New Eagle LLC

MotoTron Controls Solutions

Simulink Style Guide

Version 1.0

5-12-2010

Authored by Rick Poorman with Input from New Eagle, Woodward, & MotoTron Employees, Past and Present

(A Guide to MotoHawk Programming)

Contents

Document Purpose 3
Background 4
Project File System and Data Organization Rules 6
Vardec Rules 10
Element Usage Rules 11
Code Clarity Rules 15
Drawing Style Guidelines 19
Process Rules 24

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Document Purpose

The purpose of this document is to provide a set of implementation guidelines for New Eagle employees and customers who use Simulink with MotoHawk. These are guidelines and in no way are required to make Simulink or MotoHawk function as a method to generate operational control code. Rather, they are intended to better enable a system designer to use the same source code diagram to communicate his/her ideas to a human audience. This document sets rules that facilitate the reading and understanding of the algorithms encapsulated in the Simulink/MotoHawk model.

Along with the above, new sections capturing a few rules that help best utilize the code generation of Real-Time-Workshop are added.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Background

Simulink is a simulation tool originally developed for system modeling, serving as a graphical front-end for the MATLAB numerical analysis environment. In recent years, Simulink has also been successfully employed as a design language for controls rapid prototyping. Industry-wide attention is now focused on extending the usefulness of graphical design assets for production software development. Simulink diagrams are now used for:

- **Documentation of the code intent**
- Design verification through interpreted and compiled simulation
- Off-nominal system performance analysis
- Code generation for controls rapid prototyping
- Code-generation for hardware-in-loop testing
- Production software specification
- Production software development & code generation

In addition to being a pictorial algorithm design and simulation document, a Simulink diagram with associated MATLAB workspace objects, provide the information to auto-code an executable. This means that a single, properly designed, diagram may be directly executed in the non-real-time simulation environment, in real-time on a prototyping platform, (such as dSpace, PXI, ADIRTS, XPC) or on an embedded target. It is important to note that the Simulink drawing alone is generally insufficient to create a truly executable specification. Additional quantities such as constants (calibratables) used by a diagram need to be defined for a diagram to be executable. This is analogous to the separation between program and data.

As with any textual programming language, graphical programming languages have interpreter or compiler enforced syntax. When syntax errors are introduced in Simulink diagrams, execution halts with an error message just like any other programming language. Syntax "ambiguity" arises when diagrams are used for purposes beyond which they were originally intended. Since Simulink was originally focused on non-realtime simulation, syntax checking is limited to insuring that task can be accomplished. As the utility of diagrams is extended to a broader class of application, it is useful to place additional constraints (syntax) on the diagrams. Examples include:

- **Designing the diagram as its own documentation such that the meaning is clear to a human reader as well as the code generator**
- Avoiding blocks or structures unsupported by code generators
- Using blocks or structures supported by rapid prototyping tools
- Creating drawings in a consistent fashion for clarity and readability
- Using diagram documentation in a way that enhances readability of the diagrams and any derived code

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

The collection of rules that govern model diagram creation and modification for all desired are referred to collectively as a "style guide." Style guides vary necessarily according to the specific processes and tool chains that must be supported in an organization. Despite this, we have observed and applied a number of best practices that we believe work across processes and toolchains. We offer these as the “New Eagle-MotoTron Controls Solutions - Simulink Style Guide."

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Project File System and Data Organization Rules

Rule: All diagrams, files, libraries, specifications, etc. specifying the design of a particular project model shall be organized in a project “root” subdirectory for that project containing the model, open, close, and other functions, as well as folders for needed support files.

Rationale: Complex designs are difficult to manage at the file-system level. Co-locating design files in a common directory makes identification of key assets straightforward. This can also be zipped and shared as a complete representation of the entirety of the project. The only acceptable substitute is a full configuration management system such as might be available using Clear-Case or a similar product.

Rule: A project subdirectory shall have several subdirectories of its own. These will include: CAN, Libraries, Images, Components, m_files, etc., as necessary for storing CAN message definition files, additional library elements used by the project model, images used in model masks,Motohawk Component Files, other m-file functions, etc. These will be added to the MATLAB path at the opening of the model file and used by the project.

Rationale: This organization is neat, clean, and readies itself to effective MATLAB path management (the process by which MATLAB tracks the location of files it uses). This subdirectory structure lends itself to automation via the _open.m and _close.m . path management files that motohawk_project places in the root project subdirectory during creation of a project.

Note: CAN, Images, and Libraries are created by the motohawk_project function. Other path management additions are made by editing the _open.m file.

Rule: Where reuse of code is desired, the model (.mdl file), function or script (.m files) shall be placed in libraries, components, or zipped project structure.

Rationale: Libraries have the same structure as those created by motohawk_project. They may, but do not have to contain the same file structure as any MotoHawk project. Thus they can contain any MATLAB or Simulink file type. The library organization and sharing will facilitate reuse of good model elements and m-files. Likewise, the component acts as an encrypted form of a library. The .mhc file contains a zipped record of all necessary files to use the component in a build.

Rule: Blocks with reuse potential shall be placed in library files

Rationale: Some Simulink constructions are so pervasive throughout a design that they deserve special attention. Just as the default Simulink block library (Simulink.mdl) is available using a single command in MATLAB, a useful collection of custom model elements can be a true asset to an organization. Devices like PIDs, filters, governors, common nonlinear elements, etc. appear again and again in designs. Common elements should be placed in purpose built libraries that are maintained by projects or individuals and made available to the design team, or throughout MotoTron via the web site. Related elements may be grouped in a common library.

