Piloted Full-Motion Simulation with Simulink®

Emily K Lewis,[1]
Metis Technology Solutions, NASA Ames Research Center, Moffett Field, CA, 94035, U.S.A

A recent experiment at NASA Ames Research Center's Vertical Motion Simulator (VMS) successfully combined a real-time, human in-the-loop architecture with the flexibility of operating in the Simulink® graphical model-based engineering environment. The VMS is a large amplitude flight simulator designed to be adaptable to provide rapid integration and development of a wide variety of vehicles and support diverse aeronautical investigations. Math models are often programmed in Simulink. Typically, to run Simulink models in real time, they are converted to C code. However, the conversion and integration process can be time consuming and cumbersome. Thus, the VMS facility capabilities were expanded to allow a Simulink vehicle math model to run in the MathWorks' MATLAB environment during a piloted full-motion simulation experiment. The MATLAB Simulink based approach to driving the VMS was found to decrease development time by allowing quick integration of math model changes and providing the ability to run the same version of the model on researcher's desktop computers. This accomplishment demonstrated that the development ease of the graphical Simulink environment could be retained, while working within the real-time environment of the VMS architecture and maintaining the unique flexibility of the VMS.

Nomenclature

AFCS = Advanced Flight Control Systems
CAMAC = Computer Automated Measurement and Control
COTS = Commercial Off-the-Shelf
EFIS = Electronic Flight Instrument System
FCC = Flight Control Computer
HDD = Head Down Display
IDU = Integrated Display Units
IFR = Instrument Flight Rules
IG = Image Generator
I/O = Input/Output
MFD = Multi-Function Display
OTW = Out the Window
PCP = Pilot Control Panel
PFD = Primary Flight Display
RBP = Remote Bug Panel
SAS = Stability Augmentation System
SCRAMNet = Shared Common Random Access Memory Network
VMS = Vertical Motion Simulator
XIO = External Input/Output

I. Introduction

THE Vertical Motion Simulator (VMS) at the NASA Ames Research Center, shown in Figure 1, is a six-degree-of-freedom flight simulator designed to provide realistic motion cues for high fidelity piloted simulations. Housed in a ten-story tower, with a 60-foot vertical displacement and 40-foot lateral displacement, the VMS offers the largest

[1] Simulation Engineer Principal, SimLabs NASA Ames Research Center

vertical motion range [1, 2] of any simulator in the world. The large travels in lateral (or longitudinal) and vertical enables the VMS to provide a realistic cueing environment which results in pilot control techniques that are similar to actual flight [2]. This level of fidelity enables the VMS to deliver high quality research data that correlates well to the real world.

Over its 38-year history, the VMS has simulated a wide variety of aerospace vehicles and has supported a large number of research topics. Some areas of focus include handling qualities, guidance and display development, flight control design and evaluation, and simulation fidelity requirements. A large number of actual and conceptual aircraft have been studied, including various helicopters, Vertical/Short Take-Off and Landing and conventional aircraft, tilt-rotors, airships, spacecraft, high-speed supersonic transport, and the Space Shuttle.

Traditionally, vehicle math models simulated at the VMS were programmed by VMS engineers based on algorithms or block diagrams provided by the researchers. With advances in computing, vehicle modeling standards have evolved towards graphical development environments. Many in the aerospace simulation industry have adopted MathWorks products such as MATLAB® and Simulink® for model development. Researchers are now able to provide mature models which have been designed and tested in the Simulink environment. In these cases, integrating the Simulink models directly into the VMS environment can eliminate programming errors and reduce the simulation implementation and validation time by weeks for a typical experiment. The challenge faced by VMS engineers was to find a means of doing this efficiently and accurately while keeping within the requirements of the facility's architecture.

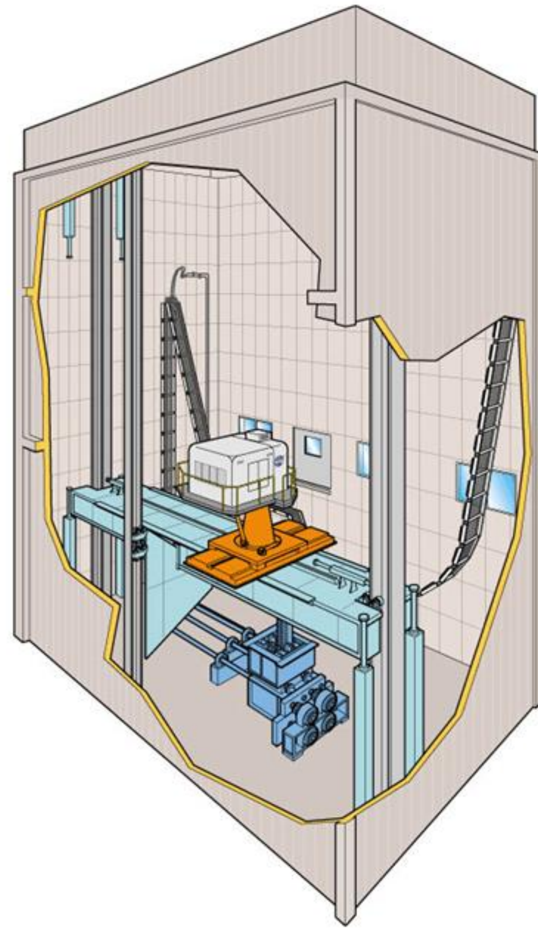

Fig. 1 Cut Away View of Vertical Motion Simulator

As MATLAB became more popular, the demand for Simulink models to be integrated with simulation facilities with an existing and mature architecture increased. For example, at the Naval Air Systems Command at Patuxent River, the legacy high-fidelity aircraft simulation environment, CASTLE (Controls Analysis and Simulation Test Loop Environment), was integrated with Simulink in order to perform flight control system development work [3]. In that case the real-time aircraft simulation set-up and execution control was transferred to the MATLAB workspace. The NASA Langley Research Center simulation engineers took a different approach for the SAREC-ASV (Simulink–Based Simulation Architecture for Evaluating Controls for Aerospace Vehicles) [4] and the B-737 Linear Autoland Model [5] efforts. To provide an efficient and portable desktop simulation capability of sufficient fidelity to effectively derive and evaluate aircraft control laws and control system components, they developed a method whereby the simulation modules were programmed in Simulink and MathWorks Embedded Coder was used to generate C code. While valuable for desktop development work where the model will be exported to multiple stations, and development work will be done on only system components, this scheme is not appropriate for a piloted simulation where all model parameters must be accessible and tunable. Several real-time and non-real-time applications in academia and industry use Simulink's target integration mechanism for off-the-shelf simulation platforms [6-8]. None of these approaches, however, were suitable for the VMS due to its stringent real-time requirements necessary for pilot-in-the-loop experiments and the necessity to simulate diverse vehicles. A more suitable approach for the VMS is the MOSAIC (Model-Oriented Software Automated Interface Converter) developed at The Netherlands' National Aerospace Laboratory (NLR), which automates the conversion from MATLAB's Real-Time Workshop® (RTW) code to a predefined Application Programming Interface [9]. MOSAIC was used successfully in several aerospace projects in Europe such as the European Space Agency's Automated Transfer Vehicle project [10]. Though this approach was suitable for the VMS, it would have required substantial modification to work within the VMS's real-time operating

environment, which includes sophisticated real-time debugging and development tools, which give it the flexibility needed to meet research requirements.

