Poseidon Executor 2008
Technical Specification

Michigan State University

In collaboration with

The Boeing Company

Boeing Representative
Jayson Vincent

Michigan State University Capstone Students
Steve Emelander: emeland4@msu.edu
Thomas Stark: starkth1@msu.edu
Nick Thrower: throwern@msu.edu
Scott Walenty: walentys@msu.edu

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

The Phantom Works division of the Boeing Company in Saint Louis, MO is requesting assistance from the talented students of Michigan State University to develop the Poseidon Executor 2008. This application will feature a high-tech Instructor Operator Station (IOS) that will be used to manage the application, along with an impressive visualization of a Poseidon P-8A submarine-hunting aircraft that will also feature the capability to toggle between automated and operator-controlled flight modes.

There are two different parts to the Poseidon Executor 2008: the Poseidon Aircraft CIGI Integration (PACI) and the IOS. The IOS and PACI interact with each other to provide a realistic and editable flight test within a 3D world.

2 PROJECT OVERVIEW

The IOS and PACI were developed as two separate segments, each having their own unique features and requirements. The IOS assumes control of everything that is needed to run the simulation including executing and killing processes. The PACI uses multiple components to control and display a flight simulation. These components include a Host Emulator (HE) that manages and configures the simulation, and a Multi-Purpose Viewer (MPV) that displays the Poseidon's flight.

2.1 Instructor Operator Station (IOS)

The IOS runs everything that is needed by PACI and also displays each process by using nested tabs. The IOS is capable of killing any process that is not desired, giving an end-to-end control over the entire process (launching and killing). Along with launching and killing processes, the IOS also has a Script Editor tool, which allows for easy setup of aircraft, waypoints, and weapons. The IOS also manages Shared Memory, the communication between the IOS and the PACI. The Shared Memory Manager allows configuration of the world through the script setup files created in the Script Editor, as well as limited control of the Poseidon P-8A.

2.2 Poseidon Aircraft CIGI Integration (PACI)

The Poseidon Aircraft CIGI Integration is based on the Common Image Generator Interface (CIGI). CIGI is a network protocol tool developed by Boeing. The PACI includes PACI Host Emulator (PHE) software to interact with the MPV via CIGI in order to display flight tests of the P-8A Poseidon aircraft. It has two different modes to control the flight. The first is File mode, where a script file contains the coordinates of the flight path, and the second is Manual Mode, where the user can control the aircraft manually via keyboard, joystick, or mouse. The user is able to transition between the two different modes in the IOS's Shared Memory Manager.

3 STATEMENT OF REQUIREMENTS

3.1 Instructor Operator Station (IOS)

- Launch and kill individual portions of the simulation for ease of user control.
- Redirect console output from any process into tabs to decrease clutter.
- Share data between IOS and PHE to give immediate updated flight details to the end-user application.
- Script Editor for PHE to give users the ability to quickly and easily edit flight scripts.

3.2 PACI Host Emulator (PHE)

- Simulate Poseidon P-8A flight with CIGI Host Emulator, which will give the user a visualized flight that can easily be understood.
- Control flight from keyboard, joystick, or script file to give the user different options to manipulate the aircraft based on preference.

3.3 Additional

- Coding and documentation standards implemented in order to give future versions an easy understanding of coding format.
- Simple all-in-one installer for users to easily install and run the Poseidon Executor 2008 program.

4 DESIGN AND IMPLEMENTATION

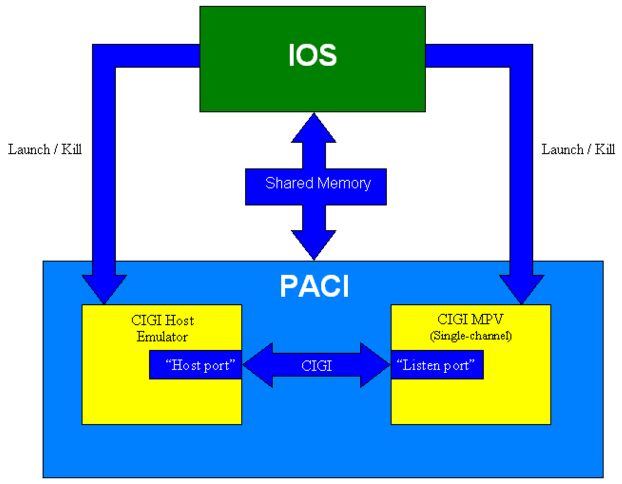


Figure 1

Figure 1 illustrates the overall architecture of the Poseidon Executor 2008. It demonstrates the ability of the IOS to launch and kill processes, such as the PHE and the MPV. It also shows communication between the IOS and PACI using shared memory, and communication between the PHE and the MPV using CIGI.

4.1 Instructor Operator Station (IOS)

The IOS is responsible for managing the simulation and being a front end for the end user. This includes handling the starting and ending of processes as well as communicating between the processes using shared memory. The IOS also has the capability of opening and creating script files.

4.1.1 Class Structure

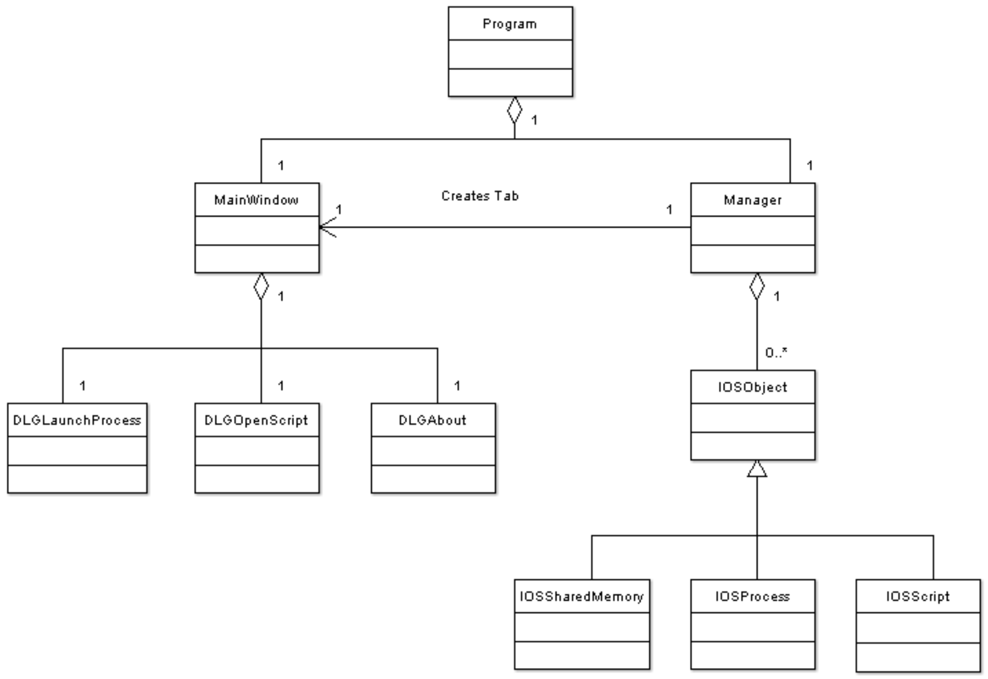


Figure 2

The overall structure of the IOS has been kept as simple as possible. Figure 2 shows the class hierarchy and for simplicity does not contain any variables or functions for the classes. Within the diagram any line with an arrow represents inheritance, the child pointing towards the parent class. Lines pointing with a diamond represent the storage of classes into another. The class touching the diamond contains the linked objects. If the line simply has an arrow on one end it represents a relationship; the direction of the arrow simply helps readability. The numbers on a line represent the quantity of objects each type may contain in the relationship. Each of the classes in Figure 2 will be described in more detail in the following sections with the exception of the dialog boxes (classes starting with DLG). Classes are represented in the diagrams by a box divided into three sections: the top section is the name of the class, the middle section contains member variables, and the bottom section shows the functions of the class.

