Applying Model-Based Design to Large-Scale Systems Development: Modeling, Simulation, Test, & Deployment of a Multirotor Vehicle

Richard Ruff[1], Cole Stephens[2], and Saurabh Mahapatra[3]
MathWorks, Inc., Natick, MA, 01760

The application of Model-Based Design to engineering design problems is well established in the aerospace industry. One of the challenges of large scale adoption is the increase in model and development process complexity. Models provide a means to communicate design information, define architectures that support team collaboration, verify specifications through simulations, and automatically generate code. This paper illustrates the use of a shared platform for communicating critical design elements within a collaborative environment. The case study presented in this paper demonstrates the development of a flight-control system for a multirotor vehicle within Simulink®.

Nomenclature

DOF = Degrees of Freedom
FCS = Flight Control Software
GPS = Global Positioning System
IMU = Inertial Measurement Unit
I/O = Input/Output
ROI = Return on Investment
SCM = Source Control Management
SVN = Subversion® (version control software)
TLC = Target Language Compiler

I. Introduction

Model-Based Design [(1) (2) (3)] offers an efficient and cost-effective way to develop complex embedded systems for a variety of applications. The design is specified as a system-level software model containing modular components on which scenario testing is conducted through simulation. The model serves as the single source of truth for the system design and becomes the basis for generating code that is deployed and tested on real-time hardware platforms. Encompassing the entire development lifecycle is the extensive use of verification and validation to promote consistency between design and requirements. To justify the return on investment (ROI) of the above paradigm [(4)], tool vendors have relied on the use of practical demonstrations or case studies that illustrate various aspects of the workflow. Such practical demonstrations typically focus on small-scale systems or specific aspects of an overall workflow. These approaches discount the additional complexities of large-scale systems such as interdependencies among multiple tool chains, distributed team collaboration, real-world implementation, and scalability issues.

Organizations that adopt Model-Based Design for large-scale systems [(5)] often discover the additional challenges associated with scaling up *simple* models to more complex ones. The discovery of such issues late in the adoption process leads to diminished ROI and introduces process risk. In other words, it is imperative for tool vendors to

[1] Principal Application Engineer, Application Engineering, 3 Apple Hill Dr., Natick, MA 01760, AIAA Senior Member
[2] Principal Application Engineer, Application Engineering, 3 Apple Hill Dr., Natick, MA 01760, AIAA Senior Member
[3] Senior Product Manager, Simulink Core, 3 Apple Hill Dr., Natick, MA 01760, AIAA Member

educate organizations adopting Model-Based Design through the use of realistic demonstrations for large-scale system development within the context of the enterprise. This paper advocates the use of such an approach thus motivating tool vendors to build realistic large-scale demonstrations or case studies that will help organizations discover critical development issues early in the process, mitigate ROI concerns, and associated risks.

In this paper, the case study is concerned with the development of a large-scale system for a quadrotor vehicle prototype. For the purposes of this paper, a large-scale system refers to any complex system that requires collaboration between myriad engineering disciplines. The principles used in this study can be applied to any system. Using Simulink®, current state-of-the-art modeling techniques for developing realistic systems within a collaborative environment are illustrated.

II. Quadrotor Model Development

Several issues commonly encountered in large-scale system development helped define the modeling platform characteristics and potential process deficiencies. For example, the design required multidisciplinary knowledge spanning rigid-body dynamics, aerodynamics, flight controls, and sensor hardware. Since such knowledge is typically distributed among several engineers on the team, the need for a common design platform that promotes collaboration became imperative.

Requirements for the modeling platform were drawn up based on two different perspectives: that of a single engineer and that of the team collaborating on the design. From the perspective of a single engineer, the modeling platform should provide rich modeling constructs specific to their domain that can enable them to express their ideas directly and concisely. A fully functional model representing a complete system would need to operate under different scenarios, so the platform should handle and track different types of Input/Output (I/O) and parameter data. It would also need to manage the complexity associated with large sets of data that grow with model size. For a mechanical system having several degrees of freedom (DOF), it should provide 3D visualization support to help engineers understand the motion of the moving parts. Since the quadrotor system has manual controls, the platform would need to support near-real-time operation for pilot-in-the-loop simulations.

From the team's perspective, the modeling platform should enable collaboration among engineers from disparate disciplines to enable parallel development. It should provide access to source control and configuration management tools from the design tool to support iterative workflows without disruption. For high integrity systems, it should automate the creation of artifacts such as documentation and reports throughout the development cycle.

A. Choice of Modeling Platform

Based on the requirements for the modeling platform and the authors' familiarity with the toolchain, Simulink® was selected for this project. Simulink® provides a graphical environment [7] for design, simulation, implementation, and testing of multidomain systems that include communications, controls, signal processing, video processing, and image processing applications. SimMechanics™ [8] was selected to model the differential algebraic equations describing the multi-DOF body mechanics in an intuitive block diagram form. For the controller that coordinates and schedules the entire flight, Stateflow® [9] was used to implement a state chart to simplify the representation of complex logic. Other tools used in the case study will be described throughout the paper whenever appropriate.

B. Facilitation of Team-Based Collaboration

The core design team consisted of two engineers who laid the foundation for the simulation framework, developed the model architecture, and spearheaded the modeling efforts. The team grew as engineers from different domains of expertise contributed to the system-level design. In the initial stages of the project, email served as the primary mechanism for ad hoc exchange of models among the team members. As the number of design components and their owners grew, the project complexity necessitated formalization of a collaboration mechanism.

Simulink® provides connectivity to Source Control Management (SCM) tools through Simulink® Projects software. For this project, Subversion® (SVN) was chosen as the SCM tool based on its cost, features, and use throughout the industry. The integration of Simulink® Projects with SVN enables all team members to work within a single environment with a common user interface. A tree-based hierarchical structure in the left navigation pane shows

different *views* of the project. For example, Figure 1 shows the view of the source control management tool configuration. This is a one-time setup done by the administrator of the repository.

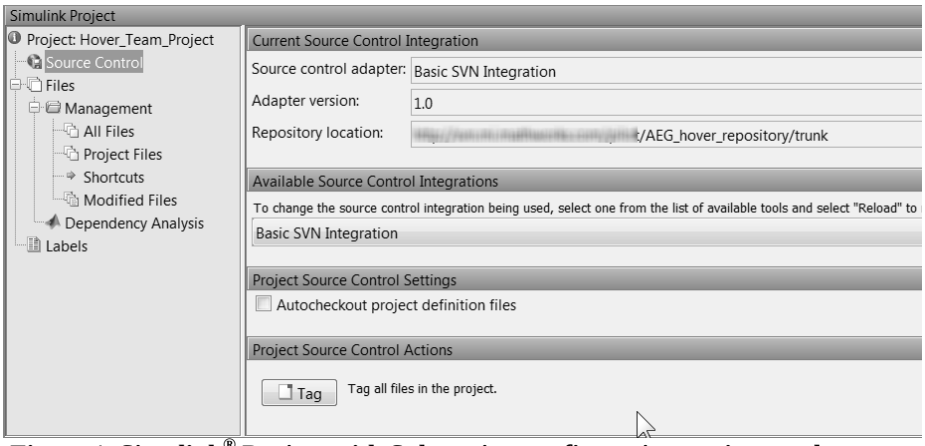

Figure 1. Simulink® Project with Subversion configuration settings such as network location and associated version numbers.

