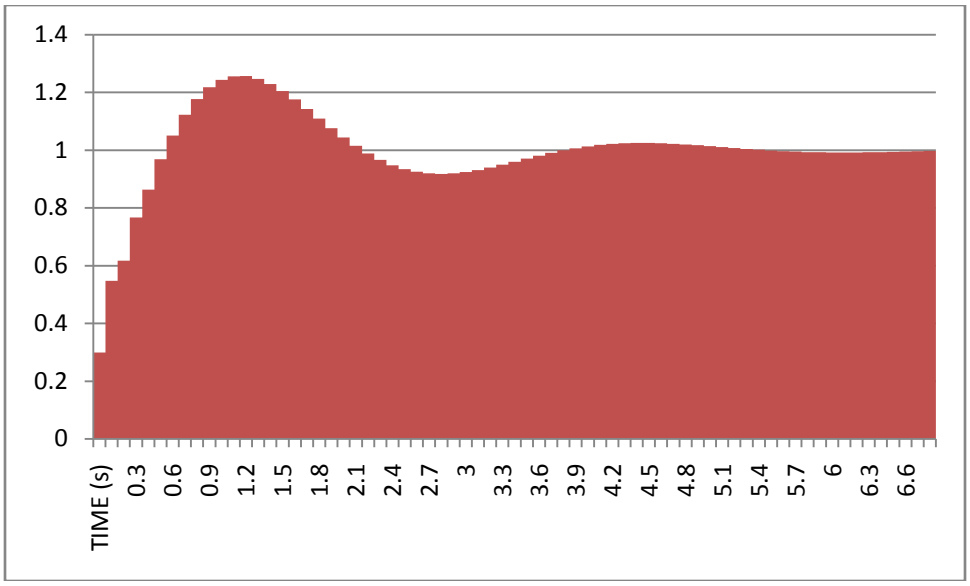

Figure 76: RTX Simulation of Wheel Position Step Response

Figure 74 provides Wheel Position data from the RTX® simulation of the plant. Relevant data points for comparison against the Matlab plot are as follows:

- 1.22 volts at 1.24 seconds
- 0.98 volts at 5.76 seconds

3.3.6 Comments on Results

In general, the results are very encouraging, with most simulated error at or below three percent. Considering these simulations are discretized versions of continuous plants, sampled at the relatively low frequency of ten hertz, and that the physical simulations were conducted in Windows and not a Real-Time system, the results are encouraging.

There are, of course, a few notable exceptions. The first physical measurements of the Spring-Mass plant and of the Car Shock Absorber system were both large when compared to the other measurements. In the case of the Spring-Mass system, granularity of the measurements becomes a problem. The expected value was 0.15, and the measured value was 0.14. This is as close as the measurements can be without being equal. The large percentage difference is more a result that the values are small than that they differ in any meaningful way. The measurement taken for the Car Shock Absorber system is not truly comparable to that taken in Matlab as the values are measured at

slightly different moments in time. The curve indicates that, should a closer measurement have been possible, the outcome would have been much better.

Finally, the values from the Windows simulation of the Bus Suspension System were not very close. When one considers the fact that the output of the plant is being magnified by over 65,000 it is easy to understand why these values differ so much, but the percent error in this case would be limiting. For this reason, the Control System Plant Simulator can not be recommended for systems that require such gross magnification of their output. Scaling Engineering Values to physical data must be kept to a more reasonable range. Worth noting, however, is the fact that the curves constructed were similar in structure and in timing. It is just the magnitude that was unfortunately inaccurate.

3.3.7 Classroom Application

The Control System Plant Simulator was developed around the idea that its primary function would be to assist in the education of control system development. During development, the Control System Plant Simulator was provided to a class of graduate and undergraduate Software and Computer Engineering students. The students used the Control System Plant Simulator to develop simple VxWorks controllers for the Car and Wheel shock absorber system described in section 3.3.5. The project required them to manually tune the coefficients of a standard Proportional-Integral-Derivative controller to minimize the overshoot and settling time of the plant. They did not know the plant model and tuned the controller experimentally. The experiment was a success as the students were able to construct reasonable controllers that produced real physical control signals to manage the simulated plant. There were no complaints or problems using the CSPS Framework for this project. This serves as evidence that the Control System Plant Simulator has the capability to become a reasonably useful tool in education.

