

# Workstation-Based Avionics Simulator to Support Mars Science Laboratory Flight Software Development

David Henriquez, Timothy Canham, Johnny T. Chang and Elihu McMahon  
*Jet Propulsion Laboratory, Pasadena, CA, 91109-8099*  
*California Institute of Technology*

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## Abstract

The Mars Science Laboratory developed the WorkStation TestSet (WSTS) to support flight software development. The WSTS is the non-real-time flight avionics simulator that is designed to be completely software-based and run on a workstation class Linux PC. This provides flight software developers with their own virtual avionics testbed and allows device-level and functional software testing when hardware testbeds are either not yet available or have limited availability. The WSTS has successfully off-loaded many flight software development activities from the project testbeds. At the writing of this paper, the WSTS has averaged an order of magnitude more usage than the project's hardware testbeds.

## I. Introduction

The MSL WorkStation TestSet (WSTS) was originally designed to serve as a development environment for flight software developers to exercise and debug their software. It was designed to be completely software-based and run on a flight software developer class Linux workstation. This allows individual users to have their own execution environment and allows device-level and high-level flight software testing when hardware testbeds are either not yet available or have limited availability. Due to its success it has also evolved into a verification and validation (V&V) environment for Systems Engineers, System Integration and Test Engineers, and Mission Operations Engineers for validating system-level behaviors and executing dry-runs of flight sequences.

In order to meet the needs of the MSL project, the WSTS was designed with certain principles in mind. The first principle was that flight software executing in WSTS should not know whether it is running against real or simulated hardware. It must serve as the virtual reality for the flight software. Any piece of engineering data visible to flight software must be modeled to meet this objective. This generally implied that there was a simulation software object for every MSL hardware element except the flight processor, which was not emulated. Each software objects was coded to accurately model the hardware behavior visible to flight software, as well as modeling the finite state behavior of the device. Hardware specifications were used as the source material for modeling the device behavior as well as the format and content of the engineering data. However, in the case of some devices like science instruments, the simulation software objects were accurate in data format but not necessarily in content because flight software behavior was insensitive to the content of the data packets. Only when flight software responses were a function of data content that the simulation software objects in WSTS accurately modeled both the device behavior and the output.

The second design principle was to maximize simulation software object commonality between WSTS and the hardware-in-the-loop testbeds. By doing so, the WSTS software objects could be reused in the hardware-in-the-loop testbeds in the event that a hardware device was not present. This is a common occurrence during the development cycle of a flight project when hardware is not yet ready for delivery to the testbeds, is never intended to be delivered to all the testbeds for cost reasons, or has to be removed for repair or additional standalone testing. By simply adding interface code so that the WSTS simulation objects have access to I/O cards, so that they electrically appear to the flight system as the missing device, system level testing can progress with simulated devices. This has proven to be extremely cost effective for MSL, especially when hardware originally intended to be built for some of the testbeds was descoped and replaced by simulated hardware.

Continuing with the second principle, WSTS also has the same user interface as the hardware-in-the-loop testbeds. The look and feel is identical to the testbeds and most of the user interface commands are common as well. However, one major difference is that the WSTS was designed to minimize idle time and run as fast as possible. T WSTS typically runs between 2 and 10 times faster than real-time, depending on the simulation complexity. The

complexity is mostly a function of the number of continuous states in the dynamic simulation that require numerical integration of differential equations. The other major difference between the non-real-time WSTS and the real-time testbeds is that users can have fine control of execution time in WSTS. Users are able to pause, resume and single-step the WSTS simulation, which is extremely valuable for analysis of flight software task behavior.

The final major principle driving the design of the WSTS was that the system must support an extensive fault injection capability because many faults can only be tested in a simulation. Hardware-in-the-loop testbeds are constrained because users usually cannot fail the real hardware. WSTS can inject faults either at a functional level like failing a thruster in a stuck-closed position, or at a very low level like setting a specific bit in an output data packet from a simulated device. This flexibility in testing can only occur in a software-based test environment. Additionally, the WSTS allows flight software testers to stress test the software in an environment that is safe and doesn't risk exposing flight hardware to unforeseen paths in the flight software. Hence, it is usually desirable to first test in WSTS before repeating the same tests in the hardware-in-the-loop testbeds.

These above design principles have guided WSTS architecture and resulted in a product that is designed to serve as a powerful development and V&V tool for all phases of the MSL mission, which includes Launch, Cruise, Entry-Descent-and-Landing (EDL) and Surface mission phases. It truly serves as a virtual reality for MSL flight software.

