1. Structured Analysis & Design Using Matlab/Simulink/Stateflow Modeling Style Guidelines -
Version 2.4.2
2. Table of Contents

1. Structured Analysis & Design Using Matlab/Simulink/Stateflow Modeling Style Guidelines ......................... 1
2. Table of Contents .............................................................................................................................................. 2
3. Document Usage ............................................................................................................................................ 4
4. Organization and Important Modeling Constructions .................................................................................. 4
  4.1 Hatley-Pirbhai Model Methodology ........................................................................................................ 4
  4.2 Context Diagram ......................................................................................................................................... 5
    4.2.1 Shared Calibration Constants ........................................................................................................... 7
  4.3 Execution Context Diagram ..................................................................................................................... 8
  4.4 Hierarchical Decomposition .................................................................................................................... 9
    4.4.1 Naming Conventions ......................................................................................................................... 10
      4.4.1.1 events ...................................................................................................................................... 10
      4.4.1.2 data .......................................................................................................................................... 10
      4.4.1.3 Summary ................................................................................................................................ 14
  4.5 Additional Model Organization & Bookshelving Requirements ............................................................. 16
5. Simulink Diagram Modeling ......................................................................................................................... 16
  5.1 Masking P-Specs ....................................................................................................................................... 17
    5.1.1 Passing Parameters Through Masks ............................................................................................... 20
  5.2 Delay ......................................................................................................................................................... 23
  5.3 Function Block ......................................................................................................................................... 24
  5.4 Annotations .............................................................................................................................................. 24
  5.5 Data Stores ................................................................................................................................................ 25
  5.6 Merge ......................................................................................................................................................... 25
    5.6.1 Vector Merging ................................................................................................................................ 31
  5.7 Signal Labeling ......................................................................................................................................... 36
  5.8 Data Scoping Rules .................................................................................................................................... 36
  5.9 Simulink Ports .......................................................................................................................................... 36
    5.9.1 Naming Conventions ........................................................................................................................ 36
  5.10 Goto/From Tags ..................................................................................................................................... 37
  5.11 Vectors ...................................................................................................................................................... 38
  5.12 S-Functions ............................................................................................................................................. 41
6. Stateflow Diagram Modeling ....................................................................................................................... 42
7. Flow Charts Using Stateflow ....................................................................................................................... 42
  7.1.1 If-Then .................................................................................................................................................. 42