4.1.1.1 Program Class

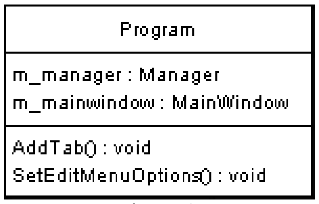


Figure 3

- ***Purpose***

  The Program class diagramed in Figure 3 is responsible for creating an instance of the MainWindow and Manager classes as well as providing a way of communicating with these classes.

- ***Variables***

  After the creation of the MainWindow and Manager classes the Program stores them for future access as m_mainwindow and m_manager. The m_manager variable is declared public so other classes within the project can access its functionality.

- ***Functions***

  The functions within the Program class are largely for accessing functionality within the m_mainwindow class without making it public. AddTab() is called by an instance of IOSObject in order to add their tab to the tabcontrol in the MainWindow class. SetEditMenuOptions() is called to set the current properties of the edit menu in order to disable and enable options.

4.1.1.2 Manager Class

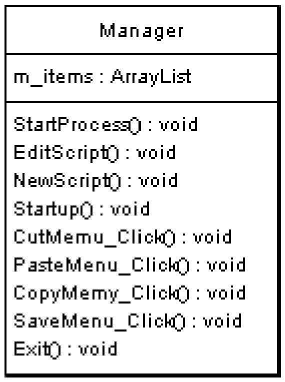


Figure 4

- ***Purpose***

  The Manager class diagramed in Figure 4 is used to store and access all instances of a process, script, and shared memory.

- ***Variables***

  The only variable in the Manager class is m_items, an arraylist containing all instances of the IOSObject class.

- ***Functions***

  Included in the manager class is the ability to start processes, edit existing scripts, and create new scripts. The Startup() function opens a startup script file when the application is launched and creates the appropriate objects. Many of the functions in the Manager class are related to options within the menu bar inside the MainWindow class. When invoked, these menu options call various functions within the IOSObject. When the IOS closes the Exit() function is called to ensure that everything launched by the IOS is properly closed as well.

4.1.1.3 MainWindow Class

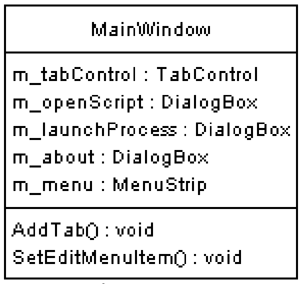


Figure 5

- ***Purpose***

  The MainWindow class diagramed in Figure 5 presents all the user interface options including all tabs created.

- ***Variables***

  The m_tabControl variable is the object where all tabs are dynamically inserted. There are several dialog boxes in the MainWindow which are displayed when certain menu items are clicked. These dialogs present the user with additional options for launching processes and opening scripts. The m_menu item is a menu strip that contains the file, edit, and help menus. There are several menu item objects that are contained within the MenuStrip that, for simplicity, are not shown in Figure 5.

- ***Functions***

  The two main functions within the MainWindow class are called to either add a tab to m_tabControl, enable, or disable an edit menu item. There are many event handlers in the MainWindow that call various functions within other classes but for the sake of simplicity they are not in Figure 5.

4.1.1.4 IOSObject Class

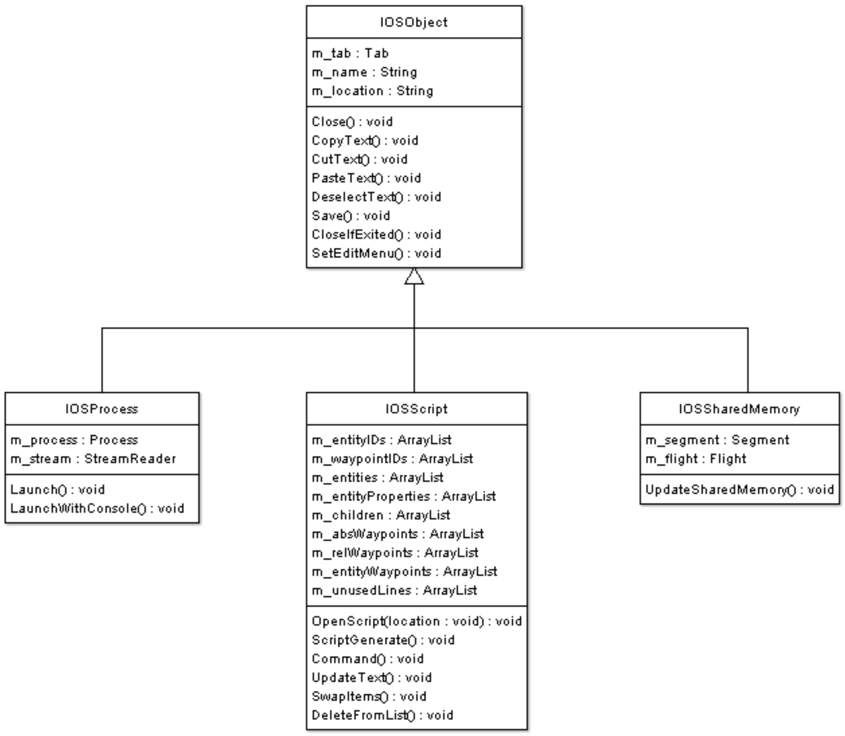


Figure 6

- ***Purpose***

  The IOSObject class is a parent class whose children have a tab within the main IOS window. This inheritance allows universal methods for communicating with these classes and the ability to store them as a single type within the Manager class. The diagram in Figure 6 shows the IOSObject class and all classes that inherit from it.

- ***Variables***

  There are three main variables that are stored in the IOSObject class. The m_tab variable is present in every child and allows the tab to be displayed within the IOS. The only case were this is not necessary is if a process is launched that is not a console application, in this case the tab is never initialized. There are also two strings to store the name of the tab (m_name) and the location (m_location) of the IOSObject. In the case of IOSSharedMemory the location is never set.

- ***Functions***

  The functions found in the IOSObject class are used to access different IOSObject's by the Manager class. Many of them include edit menu options for configuring cut/copy/paste functionality.

4.1.2 Process Management

The IOS has the ability to startup and end processes both during the launch of the IOS as well as individually while the IOS is in operation. These processes can be completely independent of the simulation itself.