4. Future Work

There are many ways the Control System Plant Simulator may be improved and extended. The Control System Plant Simulator was designed with future development in mind. From complete modifications to how plants are input and discretized, to the addition of robust user interfaces designed for specific plant simulations, the Control System Plant Simulator is a framework upon which many modifications and customizations may be made. Specific examples include

- User Interfaces. It cannot be stressed enough that end developers should take advantage of the modular design of the Control System Plant Simulator by constructing their own user interfaces tailored for individual plants. Considerable effort was invested to simplify the process of interfacing UI applications with the CSPS, not the least of which is the UiWinInterface dynimcally linked library, allowing any program that can link a dll access to the features of the CSPS. See Section 3, User Interface Development of the CSPS User Manual (Provided in Appendix B**)**
- Addition of new data acquisition devices. The data acquisition device chosen for the initial version of the CSPS was certainly limiting. While the DT-9812 unit performed admirably, the USB interface eliminates the ability to use true real-time systems as USB is inherently non-deterministic. Other developers are encouraged to provide their own data acquisition devices and their own physical port implementations that do not suffer such restrictions. This would open up real-time development for the CSPS and improve the accuracy and complexity of the plants simulated, as samples could be made at much higher rates. See the **Data Acquisition Device Development** section of the CSPS User Manual (provided in Appendix B**)** for more information on how to create and integrate new devices.
- Addition of new methods. The CSPS provides basic methods for discretization and state space realization that can certainly be improved upon. More complex and accurate methods for converting continuous plants to discrete, or for finding a set of state space equations from transfer functions exist and would only enhance

the ability of the CSPS to accurately simulate plants. Engineering Value arithmetic can be altered, and would see a performance improvement by switching from true floating point to integer based fixed point systems for example.

5. Conclusion

The goal of this Thesis was to provide a framework upon which known modeled plants could be launched and simulated as part of a Hardware-In-The-Loop system with an eye toward aiding education in controls development. Control system education benefits greatly by having students develop real controllers that are connected to real systems to monitor the success or failure of their designs. Such physical systems can be expensive or dangerous, making their use in an educational laboratory environment difficult at best. The Control System Plant Simulator provides a suite of applications that simulate plants with physical output accurate enough that students can design and develop controllers for them. The framework provides plenty of hooks upon which end users may attach their own interfaces and data acquisition systems. It is flexible and can be as complex or simple as needed.

In short, the goals for the project were met successfully. An expandable framework for the simulation of plants has been created. The CSPS is provided at no cost to all as an open source system with the hope that end users customize it, make it their own, and share their alterations with all.

Bibliography

[1] Ardence, Release Notes RTX® version 6.0. Accessed 2006, <http://www.ardence.com/>

[2] Audemard, Fabrice, Condamin, Isabelle, Hazel, Terence. "Facilitating Plant Operation and Maintenance Using an Electrical Network monitoring and Control System Simulation Tool". In the Proceedings of the Petroleum and Chemical Industry Technical Conference, 2004, 51st Annual Conference. Sept. 2004.

[3] Banerjee, S. Dynamics of Physical Systems. Professor Department of Electrical Engineering, Indian Institute of Technology. Kharagpur India, accessed 2007 <http://www.ee.iitkgp.ernet.in/~soumitro/dops/>

[4] Cheever, Erik. Linear Physical Systems Analysis Lecture Notes. Professor and Chair Department of Engineering Swarthmore College. 2006, accessed 2006 <http://www.swarthmore.edu/NatSci/echeeve1/Class/e12/E12Schedule.html>

[5] Christensson, M., Eskilson, J., Forsgren, D., Halberg, G., Hogberg, J., Magnusson, P.S., Moestedt, A., Werner, B. "Simics: A full system simulation platform" in Computer. Volume 35, issue 2. pp 50-58 2002.