Each team member was able to connect to the same repository and see the same project views, enabling more efficient file synchronization. Centralizing the project files in the repository with an appropriate directory structure helped new engineers join the project quickly because they were able to automatically synchronize their development sandboxes. Further, project-specific environment settings were accessible to all members as MATLAB® utility scripts through the Shortcuts view. Figure 2 shows the Shortcut scripts, which can be set to execute automatically with project startup or shutdown.

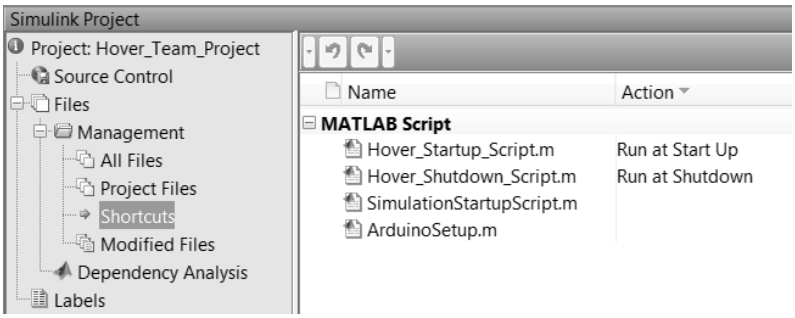

Figure 2. MATLAB® utility scripts for project environment set up accessed through Shortcuts view

The Project Files view shows various attributes, called *Labels*, that can be associated with any file. For example, it is possible to create file attributes such as checkout status, stage of review, revision number, or project member. During the check-in step, a user can change these file attributes to communicate their intentions to other team members who will see them when they update their local copies of the same file. Figure 3 shows the Labels arranged as columns in the Project Files view. Filtering capabilities, which are particularly helpful on large-scale projects such as this, were used to search and manage files based on their attributes.

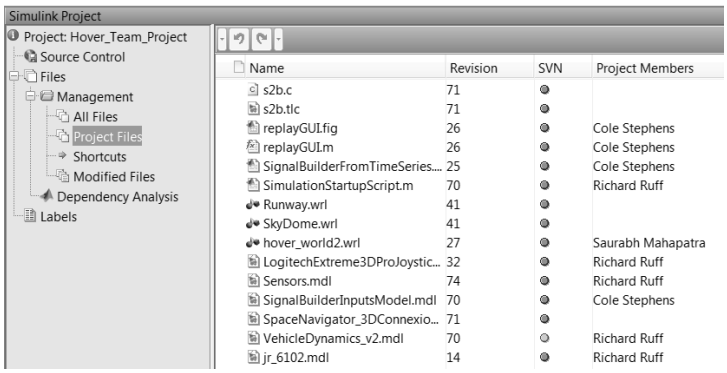

Figure 3. Labels capture various file attributes such as revision number, check-out status from Subversion (SVN), and project members.

C. Componentization for Team Collaboration

Simulink® supports the use of graphical abstraction to create a design hierarchy. Such an approach enables the system-level model to be partitioned into various design components. The criteria for such componentization are often project dependent. As a first step, a single collaborative work environment was established through the project definition outlined in the previous section. However, it was later recognized that componentization helped define boundaries for multiple contributors so that they could work in parallel.

The architecture chosen for the model becomes more important as the size and complexity of the project grows. Being able to break the system into modules allows multiple engineers to collaborate without interfering with each other. Thus it is important to determine the granularity needed for the model architecture based not only on the scale and complexity, but the number of engineers that will be collaborating. As an example, if the project will have six Guidance, Navigation and Control engineers working on it, the flight control might be broken into three separate components – one for each discipline. Further, each discipline might be broken into specific components based on the mode of operation such as takeoff, steady flight, or landing. This allows the team members to work on specific components independently and then integrate the components into the overall system.

The component representation in the software tool itself is a key factor that determines the efficiency [(10)] of the collaborative workflow. For example, Simulink® models containing design components can be stored as a single file. However, such a monolithic approach can result in increased maintenance overhead as the changes from all users would need to be reviewed, merged, and accepted. To alleviate these issues, design components can be represented in separate files that can be referenced in the top-level model. In Simulink®, this is achieved via *libraries* and via *model reference*. In this context, a library is a container for graphical subsystems, each of which can be referenced multiple times in a model. During simulation, each reference is treated as a separate graphical subsystem thereby increasing memory consumption. Thus components that are algorithmically intensive and are referenced often in the model would degrade simulation performance significantly. Libraries are used for components that are lightweight, such as unit conversion and axis transformation utilities. For algorithmically intensive components that were edited frequently and owned by a primary contributor, model reference was used. Unlike libraries which can inherit attributes like signal dimension and data type, a model reference is context independent and requires a strict definition on the I/O and software interface specification. The use of these two modeling constructs led to the need for file dependency analysis in order to understand the system architecture. This is especially useful when new engineers are added to the team. Using the built-in Dependency Viewer in Simulink®, the team generated an architectural view of the file dependencies.

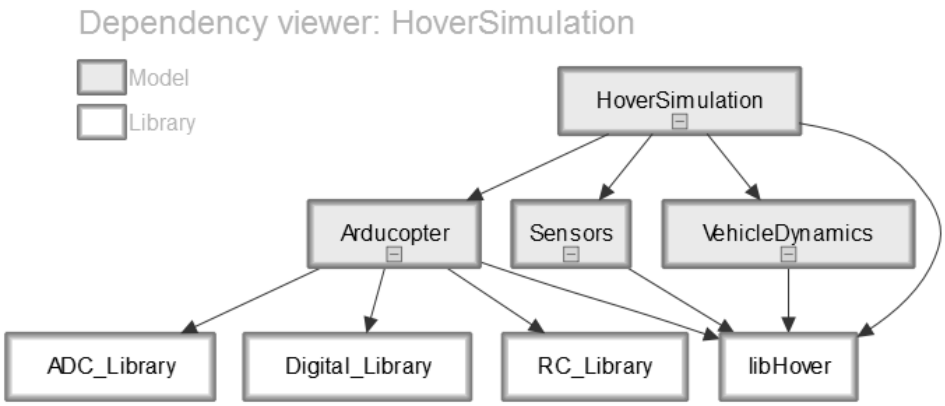

Figure 4. Dependency view of top level model *HoverSimulation*.