## II. Architecture

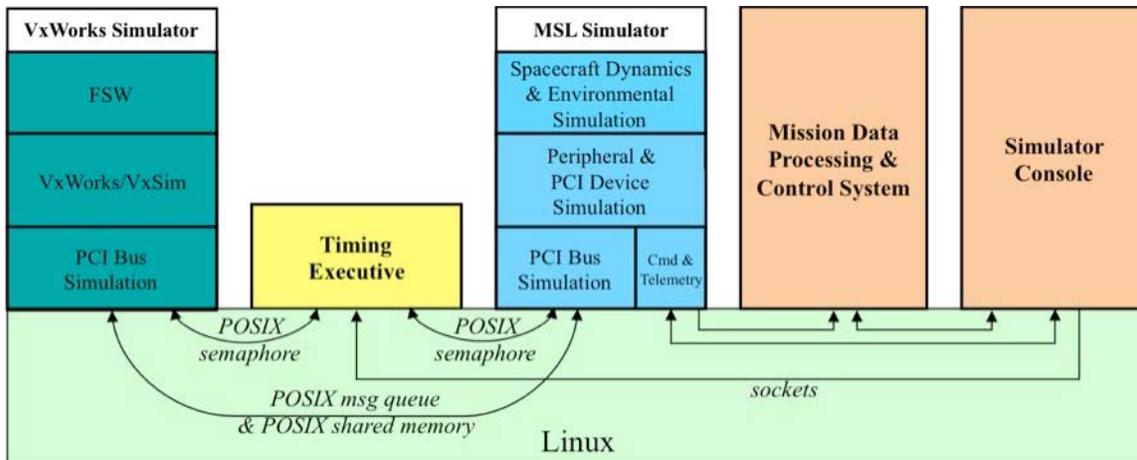
### A. Overview

The Workstation Testset (WSTS) consists of five Linux software applications executing as an integrated session on a workstation-class Linux PC. Together, the applications provide a flight-like environment for developing and testing high-level flight software, device-level flight software, flight commands and flight telemetry. The five applications of a WSTS integrated session are listed in Table 1.

Application	Function
Wind River VxWorks Simulator	MSL Flight Software execution context.
MSL Simulator	Hardware, dynamics, and physical models that simulate the MSL spacecraft and its environment in its Launch, Cruise, Entry-Descent-and-Landing and Surface mission phases.
Timing Executive	Master application that coordinates execution of the Wind River VxWorks Simulator and the MSL Simulator.
Simulator Console	Command console to configure and control the MSL Simulator.
Mission Data Processing and Control System (MPCS)	Ground Data System software that enables flight-like uplink and downlink processing, including Spacecraft Command handling and telemetry viewing.

**Table 1: Applications in a WSTS Integrated Session**

The Wind River VxWorks Simulator (VxSim) provides the familiar Wind River Target Shell interface within which MSL flight software builds are executed. Users have access to the same VxWorks system and API calls as they would on a hardware-in-the-loop testbed. The only notable difference is that the flight software can execute at a faster rate when the VxSim system tick is configured to clock faster than real-time. Out of the box, VxSim simulates the behavior of a standalone target computer and supports socket interprocess communication with external applications. The socket interface was too limited for the WSTS architecture because the external application (i.e. the MSL Simulator) simulated the other devices that are "collocated" on the same PCI bus as the target simulation. VxSim was modified for WSTS to use POSIX semaphores, POSIX message queues and POSIX shared memory for its interprocess communication with the other applications in WSTS. These additions provided the infrastructure needed to implement time slice execution control between VxSim and the MSL Simulator and also the PCI Bus Simulation that enabled simulated PCI devices in the MSL Simulator to communicate with the MSL flight software device drivers executing on VxSim.



**Figure 1: WSTS Architecture**

The Timing Executive coordinates the execution of the modified VxSim and the MSL Simulator so that the two applications stay synchronized. It is the Timing Executive that provides the user the capability to pause, resume and step the WSTS integrated session. This feature enables both debugging and fault injection capabilities.

The MSL Simulator simulates the MSL avionics, which include the PCI devices in the Rover Compute Element (RCE), the 1553 remote terminal peripherals, the GN&C sensor and actuator peripherals and also the science instrument peripherals. The simulated avionics peripherals are stimulated by dynamics and environmental simulations that simulate the Launch, Cruise, Entry-Descent-and-Landing (EDL) and Surface mission phases. The user is able to interact with the MSL Simulator in a flight-like manner, through MSL flight software, or can issue commands directly to the MSL Simulator. The MSL Simulator also provides “truth” telemetry that is archived by the same ground tools (i.e. MPCS) as flight telemetry. As mentioned above, its main interface with MSL flight software is through the PCI Bus Simulation. The MSL Simulator only advances its avionics simulation when it is given a time slice from the Timing Executive.