    7.1.2 If-Then-Elseif ................................................................................................................................... 43
    7.1.3 Sequenced If-Then Statements ....................................................................................................... 45
    7.1.4 Sequenced If-Then-Elseif ................................................................................................................ 46
    7.1.5 Compound Conditionals .................................................................................................................. 47
    7.1.6 While Loop ....................................................................................................................................... 53
    7.1.7 For loops .......................................................................................................................................... 53
    7.1.8 Multi-Way Branching ....................................................................................................................... 55
  7.2 History Junction and State Machine Reset ............................................................................................... 59
8. State Machines Using Stateflow .................................................................................................................. 57
  8.1.1 Two State Machine .............................................................................................................................. 57
  8.1.2 Guarded Transitions and The Inner Flow ........................................................................................... 58
  8.1.3 Transition Actions and Mixed State/Flow Charts .......................................................................... 59
  8.1.4 History Junction and State Machine Reset ...................................................................................... 61
  8.1.5 State Hierarchy ................................................................................................................................... 61
  8.1.6 State-to-State Transitions .................................................................................................................. 62
  8.1.7 State Maintenance Action .................................................................................................................. 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1.8</td>
<td>Concurrency</td>
<td>66</td>
</tr>
<tr>
<td>8.1.9</td>
<td>Sequenced State Machines</td>
<td>67</td>
</tr>
<tr>
<td>8.2</td>
<td>Keywords and Built-in Functions</td>
<td>72</td>
</tr>
<tr>
<td>8.3</td>
<td>Annotations</td>
<td>74</td>
</tr>
<tr>
<td>8.4</td>
<td>Additional Stateflow Considerations</td>
<td>75</td>
</tr>
<tr>
<td>8.5</td>
<td>Event Scoping</td>
<td>80</td>
</tr>
<tr>
<td>8.6</td>
<td>Event Typing</td>
<td>80</td>
</tr>
<tr>
<td>8.7</td>
<td>Wired Event Connectives</td>
<td>81</td>
</tr>
<tr>
<td>8.8</td>
<td>Broadcast Events</td>
<td>82</td>
</tr>
<tr>
<td>8.9</td>
<td>Data Dictionary</td>
<td>82</td>
</tr>
<tr>
<td>8.10</td>
<td>Port Names</td>
<td>83</td>
</tr>
<tr>
<td>8.10.1</td>
<td>Data Types</td>
<td>86</td>
</tr>
<tr>
<td>8.10.2</td>
<td>Data Assignments</td>
<td>86</td>
</tr>
<tr>
<td>8.10.2.1</td>
<td>The Data Switch</td>
<td>87</td>
</tr>
<tr>
<td>8.10.3</td>
<td>Bit Operations</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>Reusable Model Libraries</td>
<td>96</td>
</tr>
<tr>
<td>9.1</td>
<td>Standard Utilities</td>
<td>97</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Calibration Constants</td>
<td>98</td>
</tr>
<tr>
<td>9.1.2</td>
<td>FOX (library component)</td>
<td>100</td>
</tr>
<tr>
<td>9.1.3</td>
<td>TABLE LOOKUP (library component)</td>
<td>101</td>
</tr>
<tr>
<td>9.1.4</td>
<td>ROLAV_TC (library component)</td>
<td>104</td>
</tr>
<tr>
<td>9.1.5</td>
<td>ROLAV_FK (library component)</td>
<td>107</td>
</tr>
<tr>
<td>9.1.6</td>
<td>CLIP (library component)</td>
<td>109</td>
</tr>
<tr>
<td>9.1.7</td>
<td>FAULT CODES (library component)</td>
<td>110</td>
</tr>
<tr>
<td>9.1.8</td>
<td>TIMER (library component)</td>
<td>111</td>
</tr>
<tr>
<td>9.1.9</td>
<td>VARIABLE SATURATION (library component)</td>
<td>113</td>
</tr>
<tr>
<td>9.1.10</td>
<td>VARIABLE SATURATION - MINIMUM CLIP (library component)</td>
<td>114</td>
</tr>
<tr>
<td>9.1.11</td>
<td>VARIABLE SATURATION - MAXIMUM CLIP (library component)</td>
<td>115</td>
</tr>
<tr>
<td>9.1.12</td>
<td>DOC-LINK BLOCK (library component)</td>
<td>115</td>
</tr>
<tr>
<td>9.1.13</td>
<td>CALIBRATION FILE SPECIFICATION (library component)</td>
<td>116</td>
</tr>
<tr>
<td>9.1.14</td>
<td>PID BLOCK (library component)</td>
<td>117</td>
</tr>
<tr>
<td>9.1.15</td>
<td>CRIB SHEET LINK BLOCK (library component)</td>
<td>118</td>
</tr>
<tr>
<td>9.1.16</td>
<td>MULTI-PORT DE-SWITCH (library component)</td>
<td>120</td>
</tr>
<tr>
<td>9.1.17</td>
<td>Delta_Time Block</td>
<td>96</td>
</tr>
<tr>
<td>10</td>
<td>Data Dictionary Interaction</td>
<td>124</td>
</tr>
<tr>
<td>10.1</td>
<td>Pd2cmd</td>
<td>124</td>
</tr>
<tr>
<td>11</td>
<td>Model Initialization</td>
<td>125</td>
</tr>
<tr>
<td>12</td>
<td>Unit Testing Considerations</td>
<td>136</td>
</tr>
<tr>
<td>13</td>
<td>Conclusion</td>
<td>136</td>
</tr>
<tr>
<td>14</td>
<td>Index</td>
<td>138</td>
</tr>
</tbody>
</table>
3. Document Usage