4.1.2.1 Startup Processes

The processes required when the IOS is launched are stored in an XML file called startup.xml. This process list can be modified manually through the xml file. The name attribute is displayed in the tab, the location attribute is where the executable can be found, and the console attribute lets the IOS know if it needs to redirect console output into a tab. When adding to the startup.xml file it is important to keep the ordering of these attributes the same. An example content of startup.xml is shown in Figure 7.

```
<?xml version="1.0" ?>
<Startup>
        <Process name="Host" location="Hemu3.exe" console="false" />
        <Process name="MPV" location="mpv/mpv.exe" console="true" />
</Startup>
```

Figure 7

When the IOS is launched it opens startup.xml and starts each of the processes listed creating an instance of the IOSProcess class and placing it in the Manager class. In addition, if the process is a console application a tab is created within the MainWindow's TabControl. In addition to Processes, the startup script can also launch scripts and shared memory.

4.1.2.2 Runtime Management

The end user can also launch and kill processes on demand. To launch a process the user goes to the file menu and selects "Launch Process" the launch process dialog box is displayed as seen in Figure 8. As with the startup.xml file there are three attributes here. The process name will be the title of the tab if it is a console application, the process location is where the executable is located, and the checkbox allows you to mark the process as a console application or not. Once the Launch button is pressed then the process is launched and if necessary a tab is created.

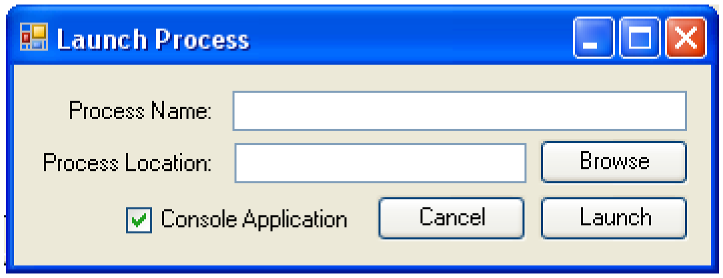


Figure 8

In addition you can also kill a process by activating the tab and going to the file menu and clicking close tab. If a window launched by the IOS is closed then the tab will also be removed. When the IOS is closed it automatically kills all processes that were launched at startup or during the simulation by the user.

4.1.2.3 IOSProcess Class

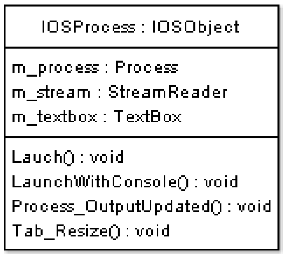


Figure 9

- ***Purpose***

  The IOSProcess class found in Figure 9 is a child of IOSObject. This class is responsible for managing processes within the IOS and displaying console output into a tab.

- ***Variables***

  The variables in the IOSProcess class include the process itself (m_process), the stream in which console output is read from (m_stream), and the textbox where that output is displayed (m_textbox).

- ***Functions***

  The Launch() function is responsible for launching any non-console application and the LaunchWithConsole() function initializes the tab and is responsible for setting up the redirection of console output. That console output is sent to the tab whenever the Process_OutputUpdated() function is called which is an event handler. Tab_Resize() simply resizes the textbox whenever the tab size has changed.

4.1.3 Shared Memory

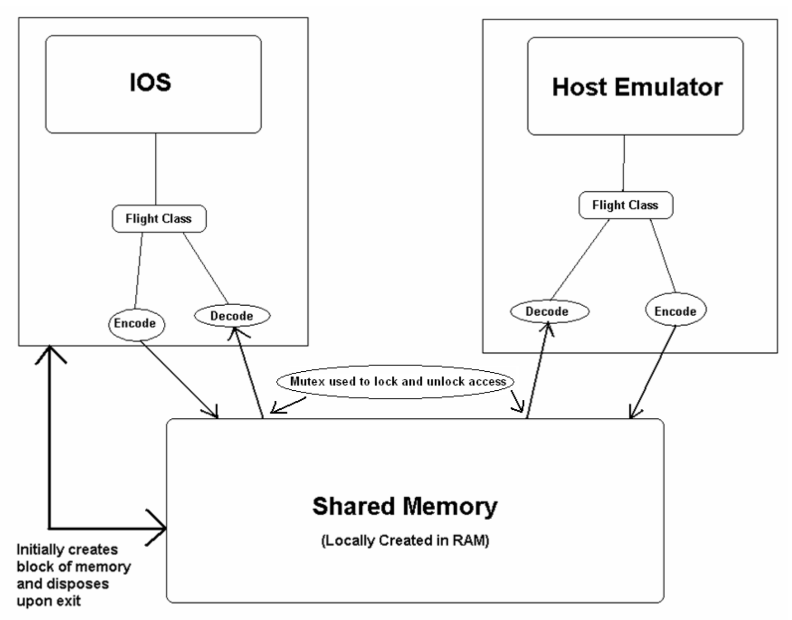


Figure 10

Error! Reference source not found. shows the functionality of shared memory between the IOS and the PHE. Although the shared memory appears to give constant control to both applications, an advanced system of mutex locks gives the data being shared a greater stability in case of simultaneous access.

4.1.3.1 Functionality

Both the IOS and the PHE will be able to view and edit the shared memory variables created by the IOS. The main purpose behind shared memory is the ability to control a program through the IOS. The user is able to define, control, and view these variables within the IOS directly. The Poseidon Executor 2008 builds shared memory for the PHE, but the ability to use shared memory with other programs can easily be implemented.

4.1.3.2 Method

The IOS creates the shared memory with a segment class that defines a block of memory to be specifically used within the PHE. This segment class has the capability to acquire and set a serializable class defined for each program. A serializable flight class for the PHE will store the properties of an aircraft during the simulation. Both the PHE and IOS have shared memory classes that use and control these segment and flight classes. The PHE constantly sets the shared memory variables with data from the simulation. The IOS continually retrieves data from shared memory and displays it to the user. The IOS gives users the ability to edit these variables. When edited, the PHE switches to automatic mode and updates the simulation with the new data.

The segment class creates a shared memory block upon startup of the IOS. It has a base class of “IDisposable” so that the unmanaged resources can be disposed of when deconstructed. Upon construction of the segment, a unique mutex is created for each program, in order to deal with concurrency. When a new segment class is instantiated, it either creates a file mapping with the Kernel32.dll windows file, or attaches to a previously created file map if shared memory has already been produced.

The mutexes are controlled through the functions lock() and unlock() to ensure complications won't arise from multiple accesses to the same block of memory.

A memory stream is inserted into the shared memory for ease of use. A binary formatter is used to format and serialize the flight class so that it may be put into the memory stream. The binary formatter is used again when receiving the memory stream from shared memory to deserialize the flight class into its original format.

4.1.3.3 Initial Conditions

The initial conditions can be set in the startup xml file also used to control which processes will be created upon launch of the IOS.