Rule: It is acceptable to nest library links.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rationale: It is acceptable to decompose subsystem element libraries down into the linkages between more elementary library elements. At the lowest level, a Simulink library will contain a mixture of standard Simulink blocks and, perhaps, reusable blocks from one or more common element libraries maintained within the organization.

Rule: Where multiple people work a given project, if no configuration management tool is in place, one person is designated as “having the baton” and all other people must check files in and out from the baton holder. Each time a new file is checked in, a new copy of the complete and updated project sub-directory is distributed.

Rationale: This practice, in effect, enforces much of the discipline of configuration management, even without a tool.

Rule: When library elements are used, the link between the library element in the model and the library shall be maintained, unbroken, where possible.

Rationale: Since the libraries are contained within a project directory tree, they are under control of the project and only updated when the project (person with the baton, see next rule below) deems appropriate. Thus any use of that project library within a project can be maintained by updating only a given library.

Rule: Links to reusable device libraries shall not be broken permanently where the fundamental purpose of the block is not changed. If something in the block needs to be changed to “fix” the block, these changes may be pushed back into the library if the link is disabled, but not broken or the library version may again be employed. Instead, sever, but do not break the link to maintain a tie with the library because if the link is fully broken, use of the reversible option is not possible. The appearance of the block should be altered to indicate that it is no longer being maintained by library updates but still maintains loose ties.

Rationale: In general, links are “broken” in order to affect some sort of modification. Confusion can result when changes are made to a local copy of what would outwardly appear to be a common library element. At execution time, this can result in difficult to explain variance in design behavior. Breaking such links can also make it impossible to effect improvements and bug fixes throughout a design by correcting problems at the source, in the original library. In the event that a link must be broken to enable a design change, this generally points to a need to consider updating the original library entry or (perhaps) adding a new variant in the library. The idea is to facilitate updating library elements where possible, but make it very clear where no update will take place.

Rule: If the function of a library block is to be changed from its original function, then all ties to the original source material in a library shall be broken. This means, ‘disable’, then ‘break’ the library link making the code stand on its own. It is also required that the original appearance of the block be altered to avoid confusion as to its origin. If this new block is to be used, place it in a library as a new block type.

Rationale: When one alters the function of a library block, it is no longer the library block from which it originated, but a new creation. Thus it needs to reflect this reality. If the new creation is

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

to be used and resaved, then move a copy of the new and altered block to the library and save it. Then replace your altered block with the new library element. This keeps both the library and your model in good form and eliminates the possibility that wrong code may be used by others reusing your project work or obtaining an altered (now with new stuff) library from you.

Rule: Simulink model files shall have the .mdl name extension.

Rationale: This is a standard extension that MATLAB expects to find. Most modern operating systems will attempt to associate .mdl files with MATLAB/Simulink. A typical design model name might be system.mdl .

Rule: Simulink library filenames shall have a _lib.mdl extension.

Rationale: Complex designs are difficult to manage as a single, large, flat .mdl file especially where multiple people collaborate on design. Simulink includes a facility for reusable library entries, which we encourage. We recommend that system level designs be composed of collections of libraries for reuse and ease of collaborative development. Libraries are not treated the same as flat .mdl files, and it is important to clearly differentiate these important file types.

Since Simulink does NOT have a unique filename extension for libraries, using a suffix of '_lib' on the root file name accomplishes this goal. For example, foo_lib.mdl would be a library element that could be used as part of a larger design called system.mdl.

Rule: For a device library, there shall be 1 (one) block at the top level of that library diagram which is the complete and usual implementation of that device's model. Additional blocks shall be designated as lesser and have their limited function described.

Rationale: Subsystem parts that comprise a larger design should be saved by themselves as a library project file tree. By maintaining a 1 to 1 correspondence between a device and its library subsystem file, locating relevant files for modification or study is greatly simplified. The exception to this guideline is signal processing libraries as discussed in the following rule.

Rule: The high level system diagrams shall contain a hierarchy organization and may contain enclosed subsystems, library links and MotoHawk components, and software organization.

Rationale: A top level system design that is decomposed into clearly defined elements for collaborative development will be a diagram containing a typical project organization, with nothing but organizational subsystems, links to libraries, MotoHawk components, and the interconnections between them. Broad use of Simulink logic in the top level diagram is discouraged for complex system.

Rule: Utility scripts that operate on diagrams for information querying, automated diagram checking and modification, automated printing, etc. shall be placed in a Library subdirectory and included with the project.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rationale: Just as certain Simulink elements are commonly used and can be organized into a central repository for the benefit of an organization, so can MATLAB utility scripts. It is common for engineers to create utilities that parse through diagrams for information, perform additional syntax or style checking, and even automatically modify diagrams. Utilities that are of sufficient value should be placed in a central repository and maintained as part of a consistent model-based development process.

Rule: Editing Simulink within a Library is risky, unwise and is thus discouraged. Library elements should remain at all times functional and saved.

Rationale: Simulink has different rules within the confines of a library file as compared to a model file EVEN THOUGH it has the .mdl extension. These differences involve model update, simulation, and other properties. It is for this reason that to properly edit library elements, they should be brought out to a model file, edited, tested, and then reintroduced back into the library and should be immediately saved. This protects all project model files using the library and is a further protection against a MATLAB crash ruining anything other than immediate work on the one item being edited.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Vardec Rules

Rule: Choose whether the project will use floating or fixed point math up front.

Rationale: This choice decides whether the use scaling and offsets in calibrations, probes, elsewhere in MotoHawk is even possible and whether fixed point versions of these blocks should be employed.