MSL Simulator commands and telemetry are routed through the Simulator Console. The Simulator Console also has an optional GUI to display Simulator telemetry and provides a command line interface and a Python interface for executing single or batched commands. These interfaces provide return status for commands that are executed on the Simulator. MSL Simulator commands and telemetry are also available from the Mission Data Processing and Control System (MPCS). However, this interface is flight-like and so Simulator commands executed from MPCS do not return status. Command status must be ascertained from the Simulator telemetry. But the advantage of MPCS is that it allows the user to side-by-side compare Simulator telemetry with the flight telemetry produced by MSL flight software. Scripts and displays that are developed for MPCS can then migrate from WSTS to hardware testbeds and can even evolve for use in operations. WSTS is able to offload the hardware testbeds by including MPCS in its integrated environment and thus provide a venue by which test procedure can be tested and refined prior to consuming time on a hardware-in-the-loop testbed.

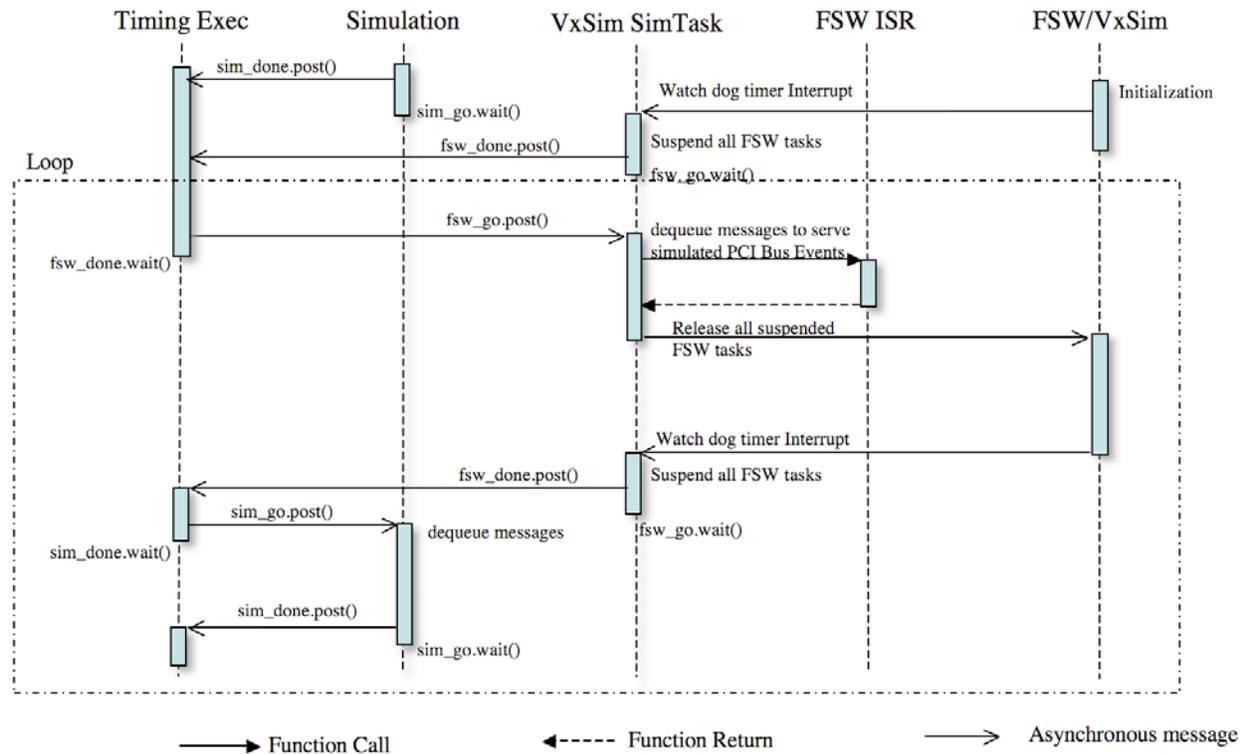
WSTS provides a high availability test venue that is only limited by the computer resources available to the user. Each WSTS integrated session executes in an independent environment, enabling multiple sessions to run on the same workstation without conflicting with one another. The commands and telemetry for each session are logged by MPCS and can be later queried for post-processing.

## **B. WSTS Execution Model**

WSTS defines a time slice as the smallest unit of execution time in which either the VxSim or the MSL Simulator can continuously run. The size of the time slice is tuned a priori to minimize context switches and still faithfully reproduce the same order of events as on the real flight system. The Timing Executive is the master program that is responsible for distributing execution time slices to VxSim and then to the MSL Simulator in a round-robin fashion, as shown in Figure 1 and Figure 2. This controlled serial execution of the aforementioned applications keeps them synchronized by not allowing either application to get ahead of the other. Maintaining synchronization is the only constraint on the WSTS execution model. So unlike a real-time simulation with specific hardware-driven deadlines, WSTS can arbitrarily throttle the advancement of the simulation without violating any

deadlines. This execution model enables faster and slower than real-time execution of WSTS as well as provide the pause, resume and step execution control.