The format of the initial conditions file is shown on Figure 11.

```
<xml>
  <Sharedmemory entity="entitypath" flight="flightpath"
     weapons="weaponspath" flightmode="false" latitude="101"
     longitude="102" altitude="103" airspeed="104" />
</xml>
```

Figure 11

4.1.3.4 IOSSharedMemory Class

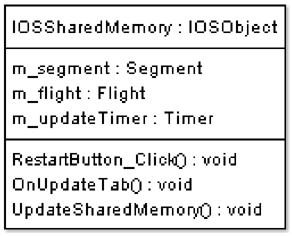

IOSSharedMemory : IOSObject
m_segment : Segment
m_flight : Flight
m_updateTimer : Timer
RestartButton_Click() : void
OnUpdateTab() : void
UpdateSharedMemory() : void

Figure 12

- ***Purpose***

The IOSSharedMemory class shown in Figure 12 is the sharedmemory tab created by the IOS to display variables in the flight class with the ability to be changed by the user.

- ***Variables***

The Segment and Flight class are instantiated within the IOS' shared memory class in order to communicate with the PHE, which has the same classes created on its end. The Timer is used to update the IOS' flight data from the current values in shared memory.

- ***Functions***

The RestartButton_Click() switches the flight mode to automatic, loads the new script files in (if needed), and restarts the simulation from the beginning. OnUpdateTab() continuously updates the flight data from shared memory with each duration of Timer. UpdateSharedMemory() sets the user-input data into shared memory where the PHE automatically sets the aircraft with the newly acquired data.

4.1.3.5 Segment Class

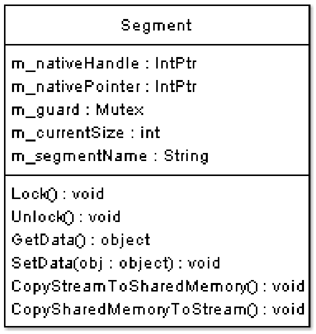


Figure 13

- ***Purpose***

The segment class shown in Figure 13 provides both the IOS and the PACI HE the resources for access to shared memory.

- ***Variables***

m_nativeHandle is used to check if an error is incurred while creating or accessing shared memory from the file map. m_nativePointer is used as a pointer to the file mapping so that access to the data can be given to other functions. The Mutex is created to give the segment a security feature to ensure the data will not be written to without permission. The currentSize and segmentName are the size of shared memory created and the name of the segment used to create a file map.

- ***Functions***

The Lock() and Unlock() functions are used to lock and unlock the mutex for privileges. The SetData() and GetData() are used to set the flight class in a binary formatted stream, and to revert the stream back into the readable flight class. The CopyStreamToSharedMemory() and CopySharedMemoryToStream() functions are used to get data from shared memory, convert it to a byte array, use a binary reader to translate the data, and visa versa.

4.1.3.6 Flight Class

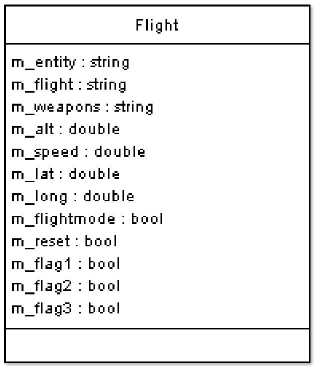


Figure 14

- ***Purpose***

  The flight class show in Figure 14 contains the information used by the IOS and the PACI HE to allow control of certain flight variables by both programs.

- ***Variables***

  m_entity, m_flight, and m_weapons are the paths to the script files needed to run the flight from the PHE. m_alt, m_speed, m_lat, and m_long are the data variables needed to display flight details to the IOS from the PHE and also to set user defined figures in the PHE. m_flightmode is the flag used by the PHE to determine whether to fly the aircraft from the flight script or from manual controls. m_reset is the flag to let the PHE know when it needs to restart the flight from scratch. m_flag1, m_flag2, and m_flag3 are extra flags available for miscellaneous purposes.

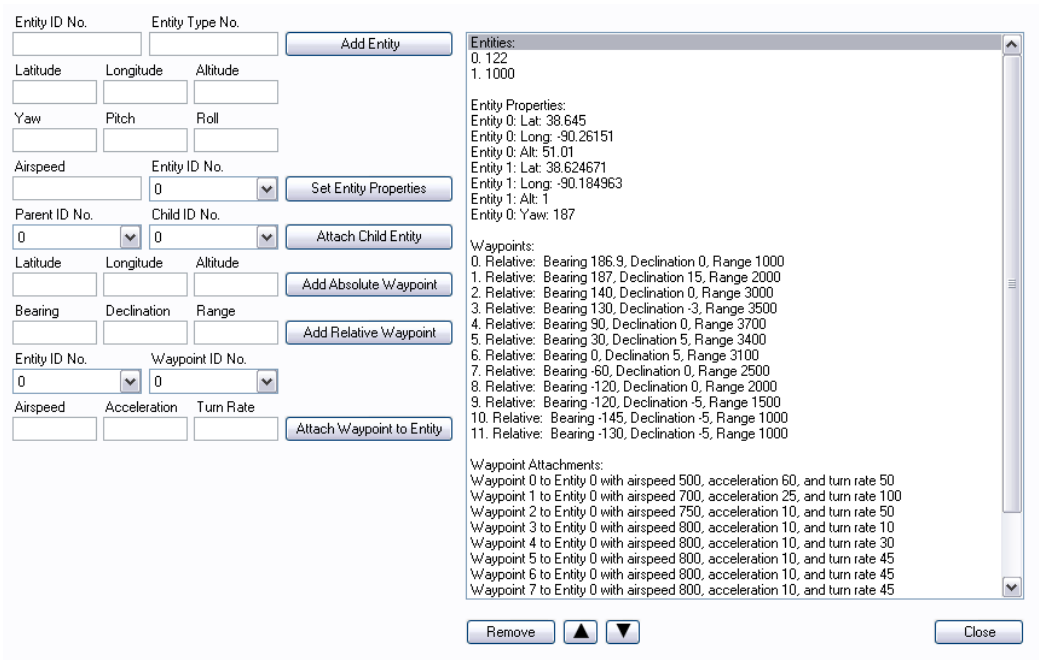
4.1.4 Script Editor

The IOS comes with a GUI-based Script Editor that makes writing and editing script files much easier.

4.1.4.1 Script File Format

The script file format used by the Host Emulator is a simple line-by-line format. Using a text editor to manage up to hundreds of these lines can be difficult.

4.1.4.2 GUI



```
Entities:
0. 122
1. 1000

Entity Properties:
Entity 0: Lat: 38.645
Entity 0: Long: -90.26151
Entity 0: Alt: 51.01
Entity 1: Lat: 38.624671
Entity 1: Long: -90.184963
Entity 1: Alt: 1
Entity 0: Yaw: 187

Waypoints:
0. Relative:  Bearing 186.9, Declination 0, Range 1000
1. Relative:  Bearing 187, Declination 15, Range 2000
2. Relative:  Bearing 140, Declination 0, Range 3000
3. Relative:  Bearing 130, Declination -3, Range 3500
4. Relative:  Bearing 90, Declination 0, Range 3700
5. Relative:  Bearing 30, Declination 5, Range 3400
6. Relative:  Bearing 0, Declination 5, Range 3100
7. Relative:  Bearing -60, Declination 0, Range 2500
8. Relative:  Bearing -120, Declination 0, Range 2000
9. Relative:  Bearing -120, Declination -5, Range 1500
10. Relative:  Bearing -145, Declination -5, Range 1000
11. Relative:  Bearing -130, Declination -5, Range 1000

Waypoint Attachments:
Waypoint 0 to Entity 0 with airspeed 500, acceleration 60, and turn rate 50
Waypoint 1 to Entity 0 with airspeed 700, acceleration 25, and turn rate 100
Waypoint 2 to Entity 0 with airspeed 750, acceleration 10, and turn rate 50
Waypoint 3 to Entity 0 with airspeed 800, acceleration 10, and turn rate 10
Waypoint 4 to Entity 0 with airspeed 800, acceleration 10, and turn rate 30
Waypoint 5 to Entity 0 with airspeed 800, acceleration 10, and turn rate 45
Waypoint 6 to Entity 0 with airspeed 800, acceleration 10, and turn rate 45
Waypoint 7 to Entity 0 with airspeed 800, acceleration 10, and turn rate 45
```

Figure 15

The Script Editor utilizes textboxes, drop-down menus, and a listbox to make management considerably easier. Figure 15 shows the GUI of the script editor.