This document describes how The MathWorks tool suite, including Matlab, Simulink, and Stateflow, can be used to model and specify powertrain controller functionality. The tools have shown efficacy in modeling controller features for gasoline and diesel applications. This style guide documents the modeling methodology that has emerged from prior trial studies involving a number of Puma Diesel features and a number of gasoline engine and transmission features.

Note: MATLAB/Simulink/Stateflow tool suite definitions and descriptions are provided in The MathWorks documents. This document clarifies how the tool suite is to be used for powertrain strategy specification.

The modeling style recommendations put forth in this document have been made to support understandability, maintenance, and operation of the model during simulation. In addition, other style recommendations are being put forth to facilitate the automatic processing of models for purposes of software unit testing, architectural building and consistency checking, automatic target ready code generation, etc. These additional capabilities can only be realized if certain modeling styles are adopted and adhered to from the beginning of model creation. Finally, many style conventions are put forth to enable the use of the model as one component in the user documentation suite available on the WEB to the calibration community.

This guide is applicable to Matlab version 5.3, Stateflow version 2.0 and Simulink version 3.0.

Questions concerning this guide or its use should be directed to Paul Smith AVT/CAPE - PCSE - Control and Software Design Engineering, via the Matlab Problem Log WEB page on:

http://www.dearborn.ford.com/csded/cgi-bin/matlab.cgi

or, via PROFS @ PSMITH13 or on phone number (313)594-1507 or e-mail psmith13@ford.com

4. Organization and Important Modeling Constructs

4.1 Hatley-Pirbhai Model Methodology

The modeling methodology is based loosely on the data flow, control flow, and data models established by Hatley and Pirbhai. In general, in the Matlab tool suite, Simulink diagrams model data flow while Stateflow diagrams model control flow. Refer to the Figure 1 below.

Process specifications (P-specs) are specified using Simulink blocks and/or Stateflow diagrams, depending on the nature of the algorithm. In general, this partitioning is straightforward, however there exist situations where judgment has been used to maintain modeling consistency. These situations are identified throughout this document. Detailed modeling guidelines for these situations are established.
4.2 Context Diagram

The top level model for each feature is a Simulink diagram that serves as the context diagram. Refer to Figure 2 below. The top level of the model should be saved in the file named feat_v<xx>.mdl where xx is the primary version number of the feature.

In this diagram Inports are used for feature inputs, Outports for feature level outputs, and Inports for scheduler function call event triggers.

Input and scheduler function call event signal labels are defined explicitly at this level. That is, all input signals must be labeled without signal inheritance at this level.

Output labels should be automatically propagated through the model from the point where the signal is first written to. Signal names that are propagated will be shown with ‘<’ and ‘>’ on either end of the name (e.g., <output1> in the example below). All input signals shall be labeled on the context diagram level. Do not inherit input signal labels on the context diagram. All data signals on the context diagram shall be defined in the data dictionary (with the possible exception of the ‘first pass’ flags used in model initialization or temporary parameters that are to be implemented as pass by value arguments on cross feature function calls). In all cases where temporary parameters of 'first pass' flags appear at the context level, the parameters must begin with the characters 'ml_'. Refer to later sections of this guide for detailed naming conventions for signals.

Additional items that may be shown on the context diagram are text annotations and a block that specifies the calibration file specification and shared calibration constant supporting blocks (terminators/grounds). There should be nothing else on the context diagram.
No Stateflow blocks appear at this level.

Since this top level of the model contains the entire algorithm, it is named feat_v<xx>. The file it gets saved to is feat_vxx.mdl. The xx corresponds to the primary version number. The feat is the approved feature abbreviation from the NDT/Feature Planner database and is also the directory name used in the configuration management system directory structure. Strict adherence to this naming convention is necessary to support the automatic WEB based serving of models as part of the user documentation set.

The subsystem shown centrally in the context diagram is called feat_Vxx_xxx_xxx, where feat is the feature abbreviation as defined in NDT/Feature Planner database and Vxx_xxx_xxx is the full specific version number for the feature being modeled. Please use '_' (underscore) characters instead of '.' (period) characters for the name of this model block because the model printing facility will create files on your PC with the name of this block while generating the user documentation package. If you use periods, these files will be assumed to be of a type that is not understood by the tools.