[6] Coelho, Claudionor, De Micheli, Giovanni, Gupta, Rajesh K. "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components". In Proceedings of the Design Automation Conference, pp. 225-230, 1992.

[7] Copstein, B., Mora, M. "An Environment for Formal Modeling and Simulation of control Systems." Proceedings of the 33rd Annual Simulation Symposium. 2000.

[8] Douglass, B. Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley Boston MA, 1999.

[9] Dorf, R., Bishop, R. Modern Control Systems Ninth Edition. Prentice Hall New Jersey, 2001.

[10] El-Hajj, A., Kabalan, K., Khoury, S. "A Linear Control System Simulation Toolbox Using Spreadsheets" Control Systems Magazine, IEEE. Volume 20, Issue 6, 2000.

[11] Flannery, Brian P., Press, William H., Teukolsky, Saul A., Vetterling, William T. Numerical Recipes in C: The Art of Scientific Computing, Second Edition. 1992, Cambridge University Press.

[12] Foss, B.A., Eikaas, T.I., Hovd, M., "Merging Physical Experiments Back Into The Learning Arena", Proceedings of the 2000 American Control Conference. 2000, Chicago IL.

[13] Franklin, G., Powell, J., Workman, M. Digital Control of Dynamic Systems. Addison Wesley Longman Inc. Menlo Park CA, 1998

[14] Franklin, G., Powell, J., Emami-Naeini, A. Feedback Control of Dynamic Systems – Third Edition. 1994 Addison-Wesley Publishing Company.

[15] Gomez, Martin. "Hardware-in-the-Loop Simulation." Embedded.com. 11/30/2001, accessed 2006 <http://www.embedded.com/story/OEG20011129S0054>

[16] Grega, Wojciech. "Hardware-in-the-loop Simulation and Its Application in Control Education". Department of Automatics, Univeristy of Mining and Mettaluargy Krakow Poland. From Frontiers in Education Conference, 1999. FIE '99. 29th Annual. Nov 1999.

[17] Hartson, Rex H. "Digital Control Simulation System." Annual ACM IEEE Design Automation Conference, 6th annual conference on Design Automation. Pages 113-144. 1969. ACM Press, New York, NY.

[18] Heinrich, Mark, Ofelt, David, Olukotun, Kunle. "Digital System Simulation: Methodologies and Examples" in the Proceedings of the 35th annual conference on Design automation, pp 658-663. 1998

[19] Inoue, A., Kroumov, V., Shibayama, K. "Interactive Learning Tools for Enhancing the Education in Control Systems." Proceedings on the 33rd Annual Conference on Frontiers in Education, 2003.

[20] Jackson, L. Digital Filters and Signal Processing Second Edition. Kluwer Academic Publishers. 1989, Norwell MA.

[21] Juang, J-C., "Controller Rapid Prototyping and its Incorporation in Control Education", Proceedings of the 4th IFAC Symposium on Advances in Control Education July 14-16, 1997, Istanbul.

[22] Klee, H "Simulation and Design of a Digital Control System with TUTSIM" IEEE Transactions on Education, Volume 34, issue 1. 1991.

[23] Klee, H., Dumas, J. "Theory, Simulation, Experimentation: An Integrated Approach to Teaching Digital Control Systems." IEEE Transactions on Education, Feb 1994, Volume 37, Issue 1.

[24] Messner, William C., Tilbury, Dawn Control Tutorials For Matlab and Simulink. A Web Based Approach. 1998 Prentice Hall New Jersey.

[25] Messner, William C., Smoker, Jason, Tilbury, Dawn, Tseng, Ian Modeling Tutorials for MATLAB and Simulink. Accessed 2006. <http://www.me.cmu.edu/ctms/>

[26] Metzger, M., Moor, S.S., Piergiovanni, P.R. "Process Control Kits: a Hardware and Software Resource" in Proceedings of the 35th Annual Conference on Frontiers in Education, 2005.