In Figure 4, the top level model *HoverSimulation* contains references to three additional models via model reference. The *Arducopter* subsystem model has dependencies on *libHover* and three other libraries, while the other subsystems reference only the *libHover* utility library.

D. Peer Reviews

As the size of the team increased, establishing a collaborative framework that facilitated peer review was essential to the success of this project. Peer review was conducted as part of a typical workflow as shown in Figure 5. Peer review consisted of activities such as:

- Comparing different versions of models and resolving conflicts.
- Generating a System Design Description (SDD) report.
- Exporting web views of the models to share the system architecture with others in the organization.

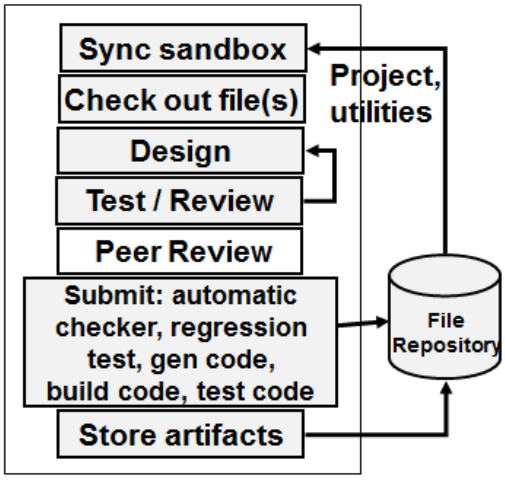

Figure 5. Typical code management workflow

These activities were completed using Simulink Report Generator® [11] [12], which can perform an XML-based comparison of two Simulink® models. This comparison not only notes different blocks used within the model, but also the underlying configuration settings such as the type of solver being used, data logging settings, and code generation settings. Simulink Report Generator® also provides the ability to create the SDD for a model. The SDD contains information on the model such as the I/O signals, the associated data types, and dimensions. The web views of the Simulink® and Stateflow® models provide access to the design to others, such as management, who need to be aware of the overall model but do not actually need to run the simulation. This is important from a licensing standpoint, because it reduces the number of tool licenses needed to just those actively working on the project.

For this project, an understanding between team members [13] for component ownership was established stipulating that, at a given time, only one user would be working on a component. This was done to avoid the costs associated with merging changes from multiple engineers. In a typical workflow, an engineer would make changes to their local version of a file when they were developing new algorithms. If these changes panned out, it would then become necessary for the engineer to commit their model changes to the repository. At this point the model comparison capability was used to compare the engineer's version with the version in the repository. This allowed the engineer to quickly see the changes they had made and confirm that they want to keep all of their changes, but also to confirm that no one else had made any changes to the model since the engineer's last update from the repository. Adopting contractual rules and using a formal process to commit files to the repository reduces the chance that someone's changes are inadvertently lost by being overwritten.

E. Modular Design Platform using Variants

A well-designed modular architecture with shared interfaces and a common core supports the synthesis of multiple design variants by enabling engineers to swap in (and swap out) components with varying attributes or implementations. These *variants* were used in the case study to handle the trade-off between model fidelity and simulation performance. To facilitate pilot-in-the-loop mode, simulation performance near real-time on a standard PC was needed.

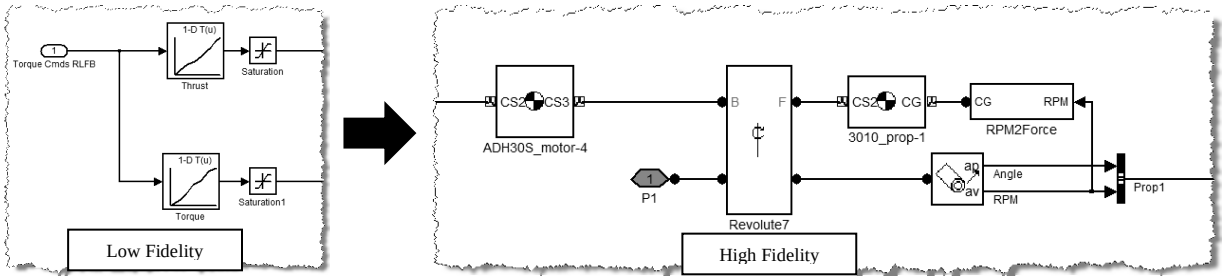

Figure 6. Thrust and torque actuator variants

Figure 6 shows two possible implementations for the plant model describing the DC motors of the system. In the lower-fidelity implementation with better performance, the thrust and force outputs were obtained as a function of the command input in a lookup table. The lookup table was constructed with measured values experimentally obtained from a test rig in the lab. For the higher fidelity model implementation, a first principles model [15] of the applied torque from a DC motor to the propeller was used. From a development standpoint, most components started out as relatively simple "back of the envelope" models that evolved into more sophisticated and more accurate representations of the actual physical system. In this design elaboration, low fidelity components are replaced by more specialized ones. Extensive use of a modeling construct known as Variant Subsystems was used to enable switching between these different implementations within the model. Using MATLAB® scripts, the engineers could programmatically select and simulate the variant of interest among the various choices available. This allows the entire model to be quickly reconfigured—without making changes to the model itself—by changing a few parameters.

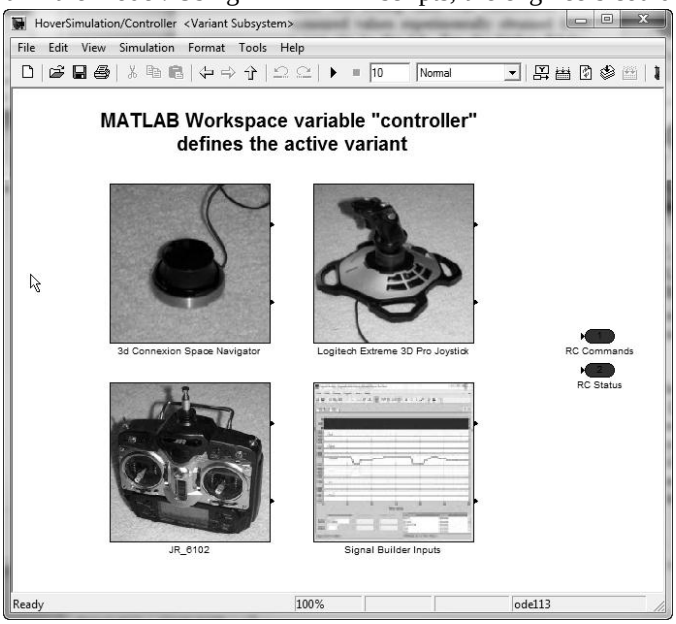

Figure 7. Controller variants

As an example, Figure 7 shows the different variants of the inputs that a user can select based on the type of test being done on the model (from top-left clockwise: 3Dconnexion SpaceNavigator™, Logitech Extreme™ 3D Pro Joystick, Signal Builder block, and JR® XP6102 6 Channel radio transmitter . In addition, Simulink® enables the user to override individual components on a subsystem or reference model basis. For example, the user could specify the entire system to be run in a low-fidelity mode through MATLAB®-based scripting. The user could then specify that the high-fidelity variant of a particular subsystem be used. This approach allows the user to improve simulation performance while concentrating on specific subsystems.