The calibration file specification shall be shown at this level. The calibration file specification block shall not been shown at any other level of the hierarchy. This block will be needed when simulating the model to provide base calibration values for all calibration constants throughout the model. It is also automatically copied by the Software Test Tool into the testing harness which is created.
4.2.1 Shared Calibration Constants

Note that only those calibration constants that are owned by other features or owned by the feature being modeled but used in another feature appear on the context diagram. This is done so interface tracking can occur from the context diagram level and include the checking for properly declared shared calibration constants. All other types of calibration constants (owned and used only by your feature) are directly referenced throughout the hierarchy where they are used as constant blocks, Stateflow workspace variables, or directly referenced in block dialogues (e.g., the CLIP block), drawing their value from the Matlab workspace. These constants can be terminated on the context diagram level. Please refer to Figure 3 below for the minimum requirement for documenting shared constants. If the modeler chooses to wire the constant signal flow from the context diagram down the hierarchy to the point in the model where the constant is used, the only stipulation is that the signal flow be terminated just before it is used in any calculation and the actual value drawn from the workspace. A key assumption is made by all automation in the process that all calibration data comes into the model directly from the workspace and is not received through a signal flowing from the context diagram level of the model. This is done to ensure consistency in the definition of constant values in the Matlab base workspace, not through a connection in the context diagram.

Figure 3
4.3 Execution Context Diagram

Perhaps the single most important element of the modeling style is the execution context diagram. By structuring the model in the manner described below, automation of many key quality control processes is possible AND practical. Without this notion, automated testing of algorithmic models versus production controller software is difficult if not impossible and automatically generating production ready software would not be practical. This structuring also provides benefits in algorithm and software design and debug.
The next diagram, Execution Context Diagram (Figure 4 above), shows each feature execution context as a separate Simulink subsystem. An execution context is defined as a piece of algorithm that runs at a specified rate (interrupt or periodic) distinct from other algorithm pieces in a given model. Examples include fixed rate processes running at 100ms, 50ms, 30ms, etc. or crank angle interrupt based processes. There should be a one-to-one mapping between scheduler function call event triggers shown on the context diagram one layer up in the hierarchy (context diagrams) and Simulink subsystems on this diagram. Data flow between contexts is also shown at this level if any exists (none shown in this example).

The individual execution context subsystems should be named **feat-context**, as shown in the diagram above where feat is defined as before and context is one of bg, pip, cid, 1ms, 2ms, 4ms, RAM_init, etc...

There are times when a feature provides a utility function, accessible by interfacing features through the context level of the feature. When this occurs, it shall be treated as a separate execution context and shown as a separate subsystem on the execution context diagram.

### 4.4 Hierarchical Decomposition

Subsystem decomposition continues until the lowest level p-specs blocks are modeled - refer to Figure 5 above. It is considered good form when no more than about seven process specifications (p-specs) and/or
control specifications (c-specs) appear on a single diagram. However, p-spec diagrams may have considerably more than seven blocks as lowest level primitive blocks are used (summers, gains, etc…).

Stateflow blocks are used to model control (process activation and process sequence) specifications in each subsystem below the execution context diagram. In general, there is one and only one C-spec per subsystem and one or more Pspecs per subsystem.

4.4.1 Naming Conventions

4.4.1.1 events

The triggering event input to a P-Spec is wired into the top (graphically) of a subsystem through the trigger port and the signal carries the naming convention do_<p-spec name>. The name of the p-spec subsystem block is also <p-spec name>. The trigger block from the Simulink/Connections menu, configured as a function call type trigger is used inside the p-spec subsystem block to achieve this.

The function call event input into a subsystem which is merely a layer in the system hierarchy comes into the side (graphically) of the subsystem block as the first input and the signal carries the naming convention trig_<hierarchy name>. The name of the subsystem the triggering event is input to is <hierarchy name>. In this case, the Inport block from the Simulink/Connections menu is used to bring the event into the subsystem.

For function call event connections, by convention and for the purposes of manual tracing of events throughout the system, the function call event outport name on a Stateflow C-Spec, the signal name starting from the C-Spec output port, and receiving event Inport or trigger port name must be the same.

For function call event connections involving reusable components, the function call Outport event name, signal name, and receiving event Inport name will be different by necessity. The Outport and the signal should be named the same, however. For example, the input port names on the library element, TIMER, will be generically named trig_timer and trig_reset. The controlling C-Spec in the model should name these events something more descriptive like trig_mytimer and trig_mytimer_reset. These more specific names would appear on the C-spec Outport and the signal connecting the C-Spec to the timer.