[27] Murrer, Robert "Technologies for Synthetic Environments: Hardware-In-The-Loop Testing". September 2003, SPIE-International Society for Optical Engines

[28] Ogata, Katsuhiko. Modern Control Engineering Fourth Edition. 2002 Prentice Hall New Jersey

[29] Olukotun, K., Heinrich, M., Ofelt, D. "Digital System simulation: Methodologies and Examples", Proceedings of the 1998 Design Automation Conference. 15-19 June 1998.

[30] Oppenheim, A., Willsky, A., Nawam, S. Signals and Systems Second Edition. Prentice Hall New Jersey. 1997.

[31] Oppenheim, A., Schafer, R. Discrete-Time Signal Processing Second Edition. Prentice Hall New Jersey1999.

[32] Roher, C., Perdikaris, G. "Digital Control System Simulation Using Personal Computers" 16th Annual Conference of IEEE Industrial Electronics Society, 1990 IECON

[33] Saco, Roberto, Pires, Eduardo, Godfrid, Carlos "Real Time Controlled Laboratory Plant for Control Education" 32nd Annual Frontiers in Education. Nov. 2002.

[34] Sasaki N., Ohyama Y., Ikebe J., "Design Exercises for Robust Controller Using a Double Inverted Pendulum", Proceedings of the 4th IFAC Symposium on Advances in Control Education, July 14-16, 1997, Istanbul.

[35] Shaw A. Real-Time Systems and Software John Wiley and Sons, New York NY 2001

[36] Thakkar Nishant. Real-Time Control System Framework. Computer Science Capstone Thesis for RIT. Version 4.1, 6/7/2006

[37] Thrasyvoulou, K., Tsakalis, K.k Spanias, A., "J-DSP-Control: A Control Systems Simulation Environment" Department of Electrical Engineering, Arizona State University, Tempe AZ.

[38] Verwer, Andy. "The Role of Process Simulation in the Control System Software Life Cycle". IEE Colloquia on the Application of IEC61131 in Industrial Control: Improve Your Bottom-Line Through High value Industrial Control Systems (Ref. No. 2000/081) 2000

[39] Zhen Li, Kyte, M., Johnson, B., "Hardware-in-the-loop real-time simulation interface software design" Proceedings on the 7th international IEEE conference on Intelligent Transportation Systems", Oct 2004.

Appendix A: Directory Structure

The following appendix consists of listings of all of the contents of each directory in the CSPS package.

The Legend for directory listings is as follows

Folder
`File`

Root Directory

The root directory is the top level directory in the CSPS package. It is organized as follows:

CSPS

- **CODE**
- **DOCUMENTATION**
  - `Control System Plant Simulator Thesis.pdf`
  - `Users Guide.pdf`
  - `Directory Structure.pdf`
- **RTX_PKG**
  - `CSPS_RTX_CompKernel.rtss`
  - `CSPS_W32_Kernel_RTX_PORT.exe`
  - `ExampleUI.exe`
  - `SimInterface.exe`
  - `SimpleUI.exe`
  - `UiWinInterface.dll`
- **WIN_PKG**
  - `CSPS_W32_CompKernel.exe`
  - `CSPS_W32_Kernel.exe`
  - `ExampleUI.exe`
  - `SimpleUI.exe`
  - `UiWinInterface.dll`

The CODE directory contains all of the source code used to create the CSPS, and will be described fully in the sections that follow.

The DOCUMENTATION directory contains the documentation associated with the CSPS. The user's guide, and original thesis are provided here.

The RTX_PKG directory contains the compiled executables and libraries needed to launch the RTX version of the CSPS. This version has been built only for use with the simulated interface, but future data acquisition devices may be ported to it.