To support piloted experiments, the VMS requires simulations to run in "hard real-time", which means that the execution of all software computations must be completed within a specific frame time, according to the system's real-time clock. The real-time system will trigger a timeout and stop running if that specific frame time is overrun. Simulink does not run in real-time [11, 12]. Instead, a pseudo real-time is achieved by including a pacer block which will slow the computations down to make the simulation time near the desired frame time. MathWorks provides such a pacer block, and more precise versions can be programmed by the user. However, since these pacers are not synchronized to a hard-real-time clock, the resultant frame rate can vary. Moreover, if the execution takes longer than one frame time, the pacer block will not help and, in this case, the simulation will actually run slower than real-time [12, 13].

A method to integrate Simulink models at the VMS was developed and successfully implemented for a number of experiments over the past decades. To run Simulink models in real time for simulations with pilots or hardware in the loop, it is standard practice to convert it to C code using MATLAB's Real-Time Workshop, which is now called Simulink Coder. Typically, at the VMS, this auto-generated code is compiled into the host executable. In this way the VMS real-time environment can control the clock. Without this control, which guarantees hard real-time execution, the model may lose synchronization and consequently would not deliver the required fidelity. Many facilities that require real-time simulation use this approach [4, 12-14]. However, the conversion and integration process can be time consuming and cumbersome, especially during development when model modifications are frequently required. The generated code is difficult to debug and gaining access to parameters for tuning can be tedious. Moreover, some Simulink blocks are not supported by Simulink Coder.

To minimize model integration time and support a recent experiment whose Simulink model could not be used to generate C code, the VMS facility capabilities were expanded to allow a Simulink aircraft math model to run in the MathWorks' Simulink environment on an external device. This is the first time a vehicle model has executed in Simulink for a piloted real-time simulation at the VMS. This experiment also required flight hardware in the loop. Communication between the host computer, Simulink math model, and the flight hardware devices required special data-transmission methods and new interface software. The interconnections and the architecture design were unique to this VMS experiment, making the interface design challenging. Nevertheless, running the model in the MATLAB environment in this way worked well. The software developed and lessons learned during this simulation can be applied for future experiments.

This paper will describe the method by which this experiment was accomplished. An overview of the VMS will be provided first. Subsequent sections will discuss the development work required, and the challenges and lessons learned to integrate and run a Simulink model in the MATLAB environment for a real-time piloted hardware-in-the-loop simulation experiment at the VMS.

II. The Vertical Motion Simulator

The VMS is a large amplitude, uncoupled, six-degree of freedom, flight simulator that delivers unmatched vertical and lateral travel. The facility's architecture, both the hardware and software, was designed to be adaptable to support diverse aeronautical investigations. An in-house developed software environment provides a robust and flexible set of development, debugging and execution tools which supports rapid simulation development and enables efficient testing.

A. Hardware

A wide range of adaptable hardware components are available which can be put together in various ways to accurately represent most any vehicle, real or notional. Figure 2 shows the two-seat cab and controllers that were used for this experiment.

1. Computer Hardware

The VMS computer system architecture is comprised of a host computer, on which the aircraft math model executes, along with a number of other separate processors. The host computer, a 1GHz Hewlett Packard Alpha running Open Virtual Memory System (OpenVMS), provides the executive and real-time Input/Output (I/O) control functions, the interface for executing and debugging models, and control of the other processors (visual, motion, etc). The host and operating system are currently being upgraded to a Linux based system. Other computers include an Image Generators (IG) to produce the outside world visual scene, and graphics engines to create the cockpit flight instruments and lab engineering displays.

Fig. 2 Advanced Rotorcraft Two-Seat Cab, Flight Hardware and Controllers

2. Communication Devices

Communication between the host and attached computational processors is handled over a real-time network using Ethernet protocols. An in-house “raw Ethernet” protocol, referred to as External I/O, or XIO, is typically used for cab flight instruments, lab engineering displays and auxiliary processing. A CAMAC (Computer Automated Measurement and Control) real-time data acquisition system by Kinetic Systems is used to provide an interface for cockpit or lab analog and discrete controls. A SCRAMNet (Shared Common Random Access Memory Network) is used for communication with the Motion Control Unit and the pilot control loaders.

B. Software

Like the hardware, the VMS software has been designed to support a flexible and rapid development capability. This is achieved by way of a set of software components, the backbone of which is the in-house developed MicroTau real-time environment [15]. MicroTau provides effective debugging and monitoring tools for developing and testing simulation models. Furthermore, it provides the control and execution functions to enable executing simulation models in real-time piloted experiments. It controls the simulation I/O processors, performs the aircraft model calculations and supplies the user interface to the real-time simulation.

Another important component is the software library which contains various modules that perform model-independent functions such as the aircraft equations of motion calculations, driving the cockpit visual scene, controlling the motion system and recording electronic data. Typically, a simulation consists of the model-independent library routines as well as other modules specific to the vehicle, such as the aircraft aerodynamic math model or the control system. Vehicle-specific model software can be developed at the VMS or by the visiting researchers. Figure 3 below depicts these basic software components. Although the VMS library and legacy code is written in FORTRAN, all major languages and software environments are supported including C, FORTRAN, Ada and Simulink. A number of Simulink models have been used at the VMS over the last decade.

The Simulink models that have previously been integrated at the VMS range in size and complexity, from a single guidance module [16] to a complete vehicle model including functional gears, turbulence model and fully moveable nacelles [17]. In each of these cases, Real-Time Workshop, now called Simulink Coder, was used to generate code from the Simulink diagrams which was then integrated into the host computer software.

During one experiment, a prediction algorithm ran in the Simulink environment on an external processor. Since this Simulink component was only used to drive symbols on a display and was not critical for flight, there was no requirement to run it in real-time. For the current experiment, however, the vehicle and control system models, which are critical to piloted flight, had to be run in the Simulink environment. This requirement presented a challenge for the

engineers as well as a change to the existing integration methods and standards. The methods by which this was accomplished, the development work required, and the problems solved will be discussed after a description of the experiment.