- ***Textboxes***

  The textboxes each come with labels so the user may know what field is being modified, as opposed to memorizing the order in which the Host Emulator reads in the data.

- ***Drop-down Menus***

  The purpose of the drop-down menus is to ensure the user cannot enter a piece of data that has not been previously declared. In this case the applicable pieces of data are entities and waypoints (both referenced by ID number).

- ***Listbox***

  The listbox displays all data in the script, and updates if the user adds or removes anything. Adding can be done by any of the "Add" buttons to the left of the listbox, and removal can be done by selecting an item and clicking the "Remove" button below, or by pressing the "Delete" key. In areas where ordering of sequence matters, in this case only one, waypoint attachments to entities, it is possible to drag elements of the listbox up and down, changing their ordering. This may also be achieved by clicking the up and down arrow buttons.

4.1.4.3 Saving and Opening

The Script Editor opens a script file by parsing it and stores the data in order to display it in the listbox. Saving to an output script file is also possible, by taking all the data currently stored and generating commands for each item in the listbox. These options are both available in the IOS's "File" menu.

4.1.4.4 Commands

The Host Emulator supports hundreds of script commands for the varied functionality it may require. The Script Editor only has support for 15 commands, ADD_ENTITY, ENTITY_AIRSPEED, ENTITY_YPR, ENTITY_POS, ENTITY_YAW, ENTITY_PITCH, ENTITY_ROLL, ENTITY_LATITUDE, ENTITY_LONGITUDE, ENTITY_ALTITUDE, ENTITY_ATTACH, WAYPOINT_ADD_ABSOLUTE, WAYPOINT_ADD_RELATIVE, ENTITY_ADD_WAYPOINT, and ENTITY_FLY_WAYPOINTS. It also adds a "RUN" command at the end of every script file it saves. Of course, if the user wishes to have more than just these 15 commands, they can always add the commands in a text editor.

4.1.4.5 Error Checking

Much of the Script Editor's ease of use lies in its ability to error check. If anything other than the desired type of data is entered into a textbox, a message will pop up, informing the user to please enter that type of data (for example, a positive integer). Along with textbox error checking is the "Add" buttons' error checking. If at any time the user attempts to add something without each required field being filled in, the Script Editor will simply ignore it.

4.1.4.6 IOSScript Class

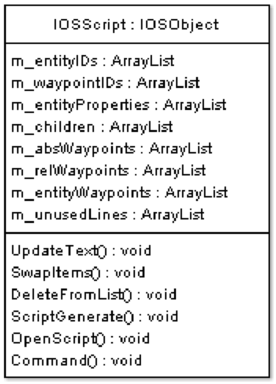


Figure 16

- ***Purpose***

  The purpose of the IOSScript class from Figure 16 is to display the Script Editor window, allowing for easier script file creation.

- ***Variables***

  Each piece of data is put into an ArrayList. There are two lists, one for the entity IDs and another for the waypoint IDs, which the drop-down menus are based on. There is also one list for each "Add" button. Whenever you click an "Add" button, one item gets pushed into the corresponding ArrayList, as well as into the entity or waypoint IDs list if adding an entity or waypoint.

- ***Swapping***

  The SwapItems() function gets called whenever the user either drags an item up or down, or clicks the up or down arrows, and also only if this is legal, i.e. where ordering matters, in this case only in waypoint attachments to entities. The swapping is a very basic swap of elements, structured as follows (pseudocode):

  ```
  temporary = element1
  element1 = element2
  element2 = temporary
  ```

- ***Deleting***

  The DeleteFromList() function gets called whenever the user clicks "Remove" or presses the "Delete" key, and again only if this is legal, i.e. if the currently selected field is an actual piece of data rather than a newline or a header. DeleteFromList() determines what index of the ArrayList to delete by keeping track of line numbers for each section in the listbox. It simply subtracts the current line number by the beginning line number of the section of data, and that is the index of the corresponding ArrayList.

- ***Textboxes***

  After the user is done entering text and then leaves a particular textbox, the FocusLeave() event handler is invoked. Within this event handler, the textbox utilizes C#'s Convert class to make the text into actual data. This is done within a try-catch statement. If this fails, a message box pops up to notify the user.

- ***Updating the Listbox***

  Every time a change is made internally in the data structures, the listbox gets notified to update those changes. The UpdateText() function first clears everything it has and then simply does a foreach loop over every ArrayList, and in these loops writes one line for each piece of data. It also adds headers and newlines to make the formatting nicer.

- ***Saving the File (to text)***

  The ScriptGenerate() function gets called whenever the user clicks "Save" in the "File" menu of the IOS. First, a save box will pop up, prompting the location and name of file. When it gets this information, a StreamWriter is created to output to this file. The algorithm works much like UpdateText(), looping through each ArrayList, but instead of outputting human-readable information to a listbox, it outputs script file format commands to a file.

- ***Opening a File (from text)***

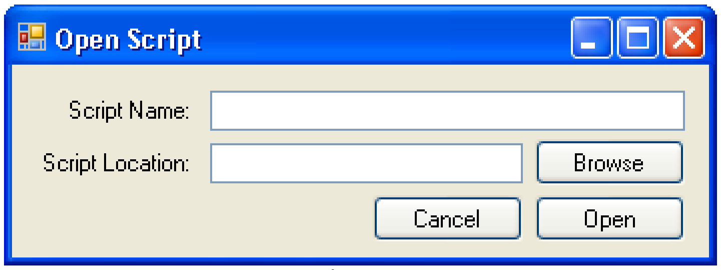


Figure 17

When the user clicks “Open Script” from the IOS’s “File” menu, the OpenScript() function is called. The Open Script dialog box shown in Figure 17 then opens. A StreamReader is then created to read from the file the user selects. If the file is invalid, a message pops up explaining this and the function will return. After the file has been read, OpenScript() loops through each line until the end of the file, splitting up lines of text into several different strings separated by whitespace delimiters. This is achieved using the C# library’s Split() function. Then it proceeds to call the Command() function. Command() takes the array of strings returned by the Split() function as a parameter. Command() invokes a switch statement on the very first string in the array, which should be the command name. It checks if this string matches any of the commands the Script Editor recognizes, and fills in the corresponding ArrayList. If at any time there is an erroneous line in the file, it simply ignores and continues, just as the Host Emulator itself does. If there are commands in the file that the Script Editor does not have defined, it stores them into the m_unusedLines ArrayList. This list is used later when saving the script; the undefined lines are written to the file, as they were before.