Rule: Thou shall fill in Vardec help, units, min and max values, enumerations, etc. when you create the variable or probe.

Rationale: These are extremely helpful in working with the software in MotoTune, and if not filled in at creation, are too often overlooked. The further reason is one of maintainability. It is far easier to share software or maintain it years later if properly commented.

Rule: Minimize the use of offsets and scaling in Vardecs except as automatically generated by the fixed-point versions of the blocks.

Rationale: While available for completeness, and very much necessary for fixed-point design work, they are rarely required in floating point designs. Because vardec offset and scaling tend to obscure function, it is preferred to use regular Simulink blocks to show scaling and offsets on a signal before entry into the probe, for example.

Rule: Choose descriptive, complete, vardec names.

Rationale: It is important to fully describe which signal is being represented. Along with the signal, the units representation is equally important and should be included. In an engine model, is “RPM” cam or crank speed?....EngineCrankSpeed_rpm or EngCamSpd [rad/s] are both better as they describe what speed is being represented and in what units it is encoded.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Element Usage Rules

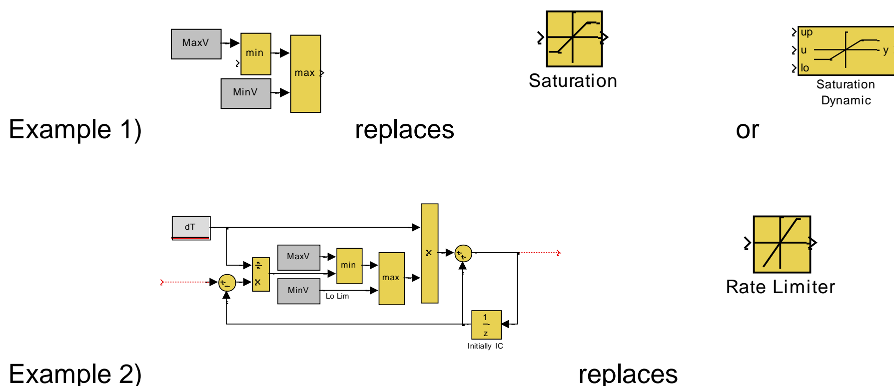
Rule: The following Simulink blocks shall only **BE USED WITH EXTREAM CAUTION** in diagrams that will be targeted to embedded systems using Real-Time Workshop as various versions of RTW-Embedded-Coder using Green Hills, Code-Warrior, and GCC compilers **MAY NOT** properly generate code for these in some versions. They are thus **not recommended** and best avoided since suitable substitutes are easily constructed from simple Simulink blocks.

- the derivative
- the discrete-time integrator
- the saturation dynamic
- the rate limiter
- the saturation

Rule: The following Simulink blocks shall **NOT** be used in diagrams that will be targeted to embedded systems using Real-Time Workshop as various versions of Simulink **DO NOT** generate code properly:

- the transport delay
- the variable transport delay
- any continuous time element

Rationale: Continuous time blocks (such as the derivative and integrator) can result in inefficient code for embedded application. In addition, many of the listed blocks reference an absolute measurement of time. For embedded applications that are meant to run indefinitely, the time measurement could eventually saturate or even roll over causing inappropriate behavior. These blocks are easily rewritten for efficient embedded application using references only to delay states and the sample period. Replacements should be placed in the widespread re-use library, and engineers discouraged from using the original Simulink blocks. Suitable replacements for these and other continuous blocks are available or can be built as shown below. When placed in a library, they can be used as conveniently as the original Simulink blocks.



THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: No "from" blocks without a matching "goto"

Rationale: By default Simulink will generate a warning. Unmatched "from" blocks are equivalent to using a variable without defining and initializing it. Real time execution results are often unpredictable.

Rule: No "goto" blocks without a matching "from"

Rationale: By default Simulink will generate a warning. Unmatched "goto" blocks can be the equivalent to allocating variable space for data storage and not using it.

Rule: Use ‘goto’ and ‘from’ blocks locally to clean the appearance of diagrams especially on diagrams with large/diverse signal fan-out or fan-in. Minimize use of global ‘goto’/’from’ pairs as they tend to obscure signal flow.

Rationale: The ‘goto’/’from’ pair have no more impact on resource usage than a drawn wire. However, global ‘goto’/’from’ pairs can obscure the signal flow. Bus creator and bus selectors are a preferred method to show signal flow between subsystems and keep diagrams manageable.

Rule: Algebraic loops shall be broken using the unit delay (1/z) block not the memory block.

Rationale: The discrete time unit delay block allows specification of exact or inherited sample times, the memory block does not. This is an important distinction when using certain triggered subsystem structures in Simulink and becomes most important in code generation for real-time embedded targets.

Rule: The block priority attribute in the "Edit/Block Properties" pull down menu shall not be used in any diagram.

Rationale: In some (even recent) versions of Simulink, the block priority attribute does not function properly in all simulation and Real-Time-Workshop instances. In addition, these block priority parameters represent hidden information in the executable specification. It is more appropriate to make order of execution clear and explicit in the diagram using a simple state machine along with trigger enabled subsystems.

Rule: Block names shall **not** include the "/" character.

Rationale: MATLAB uses the "/" character to denote hierarchical relationships in Simulink diagrams. Placing a "/" character in a block name can make creation of MATLAB scripts which seek to interrogate or modify diagrams more difficult. It may also affect the presentation in the MotoTune hierarchy if the default group function is in use.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: Within a project, when variable name abbreviations are used, they shall be standardized.