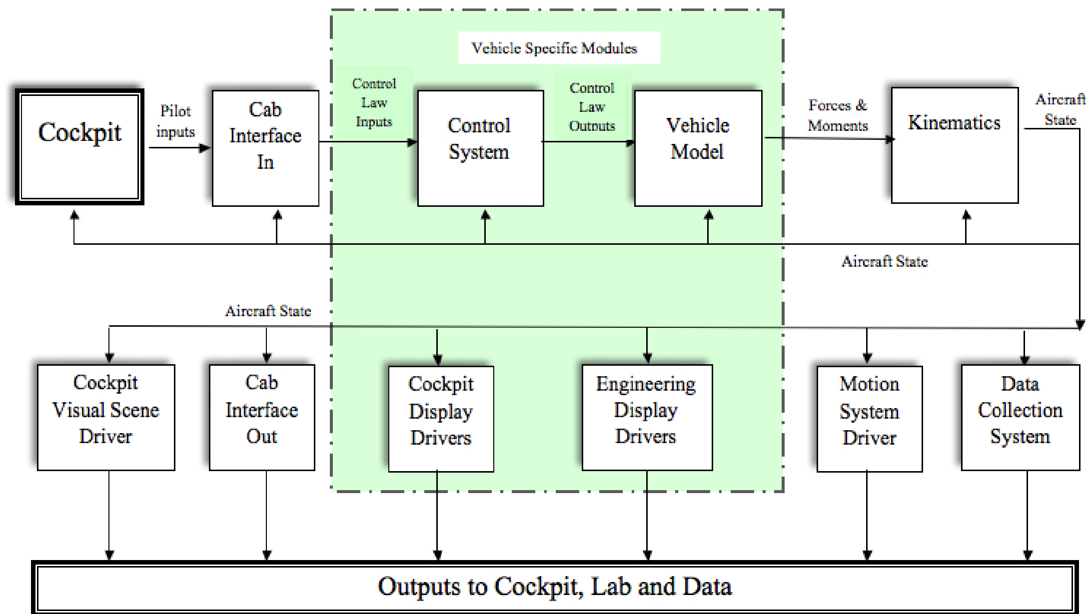

Fig. 3 Vertical Motion Simulator Software Structure

III. Experiment Description

The primary objective of this experiment was to evaluate Advanced Flight Control Systems (AFCS) for rotorcraft in a real-world, high-workload environment and evaluate pilot performance under realistic, full-mission scenarios during which the pilots' attention was often diverted away from controlling the helicopter. To satisfy this goal, it was necessary for the math model to produce a realistic helicopter response, as well as to provide the rapid modification of the AFCS's under investigation. For these reasons, the best option available to support this experiment was using a Simulink math model which included a mature helicopter simulation, as well as several new stability augmentation systems, auto-pilot, and flight director guidance. Likewise, another important component of this experiment was the use of advanced Electronic Flight Instrument System (EFIS) displays and realistic control and communication devices. Communication between the host computer, Simulink math model, and the flight hardware devices required special data-transmission methods as well as new interface software. The interconnections and the architecture design were unique to this VMS experiment, making the interface design challenging. A high-level diagram showing the hardware and software components is shown in Figure 4. Each component will be discussed, starting with the Cab Hardware, indicated by the green block on the right.

A. Experiment Hardware

Advanced flight deck EFIS displays, while informative and offering multiple functions, can be cumbersome to use and may increase the pilot workload at certain times, such as after a missed approach when they must be programmed to an alternate airport. They typically consist of multiple pages that supply a Primary Flight Display (PFD), Multi-Function Display (MFD) and Engine Indicating and Crew Alerting System (EICAS) display. The focus of this study was pilot workload during up-and-away flight in Instrument Flight Rules (IFR) conditions using realistic, full-mission scenarios during which the pilots' attention was often diverted away from controlling the helicopter. Consequently, an important component of the experiment was the use of advanced cockpit displays and realistic control and communication devices. To that end, actual flight hardware, including two Commercial Off-The-Shelf (COTS) EFIS Integrated Display Units (IDU), a Flight Control Computer (FCC), Pilot Control Panel (PCP) and Remote Bug Panel

(RBP), as well as an actual radio and transponder, were incorporated in the simulator cockpit and integrated into the simulation. One EFIS display was configured as an MFD and used for the left seat. The second EFIS display was configured as a combination PFD and MFD and used for the right seat. The EFIS MFDs included a real-world database which provided all required navigation and Integrated Moving Map functions to both seats. In addition to the COTS EFIS displays, four electronically drawn Head Down Displays (HDDs) were developed to present legacy flight instruments for the left seat, engine display and supplemental back-up dial instruments for the right seat. Lateral and Longitudinal Flight Director bars could be enabled/disabled on both the legacy and advanced displays.

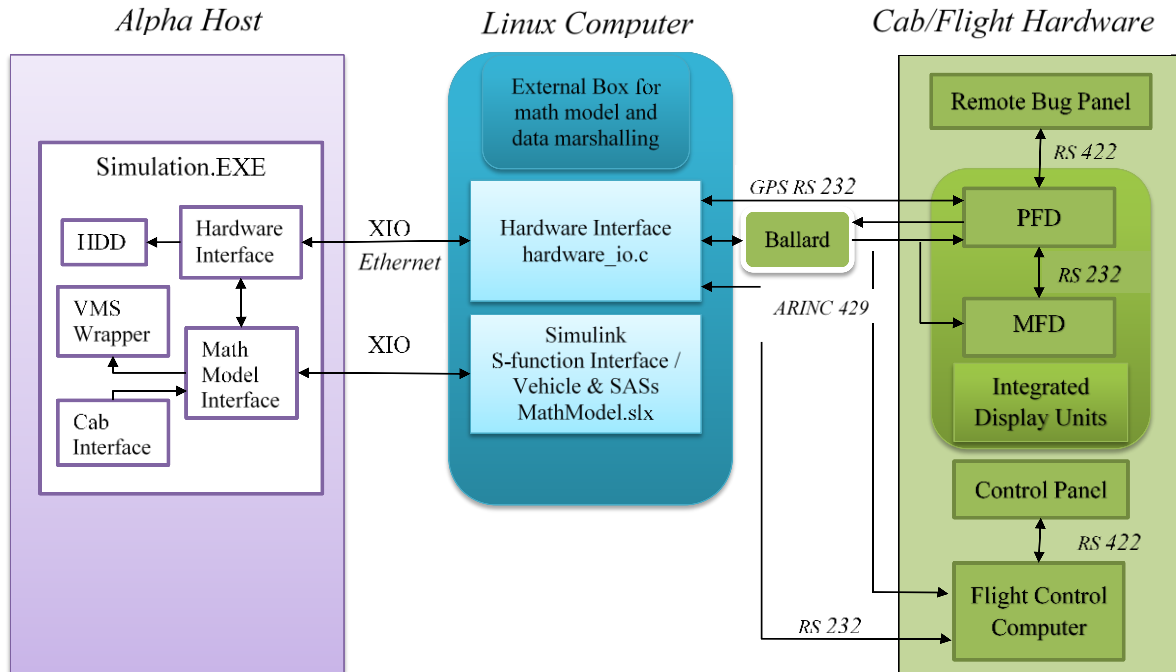

Fig. 4 Flight Hardware and Software High Level Schematic

A computer running the Linux operating system was used as an external processor to facilitate communication between the simulator host computer and the flight hardware devices, which are shown as the blue, lavender and green blocks respectively in Figure 4. New VMS data transmission methods were required and are indicated in Figure 4 as the connections between the green and blue blocks. New interface software was also required, shown as the interface blocks within the blue box.

1. Special Flight Hardware

This experiment's focus on realistic pilot workload resulted in the inclusion of a number of real-world avionics devices. These included the following:

- Remote Bug Panel (RBP) EIFS function controller,
- Pilot Control Panel for mode selection and annunciation,
- Flight Control Computer (FCC) which provided the SAS and Outer-loop Auto Pilot (AP) control mode functions,
- Bendix King KY196 Radio and Transponder.

The full cockpit front panel in shown in Figure 5.