4.2 Poseidon Aircraft CIGI Integration (PACI)

The Poseidon Aircraft CIGI Integration portion of the software will include Host Emulator software to interact with the MPV to display flight tests of the P-8A Poseidon aircraft. It will have two modes with which to control the flight path. The first is File mode in which a script file will contain the coordinates of the path the aircraft will take. The other is an Operator Mode in which the user will control the aircraft manually, via keyboard, joystick, or mouse. The user will be able to seamlessly transition between the two different modes in the IOS’s Shared Memory Manager.

4.2.1 CIGI

CIGI, or the Common Image Generator Interface, is a network protocol interface that acts as a middleman between a Host and an IG. CIGI requires that both the Host and the IG implement the CIGI Class Library. This Library acts as the connection between each application allowing the Host and the IG to communicate with each other.

4.2.1.1 CIGI Features

CIGI is developed for use in high performance graphics environments. CIGI defines an extensive list of features that are common in both the Host and IG. Some of the basic features include the following.

- Entity
- Database
- Sky
- View
- Position

There are also more advanced features that allow for a more realistic simulation. The following list contains some of the more complicated features in CIGI.

- Collision Detection
- Trajectory
- Articulated Part
- Child Entity
- HAT/HOT mission control
- Weather Effects
- Line Of Sight (LOS)
- Motion Tracker
- Sensor

4.2.1.2 CIGI Packets

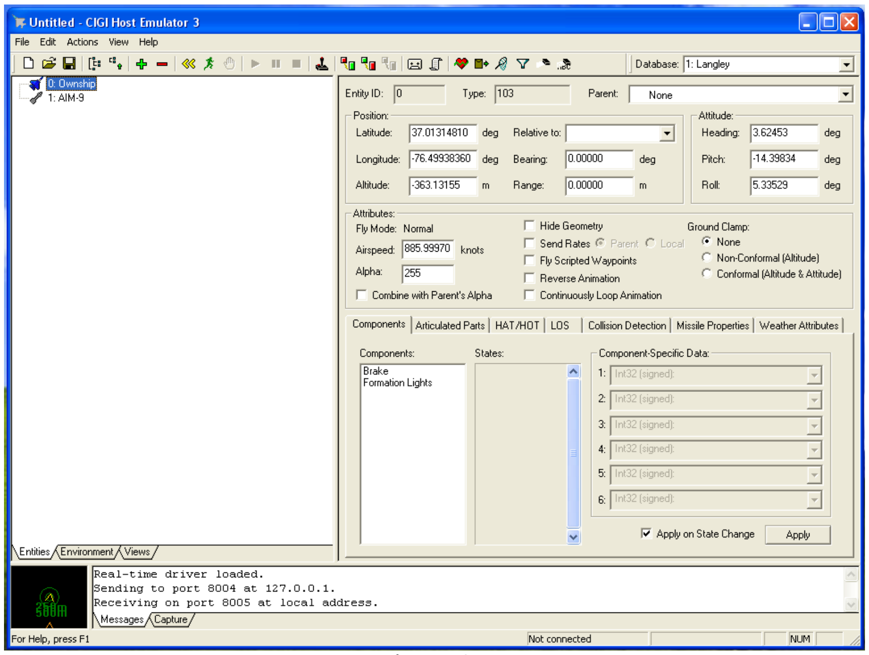
CIGI achieves the transfer of data from one application to another by using network packets, similar to UDP. Each feature of CIGI can be represented by a packet. Any new information can be sent from one application to the other by wrapping it in a packet and sending it through CIGI.

- ***Networking Capability***

  Since CIGI is a network protocol tool like UDP, networking over multiple machines is possible. One single Host machine can network to multiple Image Generators on several different machines.

4.2.2 Host Emulator (HE)

The Host Emulator or HE is a tool developed by Boeing to test CIGI. It is an emulator of a Host machine with the capability to use CIGI packets as a communication layer. The Host Emulator is divided into two processes, a main and a driver. The main window of the original Host Emulator is shown in Figure 18.



Database: 1: Langley

0: Ownship
1: AIM-9

Entity ID: 0 Type: 103 Parent: None

Position:
Latitude: 37.01314810 deg Relative to:
Longitude: -76.49938360 deg Bearing: 0.00000 deg
Altitude: -363.13155 m Range: 0.00000 m

Attitude:
Heading: 3.62453 deg
Pitch: -14.39834 deg
Roll: 5.33529 deg

Attributes:
Fly Mode: Normal
Airspeed: 885.99970 knots
Alpha: 255
☐ Combine with Parent's Alpha
☐ Hide Geometry
☐ Send Rates [✓ Parent] Local
☐ Fly Scripted Waypoints
☐ Reverse Animation
☐ Continuously Loop Animation
Ground Clamp: [✓ None] Non-Conformal (Altitude) Conformal (Altitude & Attitude)

Components:
Brake
Formation Lights
States:
Component-Specific Data:
1: Int32 (signed):
2: Int32 (signed):
3: Int32 (signed):
4: Int32 (signed):
5: Int32 (signed):
6: Int32 (signed):
☑ Apply on State Change

```
Real-time driver loaded.
Sending to port 8004 at 127.0.0.1.
Receiving on port 8005 at local address.
```

For Help, press F1 Not connected NUM

Figure 18

4.2.2.1 Main Process

The main process is a win32 process that handles the user interface and most of the file I/O.

- ***Graphical User Interface***

  The HE is written in C++ and uses the MFC library to display graphical user information.

- ***DEF Files***

  The HE utilizes text files (.def) in synchronization with the corresponding Image Generator's own DEF files to define entity, terrain, and view configurations.

- ***Scenario Files***

  The user may save and load scenario files (.sf3), which contain all information about a particular use of the HE. This includes views, terrains, and entities as well as their properties.

- ***Script Files***

  A simple line-by-line format of text file (.scp) is employed by the HE to enable control via external script file. The commands are all included in the Host Emulator's help file, and contain almost everything you can do in the HE itself. Exceptions include firing a weapon and changing target.

- ***Joystick Input***

  The HE maps joystick input to entity flight control. With the joystick the user can control the airspeed, the heading (yaw, pitch, roll), the firing of a weapon, and the changing of a target. The controls for the joystick are described in Figure 19.

CONTROL	ACTION
Forward	Pitches the nose of the entity down.
Back	Pitches the nose of the entity up.
Left	Rolls the entity to port.
Right	Rolls the entity to starboard.
Rudder Pedals or Z Axis	No action.
Point-of-View Hat Up	Increases the entity's airspeed.
Point-of-View Hat Down	Decreases the entity's airspeed.
Button 1	Fires a missile if one is available.
Button 2	Selects the next target for the missile.
Button 3	Freezes or resumes the simulation.
Button 4	Toggles the flight control mode.
Button 5	Changes the sensitivity of the joystick.