All block names and signal names are case sensitive within the model. Lower case is used by convention for all names, except when calibration constants are named.

4.4.1.2 data

All non-event signal names (data flows) should be unique, independent of case, and should match the name stored in the parameter dictionary. All signals that represent RAM locations in the software must have their signal lines labeled with the name of the parameter as it exists in the parameter dictionary. Any signal that must be labeled but does not represent data stored in the parameter dictionary (temporary data or data signals that only exist to aid in modeling) should have the characters ml_ as the first part of the signal label. Refer to the parameter dictionary for the complete set of attributes for any given parameter including data type, visibility, owner, etc…

As an example of naming conventions at the lower levels within the model, please refer to Figures 6 and 7 below.
Naming convention illustration

- `block_name` = hierarchy subsystem name
- `trig_block_name` = c-spec triggering event name
- `block_name_ctl` = receiving c-spec name

Figure 6

And the contents of the `block_name` hierarchy layer subsystem:
There are times when subsystems will be formed entirely of other subsystems for readability purposes. Any subsystem layer inserted into the hierarchy for documentation readability purposes should merely be given a descriptive name, no naming convention needs to be followed in this case. This is commonly done to make a diagram more readable, by artificially creating layers of hierarchy within the model so no more than about seven subsystem and/or Stateflow blocks appear on a page. In this case, multiple `trig_` and/or `do_` input event signals will be entering the subsystem blocks as inputs through Inports.

The naming convention must be followed for these events only at the destination p-spec or c-spec, not at the layers introduced to make the diagram more readable. Refer to the figures 8 and 9 below.

The top layer of a block inserted artificially into the hierarchy looks like that shown in Figure 8.
Inside the block labeled 'hierarchy', all the naming conventions are followed as shown below:

In the case where one c-spec receives more than one triggering event, which can occur, at least one of the enabling events should follow the naming conventions described here. The other event(s) should be given descriptive names, however no convention is put forth for those names. Refer to Figure below:
4.4.1.3 Summary

In summary, the following conventions are followed:

1. Any Simulink subsystem block that is performing a basic data transformation is called a P-sepc.
2. All P-spec subsystems must be masked (refer to section Simulink Diagram Modeling below).
3. All P-spec blocks are triggered by a function call type trigger inserted inside the masked subsystem.
4. All P-spec blocks triggers are named `do_<p-spec subsystem block name>`.
5. All function calls which trigger a P-spec enter through the TOP of the masked P-spec subsystem through the trigger port and must originate from within a Stateflow C-spec block.
6. Any Stateflow block which outputs function calls is a control flow specification or, Cspec.
7. Any Stateflow block which outputs only data is a Pspec.
8. Only Stateflow blocks which are Cspecs need have a enabling event function call signal wired into the TOP of the Stateflow block.
9. Stateflow blocks which are Pspecs can free float inside a masked P-spec subsystem block. In this case, data flow dependencies dictate order of execution within the Pspec.
10. Any Simulink subsystem block that is a layer in the hierarchy shall have at least one Inport that is a function call starting with the characters `trig_`.
11. All function calls starting with `trig_` shall terminate on the `input_events()` port of a Stateflow Cspec.
12. The full naming convention of a function call trigger enabling a layer in the hierarchy named `<blockname>` is `trig_<blockname>` and the cspec receiving the function call should be named `<blockname>_ctl`.
13. All function call signals starting with `trig_` shall enter through the side (graphically) of a Simulink subsystem block and must be the first Inport(s) to the subsystem block.
14. Any Stateflow block which is a Cspec should be triggered by a function call directly through the `input_events()` port on TOP of the Stateflow block with a function call starting with the characters `trig_`.

Figure 10
15. Only alphanumeric characters and the underscore, `_` characters are allowed in block (and signal and port) names. Spaces, dashes, dots, carriage returns, slashes, & and other similar characters should not be used.