The Timing Executive implemented the above execution model by sharing a pair of POSIX semaphores (i.e. “go” and “done”) with the modified VxSim application (see previous section) and another pair of POSIX semaphores with the MSL Simulator application. Each application waits for the release of the “go” semaphore to begin execution and then releases the “done” semaphore when finished. After either application is “done” with its time slice, it goes back to the top of its loop and waits for the “go” semaphore again, remaining inactive until it is released again by the Timing Executive. The Timing Executive guarantees that one side receives its “go” semaphore only when the other side has sent its “done” semaphore; thereby ensuring that one side cannot run and interfere with other. This sequential execution avoids race conditions between the flight software running on VxSim and simulated hardware running on the MSL Simulator and maintains synchronization. A summary of the WSTS execution model in the WSTS is shown in a sequence diagram in Figure 2.



**Figure 2: WSTS Sequence Diagram**

### III. Flight Software

#### A. Flight Software Tasking Model and Synchronization

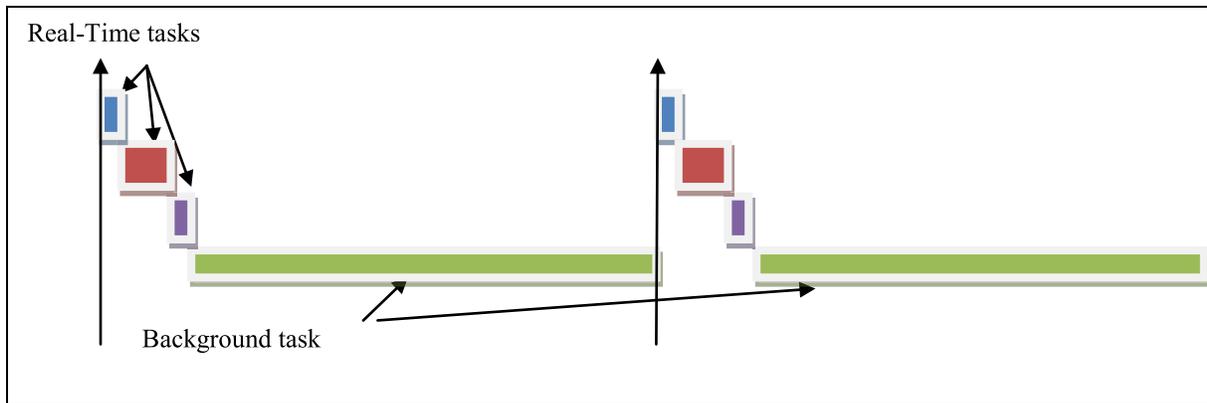
MSL flight software can generally be categorized in to two types of activities: real-time and background tasks. Real-time tasks can be described as those tasks that must be procedurally and temporally correct. In other words, they must do the right thing, and do it at the right time. Real-time tasks are given a high priority in to ensure that critical activities are completed on time. Background tasks are job-related tasks that do not have a real-time deadline. They are typically episodic in nature. Consequently, they are given lower priorities than the real-time tasks.

The flight software real-time task cycles are initiated by hardware-based timing signals. A high-accuracy, high-precision time source sends hardware interrupts on a programmed interval. For the MSL software, that interval is 64Hz. MSL flight software ties the VxWorks system tick to the 64Hz RTI to control the OS preemption and round-robin task scheduling. Additional asynchronous hardware interrupts cause real-time tasks to cycle when different hardware activities complete. The frequency of these interrupts is dependent on the operation of the hardware being

used. These interrupts act as events in the system to drive software execution. Higher-level behavior in the system is driven by ground commands uploaded to the spacecraft. These ground commands can either cause background tasks to commence or activate a real-time task.

In WSTS, the MSL Simulator is the source of all hardware interrupts to MSL flight software. This gives the MSL Simulator control over the VxWorks system tick and thereby the frequency that the real-time tasks cycle. But it does not give it control over the execution of the background tasks. Flight software synchronization for WSTS requires that the background tasks also suspend their execution. In order to freeze execution of all tasks, MSL flight software programs an additional watchdog timer when it is built for VxSim. This watchdog timer expires after a time slice of execution time has been consumed. The watchdog timer expiration places the flight software in interrupt context and the watchdog service routine uses the “go” and “done” semaphores to implement its side of the WSTS execution model. Once the watchdog service routine receives its “go” semaphore, it reprograms the watchdog timer and returns so that flight software can return to task context. The addition of the watchdog timer enables MSL flight software and the MSL Simulator to execute in a coordinated manner.