Rationale: Common variable sub-words are often abbreviated by designers. It is important to create a standardized list of agreed upon abbreviations within a project to avoid confusion in collaborative workgroups. (e.g. Is "temp" an abbreviation for temporary or temperature?)

Rule: Large bus structures shall not cross rate/conditional execution boundaries. Correct use of signal bus structure is encouraged to move signals about the model in groups.

Rationale: While correct use of the bus structure is encouraged, the bus structure is not always the right answer when moving signals throughout a model. Having signal busses cross subsystem rate or conditional execution boundaries can force the code generator to create temporary RAM (stack) variables storing the condition of each and every parameters in the bus at this point in the calculation as the code cannot know which information it will need or not need downstream relative to the conditions of execution. Before crossing a subsystem boundary where execution is not all in the same rate or is conditionally executed, only the signals of interest should be separated from the main bus and then allowed to cross the boundary. This generates code with a minimum of overhead because temporary stack variables are still used, but only the needed information is stored with nothing extra.

Rule: Use of the Data Store blocks in MotoHawk should be minimized as a method for moving data about a model and restricted to uses where it is appropriate to the creation of global variables and data structures.

Rationale: The MotoHawk 'Data Definition' blocks create global variables which present a tempting way to move information about a model. They do this well, but have the unintended property of breaking Simulink's ability to tell in what relation to other blocks the global data is to be updated. It is possible to use multiple data writes for a given data definition as well as multiple data reads. The signal flow that tells Simulink the execution order of the blocks breaks in this scheme. Thus any execution order may be generated relative to where the data is calculated and where it is used as dictated by other Simulink blocks and not the needs of the data store. Data may be calculated one time tic ahead, on the present time frame, or one time tic delayed from the current frame where the data is used…..and that is if all the instances are running in the same time frame. Throw in conditional execution, alternate time frames, rotation based frames, interrupt frames (e.g. CAN receive frames) and any notion of coherence goes out the window without great skill and care by the programmer.

If the bus structure rules above are followed, they present the preferred method for moving data about the model as busses produces no overhead unless crossing execution boundaries and little overhead when done correctly but adamantly maintains both actual coherence and the documentation to report it to the human reading the diagram. In the above, additional skill is required of the programmer and model reader/reviewer to document and interpret execution order.

Rule: Use of the Simulink 'Merge' block should be done with great caution. All inputs and the output of the 'Merge' block refer to the same temporary variable location on the stack as implemented in the generated code.

Rationale: More than one input can write to the 'Merge' stack memory location during the code execution and the temporary variable output reflects the value that last writes to the location. Thus if more than one input condition is active, the last to execute will be the one which sets the

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

output of the ‘Merge’ block to be used downstream. In some implementations and where well defined rules for input execution order or limited execution are left absent by the model builder, execution order is still enforced by Simulink and may simply come down to the order in which the blocks were drawn which influenced how the blocks entered into the model file. Do not let this control the action of the system. Duplicate code drawn in another order may act differently if you depend on this rule of Simulink!

Rule: Use the ‘If’ statement or ‘Case’ statement to enforce single-option-from-many selection such as into a merge where multiple options from which to select exist.

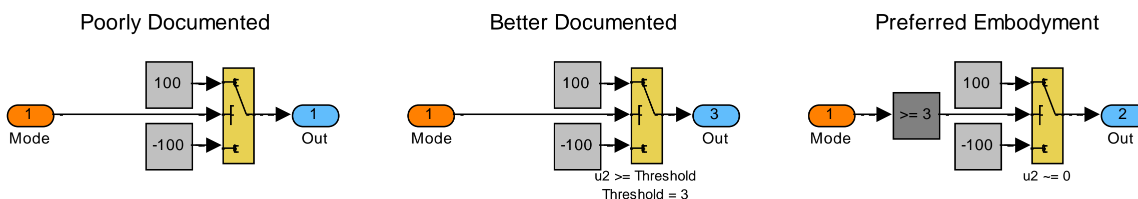
Rationale: It is preferred to use the ‘If’ statement or ‘Case’ statement constructs in place a multiple parallel logic constructs to drive conditional execution for selection by ‘Merge’ blocks because the very nature of these statements enforce only one answer be true at a time. The ‘Case’ statement generates mutually exclusive control signals from a case number or numbers (whichever case definition is true first) just as the ‘If’ statement generates mutually exclusive control signals from any input as defined by the ‘If’ statement definitions. The key is the mutual exclusivity and definite order of checking which might be true! Drawing the same logic where multiple things can be true, but with a preferred order of selection is complex and risks the ‘Merge’ block making its “own” decision based on execution order of multiple enabled signals.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Code Clarity Rules

Rule: All necessary information to recreate a diagram must be available on a printed paper copy of the model.

Rationale: Access to the original model may not be available for code review or when attempting to reuse good ideas from past programs. A paper copy should be sufficient to reproduce the working code. See the example below: All three models function the same way, but only the center and right are fully documented. Note: If **only** Boolean switches are used throughout an entire model, the "u2 ~= 0" notation on the right most drawing may be removed.



Rule: Code one idea at a time. Make clear the purpose of the coded idea.

Rationale: While it is possible to encode very complex concepts within just a few blocks or even a single block using the under-hood settings, this practice should be discouraged if it hides the intent of the code. Again see above rule about the paper copy!

Rule: Use Simulink for signal flow diagrams, use StateFlow for stateful logic diagrams.

Rationale: If both programs are available, diagrams are often much simpler and greatly clarified by choosing the tool best designed for the coding application. Since the two programs work seamlessly together, one does not have to choose one over the other. The choice of, "is this Stateflow or Simulink?" should be based on what is being done by the code. Stateful behaviors are simply easiest to make clear using StateFlow while Simulink is especially easy to encode signal flow concepts.