Fig. 5 Cockpit Front Panel

2. Special Data Transmission Hardware

The communication scheme for this experiment was comprised of the current VMS standard systems--SCRAMNet, CAMAC and XIO--as well as some special data transmission methods, listed below. Although these are industry standard, they were new to the VMS. The variety of the communication methods and their novelty to the VMS made the total I/O system complex. The I/O scheme will be discussed later in the interface section. The new data transmission systems included:

- ARINC 429 (Aeronautical Radio, Inc.) an industry standard data transfer specification for aircraft avionics was used for the EFIS displays and FCC communication;
- RS-232 (Recommend Standard 232) a technical interface standard for serial data transmission between two devices was used for the FCC, GPS data and EFIS cross link;
- RS-422 (Recommend Standard 422) an improved data transmission protocol that uses differential transmission which enables longer cable lengths was used for the RBP, PCP and FCC communication.

The standard XIO scheme was the communication method used for the Linux computer on which the hardware I/O and Simulink math model ran, which will be discussed next.

3. External Linux Computer

An external processor, shown in the blue block labeled “Linux Computer” in Figure 4, was used to run the Simulink math model as well as the hardware interface data marshalling logic. During the experiment, this computer was secured in a rack at the back of the simulator cab. Initially, SCRAMNet was planned for the communication between the host and the Linux box. However, during development, it was decided that Ethernet XIO connection would be beneficial for two reasons. First, with an Ethernet interface a data package message is sent that would be synchronized to the start of frame. For this reason, Ethernet would better enable near real-time because it would take advantage of the hardware clock for communication synchronization. Second, software had already been developed for a previous experiment which could be leveraged. This solution entailed less time and reduced the possibility of errors. Nevertheless, some improvements to the XIO logic were required for the math model to work properly with the VMS infrastructure. These will be discussed in the Software section next.

B. Experiment Software

The experiment software set consisted of three main components, each of which had a unique binary, and all of which had to be executing for the simulation to run correctly. These were:

1. The Simulink math model, which ran on the Linux computer (the blue block in Figure 4) included the Vehicle Model, Control System and Interface modules.
2. The flight hardware interface program, which also ran on the Linux computer.
3. The primary executable (the lavender block in Figure 4) which ran on the host computer, included the VMS real-time operating system, as well as the graphics and motion drivers.

Each of these software components is discussed in the sections below.

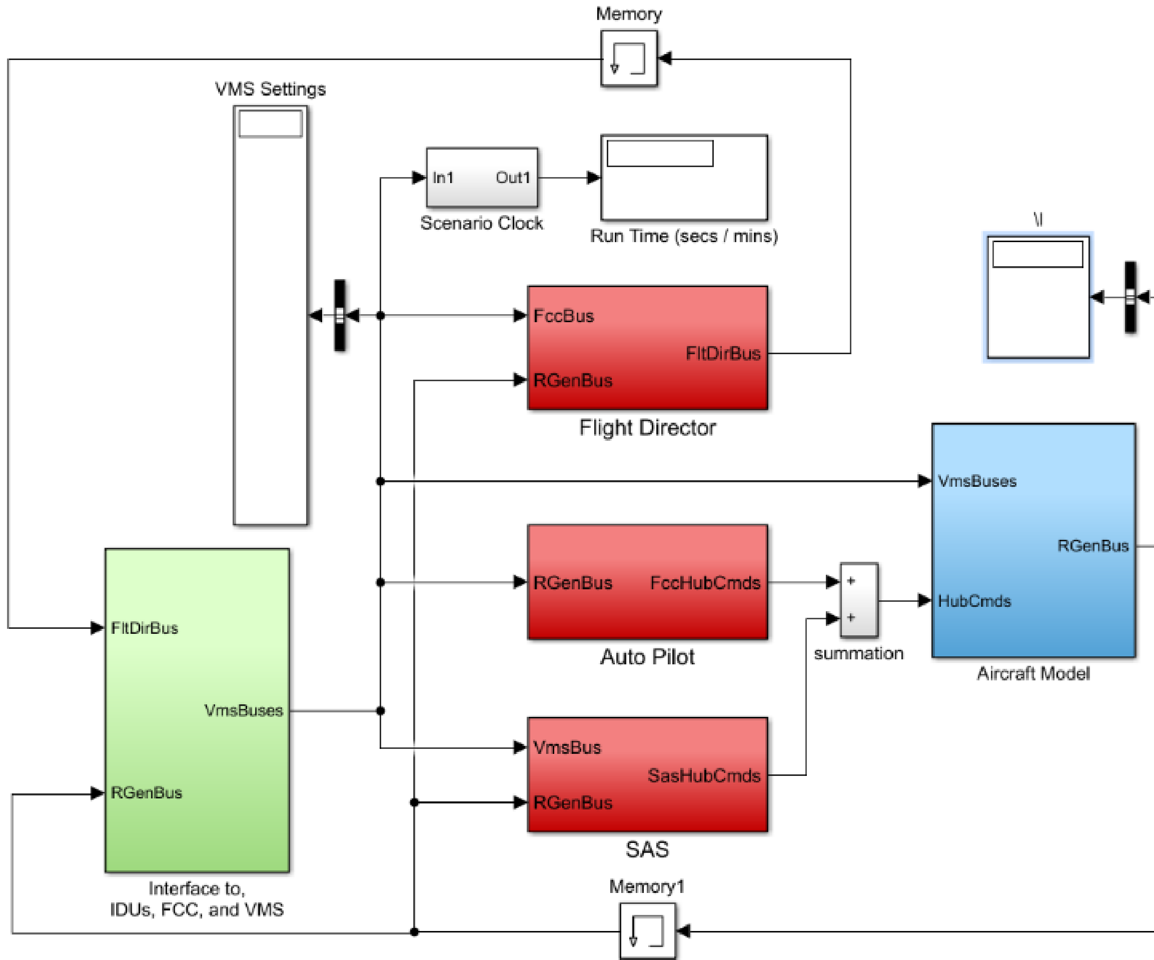

Fig. 6 Simulink Math Model Diagram

1. Simulink Math Model

The math model was provided by the researchers and delivered in Simulink. The Simulink math model, shown in Figure 6, included the aircraft model, Advanced Flight Control System (AFCS) components, and the VMS interface S-function. The S-function (system-function) is a mechanism by which Simulink blocks may reference non-MATLAB language computer code. When such code blocks are converted to S-functions, they can be referenced by Simulink during the link stage. While different s-function types are available, the Level-2 S-function supports the superset of all features and as such is typically the one used. However, to generate code from a Level-2 S-function, a Target Language Compiler (TLC) file for the S-function must first be written, which was out of the scope of this effort.

The aircraft model was a generic helicopter simulation developed to produce a realistic rotorcraft response while being computationally efficient and configurable to any specific helicopter by means of a set of variable parameters. It was written in C++ and compiled into a Simulink S function and is represented as the blue block on the right in Figure 6. The AFCS blocks incorporated a variety of Stability Augmentation Systems (SAS) options, auto-pilot and flight director guidance, and are shown as red blocks in the center of the diagram. Last, the Simulink/VMS interface

block is shown in green in the lower left. Since the delivered vehicle model was written in C++, C code could not be generated, which is the standard VMS integration method. Consequently, it was decided to run the Simulink math model in the MathWorks' MATLAB environment on an external Linux computer, making this the first time the vehicle model has run in Simulink for a piloted real-time simulation at the VMS. The method by which this was accomplished, and the development work required, are discussed in the sections below.