The WIN_PKG directory contains the compiled executables and libraries needed to launch the Win32 version of the CSPS. This version has been built to work with Data Translation's DT-9812 data acquisition device, through future data acquisition devices may be ported to it. Note that NO aspect of this version of the CSPS runs in real time.

CODE Directory

The CODE directory contains all of the source code and projects needed to create the CSPS.

- **CODE**
  - **CSPS_RTX_CompKernel**
    - **Utils**
      - LocalDefinitions.h
      - StdAfx.h
    - CSPS_RTX_CompKernel.cpp
    - CSPS_RTX_CompKernel.dsp
    - CSPS_RTX_CompKernel.h
    - CSPS_RTX_CompKernel.plg
  - **CSPS_W32_CompKernel**
    - **Utils**
      - LocalDefinitions.h
    - CSPS_W32_CompKernel.cpp
    - CSPS_W32_CompKernel.dsp
    - CSPS_W32_CompKernel.plg
    - StdAfx.cpp
    - StdAfx.h
  - **CSPS_W32_Kernel**
    - **Utils**
      - LocalDefinitions.h
    - CSPS.ico
    - CSPS_W32_Kernel.cpp
    - CSPS_W32_Kernel.dsp
    - CSPS_W32_Kernel.plg
    - IconScript.aps
    - IconScript.rc
    - resource.h
    - StdAfx.cpp
    - StdAfx.h
  - **CSPS_W32_Kernel_RTX_PORT**

    - **Utils**
        - LocalDefinitions.h
        - StdAfx.h
    - CSPS.ico
    - CSPS_W32_Kernel_RTX_PORT.cpp
    - CSPS_W32_Kernel_RTX_PORT.dsp
    - CSPS_W32_Kernel_RTX_PORT.h
    - CSPS_W32_Kernel_RTX_PORT.plg
    - IconScript.rc
    - resource.h
    - StdAfx.h
- **Shared**
- **SimInterface**
    - **Utils**
        - LocalDefinitions.h
    - InputScriptRunner.cpp
    - InputScriptRunner.h
    - OutputPrinter.cpp
    - OutputPrinter.h
    - PhysicalPort.h
    - SimInterface.cpp
    - SimInterface.dsp
    - SimInterface.h
    - SimInterface.plg
    - SimPortInterface.cpp
    - SimPortInterface.h
    - StdAfx.cpp
    - StdAfx.h
- **SimpleUI**
    - **Utils**
        - LocalDefinitions.h
        - UiWinInterface.h
    - SimpleUI.cpp
    - SimpleUI.dsp
    - SimpleUI.plg
    - StdAfx.cpp
    - StdAfx.h
- **UiWinInterface**
    - **Utils**
        - LocalDefinitions.h
    - StdAfx.cpp
    - StdAfx.h
    - UiWinInterface.cpp

    - UiWInInterface.def
    - UiWinInterface.dsp
    - UiWinInterface.h
    - UiWinInterface.plg
- **VB**
    - CSPS_Main.frm
    - ExampleUI.vbp
    - ExampleUI.vbw
    - GlobalModule.bas
    - InitialCondForm.frm
    - IOUpdateForm.frm
    - NonLinearForm.frm
    - NumDenomForm.frm
    - NumDenomMatrix.frm
    - PseudoPortForm.frm
    - SchedulePortForm.frm
    - SetDMethodForm.frm
    - StateSpaceForm.frm
    - ZPKForm.frm
    - ZpkMatrixForm.frm

The CSPS_RTX_CompKernel directory contains all of the code needed to create the RTX port of the Computational Kernel. Note that this project accesses a large number of code files stored in the shared directory.

The CSPS_W32_CompKernel directory contains all of the code needed to create the Win32 port of the Computational Kernel. Note that this project accesses a large number of code files stored in the shared directory.

The CSPS_W32_Kernel directory contains all of the code needed to create the Win32 port of the system Kernel. Note that this project accesses a large number of code files stored in the shared directory.