Rule: Separate the "what/when from "how" in your code. Capture these ideas separately so that they may be edited independently.

Rationale: Changes to a program often come as change to what happens, when it happens, or how it happens. Rarely does a change come as all three, or even two. This is, in effect, more than one change. By keeping the ideas of what happens, when it happens, and how is it executed, separate from each other within your code, you make edits to these items simple and straight forward with no more complexity than is essential to the action.

It can be helpful to use StateFlow to encode the "what" and "when" aspects of the design as it allows capture of this information clearly. The "how" is often signal based and flows from the "what" and "when" as encoded in conditional Simulink sub-systems or switched signals. As ideas

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

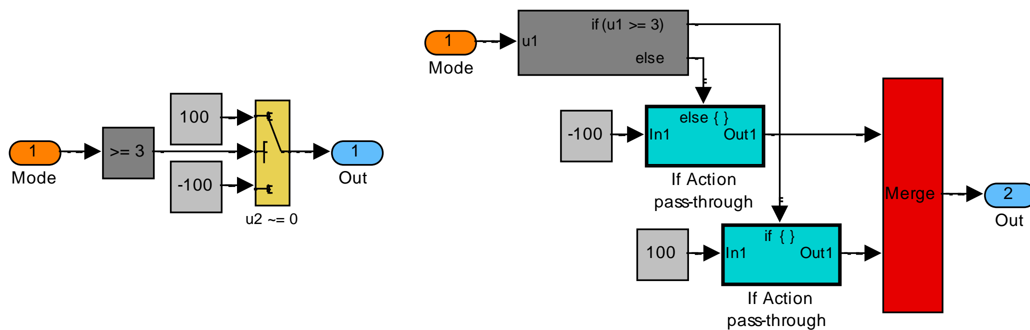
become more complex, this simple division can make editing these behaviors order-of-magnitude easier. Further, unintended errors to “what” or “when” things happen will not be introduced just because a change to “how” an action is implemented is needed, for example.

Rule: Conditional and comparison block outputs shall be treated as true boolean values.

Rationale: The default Simulink data type is a double precision floating pointing value. Boolean results in diagrams (e.g. results from less-than or greater-than operators) may be improperly represented as floating point zeroes or ones, uint8 values, or other depending upon the block used. It is tempting, but incorrect, to use these results directly in arithmetic operations. (e.g. Multiplying a "boolean" 1 or 0 by a computed quantity to mask it from subsequent logic) Mixing data types in this manner is confusing to software engineers and makes porting resultant code to some processors difficult. In later revisions, Simulink gets it “mostly right”, but that is just asking for trouble.

Rule: Switch blocks driven by logical expressions shall use the “not equal to Zero”, (boolean) input selection.

Rationale: Switch blocks should be driven by Boolean inputs. From the previous rule, we know that some versions of Simulink improperly treat Boolean results as double precision floats. To insure proper switch operation in code generation, the default switching logic should use the ~=0 choice. This makes the logic making the choice very clear to the human reader as well as generating very clean, fast acting code. With Simulink 7.0 and later, this is as clean or better than an if statement with multiple enabled subsystems (recommended for 6.51 and earlier only). This method allows use of a “=0” or “~=0” processor flag in directing the transition saving processor time.



These produce similar code

Rule: Block names should be hidden if the block's icon clearly describes its function

Rationale: Block names take up unnecessary diagram real-estate if the purpose of the block is clear from its icon. For example, the icon of a summing junction makes its function clear. Having a label on the block that says "summing junction" is redundant and wastes space.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: Custom icons on blocks or subsystems should clearly indicate the block's function.

Rationale: Icons should be used to add clarity to the diagram. Icons placed on a block replace the default input/output port labeling. The designer must ask themselves which is the clearest way to describe the block's function and interface and then implement the clearest way possible. Remember, the diagram is there to help someone else, or you many years from now, figure out "what is the intent of this diagram." We encourage the clever use of pictures such as in the examples below (from Jason Barta and Rick Poorman) to entice the reader to look further into the code and to serve as navigation aids through the hierarchy.

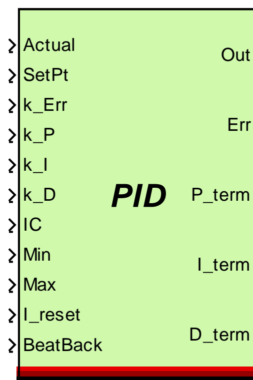
Limp-Home-Code

CAN_Communications

Rule: Block encapsulation masks should be used when encapsulation of the block will help make the behavioral of the system below clearly understood If the mask does not request input data for customization or multiple instance, double clicking should reveal the Simulink below.

Rationale: Electing to mask a block with dialog values and documentation makes it more cumbersome for a designer to examine the underlying diagram unless a forcing function is used on the open-block callback. It is recommended to create such masks only when understanding of the block's function can be enhanced by the encapsulation. Again see above and below.