Figure 19

- ***Relative Waypoints***

  The Driver handles all motion calculations, and this includes entity waypoints, but it only understands absolute waypoints (latitude, longitude, altitude). If the user decides to supply relative waypoints (bearing, declination, range) rather than absolute waypoints, the main process of the HE calculates the absolute waypoint from the relative, and then feeds the Driver the absolute waypoint.

4.2.2.2 Driver

The main process will spawn an instance of the driver program. The driver handles motion calculations, network I/O, and file I/O (during recording and playback). It is built on RTX, a hard real-time extension to Windows by Ardence. However, this is translated to plain Win32 via a façade API.

4.2.3 Multi-Purpose Viewer (MPV)

MPV, an Image Generator (IG), is another tool developed by Boeing to interact with CIGI. An image generator produces images, visuals, or graphics for display. MPV is based on the Open Scene Graph, or OSG, and renders everything using the libraries within OSG.

4.2.3.1 Configuration

The MPV uses text-based files called "def" files. The syntax of these files is entirely compatible with the Host Emulator. So, using both MPV and HE makes for a convenient setup. The .def files are used to configure terrain, models, and the sky that is rendered within the MPV.

4.2.3.2 Single vs. Multiple Channels

MPV and CIGI support the concept of several rendering channels. The MPV can be configured to display multiple configurations with different unique entities and terrains. This setup requires the use of an MPV manager that manages each channel and delivers CIGI packets to the appropriate MPV display. However, there is no need for multiple rendering channels, as the Poseidon Executor is a one-machine or computer setup. So, this is a single-channel configuration, where the host communicates directly with a single instance of the MPV.

4.2.3.3 Visualization

The different kinds of visuals in the MPV are the terrain models, like landscapes, the entity models, such as the aircraft, and the skydome model. These models are also organized into the same categories within the PHE. The MPV can render models in any format supported by OSG, which include the OpenFlight or .flt, 3D Studio Max or .3ds, and AC3D or .ac.

- ***Terrain***

  OpenFlight is the leading 3D visual database standard for simulation in the world. It has features such as levels of detail (LOD), culling volume, switch nodes, drawing priority, and binary separating planes. The sample terrain included with the Poseidon Executor is stored in the OpenFlight format. It is a terrain developed by TrianGraphics freely available for non-commercial use. In order for the terrain to be included in commercial applications, a TrianGraphics company logo has been added to the MPV display in the lower left corner.

- ***Models***

  3D Studio Max is a commonly used 3D modeling suite. It has a vast array of modeling capabilities. This software can export in multiple formats, including 3ds. The 3ds format saves texturing as well as modeling data and allows for high quality models to be exported by 3D Studio Max. Most of the models displayed by the MPV are stored in this 3ds format.

- ***Skydome***

  AC3D is a 3D design program that has been available since 1994. The software is used by designers for modeling 3D graphics. Unlike other 3D software AC3D does not refer to polygons. AC3D is based on the concept of surfaces. A surface can be a polygon, polygon outline, or a line. An object is represented as a collection of surfaces. MPV uses the native AC3D format “ac” to load and render a skydome. MPV can render skydomes with transparency as well as multiple domes simultaneously.

- ***Special Effects***

  Special effects, such as missile trails and explosions, cannot be represented by simple solid models or polygons. MPV has a very powerful special effects system built in, and can be configured, like anything else, through .def files.

4.2.4 PACI Host Emulator (PHE)

The Host used by PACI is the PACI Host Emulator or the PHE. The PHE is a modified version of the HE, and includes all the common features of the HE as well as some new implementation for the PACI system.

4.2.4.1 Manual Control with Keyboard

The PHE has the extended functionality of being controlled by a keyboard. This includes the ability to control a plane in flight, set targets, and fire missiles. The new keyboard mappings are defined in Figure 20.

KEY	ACTION
Up Arrow	Pitches the nose of the entity down.
Down Arrow	Pitches the nose of the entity up.
Left Arrow	Rolls the entity to port.
Right Arrow	Rolls the entity to starboard.
Page Up	Increases the entity's airspeed.
Page Down	Decreases the entity's airspeed.
"H"	Holds current pitch and roll.
"F"	Fires a missile if one is available.
"T"	Selects the next target for the missile.
"F12"	Toggle keyboard control

Figure 20

4.2.4.2 Shared Memory with IOS

The PHE can communicate with the IOS using Shared Memory. The capability to create a connection between the IOS and PHE allows for bi-directional communication between the separate processes. The PHE uses this connection to update the IOS with the position and speed of the Poseidon. The PHE also depends on shared memory as a source of information regarding which scripts to load and when. The PHE can also update its own information when a change is initiated by the IOS. Figure 21 illustrates the internal structure of the Shared Memory class that is used within the PHE.

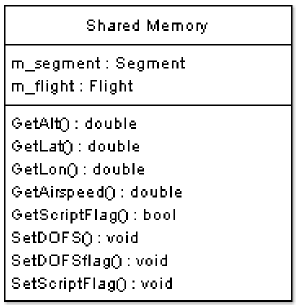


Figure 21

- ***Purpose***

  The Shared Memory class in Figure 21 is implemented within the PHE. This class allows the application to interact with data stored in shared memory. This interaction serves as a means of communication among processes that access the same memory.

- ***Variables***

  The shared memory class contains one reference to memory within the m_segment member. The m_segment variable is attached to an existing instance of shared memory that has been created by the IOS. The Flight class represents the structure of data that is stored in memory. This class has its values updated when shared memory is accessed, and is also used by the PHE to populate shared memory.

- ***Accessing Shared Memory***

  Flight data that is stored in shared memory is accessed using the functions GetAlt(), GetLat(), GetLon(), and GetAirspeed(). The PHE uses this information to update its own internal flight data when a manual update is requested.

- ***Setting Shared Memory***

  The PHE constantly updates the shared memory segment using the SetDOFS() function. It updates shared memory with the latitude, longitude, and altitude of the Poseidon. The PHE also uses SetSpeed() to update shared memory with the current airspeed of the plane.

- ***Communication***

  The PHE and IOS must communicate with each other using shared memory. The processes achieve this communication using flags. There are multiple flags used for each separate operation that must be fulfilled.

- ***Updating the PHE Flight Data***

  When the IOS manually updates flight information the SetDOFSflag is set to true. This will inform the PHE that a manual update has been requested. The PHE will then revise its own internal flight data using the values that have been stored in shared memory by the IOS.

- ***Restarting the PHE***

  The IOS can restart the simulation by setting the m_reset flag to true. When the PHE receives a true value for the m_reset flag, it will first clear itself of all entities and weapons. The PHE will then reload the same entities, weapons, and flight path, if m_flightmode is true.

- ***Automatic Configuration***

  The PHE is configured to automatically run scripts on startup. It will run the scripts that have been stored in shared memory by the IOS. These scripts include entities, weapons, and if enabled a flight script that sets waypoints. This automatic configuration allows a user to run the PHE without having to interact with it.

- ***Manual Flight vs. Flight from Script***

  The PHE has two different forms of flight, manual mode and file mode. When in manual mode, the PHE functions the same as the HE. When in file mode, the PHE will load and run a flight script stored in shared memory. This script will load waypoints, attach them to the Poseidon, and begin flying the plane through each waypoint. When the last waypoint is reached, the PHE will reset the scripts and start over from the beginning. While in file mode, a user may enter manual mode from the IOS and the PHE will immediately accept keyboard, mouse, or joystick input as it continues moving the plane with its current direction and speed.