The CSPS_W32_Kernel_RTX_PORT directory contains all of the code needed to create the RTX port of the system Kernel. Note that this project accesses a large number of code files stored in the shared directory.

The Shared directory contains all code used by more than one project. There is a significant number of files stored here, and it will be discussed in a future section.

The SimInterface directory contains all of the code needed to create the simulated interface to be used with the RTX port of the Control System Plant Simulator.

The SimpleUI directory contains all of the code needed to build the example console user interface.

The UiWinInterface directory contains all of the code needed to build the UiWinInterface.dll library that may be used to simplify the user interface development process.

The VB directory contains a visual basic project that provides an example of a Visual Basic GUI that uses the UiWInInterface library to communicate with the CSPS.

Shared Directory Map

The Shared directory contains all code that is used by more than one process space. The vast majority of the CSPS code is stored here.

- **Shared**
  - **CompKernel**
  - **Descriptors**
  - **Interfaces**
  - **Kernel**
  - **OsAbstraction**

The shared directory contains five directories, each of which will be described in detail in the following sections.

CompKernel Directory Map

The CompKernel directory is a subdirectory of the Shared directory. It contains all of the code specific to the computational kernel. This code is shared between different ports of the Computational Kernel process.

- **CompKernel**
  - **Plant**
    - Plant.cpp
    - Plant.h
    - PlantWatchdog.cpp
    - PlantWatchdog.h
  - **Ports**
    - **DtPorts**
      - **Lib**
        - Oldaapi32.lib
        - Oldaapi.h
        - Oldacfg.h
        - Oldadefs.h
        - Oldsptch.h
        - Olerrors.h
        - OLMEM32.lib

        - Olmem.h
        - Oltypes.h
      - DTPhysicalPort.cpp
      - DTPhysicalPort.h
      - DTPhysicalPortCache.cpp
      - DTPhysicalPortCache.h
    - **SimPorts**
      - SimPhysicalPort.cpp
      - SimPhysicalPort.h
      - SimPhysicalPortCache.cpp
      - SimPhysicalPortCache.h
      - SimPortInterface.cpp
      - SimPortInterface.h
    - AnalogPseudoPort.cpp
    - AnalogPseudoPort.h
      BinaryPseudoPort.cpp
    - BinaryPseudoPort.h
    - PhysicalPort.cpp
    - PhysicalPort.h
    - PhysicalPortCache.cpp
    - PhysicalPortCache.h
    - PortManager.cpp
    - PortManager.h
    - PseudoPort.cpp
    - PseudoPort.h
- **Utils**
  - EngineeringValue.cpp
  - EngineeringValue.h
  - EngineeringValueMatrix.cpp
  - EngineeringValueMatrix.h
  - PortValuePair.h

The Plant directory contains the code related to the plant object in the Computational Kernel process. This includes both the plant and the plant watchdog.

The Ports directory provides all code related to ports in the system. Physical ports, including specific implementations of the physical ports such as the DT Ports and the Simulated Interface Ports, pseudo ports, and port manager code is all stored here.

The Utils directory contains the utilities used by the Computational Kernel processes. This includes all engineering value related code, as well as the PortValuePair descriptor.

Descriptors Directory Map

Descriptors are header files that contain the structures that get passed through the CSPS. The Descriptors directory in the Shared folder contains the major descriptors for the system.

- **Descriptors**
  - PhysicalPortDescriptor.h
  - PseudoPortDescriptor.h
  - StateSpaceDescriptor.h
  - TFDescriptor.h

Interfaces Directory Map

The Interfaces Directory in the Shared folder contains all interface related code for every interface in the system.