F. Complex Logic Representation

A key architectural design choice was made to represent the entire flight control algorithm in a single state chart in Stateflow®. Figure 8 shows the top-level state chart which uses parallel states to break the flight control algorithm into four components: *Initialization*, *Navigation*, *Guidance*, and *Control*. Parallel states can be active simultaneously unlike exclusive states where only one state can be active such as "on" or "off" states for a light switch.

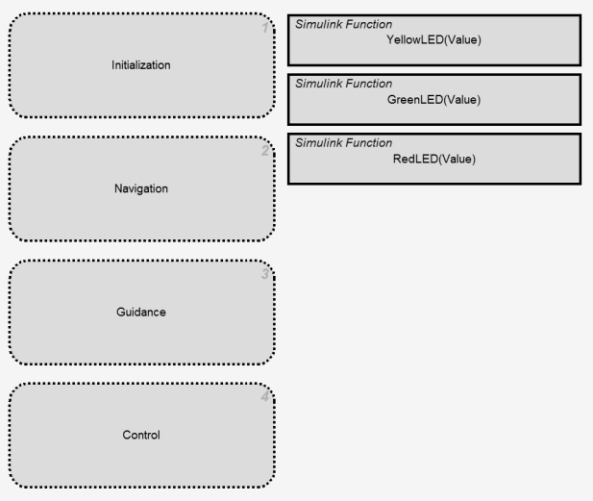

Figure 8. Flight control system state machine

Figure 9 shows the contents of the *Control* subchart. This updates the flight controls and processes the motor commands at every time step when activated. This chart also controls the execution order of the functions associated with the state. The flight control commands will be updated first via the *Controls* function, followed by the motor actuator commands via the *Motor_Command_Processing* function.

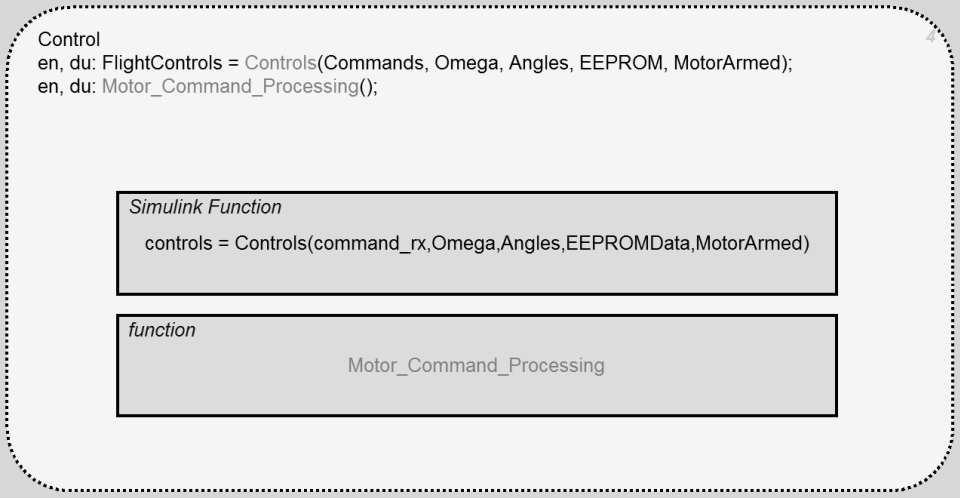

Figure 9. Contents of the control state in the flight control system state machine

Note that *Controls()* is a Simulink® function, a construct that allows a Simulink® algorithm to be embedded within the state chart. This allows the modeler to create a state-chart-centric view of the design as is required in the design of scheduling algorithms. Figure 10 shows the contents of the Simulink® function in Figure 9.

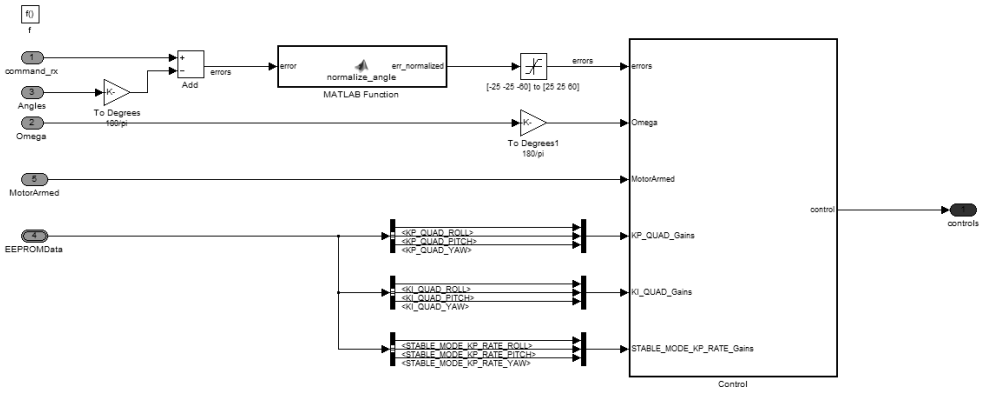

Figure 10. Simulink® function of Controls as a block diagram

A key lesson learned on the project was that choosing an appropriate modeling technique for each design task leads to greater efficiency. The ability to represent control flow with state chart semantics alongside algorithmic representation with block diagrams within a single modeling environment helped simplify a complex system model.

III. Testing: Simulation, Analysis, and 3D Visualization

Simulation in Simulink® provided several key benefits over the course of development such as evaluating new design ideas, verifying behavior, testing algorithms, validating functional requirements, and enabling pilot-in-the-loop interaction.

A. Execution

The definition of *execution* varies within different tool chains. In the context of this paper and the use of Simulink®, it refers to a time-marching solution of a set of mathematical expressions and differential equations. These represent the dynamics of a system or an algorithm whose behavior changes with time. This definition requires more computational power than an equivalent set of simple data calculations.