Additionally, MSL flight software uses a different idle task when it is built for the WSTS. The time flight software spends in the idle task is wasted CPU time. The modified idle task “shortcuts” the wait time by expiring the watchdog timer and giving up its time slice. This modification was implemented to optimize the system performance of WSTS by minimizing the busy waiting.



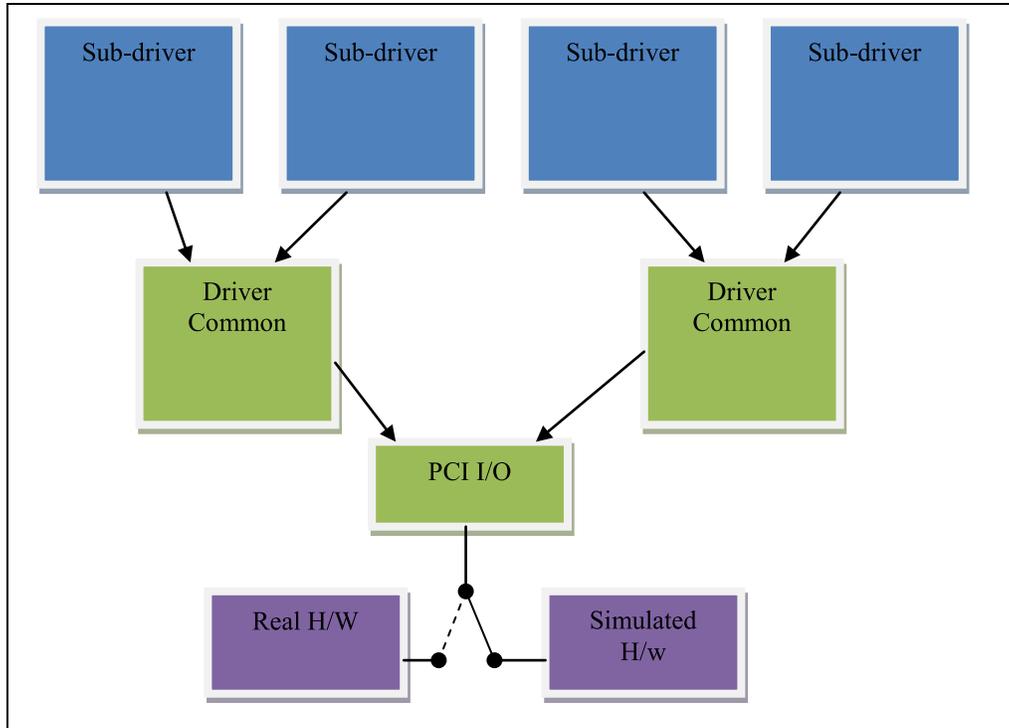
**Figure 3: Flight Software Tasking Model**

## B. Flight Software Layering

MSL flight software can be decomposed into three layers: the driver layer, the service layer, and the application layer. The application layer controls the higher-level behavior of the system, and performs activities such as sequencing, navigation, spacecraft health and configuration, and science and engineering management. The service layer provides centralized services like data management, inter-process communication, logging, and file systems. The driver layer is responsible for controlling the hardware interfaces in the system. And it is the driver layer where the interfaces to the MSL Simulator are accommodated. The layering insulates the higher-level flight software from differences in the interface code, and ensures that the maximum possible amount of flight software code is common between the WSTS platform and the hardware-in-the-loop platform.

The MSL flight software driver layer is further subdivided into three layers: the hardware I/O layer, the board common functions, and the board sub-functions. The other two layers use the hardware I/O layer to access the PCI address space. This layer is the point at which interprocess communication with the MSL Simulator would need to take place. The layer has a platform-independent interface defined, and alternate implementations are compiled depending whether the software is built for the real PCI bus or the PCI Bus Simulation. This allows the driver layering also enables the drivers to also have common code between the WSTS platform and the hardware-in-the-loop platform. The MSL flight software layers above the hardware I/O layer cannot differentiate between the driven by WSTS or by real hardware.

There are two flight software drivers that could not be accommodated by the above layering: the Direct Memory Access (DMA) driver and the Non-volatile Memory (NVM) driver. The DMA driver required an alternate WSTS implementation because that driver is responsible for transferring large buffers of data over the PCI bus using hardware built into the RAD750 flight computer that is not modeled in the MSL Simulator. It was also decided that the NVM storage interface would be implemented as local file storage in WSTS for performance reasons.



**Figure 4: Flight Software Driver Layering**

#### **IV. MSL Simulator**

##### **A. MSL Simulator Layering**

The MSL Simulator also uses a layered approach to implement its software. Its four different layers are: the PCI Bus Simulation, the PCI Board Simulation, Peripheral Device Simulation and the Dynamics Simulation. At the top level, the Dynamics Simulation layer is populated by spacecraft and environment dynamics models that compute various simulated physical states that are fed into simulated hardware objects in the Peripheral Device Simulation Layer. In the case of closed-loop dynamics simulation, objects in the Peripheral Device Simulation layer would also produce stimulus that affect the dynamics and environmental models.