Graphically, these rules look like the following:

![Diagram showing masked Simulink subsystem block, hierarchy layer, and Stateflow Cspec]

**Figure 11**
And underneath the **hierarchy** block, the function call trigger terminates on a Stateflow Cspec:
4.5 Additional Model Organization & Bookshelving Requirements

- All simulation instrumentation blocks shall be removed from the model prior to bookshelving of the feature. This includes, scopes, displays, workspace import and export blocks, etc…

- If execution cycle delta time for fixed rate processes is required within a model, a special constant shall be defined in the workspace following the naming convention ML_<feat>_TASK<#>_DT, where <feat> is the standard feature abbreviation and <#> is 1-5 (i.e., on each for 100ms, 50ms, 30ms, 16ms, 8ms). As an example, if the execution cycle time for the 100ms task rate is needed within the EGO feature, the constant ML_EGO_TASK1_DT = 0.1 shall be defined in the workspace. This way, the task can easily be moved to a new rate and only one constant need be changed. A special block has been added to the standard utilities to output this constant and must be used when cycle time is needed in the feature level model. Refer to the section entitled 'Standard Utilities’ later in the guide for a description of this custom block. Of course, if execution cycle time is needed for variable (interrupt) rate tasks, it must be calculated internally within the task.

- In general, in order to bookshelf a model, the model update function must execute with no warnings or errors. Additionally, extensive simulations must be completed prior to bookshelving to ensure a complete and correct specification of functionality has been achieved. Also, the model must pass a peer based style review to ensure compliance with these written guidelines.

5. Simulink Diagram Modeling

Pspecs (data transformation equations) will be modeled in block diagram form using Simulink primitive (gains, summers, multipliers, switches, etc ...) blocks (see example above).

Each P-Spec should be given a descriptive name. Each P-Spec will be triggered to execute as a function call as shown in Figure 12 below.
This function call is invoked from within a C-spec in a Stateflow diagram.

\[
\text{output1} = [2 \times (\text{input1} + \text{input2})] + \text{input3}
\]

**Figure 13**

### 5.1 Masking P-Specs

All P-Specs should be masked and a form of pseudo-code or other suitable description of the equation being performed in the p-spec should be placed in the mask icon. This is done to facilitate the use of the Simulink diagram in the user documentation package. In the example above the mask should contain something like that shown in Figure:

\[
\text{output1} = [2 \times (\text{input1} + \text{input2})] + \text{input3}
\]

**Figure 14**

After the P-Spec subsystem is masked, a double click on the subsystem block will bring up the mask window as shown below. The detailed block diagram that represents the equation(s) contained in the P-Spec is now hidden below the mask. In order to view the block diagram, go to the **Edit, Look Under**
**Mask** menu picks from the model window. The text for this mask dialogue window comes from the mask definition and is shown below in Figure 13.

![Figure 15](image)

When creating the mask, the windows below will need to be filled in. These are accessed initially when the **Edit, Mask Subsystem** menu picks are chosen from the model window. They can also be edited after initial mask creation through the **Edit, Edit Mask** menu picks from the model window.

The first set of information that should be filled in is under the **Icon** tab in the mask editor. Refer to Figure 14 below. Here the **Mask Type** should be entered as the name of the P-Spec being masked. This will appear in the mask dialogue border when double clicking on the mask subsystem later. See the example window above.

In addition, in the **Drawing commands:** window, the contents of the masked P-Spec icon should be specified. This example uses the `fprintf()` statement. This is very C-like and you should refer to the Matlab help for more details of the syntax. One important note however, the text in the Icon will be automatically centered. Some manual spacing may be necessary to get a nicely formatted description of the equation(s) performed in the P-Spec.
Figure 16

The second part of the Mask dialogue that should be edited is under the **Documentation** tab in the mask editor. Refer to Figure 17 below. Here the block description should be added. This is the text that will appear inside the mask dialogue window when the masked subsystem is double clicked. This should be a sufficiently detailed description of the equation and requirements being satisfied with the equation. The block help can be optionally filled out as well, but unless this is a re-usable utility type function being masked, it is not likely that the block help will be used at this time. At some time in the future, if tighter integration of automated calibration procedures is made with the algorithm models, then this part of the mask could become useful.
5.1.1 Passing Parameters Through Masks

When masking a block in Simulink, a unique mask workspace is created for the masked subsystem whenever the mask creates one or more variables in the initialization tab of the mask editor. If there are not any new variables created, the blocks under the mask use the base workspace. Blocks within the masked subsystems that do create mask workspace variables cannot reference the base workspace.