One of the key benefits of the approach used on this project is that each model reference in the system is itself a complete model that can be executed. This enables the engineer responsible for that component to test and verify the behavior using representative sets of inputs independent of the complete system. Once the engineer is satisfied with the performance, the component can be included in the overall system simulation to test interactions between components.

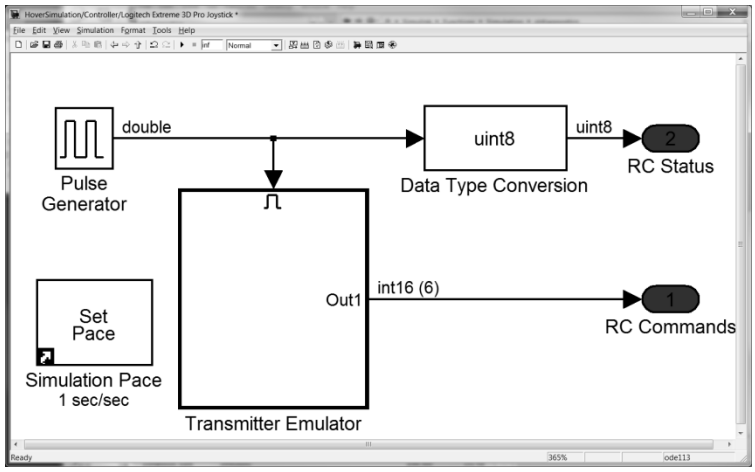

Figure 11. Joystick model with a Simulation Pace block

It was observed that for the full system model, simulation using predefined test cases [16] defined via the Signal Builder block was faster than real-time. In other cases, the simulation and visualization must occur in near-real-time, for example when running a pilot-in-the-loop simulation with an external device such as a joystick for input commands. Figure 11 shows how a joystick is modeled to generate the inputs to the system to emulate the commands seen by the radio receiver on the actual hardware. The *Simulation Pace* block was used to synchronize the simulation such that one second of simulation time would correspond roughly to one second of actual clock time. As this block is only used in the pilot-in-the-loop variants, simulations for other variants could execute faster than real time. This

enabled the engineer to easily adapt the simulation to various needs: slowing it down when using manual input commands and executing as fast as possible when running specific test cases with internally defined inputs.

B. Large-Scale Simulation Data Analysis

Understanding the simulation results was critical to the success of the project. Two methods were used: Simulink® scopes and logged signals. Scopes offer quick and easy access to simulation data from signal lines. Logging streams the signal data to a MATLAB® variable for later post-processing.

For a large-scale system, managing the simulation data can be a challenge. For analysis, certain signals were tagged to enable tracking and review. To compare different runs, it became necessary to use the logging capability instead of relying on scopes because scopes are reset for each simulation run. Figure 12 shows options to specify on a signal line so that the data is logged during the simulation into the MATLAB® workspace.

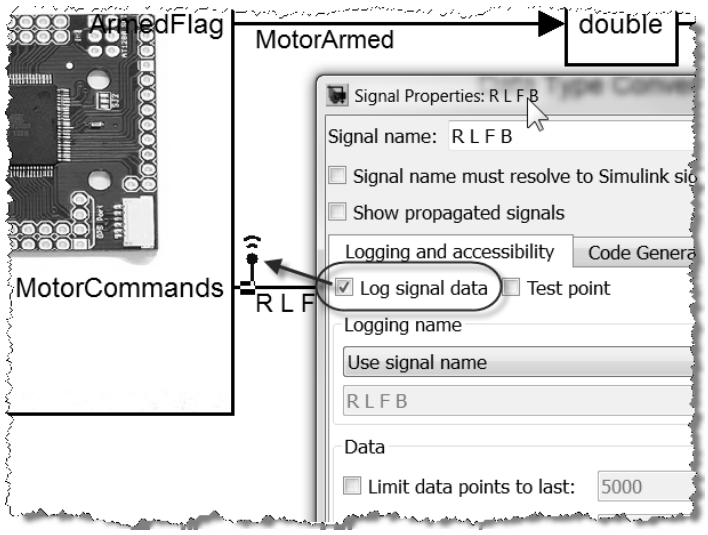

Figure 12. Options for enabling logging of signals

Once these signals were logged, the Simulation Data Inspector tool was used to inspect the logged signals following each simulation. This tool, shown in Figure 13, offers a mechanism to manage the logged data from the various signals within multiple simulation runs. When changes are made to models and simulation parameters, another simulation run is made and the new data is recorded. This new data can then be compared against baselines or previous runs. As an example, the user can quickly determine the effect of modifying the controller gains to determine if the overshoot improved, evaluate changes in the settling time, and so on.

As noted previously, a joystick was used in a pilot-in-the-loop simulation to let the pilot experience the vehicle behavioral response to commands. These handling quality tests were subjective in nature and required simulation tuning for improving each pilot's experience.

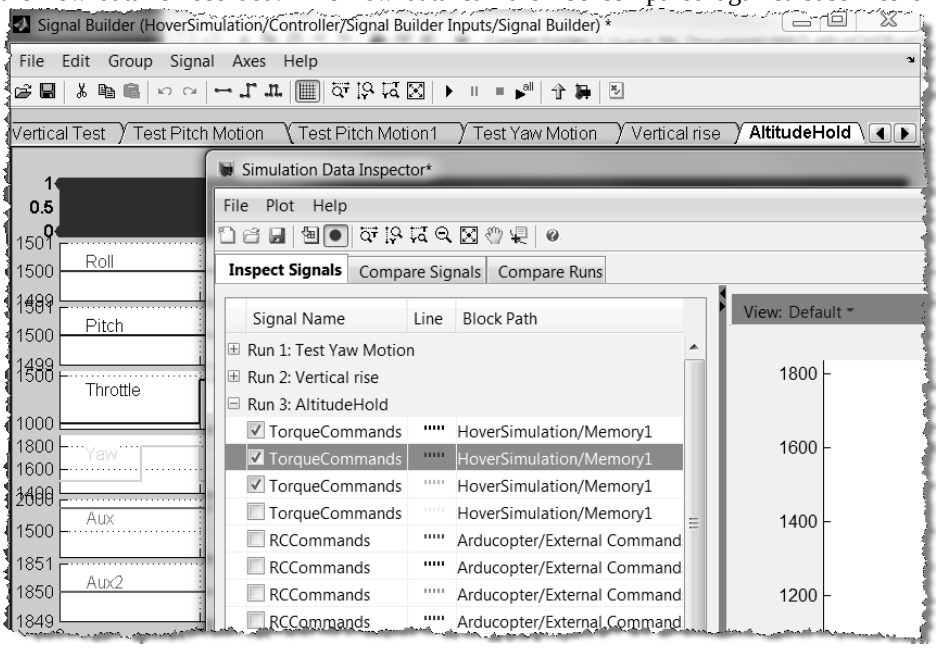

Figure 13. Signals across simulation runs in Simulation Data Inspector

C. 3D Visualization

3D visualization was implemented using Simulink 3D Animation™ [17] [18]. Figure 14 shows a screen capture of the 3D visualization. The 3D scene was authored in Virtual Reality Modeling Language (VRML) and assembled using exported CAD images, Google Earth images, and camera images of the MathWorks campus in Natick, MA. Output

simulation data of the vehicle's states from the Simulink® model was input into a VR Sink block that interfaces to the VRML scene. The polygonal optimizations of the scene led to a realistic 3D rendering of the simulation in near-real-time for pilot-in-the-loop simulations.