The Peripheral Device Simulation layer implements the functions are the window-to-reality for the flight. All the software objects in this layer rely on a data bus (serial or 1553) to interface with the simulated hardware in the PCI Board Simulation layer. It is the responsibility of the simulated peripheral devices to produce engineering data that will be ultimately consumed by the flight software.

The next layer down is the PCI Board Simulation layer. The simulated devices in this layer implement the PCI register interface behavior that is visible to flight software and are the conduits for engineering data to and from the Peripheral Device Simulation Layer.

At the bottom is the PCI Bus Simulation layer. This layer is responsible for all data transferred to flight software. This layer provides the infrastructure to generate PCI interrupts and initiate PCI DMA transfers to the flight software.

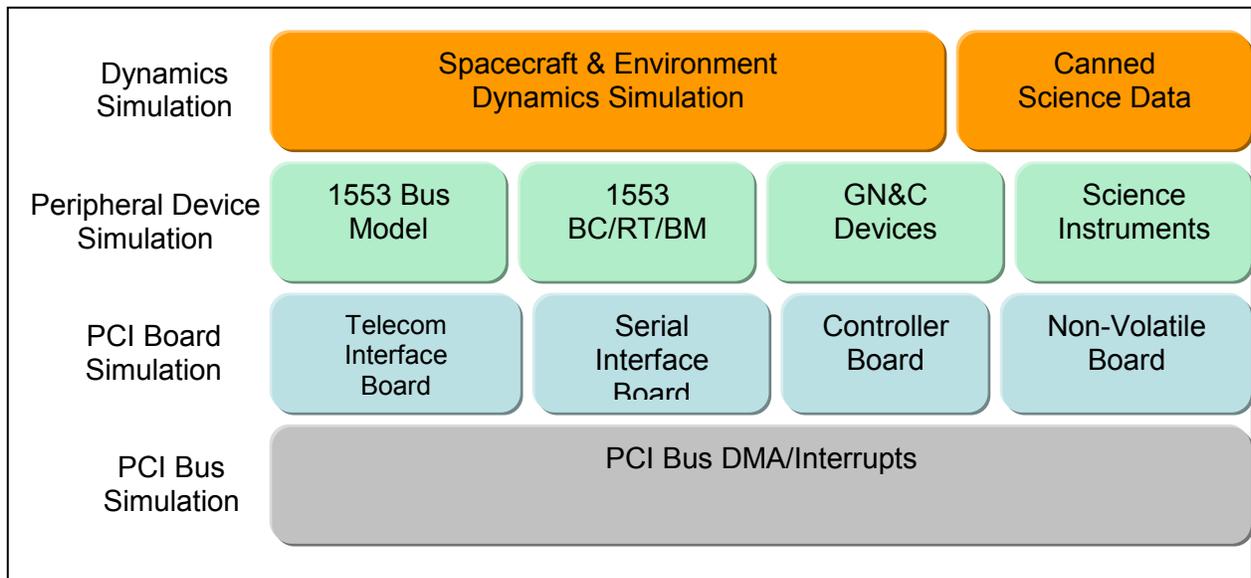


Figure 5: MSL Simulator Layering

**B. MSL Simulator Scheduling Model**

The MSL Simulator is a multi-thread process with a main thread, a thread for processing simulation commands and several threads for sending the telemetry down to MPC. It is in the main thread that computes all the simulated device objects and dynamics models are scheduled and executed.

The MSL Simulator must execute all the different types of simulated device and dynamics models in the right order. Certain devices need to run at one rate to periodically generate timing interrupts and bus events, while the integration of the dynamics model might need to run at a different rate than those. This presents a challenge to coordinate all these events to occur at the right order. The MSL Simulator is able to execute the various objects in the proper rate group with its master event-based scheduler to sort the events and to determine the execution order. The event-based scheduler treats every simulation computation as a schedulable event. Events can be periodic or aperiodic. The MSL Simulator is able to advance for any arbitrarily sized time slice and still execute the events in the proper order (see Figure 6). Running the scheduler as a single thread also eliminates concurrency issues of running all the simulated devices simultaneously at different rate groups. Most simulated devices in the Peripheral Device Simulation Layer can be executed in the event scheduler without loss in fidelity. However, device objects in the PCI Board Simulation Layer interact very closely with the flight software. This class of simulated device required additional capability to simulate the register-level interfaces that are visible to flight software.