4.2.4.3 C# wrapping

The PHE and IOS are written in C++ and C# respectively. In order to access shared memory from both applications a single language needed to be implemented within the IOS and PHE. The Shared Memory classes are written in C# and therefore needed to be wrapped by c++ inside of the PHE. This was accomplished by converting the PHE code to managed c++. Using managed code the PHE is able to instantiate reference classes defined in c#. These reference classes are then used as though they were standard c++ classes. Once wrapped properly the c# shared memory classes can be used within the PHE to access shared memory that has been created in another application written in another language.

4.2.4.4 Driver

The PHE driver controls flight paths as well as trajectories for launched weapons. The driver has been updated to more accurately represent the Poseidon’s weapons system. When fired, weapons now drop until level with their targets, and then home in on them.

5 CODING STANDARDS

This section describes the details of the standards that were used during the development of the Poseidon Executor 2008. These standards are a hybrid of several other well know coding standards. All code within the Instructor Operator Station (IOS) abides by these guidelines. Code that was added to the Host and MPV for the project, however, does not follow these standards and instead follows the standards previously defined by those projects' developers.

5.1 Internal Documentation

Every file within the project contains a file header and each function has its own function header.

5.1.1 File Header

The file header can be found at the top of every file. The first line of the header has the file name followed by two dashes and the project it associates with. The header also includes a description, a list of authors, and the date that it was created. Figure 22 is an example file header with proper spacing.

```
//=============================================================
// Filename:    FileName.cs  -- IOS
//
// Description: This is an example of a description that would
//              describe what is included inside of a file and
//              the functionality that it provides.
//
// Authors:     Team 2 Boeing
//                  Steve Emelander
//                  Tom Stark
//                  Nick Thrower
//                  Scott Walenty
//
// Date:        Spring 2008
//=============================================================
```

Figure 22

5.1.2 Function Header

Functions are documented using C# XML Documentation Comments. This allows intellisense within Visual Studio to provide useful information to the program about a function's use. The summary tab describes what the function does, the return tag describes the data that is returned, and the parameter tag describes one of the parameters that the function takes in. Each tag should contain one or more complete sentences. Figure 23 shows a simple function with an example function header.

```
/// <summary>
/// Simple example to show off a function header.  This will take an input
/// variable and double it.
/// </summary>
        /// <param name="input">The original input to be doubled.</param>
/// <returns>The value after the input has been doubled.</returns>
private int DoubleInput(int input)
{
                return input * 2;
}
```

Figure 23

5.1.3 Class Header

A class header is similar to a function header. It uses the C# XML Comments and puts a description of the class within the summary tab as demonstrated in Figure 24.

```
/// <summary>
/// This would describe a class and what it does.
/// </summary>
class ClassExample
{
}
```

Figure 24

5.1.4 Additional Comments

A programmer can add comments within the code to further clarify an aspect of his/her code. There is no specific standard for this sort of comment.

5.2 Naming Conventions

Classes, Functions, and Variables all have specific naming conventions to help make their functionality easier to understand and read.

5.2.1 Classes

Each class has a capital first letter for each word and should only contain letters a-z. If the class is a dialog box it should be prefixed with DLG. Similarly, if the class is a part of the IOS that can be launched within a tab it is prefixed with IOS.

5.2.2 Functions

Like classes, functions have a capital first letter for each word. For event handlers the function is named after the object where the event occurred followed by an "_" and the type of event that occurred. For example, MainWindow_Close() is a function that is called when the Main Window has been closed.

5.2.3 Variables

To identify the scope that a variable is accessible within, each variable is prefixed with "m_" for member variables and "g_" for global variables. If the variable is not prefixed then it is only accessible within function it is created in. Following the prefix, the first letter of the variable name is lower case, but each word after the first starts with a capital letter. This helps with readability by differentiating functions and classes from variables.

5.3 Organization

The code has been organized uniformly throughout the project so that future programmers can find and understand the code quickly.

5.3.1 Regions

All code in every class is organized into regions in order to encapsulate aspects of a class. Each region is defined by the pound symbol and the keyword "region" followed by the region's name. The region is then closed with another pound symbol and the keyword "endregion". Figure 25 is an example of a blank constructor inside of the Constructor region.

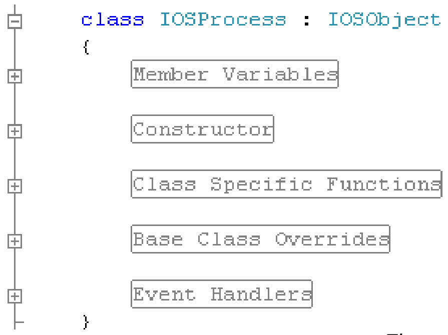
```
#region Constructor

public ClassName()
{
}

#endregion
```

Figure 25

Doing this allows you to click the +/- button on the #region line to hide a region away. Figure 26 shows the content of the IOSProcess class with all the regions minimized.



```
class IOSProcess : IOSObject
{
    Member Variables

    Constructor

    Class Specific Functions

    Base Class Overrides

    Event Handlers
}
```

Figure 26

5.3.2 Ordering

Regions are declared in order as follows:

- Member Variables
- Constructor – including functions only called by the constructor.
- Class Specific Functions – any function that is not an event handler, an override, or virtual.
- Base Class Overrides – any function that uses the override keyword to override the function declared in its parent class.
- Template Functions – any function that uses the virtual keyword to allow for overriding of its child classes.
- Event Handlers – and function that has an event assigned to it.

6 SCHEDULE

01/23-01/29 (Week 1)

- IOS - UI designed and implemented
- PACI - Manipulation of entities
- PACI - Loaded terrains

01/30-02/05 (Week 2)

- IOS – Run/kill processes
- IOS – Load process output into window
- PACI – Locate models/terrains/databases
- PACI – Create scenario

02/06-02/12 (Week 3)

- IOS – Load processes into tabs
- PACI – Automated scenario configured
- PACI – Manual flight mode

02/13-02/19 (Week 4)

- IOS – Load processes into tabs (to be completed)
- Alpha demonstration

02/20-02/26 (Week 5)

- IOS – Begin shared memory
- PACI – Flight from File

02/27-03/01 (Week 6)

- IOS – configure shared memory with C# and C++
- PACI - Flight mode switch

03/02-03/08 (Week 6-7)

- Michigan State University Spring Break

03/09-03/11 (Week 7)

- IOS – Flight class structure configured
- PACI – Missile creation/projection

03/12-03/18 (Week 8)

- IOS – IOS shared memory integrated
- IOS – Script Editor created

03/19-03/25 (Week 9)

- IOS – Script Editor implemented
- PACI – PACI shared memory implemented
- Beta demonstration

03/26-04/01(Week 10)

- IOS – Final startup file configured
- PACI – configure startup from shared memory
- Beta demonstration

04/02-04/15 (Weeks 11&12)

- Project Video script completed
- Project Video storyboard completed

04/16-04/22 (Week 13)

- Project Video voices addition and completion