Valve

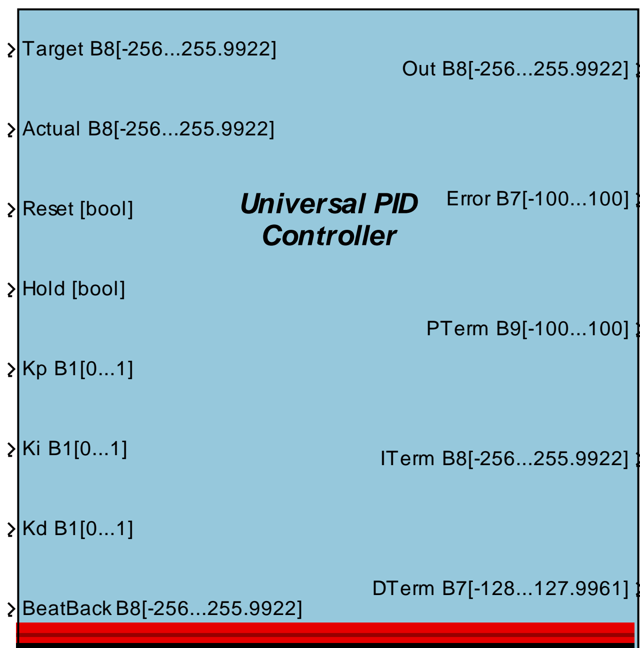


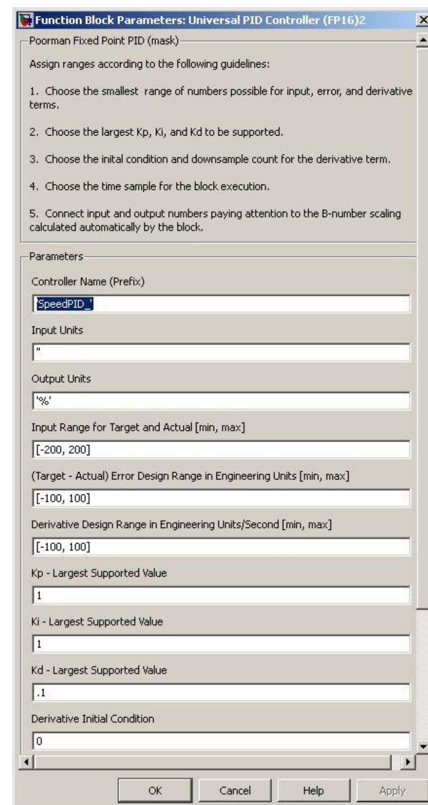
Universal PID

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: Masked blocks with custom icons shall contain interface documentation in the block description field of the mask if the block does not behave as unmasked and have labeled ports.

Rationale: If not properly constructed to include labels on ports, masked blocks with custom icons by themselves provide no indication of their wiring interface. Without easily accessible documentation, the designer must look under the mask and inspect the diagram to determine the interface for wiring. This is not possible with a paper copy and thus cannot be recreated from a paper copy of the code without seeing subsequent layers which must clearly show the interface requirements.





THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Drawing Style Guidelines

Rule: In general, signal flow shall be from top down and left to right.

Rationale: This flow of information is comfortable to most Western readers, and admittedly represents an anglo-centric bias. However, one should not follow a slavish devotion to this diagram structure since feedback loops generally lend themselves to circular signal flow. The goal should always be to enable to diagram to clearly document its own function.

Rule: Wire crossings shall be minimized and unnecessary bends in wires shall be eliminated.

Rationale: Drawings are always more difficult to read when there are a multitude of wire crossings and convoluted wire paths. Designers should frequently challenge their peers' and their own drawing choices, such as input and output ordering, that might result in a cleaner wiring diagram. While the initially chosen I/O order may make since when a diagram is first created, it may be undesirable as a design grows and evolves. Make changes as necessary to keep the diagram neat. When the desired functionality is achieved, the computer is happy. However, the diagram is only **done** when it is made neat and understandable to the human reader/reviewer.

Rule: Wire crossings shall never connect. 'T' intersections shall always connect.

Rationale: Although Simulink adds little dots to show wire connections. These often get misplaced when elements of a drawing are moved. Worse yet, if a drawing is several generations old in photo copy on anything but top line equipment, information can be lost on which wires cross and which connect. Make it clear, follow this rule as it will protect your information on even poor paper copies It also makes your drawings much easier to read.

Rule: Where possible, diagrams shall contain few enough blocks as to fit on a single 1420/1080 screen comfortably and be readable when printed on standard 8.5" x 11" or A sized paper.

Rationale: Large, cluttered, diagrams are difficult to understand. If a diagram is hard to understand, it is likely also hard to modify. The level of complexity of a diagram should be scoped such that it is manageable. Block hierarchies are useful constructs for simplifying the appearance of diagrams. An improved form of the block hierarchy is the use of linked libraries. Using hierarchies of linked libraries enables collaborative development and easy unit test of simpler elements.

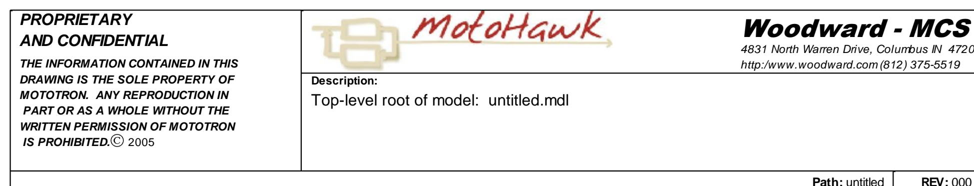
Rule: Each diagram shall be clearly named with a consistent font and location such as on the title block. Use of the MotoHawk Annotations Palate is also encouraged to guide the reader through the code.

Rationale: It is easy to get lost in a deeply nested, hierarchical, design. Simply naming each sub-diagram can help engineers get their bearings. Always placing the block name, using the same font and at the same place in the diagram, for example the upper left corner or on a title block, creates an unobtrusive but useful piece of data that can be relied upon regardless of which diagram the engineer is working on.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: Diagrams shall contain model hierarchy information using the Title Block from the Annotations Library or an equivalent.