For simulations where simulation performance was a priority, the 3D animation was not used. Instead, simulation outputs were analyzed through 2D visualization mechanisms such as Simulink® scopes or Simulation Data Inspector.

The combination of scopes, logged data, and 3D animation provided a powerful tool to understand the behavior of the system. Combining these methods provides insights that are not available with just one or two of these methods. Looking at a scope, the user cannot compare multiple simulations; comparing plots of the vehicle attitude does not lend itself to understanding the motion of the vehicle; and 3D animation does not help the user to understand mode transitions in the flight software. Together, however, these methods enable the engineer to understand all of these aspects more fully.

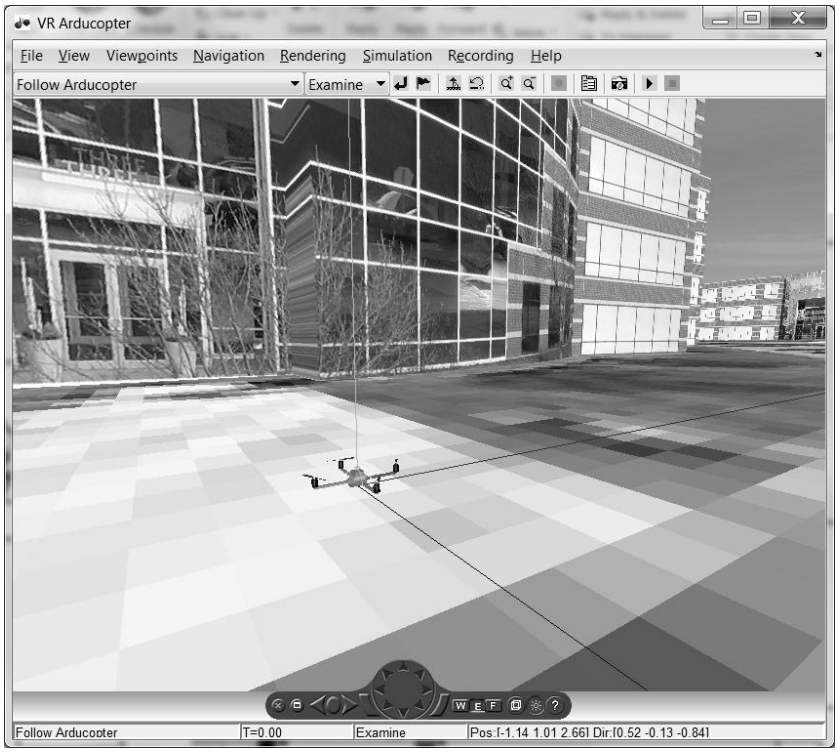

Figure 14. 3D Scene of the Quadrotor on MathWorks Campus

IV. Controller deployment on physical hardware with flight testing

A. Choice of Physical Hardware

As a test bed, the ArduCopter hardware shown in Figure 15 was chosen. This is commercial-off-the-shelf hardware available from jDrones (http://store.jDrones.com). It makes use of an ArduPilot Mega 1.0 board with an Arduino® ATmega2560 processor. This is paired with the ArduPilot Mega IMU (Inertial Measurement Unit) Shield. This IMU has a sensor suite comprising an Analog Devices ADX330 accelerometer, an InvenSense triple axis gyroscope, a Bosch® Sensortec BMP085 digital pressure sensor, a Honeywell® HMC5883L triple axis magnetometer, and a MediaTek® MT3329 GPS system. A Maxbotix® LV-EZo Ultrasonic Range Finder sensor was added to the vehicle for measuring altitude. To capture the flight data, a Logomatic v2 Serial SD Datalogger board was used. The current implementation of the control laws uses accelerometer, gyroscope, and range finder data.

Figure 15. ArduCopter

B. Flight Control Software (FCS) Development

With the ArduCopter hardware platform, existing open source ArduCopter software was used to streamline the flight control development. This choice provided existing algorithms in the code base as a baseline for comparison during flight testing. The existing open source algorithmic code was re-implemented in Simulink® and Stateflow® and tested via simulation. Stateflow® was used to control the overall flow of the FCS as described in the Complex

Logic Representation section. This is shown schematically in Figure 8. The *Initialization* state carried out basic calibration during system startup or reset.

C. Embedded Software Generation

Following the implementation and simulation testing of the FCS in Simulink®, the development team needed to convert it to C code targeted for the Arduino® ATmega1280 or ATmega2560 processor. Note that the ArduPilot Mega 1.0 is currently available with either processor. To accomplish this, automatic code generation in Simulink® was used with the Arduino® target package available for download from MATLAB® Central (http://www.mathworks.com/matlabcentral/fileexchange/30277-embedded-coder-support-package-for-arduino). To interface directly with the ArduCopter hardware, additional low-level C-authored S-function interface blocks were created in Simulink®. These are shown as gray blocks in Figure 16. These driver blocks capture the data from accelerometers, gyros, sonar range finder, and the receiver as inputs to the FCS and send the resulting commands to the engine speed controllers. For this implementation, these blocks have no impact on simulation and act as a simple pass-through of the simulation data (Refer to **Appendix A**: ***Source code for the ADCRead S-function (Arduino_ADCRead_sfcn.c)*).** However, they result in hardware interface function calls being inserted into the generated code. This is accomplished by the creation of a custom Target Language Compiler TLC file for each S-function block. The TLC file specifies how the C code is generated for a given S-function (Refer to **Appendix B**: ***Source code for the ADCRead tlc file (Arduino_ADCRead_sfcn.tlc)***).