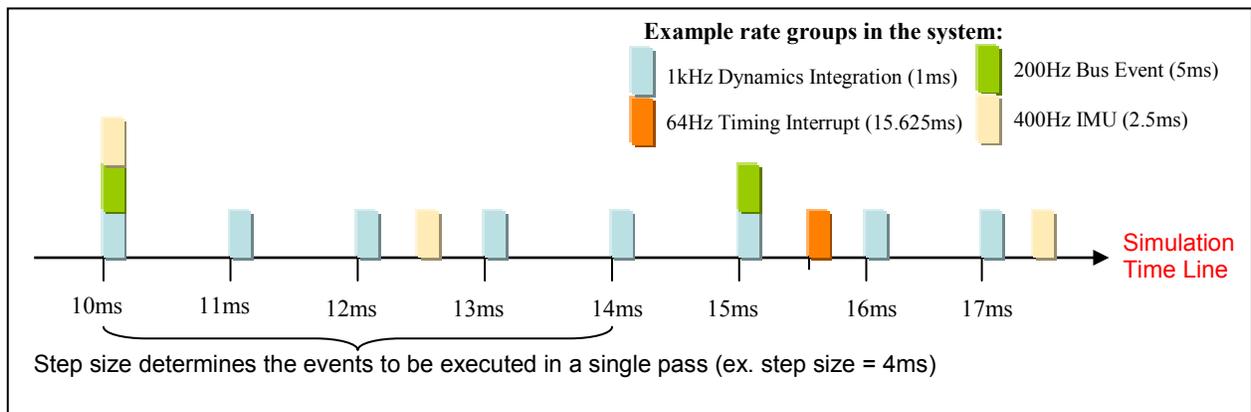


Figure 6: MSL Simulator Event-Based Scheduler

### C. PCI Board Simulation & PCI Bus Simulation

At the lowest layer, the PCI Bus Simulation implements the flight-like interface between the flight software and the MSL Simulator. The PCI Bus Simulation had to address the following PCI bus behaviors<sup>2</sup>:

1. Register & memory accesses
2. PCI device configuration and enumeration
3. Interrupts
4. DMA transfers

WSTS uses a combination of POSIX shared memory and POSIX message queues to implement its PCI Bus Simulation. The shared memory is used to represent the PCI memory space that is mapped to the flight processor memory space in VxSim. The flight software PIO reads and writes are redirected to this shared memory to simulate access to the card hardware. PCI DMA transactions and PCI interrupts are simulated by messages sent via the message queues. These POSIX interprocess communication mechanism are instantiated at run with unique identifier. This allows multiple instances of the WSTS to run on each workstation without interfering with each other. Both the simulation and the flight software applications attach to the shared memory and shared message queues during initialization. The shared memory also contains the PCI configuration header space that stores the PCI device and function information. These regions are initialized by the simulation and accessed by the flight software during its initialization. Flight software become aware PCI device base address offsets and interrupts in the same manner as it would in a real rover computer; the only difference is that the offsets refer to simulated PCI device address regions in the shared memory.

Simulating programmed I/O behavior of the PCI devices presents another challenge because the flight software directly interacts with these simulated devices by reading device registers that often require immediate. The WSTS round-robin execution paradigm does not allow the MSL Simulator to run and respond while the flight software has the control in a time slice. In the real hardware, the response triggered by the register access can occur on the order of microseconds. This is much shorter than the shortest period of a reasonably sized time slice. Since the MSL Simulator cannot respond while the flight software is running, part of the simulation logic needs to reside in the execution context of the VxSim in order to respond support programmed I/O. This is accomplished by capturing every PCI access from the flight software into a special function call. This function call distributes the PCI accesses to a proxy, simulated device that resides in the VxSim. This simulation proxy is simply a subset of the PCI device simulation that implements special register behaviors that require immediate responses to stimulus. The register access responses are immediately written to the appropriate register address in the VxSim context and also forwarded to the MSL Simulator using the shared memory. This allows the simulated devices in the MSL Simulator to synchronize with the proxy simulations running in VxSim by picking up these requests and continue with the processing of these events.