Despite initial concerns that the model may not run in real time or that synchronization may be lost, this method worked well and provided several advantages. Namely, model changes could be made to the Simulink math model on the researcher's computer and very quickly integrated into the VMS code base via the Linux computer. This also enabled the engineering team to copy the exact version of the math model into a desktop version of the simulation which was used for pilot training. This provided confidence in the training simulator integrity.

The Simulink math model communicated with the host via Ethernet XIO at 100 Hz. Due to reasons that will be discussed in the Challenges section below, the restriction that stops the simulation if a frame overrun occurs was disabled, and the hardware clock was utilized for I/O synchronization between the host computer and the Linux computer and monitored overruns when they occurred. The control inputs from the cab were processed on the host and, at the end of the frame, sent to the Linux computer. These signals were used by the Simulink math model to update the vehicle state and SAS commands during the subsequent frame, and then returned to the host where they were processed to drive the motion, visuals and displays. Because communication happens on fixed frame boundaries, this arrangement incurs a minimum of two frames of delay, or 20 millisecond (ms), between the cab outputs and the resultant motion and visual updates. As the primary focus of this study was workload during up-and-away flight, this delay was deemed acceptable by the researchers.

The initial version of the Simulink math model included numerous occurrences of, and calls to, Windows-specific code, which would not compile or run on the Linux computer. These all had to be replaced with Linux compatible software, some of which did not exist and had to be programmed from scratch. Many other details, such as the units and ranges of the inputs and how to set the initial conditions, had to be coordinated between the math model, Simulink interface and host interface to ensure accurate and appropriate communication.

One such change was the addition of memory blocks, which implement a one-time-step delay between the math model (vehicle and flight director) outputs and the VMS Interface S-function. These were needed to break algebraic loop errors. This change fixed the Simulink errors, but resulted in one additional frame delay, making the expected minimum delay incurred ~3 frames, or 30 ms.

Another change to the Simulink diagram was the addition of the block to perform the I/O between the VMS host and the vehicle and control system models. The Host-Simulink I/O Interface consisted of two components that were written in-house and ran on the Linux computer as shown in Figure 4: an interface module written in C (VMS_Host.c) and a configuration file (VMS_Host.cfg). The configuration file defined the host node name and the input and output port numbers and widths. Having multiple ports of differing sizes had not been done before at the VMS and required some research. The interface module defined the Simulink Level-2 S-Function that performed the I/O interface between the VMS host and the Simulink math model. It declared the function name, loaded the configuration file, and called the Simulink math model entry points, such as mdlStart and mdlUpdate. The S-Function C routine was compiled, along with the VMS XIO library, into a MATLAB (Linux 64-bit) binary MEX (MatLab EXecutable) file. A binary MEX file is a dynamically linked function comprised of a compiled C or FORTRAN subroutine that can be loaded and executed within MATLAB as if it were a built-in function. In this case, the interface C routine MEX file was linked to a level-2 S-function block in the Simulink math model. A Simulink S-function block was added into the delivered Simulink math model and connected to the MEX function by setting the S-function name to that declared in the MEX file. This S-function interface block is represented in Figure 6 as the green block in the lower left. The Simulink math model, including the Simulink I/O interface block, ran on the Linux computer, along with the hardware data marshalling executable. In order to run correctly, the Simulink math model required some input data from the flight hardware.

2. Linux Computer Flight Hardware Data Marshalling Software

Since some devices were flight hardware and not designed for a simulator, interfacing with them was challenging. The host sent data signals through XIO at one rate. The hardware interface had to decimate these data and package it for several different devices, each of which required data at different rates and formats, some at very high baud rates. For example, the ARINC 429, a specification for transferring digital data between avionics, was required for data transmission with the EFIS displays. To support this effort, a Ballard Technology's ARINC 429 Avionics Data Bus Interface box was obtained which did much of the communication scheduling. Nevertheless, a significant amount of new code was required to handle the data transmission and marshalling logic. This software was written in C and ran

on the Linux computer. It initiated the communication threads and then controlled the data exchange between the host and flight hardware.

Communication between the flight hardware data marshaling interface and host computer was accomplished using the standard XIO scheme, which was set to send and receive asynchronously in a manner similar to that for the Simulink math model interface. Figure 4 above shows a high-level diagram of the software and flight hardware I/O connections. The host computer, which runs the primary executable as well as the MicroTau real-time environment and I/O control program, is represented by the lavender block on the left in Figure 4, with some significant components of the primary executable presented as the white blocks.

3. Primary Executable

The primary executable that ran on the host computer was created for this experiment. It included library routines for standard actions such as driving the visual image generator and motion system, as well as simulation-specific source code including a Simulink math model interface, flight hardware interface and VMS wrapper.

A routine that interfaced the Simulink math model with the VMS host framework was written in FORTRAN and ran on the host computer in the primary executable binary. This module handled the data sent to the Simulink math model (e.g. cab inputs, environment, FCC outputs), and received from the Simulink math model (aircraft state, controller trim positions, flight director commands) via the XIO data transfer. Some important functions it performed included:

- math model mode mapping logic;
- simulation initialization by triggering MATLAB script files;
- SAS selection logic;
- math model output conversions required for other modules;
- math model guidance output processing to drive the HDDs and lab engineering displays;
- an Interface Alive counter used to determine when synchronization with Simulink was lost.

Additionally, the math model interface mapped the vehicle model outputs to standard VMS variables which were then sent to a VMS wrapper on the host computer.

For typical VMS simulations, the aircraft kinematics are performed in a standard VMS library routine, as indicated in Figure 3. However, for the current simulation, the delivered Simulink math model provided all the vehicle dynamics. For this reason, the standard kinematics routine was not used. Instead, a wrapper was created to accept the Simulink math model outputs and make all additional calculations needed to drive the visuals, motion, displays and tasks. The VMS wrapper primarily handled transformations to the local frame, conversions to pilot station and supplied some other state variables.

A routine that interfaced the host and the flight hardware was also written in FORTRAN and ran on the host computer in the primary executable binary. This module handled all the variables that were sent between the host and the flight hardware. Data sent to the flight hardware included aircraft state, cab buttons and trim hat deflections. Data received from the flight hardware included flight director modes, guidance commands and moving map information.

A routine that interfaced the simulator cab and host computer was also written in FORTRAN and ran on the host computer in the primary executable binary. The cab interface received the raw data from the cab, usually in engineering units or as discretes, and processed it for shipment to the Simulink math model and flight hardware. The cab interface also processed the signals sent from the Simulink math model to the cab. For example, during initialization, it calculated the signals to back drive the inceptors to the trim positions, and ramped out the control loader forces. Similarly, while running, it handled the logic to back drive the sticks as a function of the trim hat inputs, autopilot commands or other during certain SAS modes.

IV. Results

Despite worries that running the vehicle model in the Simulink environment would result in loss of synchronization and the inability to run in real-time, after the challenges were resolved, this method worked well with very few frame overruns. The full test matrix was completed with significant improvement in the model development and modifications efficiency. The Simulink based approach to driving the VMS increased the efficiency for integrating math model changes and increased the confidence that identical versions of the model were used for doing model analysis and pilot training. Development work on the SASs continued throughout experiment, trying out new ideas and making improvements based on pilot comments and research team observations.