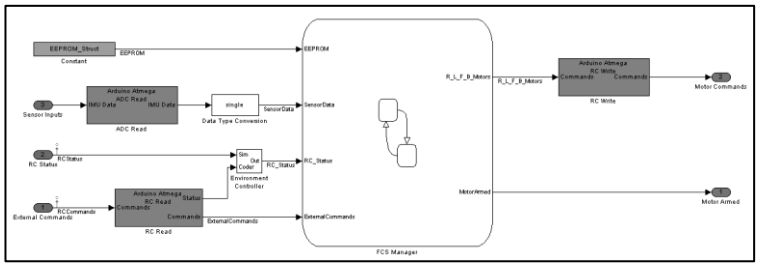

Figure 16. Flight Control Software Model

Using the Arduino® target with the Arduino® IDE (arduino-022 for this project) and the custom interface blocks, machine code for the FCS can be automatically generated and downloaded directly to the ATmega1280/2560. The target processor is connected to the development computer via a USB interface cable. The Arduino® target package provides the necessary tools for compiling and linking the generated source code with the Arduino® IDE libraries to a hex file and deploying it to the ATmega1280/2560 via serial communication. Thus, the use of custom interface blocks with the Arduino® target package's support for generation of the main function eliminates manual integration of the generated C code into a hand-written code framework. This leads to faster iterations and enables the use of a single development environment. Furthermore, this also enables processor-in-the-loop testing, which helps determine processing margins early in the design process. Additionally, it can be used to study the effects of processing platform differences, such as word sizes, and how they affect the performance on a 64-bit PC versus an Atmel processor.

D. Deployment with Flight Testing

The testing of algorithms on the actual hardware is the final step of deployment. Initially, only the IMU data was used for the FCS. While simulation of the control algorithm showed this was adequate to control the vehicle], flight testing indicated that this approach required significant effort on the part of the pilot to maintain control. To alleviate this issue, a sonar altimeter was incorporated into the vehicle and the FCS was updated to use the new input data. Despite this change late in the process, it was implemented rapidly. The iterative model-to-code workflow enabled the engineers to regenerate the FCS code and download it on the hardware. Subsequent flight tests showed significant improvement in the stability of the vehicle in maintaining altitude within the range of the sensor. This iteration was accomplished in a few hours compared to days typically required when using handwritten and manual code integration processes. Also, these late changes were readily available to other team members due to their storage in the Subversion repository.

VI. Conclusion

Collaboration among various engineering disciplines during large-scale system development is supported by a workflow centered on Model-Based Design. Return on investment can be improved if the chosen development tool supports critical design engineering capabilities, including:

- The ability to create an architectural specification of the complete system that facilitates comprehension at different levels of fidelity.
- The ability to integrate knowledge and expertise from engineers in multiple disciplines.
- The ability to support collaborative workflows.
- The ability to simulate complete system behavior, conduct testing to verify requirements, and automatically generate code for implementation on hardware.

The use of a single tool further improves the communication among teams of engineers through the expression of and sharing of design ideas within a common design environment. Such an approach accelerates the design process by eliminating the need to translate a design expressed in one environment into another. Continuous testing and verification improves the robustness of the design while reducing development time. As a result, Model-Based Design enables engineering teams to adhere to the axiom "Test what is flown; fly what is tested".

Appendix

A. Source code for the ADCRead S-function (Arduino_ADCRead_sfcn.c)

```
/*
 * sfuntmpl_basic.c: Basic 'C' template for a level 2 S-function.
 *
 *  -------------------------------------------------------------------------
 *  | See matlabroot/simulink/src/sfuntmpl_doc.c for a more detailed template |
 *  -------------------------------------------------------------------------
 *
 * Copyright 1990-2002 The MathWorks, Inc.
 * $Revision: 1.27.4.2 $
 */


/*
 * You must specify the S_FUNCTION_NAME as the name of your S-function
 * (i.e. replace sfuntmpl_basic with the name of your S-function).
 */

#define S_FUNCTION_NAME  Arduino_ADCRead_sfcn
#define S_FUNCTION_LEVEL 2

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */
#include "simstruc.h"

//#define CH(S) ssGetSFcnParam(S,0)

/* Error handling
 * --------------
 *
 * You should use the following technique to report errors encountered within
 * an S-function:
 *
 *       ssSetErrorStatus(S,"Error encountered due to ...");
 *       return;
 *
 * Note that the 2nd argument to ssSetErrorStatus must be persistent memory.
 * It cannot be a local variable. For example the following will cause
 * unpredictable errors:
 *
 *      mdlOutputs()
 *      {
 *         char msg[256];         {ILLEGAL: to fix use "static char msg[256];"}
 *         sprintf(msg,"Error due to %s", string);
 *         ssSetErrorStatus(S,msg);
```

```
 *         return;
 *       }
 *
 * See matlabroot/simulink/src/sfuntmpl_doc.c for more details.
 */

/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl_doc.c for more details on the macros below */

    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }
//    ssSetSFcnParamTunable(S,0,true); /* make the channel tunable */

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 7);
    ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
    ssSetInputPortDataType(S, 0, SS_INT16);
/*
     * Set direct feedthrough flag (1=yes, 0=no).
     * A port has direct feedthrough if the input is used in either
     * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
     * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
     */
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 7);
    ssSetOutputPortDataType(S, 0, SS_INT16);


    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    /* Specify the sim state compliance to be same as a built-in block */
    ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE);

    ssSetOptions(S, 0);

    ssSetModelReferenceNormalModeSupport(S, MDL_START_AND_MDL_PROCESS_PARAMS_OK);
}



/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
```

```
//    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);

}

#define MDL_SET_WORK_WIDTHS   /* Change to #undef to remove function */
#if defined(MDL_SET_WORK_WIDTHS) && defined(MATLAB_MEX_FILE)
/* Function: mdlSetWorkWidths =================================================
 * Abstract:
 *      Set up run-time parameters.
 */
static void mdlSetWorkWidths(SimStruct *S)
{
//    const char_T    *rtParamNames[] = {"Channel"};
//    ssRegAllTunableParamsAsRunTimeParams(S, rtParamNames);
}
#endif /* MDL_SET_WORK_WIDTHS */


#define MDL_PROCESS_PARAMETERS   /* Change to #undef to remove function */
#if defined(MDL_PROCESS_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlProcessParameters =============================================
 * Abstract:
 *      Update run-time parameters.
 */
static void mdlProcessParameters(SimStruct *S)
{
    /* Update Run-Time parameters */
    ssUpdateAllTunableParamsAsRunTimeParams(S);
}
#endif /* MDL_PROCESS_PARAMETERS */


#define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
#if defined(MDL_INITIALIZE_CONDITIONS)
  /* Function: mdlInitializeConditions ========================================
   * Abstract:
   *    In this function, you should initialize the continuous and discrete
   *    states for your S-function block.  The initial states are placed
   *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
   *    You can also perform any other initialization activities that your
   *    S-function may require. Note, this routine will be called at the
   *    start of simulation and if it is present in an enabled subsystem
   *    configured to reset states, it will be call when the enabled subsystem
   *    restarts execution to reset the states.
   */
  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */



#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
  static void mdlStart(SimStruct *S)
  {
  }
#endif /*  MDL_START */



/* Function: mdlOutputs =======================================================
 * Abstract:
```

```
 *    In this function, you compute the outputs of your S-function
 *    block.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
//    uint8_T *p1 = (uint8_T *) ssGetRunTimeParamInfo(S, 0)->data;
    const int16_T *u = (const int16_T *) ssGetInputPortSignal(S,0);
    int16_T       *y = ssGetOutputPortSignal(S,0);
    int count;
    for (count = 0; count < 7; count++)
    y[count] = u[count];
}



#define MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate ======================================================
   * Abstract:
   *    This function is called once for every major integration time step.
   *    Discrete states are typically updated here, but this function is useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */



#define MDL_DERIVATIVES  /* Change to #undef to remove function */
#if defined(MDL_DERIVATIVES)
  /* Function: mdlDerivatives =================================================
   * Abstract:
   *    In this function, you compute the S-function block's derivatives.
   *    The derivatives are placed in the derivative vector, ssGetdX(S).
   */
  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */



/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
}


/*=============================================================*
 * See sfuntmpl_doc.c for the optional S-function methods *
 *=============================================================*/

/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

B. Source code for the ADCRead tlc file (Arduino_ADCRead_sfcn.tlc)