Rationale: While naming diagrams is a helpful navigation aid, it is not sufficient. Model page location within the hierarchy must also be presented on a diagram. For example, an element called "Oscillation Over-thruster" might be embedded in a higher level diagram called "Auxiliary System Controls," which in turn is part of the overall "Jet Car Controller." Using the annotations library Title Block makes this automatic. Without it, naming a lower level requires something like "Jet Car Controller:Auxilliary System Controls:Oscillation Over-thruster". Reading the hierarchy notation allows an engineer to immediately know where they are in the design by looking at only the one diagram in front of them.



Rule: Diagrams shall be commented.

Rationale: While some diagrams can be clearly understood simply by inspection, many cannot. The use of diagram annotation places documentation where it may be most useful, in the diagram itself. Just as uncommented textual computer code can be difficult for an engineer to understand, so can a graphical program. No software engineer would accept the practice of not documenting their code. Algorithm engineers programming in Simulink should be held to the same standard. Fortunately, the annotations library provides a variety of features for this purpose.

Rule: Wire annotation shall be used to enhance diagram clarity.

Rationale: An important tool for engineers to use in commenting their diagrams is "wire annotation." Text may be tagged to wires simply by double clicking on a wire. This is most important where this information is needed from one diagram to another as in wire bus construction. Further, use of the wire label allow the use of probes that draw names from the wire label.

Rule: Wire names should be descriptive and include units in the square brackets. The preferred format is “prefix_name [units]”

Rationale: Wire names are often used to set probe names, signal names in bus creators, etc. In each of these cases, ambiguity is removed at the other end where the signal is used if it contains units as part of the name. e.g. IntakeManifold_Pressure [Pa] is clearly in Pascals while ManifoldPressure could be in Pa, PSI, InHg, or something like counts or volts. <u>Make sure to include the space before the square bracket</u>. This will take on more useful significance as MotoHawk improves, but for now, makes it clear that the units are not an operator on the variable name.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Rule: Spend time under the hood – open the dialog box for every Simulink element used. Choose options wisely. Document unusual choices with notes or call-backs.

Rational: Making the proper selections for data-type source, whether or not to saturate on overflow, how to round integer values, all play an important role in determining the quality, size and operation of the code generated. Become familiar with these operations, set the choices appropriately. Document unusual selections using the block properties annotations tab. Go as far as to create a library with properly set up, color coded, blocks. For example, color the block with the "standard" choices the default yellow, then use other shades of yellow for similar blocks with non-usual selections to help call attention. Place a key which explains the color coding and selections somewhere on the page.

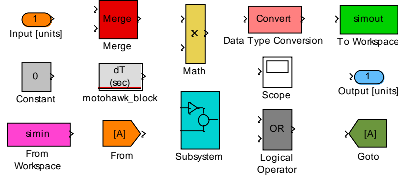
Rule: A single, consistent, color scheme shall be used for all diagrams within a model.

Rationale: Inconsistent color usage decreases efficiency when navigating or studying diagrams. Consistent use of color makes instant visual identification of specific design elements possible. The following color assignments have been found to be non-distracting and enhance efficiency in navigating diagrams. They also produce varying shades of gray on black-and-white printers preserving much of the color information and allow easy readability of black on color or black on gray for all the below colors. Further they work well whether presented on computer monitor, projector, or printed paper.

Background Colors

- **White** background for all diagrams
- **Orange** background for input nodes and "from" blocks
- **Light Blue** background for output nodes
- **Yellow** background for math blocks to make them stand out from the background
- **Light Gray** (from the custom palette) background for constants
- **Gray** background for logical operators and comparisons
- **Magenta** background for "from workspace" and any other prototype/target specific calibratable input blocks
- **Blue or Green** background for to-workspace - prototype/target specific test point output blocks
- **Dark Green** background for "goto" blocks
- **Cyan** background for grouped subsystems or library links to grouped subsystems
- **White** background for all elements which do nothing in code-generation but are required for simulation or to make Simulink happy. This makes them all but disappear in the diagram.
- **Red** background for attention needing critical operations like merges, or trigger blocks.
- **Light Red** (from the custom palette) background for items of important but lesser interest than above like function calls, data type conversions where attention is wanted, etc.

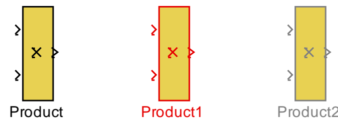
THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)



Examples shown:

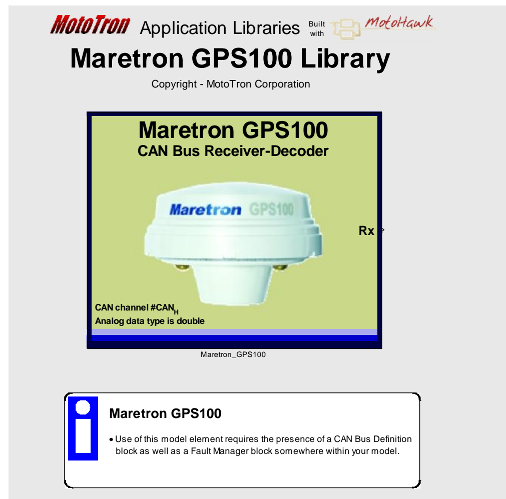
Foreground Enhancements

- **Black** foreground for all permanent elements.
- **Red** foreground for warning messages
- **Gray** foreground for optional, temporary, or experimental features (grayed out look)