Pilots trained for a full day using a desktop simulation set up in the pilot briefing room. In the VMS they performed formal evaluations for realistic high-workload full-mission scenarios, that were about sixty minutes, during which

they were required to file flight plans, communicate with Air Traffic Control (ATC) and operate Electronic Flight Instrument Systems while flying take-offs and missed approaches.

Occasionally the Simulink model did loose synchronization with the host, but synchronization was quickly restored so that the run could continue without the need to reset. All experiment evaluation runs were reviewed to determine if any undetected frame overruns occurred. For most runs, many of which were on the order of an hour long, no overruns were observed. Eight of the 100 evaluation runs encountered delays of 2 or more frames greater than the expected 3 frames. During these runs, this additional delay occurred on the order of twenty to fifty times over the full course of the run, for a maximum of 0.03% of frames with unexpected delays.

Figure 7 shows the average Linux computer math model execution time for each evaluation run, the values for which were about 0.9 ms, +/- 0.4 ms which is well below the 10 ms frame time. Figure 8 shows the maximum Linux computer model execution time per run, which ranged between approximately 2 ms and 10 ms. When the execution time approaches 10 ms, this can result in an extra frame delay. In order to review the Simulink diagram or MATLAB files during the experiment, the Linux computer windows were sometimes not left minimized, and the mouse was used. This is a likely cause for the near 10 ms maximum execution times. In order to run in hard real-time, the model execution time would have to remain below 10 ms, for the entire run.

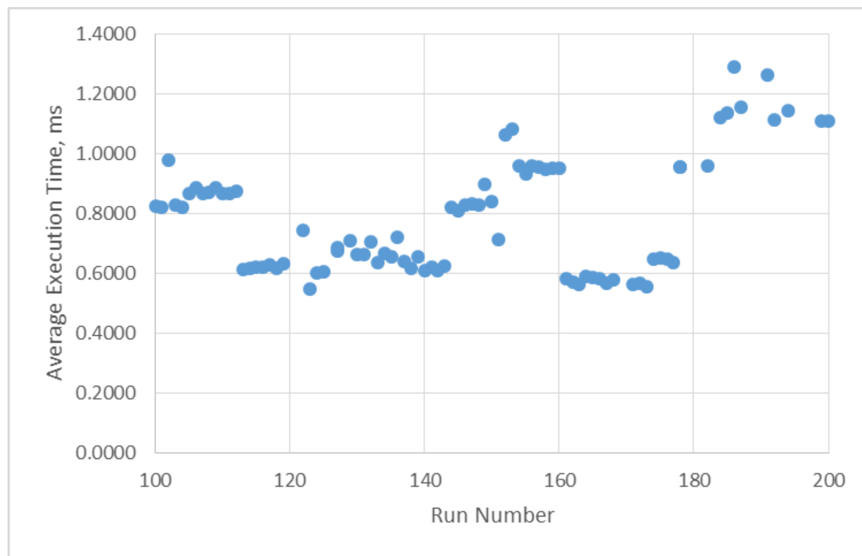

Fig. 7 Simulink Math Model Average Execution Time (ms) Per Run

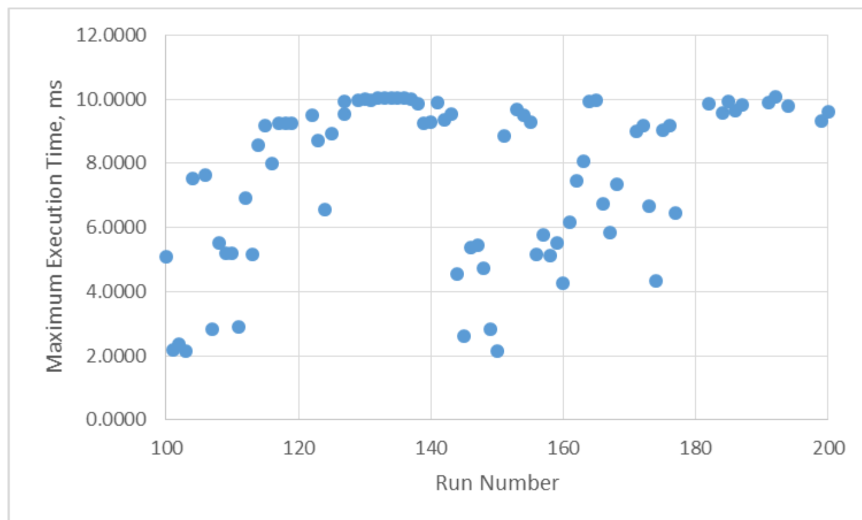

Fig. 8 Simulink Math Model Maximum Execution Time (ms) Per Run

Three times toward the end of the experiment the model lost synchronization with the host during an evaluation run. This was not a problem with the Simulink model execution. Instead it appeared that the host lost total communication with the Linux computer, which caused the simulation to freeze for a number of seconds. It appeared that the X-Server window on the PC being used to connect to the Linux computer, which is controlled by the PC process, had gone to sleep. Communication was reestablished by waking up the X-Server window on the PC by cycling the power button of the PCs monitor. After doing this, synchronization was reestablished and all runs then continued without incident. Figure 9 shows an example of such a run where the simulation lost synchronization in this way. The top plot of Figure 9 shows the Linux computer model execution time. The lower plot shows the number of frames for which the model update to the host was delayed, as determined by a real-time counter on the host computer. By comparing the model execution time, which never exceeded 4.5 ms, to the very large 53 frame delay, one can see that the overrun was not caused by the math model execution but by some other event in the system on the Linux computer.

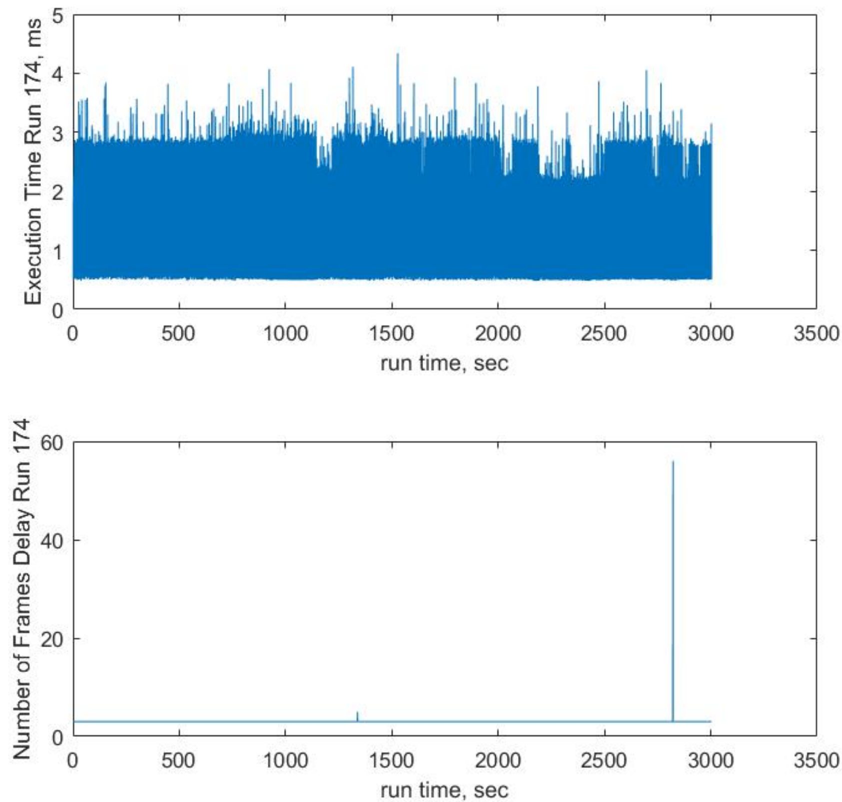

Fig. 9 Linux Math Model Execution Time and Number of Frames Delay Incurred for one Run