```
%% File : Arduino_ADCRead_sfcn.tlc
%%
%implements Arduino_ADCRead_sfcn "C"

%% Function: BlockTypeSetup ===================================================
%function BlockTypeSetup(block, system) void

  %% Ensure required header files are included
  %if EXISTS("_DONE_ADC_BLOCK_TYPE_SETUP_") == 0
    %assign:: _DONE_ADC_BLOCK_TYPE_SETUP_ = 1
    %if (CompiledModel.TargetStyle!="SimulationTarget")
      %<LibAddToCommonIncludes("ArduinoTargetHeaders.h")>
    %endif
  %endif
%endfunction

%% Function: Start ============================================================
%%
%function Start(block, system) Output
  %% Only insert the init once
  %if EXISTS("_DONE_ADC_INIT_") == 0
    %assign :: _DONE_ADC_INIT_ = 1
    /* %<Type> (%<ParamSettings.FunctionName>): %<Name> */
  ADC_Init();
  %endif
%endfunction

%% Function: Outputs ==========================================================
%%
%function Outputs(block, system) Output

  %if (CompiledModel.TargetStyle!="SimulationTarget")
       %<LibBlockOutputSignal(0, "", "", 0)> = ADC_Read((uint8_T)1); /* Gyro X */
       %<LibBlockOutputSignal(0, "", "", 1)> = ADC_Read((uint8_T)2); /* Gyro Y */
       %<LibBlockOutputSignal(0, "", "", 2)> = ADC_Read((uint8_T)0); /* Gyro Z */
       %<LibBlockOutputSignal(0, "", "", 3)> = ADC_Read((uint8_T)4); /* Accel X */
       %<LibBlockOutputSignal(0, "", "", 4)> = ADC_Read((uint8_T)5); /* Accel Y */
       %<LibBlockOutputSignal(0, "", "", 5)> = ADC_Read((uint8_T)6); /* Accel Z */
       %<LibBlockOutputSignal(0, "", "", 6)> = ADC_Read((uint8_T)7); /* SONAR */
  %endif
%endfunction

%% [EOF]
```

Acknowledgments

The authors would like to acknowledge the significant contribution and inspiration of Richard A. Benson, Consulting Application Engineer at MathWorks.

References

[1]Barnard, P. *Graphical Techniques for Aircraft Dynamic Model Development.* Rhode Island : AIAA Modeling and Simulation Technologies Conference and Exhibit, August 2004.

[2]Barnard, P. *Software Development Principles Applied to Graphical Model Development.* San Francisco : AIAA Modeling and Simulations Conference and Exhibit, 2005.

[3]Turevskiy, A, Gage, S and Buhr, C. *Model-Based Design of a New Light-weight Aircraft.* South Carolina : AIAA Modeling and Simulation Technologies Conference and Exhibit, 2007.

[4]Lin, J. Measuring Return on Investment of Model-Based Design. *EE Times.* May 2011.

[5]Aberg, R and Gage, S. *Strategy for Successful Enterprise-Wide Modeling and Simulation with COTS software.* Rhode Island : AIAA Modeling and Simulation Technologies Conference and Exhibit, August 2004.

[6]MathWorks. *Simulink User's Guide.* Natick : MathWorks, 2012.

[7]Ledin, J, Dickens, M and Sharp, J. *Single Modeling Environment for Constructing High-Fidelity Plant and Controller Models.* : AIAA, 2003.

[8]MathWorks. *SimMechanics User's Guide.* Natick : MathWorks, June 2012.
[9]MathWorks. *Stateflow User's Guide.* Natick : MathWorks, June 2012.
[10]Anthony, M and Friedman, J. *Model-Based Design for Large Safety-Critical Systems: A Discussion Regarding Model Architecture.* San Diego : AUVSI's Unmanned Systems North America, 2008.
[11]Mahapatra, S. *Automatic Report Generation in Model-Based Design.* Chicago : Society of Automotive Engineers (SAE) Commercial Vehicle Engineering Congress, 2010.
[12]MathWorks. *Simulink Report Generator User's Guide.* Natick : MathWorks, June 2012.
[13]Walker, G, Friedman, J and Aberg, R. *Configuration Management of the Model-Based Design Process.* Detroit : Society of Automotive Engineers (SAE) World Congress, 2007.
[14]Mahapatra, S. *Enabling Modular Design Platforms in Model-Based Design.* Portland : AIAA Modeling and Simulation Technologies Conference and Exhibit, 2011.
[15]Denery, T, et al. *Creating Flight Simulator Landing Gear Models Using Multidomain Modeling Tools.* Keystone, Colorado : AIAA Modeling and Simulation Technologies Conference and Exhibit, 2006.
[16]Ghidella, J and Mosterman, P. *Requirements-based testing in aircraft control design.* San Francisco : AIAA Modeling and Simulation Technologies Conference and Exhibit, 2005.
[17]Mahapatra, S. *Enhancing Simulation Studies with 3D Animation.* Portland : AIAA Modeling and Simulation Technologies Conference and Exhibit, 2011.
[18]MathWorks. *Simulink 3D Animation User's Guide.* Natick : MathWorks, June 2012.