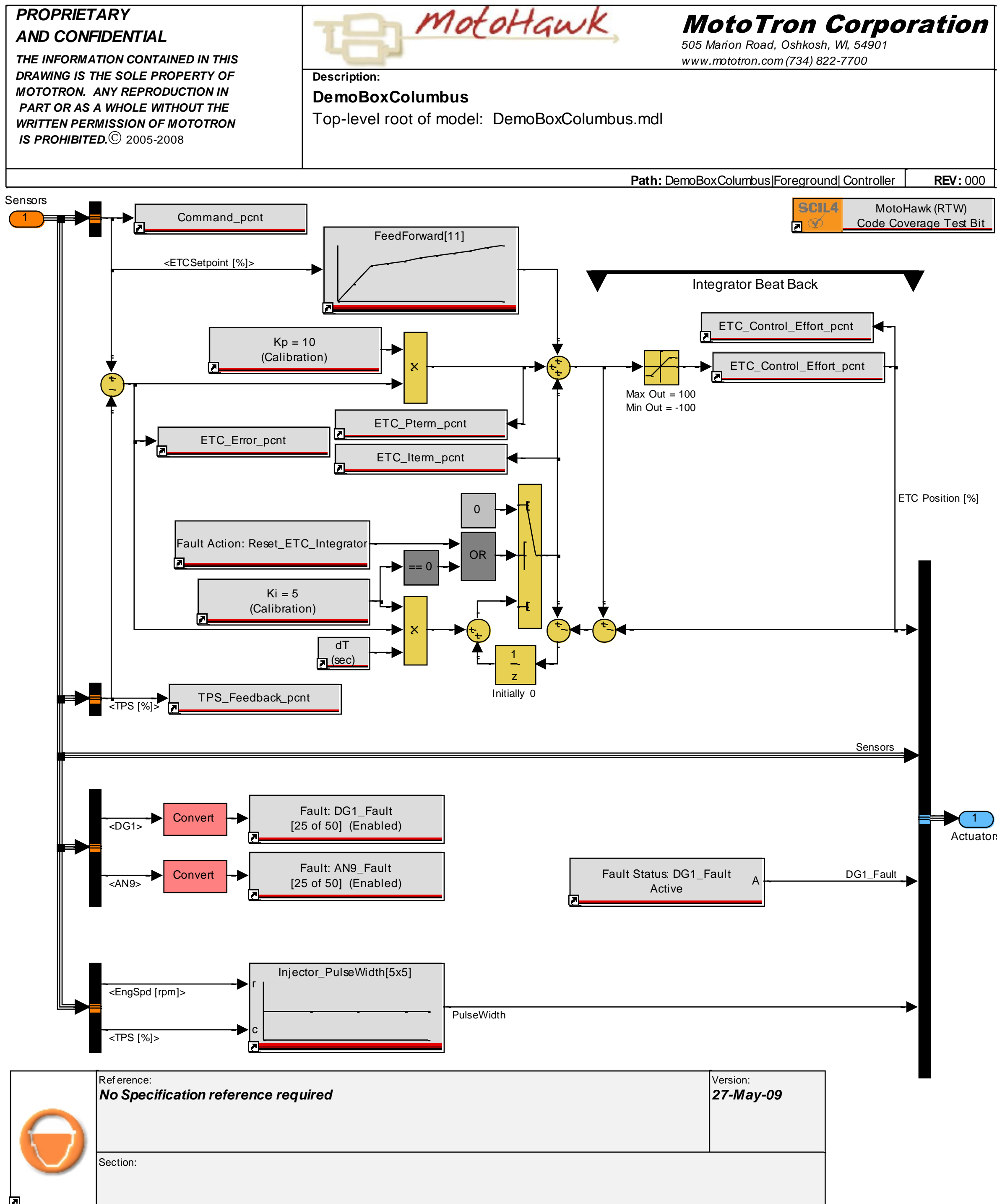
Library Backgrounds

- As in the library example below, a **color other than white** shall be used for the background of a library. This allows user selection of color for information purposes but lets any user, even those who see the library for the first time, know they are in a library and not in a model. See example below:



THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)

Example: The diagram below containing the PI controller (with beat-back and integrator resets) used in the MotoHawk Sales Demonstration Box encompasses the practices, colors, and documentation styles described above.



Process Rules

Rule: Diagram changes shall be committed to a configuration management system (tool based or technique based) for change management.

Rationale: Simulink diagrams are source code, and as such, should be subject to the same "best practices" embraced by the software engineering community at large. Use of configuration management tools and techniques to manage diagram, library, input, and utility file change history should not be considered optional if the project is even slightly complex and involves more than one design engineer. The ability to manage change, back up from mistakes, and leverage tools that enable automatic assembly of a prescribed collections of files is key to highly efficient development and application of model-based controls. Tools like Source-Forge, Clear-Case, Continuous, Perforce, Bit-Keeper, SourceSafe, CVS, and others may all be integrated into good work flows.

Rule: All diagrams shall be tested (and pass) using the "diagram update" (control-D) menu option prior to commitment to the file system.

Rationale: Successful diagram updating is a minimum requirement for possibly being able to perform a non-real-time simulation or invoking the Real-Time Workshop code generator. Performing a diagram update requires that any fatal Simulink syntax errors have been removed and that all calibratable inputs have been defined properly in the workspace. Designers should verify that their elements can be updated before committing them to the file or configuration management system. Diagram update checks may also be automated using the MATLAB scripting facility.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE SOLE PROPERTY OF NEW EAGLE LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF NEW EAGLE IS PROHIBITED (2010)