V. Challenges

This section discusses some of the challenges encountered when the Simulink environment with the VMS real-time architecture. The host frame rate ran at a consistent 100 Hz or 10 ms. The execution time of the Simulink math model on the Linux computer was, on average, about 1 ms, which is well within one frame. Initially, an attempt was made to run the Simulink math model synchronously with the host whereby, if the execution time exceeded the defined frame time, an overrun would occur and the experiment would stop running. Occasionally, however, it spiked up to 8 or almost 10, even on rare occasions going over 10 ms. When running synchronously, each large (near 10 ms) execution spike would cause a frame overrun, making this arrangement impractical for the length of runs planned. For this reason, it was agreed that best option was to run the Simulink math model asynchronously and rely on the hardware clock to ensure real-time I/O. Since the execution time spikes only occurred rarely, and for one frame at a time, it was

not noticeable to the pilot and was deemed acceptable by the researcher. An example plot of the Linux computer model execution time for a typical evaluation run is shown in Figure 10 below.

A. Latencies

During initial fixed-based development, a latency between the host and math model response was observed that would incrementally increase to the point where it caused instabilities with the autopilot. The latency slowly increased from the expected 3 frames up to 13 frames (resulting in response delays of 0.13 seconds, or 10 frames more than expected) where it would remain until the model was re-synched by stopping and re-starting the simulation. In an effort to isolate the cause of this problem, some diagnostics were used to observe the packet transfer rate. It was determined that the Simulink math model data were not always sent on a consistent basis. It would, in fact, not send any data for a few frames, and then would send several packets at once. The XIO logic, however, which assumed a real-time data stream, employed a First In, First Out (FIFO) processing method such that all packets were used in the order received. It was surmised that the observed latency was due to the Simulink math model sometimes taking longer than one frame to complete execution, an occurrence that resulted in packets incrementally building up in the buffer. The XIO buffer was flushed after 10 frames, hence the observed maximum of 13 frame latency. The XIO logic was subsequently modified such that only the most recent data was used and the rest was ignored. Though this was not a (hard) real-time solution, it did fix the incremental latency problem.

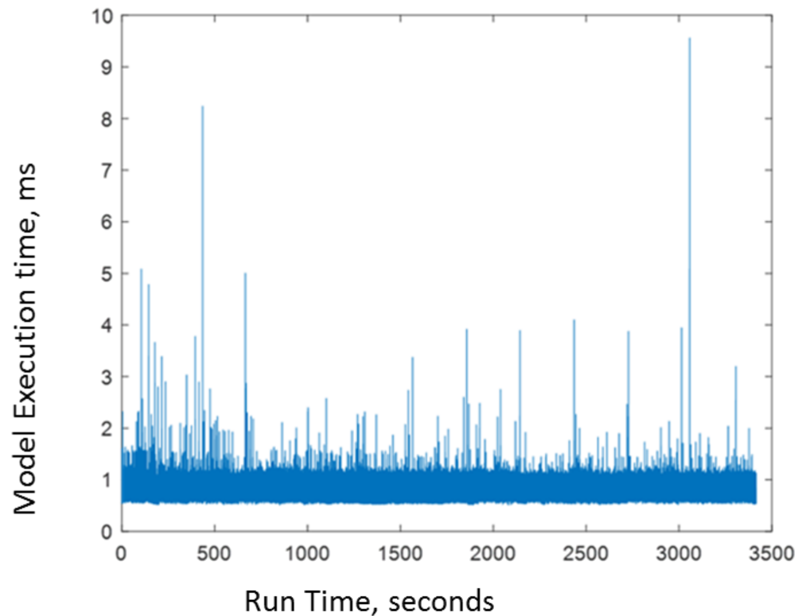

Fig. 10 Example of Simulink Math Model Execution Time (ms) for a typical run

When the Simulink math model was initially used to drive the Out-The-Window (OTW) visual scene, it was observed that the visuals were irregular and jagged. This was determined to be a result of the precision of the inputs to the OTW drivers. The Simulink math model output aircraft position in double precision latitude and longitude. Double precision is needed when using latitude and longitude to drive the visuals since the position changes between frames for a vehicle order of magnitude happens in the lower significant figures. However, since XIO only transferred floats, the doubles were automatically downcast to single precision, which resulted in the OTW visual's jumping. The XIO interface was expanded to handle double precision and this did provide smooth visuals.

Unfortunately, some vehicle response latency still occasionally occurred. The team learned that, when running a Simulink math model, MATLAB launches a large number of processes, some of which run at a lower priority causing the latencies. The team researched ways to set thread priorities to ensure all necessary Simulink processes were given a high priority and thereby guarantee real-time execution; however, a solution was not found. The best option was to minimize the number of MATLAB processes, and ensure the Linux computer math model execution time stayed safely below the frame rate. This was accomplished during experiment operations by minimizing all windows on the

Linux computer and not moving the mouse during evaluation runs. Further investigations will be performed into controlling thread priorities.

To minimize the real-time performance issues while running with the Simulink development environment the Linux computer screen saver was disabled and all Simulink and MATLAB windows were minimized unless actively in use. Logic was programmed on the host to create an alert – three rows of lights flashing at the terminal - when the math model latency exceeded four frames. This worked well and was very useful in quickly identifying occasional synchronization problems.

B. Experiment Validation

Numerous dynamic checks, including steps and doublet in all four axes, were run during the experiment development to validate the model and verify the interface programming and implementation. Thereafter, dynamic checks were run whenever significant model development work occurred (i.e. throughout the simulation) to verify no changes were inadvertently introduced into the model. A discussion of the vehicle response is out of the scope of this paper.

Piloted frequency sweeps, flown by the researcher, were recorded and used for frequency response analysis which verified that the math model after VMS integration was the same as the designed version.

To measure and document the experiment time delays, a Transport Delay Measurement (TDM) test was performed using the standard VMS procedures without the Linux computer in the loop. For the standard TDM test, a square wave command was generated on the host and used to drive the pitch position of a black and white board in the OTW visual scene, on a graphic engine display and to drive the motion system. This arrangement bypassed the control loaders to reduce measurement noise. The time delay due to the control loader analog/digital converter is known to be one-half of the base frame time, which in this case would be 0.005 seconds. Using this scheme, and an Image Dynamic Measurement System Mark 2 (IDMS-2), a tool designed to measure latencies [18], the following time delays for the standard VMS configuration were found:

- 70 ms host to OTW Transport Delay
- 50 ms host to motion system Transport Delay
- 40 ms host to HDD Transport Delay

The TDM test was repeated with the procedures modified to incorporate the current experiment configuration including the external Linux computer. In this case, the square wave command was sent from the host to the external Linux computer using a Simulink model input port. The signal was sent directly back to the host, bypassing the Simulink math model dynamics. The square wave command was then used to drive the OTW, cab and lab displays and the motion system. The results of the modified test showed exactly the expected additional 3 frames of delay, totals which meet the minimum transport delay requirement of 100 ms for Level D helicopter training simulators as specified by the FAA's US Title 14 Code of Federal Regulations (CFR) Part 60 [19]. The same regulation specifies a minimum transport delay requirement of 150 ms for airplane training simulators.