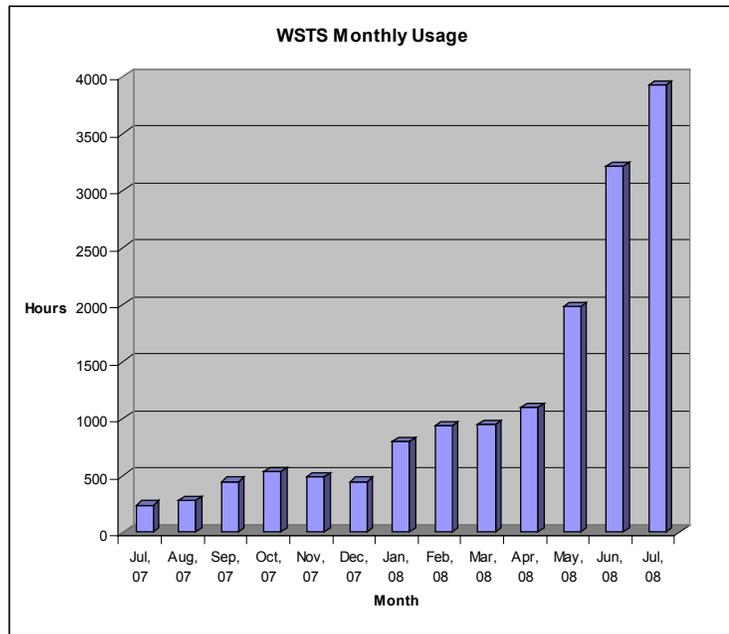
Interrupts and burst transactions (i.e. DMA transfers) are events implemented as messages in a POSIX message queue in the PCI Bus Simulation. There is a queue for each transfer direction: one for events directed at the MSL Simulator, and one for events directed at the flight software in VxSim. PCI devices issue interrupts to signal the processor. The processor, upon receipt of these interrupts, invokes Interrupt Service Routines (ISRs) that are registered by the flight software. The PCI interrupt message sent to the flight software contains the interrupt line information as the parameter. When the flight software dequeues the message, it determines which interrupt was sent and calls the correct interrupt handler.

Direct Memory Access (DMA), also known as PCI burst transactions, is handled differently in WSTS than programmed I/O. The PCI bus allows burst transactions whereby an initial address and a block of data are presented to the bus, allowing it to efficiently transfer large blocks of data between a PCI initiator and a PCI target. The PCI Bus Simulation handles this differently in that it does not write the data anywhere in the shared memory regions of the cards. Instead, the data is written to one of two allocated staging areas within in the shared memory. The DMA write staging area in the shared memory is used to hold blocks of data that would be transferred from the flight processor to the simulated PCI cards, or from the simulated PCI cards to the flight processor. During each time slice, the flight software and/or the simulated devices will stage buffers in this area, and the buffers will be picked by its counterpart during its next time slice. This write is combined with a DMA event to simulate a DMA transfer. Similarly for DMA reads, a separate staging area is allocated in the shared memory.

## V. Conclusion

The first version of WSTS was deployed to MSL in October 2006 with only a compute element simulation. At the writing of this paper, the 30<sup>th</sup> release (wsts-8.02) of WSTS was just delivered to MSL. In all this time, the WSTS architecture and design principles have held up. WSTS has not needed any major redesign as more capabilities have been added to it. On the contrary, WSTS has been successfully used in closed-loop cruise testing, closed-loop motor control software development and in developing flight software for hardware that has not been delivered to the hardware-in-the-loop testbeds. The MSL Flight Software Development Team and the MSL System Integration and Test Team have made extensive use of the capabilities in WSTS. The procedures and scripts development by these teams can be developed on WSTS and reused on the hardware-in-the-loop testbeds. These early successes for WSTS have encouraged the MSL project to make more use of it a V&V tool for all phases of the MSL mission.

WSTS has already been credited to uncovering flight software defects and also defects in the flight hardware where it deviated from its specification. WSTS has proven to be a valuable development tool and has helped decrease the number of flight software development shifts on the hardware-in-the-loop testbeds. At the writing of this paper, the WSTS has already averaged an order of magnitude greater usage than the project's hardware-in-the-loop testbeds. As MSL approaches its launch date, the WSTS usage can only increase.



**Figure 7: WSTS Usage Since July 2007**

### Acknowledgments

The authors would like to acknowledge the contribution of various teams and individuals that have influenced the design and development of WSTS. WSTS is an evolution the Mars Exploration Rover (MER) avionics simulator developed by Micah Clark and Robert Steinke. These same individuals also developed the modeling framework technologies used in WSTS for the JPL, in-house Mars Technology Development Project. WSTS also used the PCI Bus Simulation technology initially developed the Simulation Software Team on the JPL, in-house Multimission System Architecture Platform Project. And the authors would also like to thank Monte Ratajczyk, inventor of LMA's SoftSim, for his encouragement and advice on implementing a workstation-based simulator.

The authors would like to thank the hard work and dedication by the WSTS Development Team that delivered a product that already had a positive impact on the MSL. The authors would also like to thank the MSL Flight Software Development Team whose close collaboration was fundamental to successfully deploying WSTS.

### References

- <sup>1</sup>Wind River, *Wind River VxWorks Simulator User's Guide*, version 6.1, Wind River Systems, Inc., Alameda, CA 2005.
- <sup>2</sup>Tom Shanley, Don Anderson, *PCI System Architecture (4<sup>th</sup> Edition)*, Addison-Wesley Professional, 1999.