- **Interfaces**
  - **CompKernelInterfaces**
    - CompToWinOutgoingInterface.cpp
    - CompToWinOutgoingInterface.h
    - WinToCompIncomingInterface.cpp
    - WinToCompIncomingInterface.h
  - **DataTypes**
    - CompToWin.cpp
    - CompToWin.h
    - ConfigParams.h
    - UiToWin.cpp
    - UiToWin.h
    - WinToComp.cpp
    - WinToComp.h
    - WinToUi.cpp
    - WinToUi.h
  - **KernelInterfaces**
    - CompToWinIncomingInterface.cpp
    - CompToWinIncomingInterface.h
    - UiToWinIncomingInterface.cpp
    - UiToWinIncomingInterface.h
    - WinToCompOutgoingInterface.cpp
    - WinToCompOutgoingInterface.h
    - WinToUiOutgoingInterface.cpp
    - WinToUiOutgoingInterface.h
  - **UiInterfaces**

        - UiToWinOutgoingInterface.cpp
        - UiToWinOutgoingInterface.h
        - WinToUiIncomingInterface.cpp
        - WinToUiIncomingInterface.h
      - IncomingInterface.cpp
      - IncomingInterface.h
      - OutgoingInterface.cpp
      - OutgoingInterface.h

The CompKernel directory contains the code necessary to establish the Computational Kernel's side of the Win32 Kernel-Computational Kernel interface.

The DataTypes directory contains the code that defines all of the individual data types passed between interfaces. Interface semaphore names, Shared memory structures, and interface structures are defined in these files.

The KernelInterfaces directory contains the code necessary to establish the Win32 Kernel's side of the Win32 Kernel-Computational Kernel interface, and the Win32 Kernel's side of the Win32 Kernel-User Interface Process interface.

The UiInterfaces directory contains the code necessary to establish the User Interface's side of the Win32 Kernel-User Interface Process interface.

Kernel Directory Map

The Kernel Directory in the Shared folder contains all code used by the Win32 Kernel processes.

- **Kernel**
  - **Managers**
    - FileManager.cpp
    - FileManager.h
    - LogManager.cpp
    - LogManager.h
    - PlantConfigurationManager.cpp
    - PlantConfigurationManager.h
    - PlantFileManager.cpp
    - PlantFIleManager.h
    - PortConfigurationManager.cpp
    - PortConfigurationManager.h
    - PseudoPortFileManager.cpp
    - PseudoPortFileManager.h
    - SystemManagement.cpp
    - SystemManagement.h

- **Utils**
  - Complex.cpp
  - Complex.h

The Managers subfolder contains all of the manager code that makes up the bulk of the Win32 Kernel process.

The Utils subfolder contains the Complex structure, used for complex arithmetic.

OS Abstraction Directory Map

The OsAbstraction directory contains all of the Operating system abstraction files. Should the CSPS be ported to a different platform, these files would have to be altered.

- **OsAbstraction**
  - FileInterface.cpp
  - FileInterface.h
  - InterfaceSemaphore.cpp
  - InterfaceSemaphore.h
  - SharedMemoryInterface.cpp
  - SharedMemoryInterface.h
  - Thread.cpp
  - Thread.h

Complete Directory Map

The following listing provides every directory in the package. Note that files have been hidden in this listing.

CSPS
- **CODE**
  - **CSPS_RTX_CompKernel**
    - **Utils**
  - **CSPS_W32_CompKernel**
    - **Utils**
  - **CSPS_W32_Kernel**
    - **Utils**
  - **CSPS_W32_Kernel_RTX_PORT**
    - **Utils**
  - **Shared**
    - **CompKernel**
      - **Plant**
      - **Ports**

        - **DtPorts**
            - **lib**
        - **SimPorts**
    - **Utils**
- **Descriptors**
- **Interfaces**
    - **CompKernelInterfaces**
    - **DataTypes**
    - **KernelInterfaces**
    - **UiInterfaces**
- **Kernel**
    - **Managers**
    - **Utils**
- **OsAbstraction**
- **PhysicalPorts**

- **SimInterface**
    - **Utils**
- **SimpleUI**
    - **Utils**
- **UiWinInterface**
    - **Utils**
- **VB**

- **DOCUMENTATION**
- **RTX_PKG**
- **WIN_PKG**

Appendix B: User Manual