Taking these numbers into account, when delays from the Simulink math model are larger than the expected 3 frames, the OTW Transport Delay could exceed the 100 ms requirement. Moreover, if the expected ½ frame time delay (5 ms, or 0.005 sec, for a 100 hz cycle time) due to the control loaders is factored in, the OTW transport delay would be 105 ms, or 0.105 sec, which is slightly over the 100 ms minimum transport delay requirement for helicopter training simulators. For future experiments, it will be necessary to find ways to improve the transport delay performance. Some plans for this were discussed in the results section.

VI. Summary

A piloted simulation experiment was performed at the Vertical Motion Simulator during which the vehicle model and control systems ran in Simulink on an external processor utilizing flight hardware. A Linux system was used to run the Simulink math model and handle the data marshalling between the host computer, math model and the flight hardware. The math model executed in the Simulink environment on an external Linux computer and communicated with the host asynchronously at 100 Hz via an Ethernet connection. The Simulink math model execution time was approximately 1 ms on average, which was well within the 10 ms frame time. Ethernet was chosen as the data transmission method because, with such an interface, a data package message is sent that is synchronized to the start of frame, allowing the host hardware clock to be utilized for input/output synchronization. Communication between the host and external devices occurs on the frame boundaries. For this reason, a minimum of two frames delay between

the pilot inputs and the resultant motion and visual updates was incurred. The resultant 20 ms delay was deemed acceptable for this study. Although some latency problems, and resultant instabilities, were initially experienced, solutions were found such that the model was stable and successfully flown for the complete experiment.

Pilots performed formal evaluations for hour-long, high-workload, full-mission flights. Not only was the full test matrix completed, but significant improvements in efficiency were realized in control system development and enhancement work. Moreover, the Simulink based approach to driving the VMS was found to decrease development time by allowing quick integration of math model changes and providing the ability to run the same version of the model on researcher desktop computers.

VII. Future Work

In order to achieve hard real-time, some improvements are planned for future work. One likely and straight forward improvement will be made to the math model interface S-function, the current implementation of which produced 3 frames of delay. It is expected that if this were to be split into two blocks, an Input block and an Output block, then the Memory blocks (that were added to break algebraic loops) could be removed, which should reduce the delay from three frames to two.

Also, obtaining a faster computer will mitigate any model execution time overrun problems. Moreover, since fame overruns and loss of synchronization were not caused by the model execution but some other system events, finding a way to increase all the MATLAB and Simulink processes to high priority could minimize the delays incurred and allow actual hard real-time.

Finally, when the VMS environment is upgraded to a Linux system, the necessity to run Simulink on an external processor will be removed.

VIII. Acknowledgments

I would like to gratefully acknowledge Bosco Dias for his programming skills and dedication to excellence, Marty Pethtel for his detailed system architecture design, Alfredo Arencibia for his expertise, and to the entire VMS team for their skill and hard work on this experiment. I would also like to thank William Chung and Gordon Hardy for their advice and insight.

IX. References

[1] Danek, G. L., “Vertical Motion Simulator Familiarization Guide”, NASA TM-103923, 1993.

[2] Aponso, B. L., Beard, S. D., and Schroeder, J. A., “The NASA Ames Vertical Motion Simulator – A Facility Engineered for Realism”, NASA Ames Research Center, Royal Aeronautical Society Spring 2009 Flight Simulation Conference, London, UK.

[3] Magyar, T. J., Page, A. B., “Integration of the CASTLE Simulation Executive with Simulink,” Naval Air Systems Command, Patuxent River. MD, AIAA May 2001.

[4] Christhilf, D.M, and Bacon, B. J., “Simulink-Based Simulation Architecture for Evaluating controls for Aerospace Vehicles,” NASA Langley, AIAA, pp2,3.

[5] Hogg, E.F., “B-737 Linear Autoland Simulink Model”, NASA/CR-2004-213021.

[6] Allen, M. J., Beyer, E.W., Hales, S.A., and Ilvedson, C. R., “Leveraging MathWorks Tools to Develop a Simulation Framework for Diverse Customers,” AIAA Modeling and Simulation Technologies Conference, August 2011, Portland, Oregon.

[7] Landers, S., “Real-time Pilot-in-the-Loop and hardware-in-the-Loop Simulation at Gulfstream”, ADI Users Society, San Diego, California, December, 2007.

[8] Quaranta, G., Mantegazza, P., “Using MATLAB-Simulink RTW to Build Real Time Control Applications in User Space with RTAI-LXRT”, Dipartimeno di Ingeegneria Aerospaziale, Politecnico di Milano, Milano, Italy, 2001.

[9] Moelands, J. M., et al, “Automatic Model Transfer from MATLAB/Simulink to Simulation Model Portability 2”, NLR-TP-2006-674, National Aerospace Laboratory NLR,SESP, Noordwiijk, November 2006.

[10] Bodeman, C. D. and DeRose, F., “The Successful Development Process with MATLAB Simulink in the Framework of ESA’s ATV Project”, IAC-04-U.3.b.03, Vega IT GmbH, Darmstadt, Germany.

[11] Kemper, J. and Cotting, C, “Simulator Design for Flying and Handling Qualities Instruction” AIAA SciTech Forum, 2016, pp 11-12.

[12] Ackerman, Kasey Alan, “Development Of A Pilot-In-The-Loop Flight Simulator Using Nasa’s Transport Class Model” Thesis University of Illinois at Urbana-Champaign, 2014, pp8,23-24.

[13] Lu, P. and Geng Q., “Real-time Simulation System for UAV Based on MATLAB/Simulink” Deep Space Exploration Technology and Experimentation Project, IEEE, 2011.

[14] Gerlach, T. et al, “Running High Level Architecture in Real-Time for Flight Simulator Integration,” DLR, Institute of Flight Systems, AIAA, 2016, pp1, 6.

[15] MicroTau Users Guide, Contract No. NAS 2-98084, 1999, NASA Ames Research Center, Moffett Field, California, 94035.

[16] Hardy, G., “Programmable Portable Guidance Display Users Manual,” NASA TM-20130215983, NASA Ames Research Center, Moffett field, California, March, 2013.

[17] Malpica, C., Lawrence, B., et al “An Investigation of Large Tilt-Rotor Hover and Low Speed Handling Qualities,” American Helicopter Society, AHS Annual Forum 67th Annual Forum,
Virginia Beach, VA, May 3-5, 2011.

[18] Lehmer, R.D., and Chung, W.Y.: “Image Dynamic Measurement System (IDMS-2) for Flight Simulation Fidelity Verification,” Logicon Information Systems and Services, AIAA-1999-4035, August 1999.

[19] FAA, “Flight Simulation Training Device Initial and Continuing Qualification and Use,” US Title 14 CFR Part 60, Department of Transportation, US, March 30, 2016. .