For a single layer of masked subsystem of this type, any base workspace parameters must be established through the mask initialization commands. For example, if the calibration constant XYZ is required in a calculation under the masked P-spec, where the mask also has a checkbox variable, X, created, a special command must be entered in the mask initialization commands tab of the mask editor. Refer to the diagram below:
Figure 18

Under the mask we see:

Figure 19

When double clicking on the calc_out1 block, the following mask dialog is seen:
Notice, at least one variable has been created by the mask editor to hold the value of the checkbox. Because of this, the base workspace is no longer directly accessible to the blocks under the mask. Any base workspace variables needed under the mask must be entered in the mask dialog. The base workspace variable XYZ has been entered as the value that is needed below the mask. The mask editor initialization tab is shown below which is used to accomplish this:

```
set_param(strcat(gcb,'/{Constant}',Value,'c'))
```

Figure 20

Figure 21
When the base workspace variable XYZ is entered in the mask dialog, the mask initialization command takes the value in XYZ and assigns it to the mask workspace variable, c. The set_param() command then assigns the value in the variable c to the constant block (below the mask) ‘Value’ parameter, completing the transfer from the base workspace to the block under the mask.

When masking a Stateflow block which requires access to base workspace variables, one need only scope the workspace variable in the Stateflow Explorer to the MACHINE level for proper simulation.

### 5.2 Delay

In order to use the ‘last pass’ value of any particular parameter a delay should be modeled. To represent a delay in the P-spec in Simulink refer to Figure 18:

![Figure 23](image-url)

This example illustrates the use of the delay block. In this example, the delay is set up to inherit the timing from the parent subsystem (in this case in sync. with the function call).

The example uses the 1/z discrete time delay. This should be used inside Simulink p-specs.

The 1/z block will sample its input and hold it for one execution cycle. Using the triggered subsystem according to these guidelines, that means each time the do_delay event is generated and the equation contained within the subsystem delay shown above is executed, the var_old will contain the value of var from the last time the do_delay event was generated. In the example above, var=\(input1 + var\_old\), where var_old is the value of var the last time do_delay was invoked. Initial conditions for the var_old can be established through the dialogue for the 1/z block. Assuming the initial condition for the 1/z block is set to zero and input1 is held constant at 2, the following table summarizes the behavior of the above model:
The update rate for this delay block should be set to inherit the timing characteristics of the parent diagram. This is accomplished by double clicking on the 1/z block and under the field of **Timing**, enter `-1`. **If this is not done, the model will not be able to be simulated.** This tells Simulink how often to sample the 1/z input and synchronizes it with the triggering function call.

Refer to the delay block dialogue shown in Figure 19:

![Unit Delay](image)

**Figure 24**

### 5.3 Function Block

The simulink/Fcn block cannot be used in algorithm modeling. Any functionality that the Function block provides can be modeled with atomic simulink blocks. In addition, the atomic simulink blocks allow for naming and monitoring of intermediate calculations, allow for intermediate parameter typing when modeling fixed point operations and allow for more efficient code generation.

### 5.4 Annotations

Simulink provides free form text blocks for on diagram annotation as is visible in the examples above. Diagrams should be annotated with enough commentary to make the diagrams understandable. These diagrams are *part* of the user documentation suite. The more information the modeler provides in the form of comments, the more appropriate these diagrams will become for this purpose.

Drop shadows should not be placed around annotations due to issues while printing the models out.

Note: Hyperlinks to documents may be made through the Doc block on Simulink diagrams. See the Reusable Libraries section later in this document.
5.5 Data Stores

Data Stores are not to be used in algorithmic modeling. In earlier versions of this guide, data stores were used to manually implement the merge functionality. With the newer versions of Simulink, a built in block performs the functionality that the data store used to provide.

5.6 Merge

The MERGE operation is used when a parameter is written to in more than one subsystem block. The signal must be merged to form the final value of the parameter.

This is especially important when writing to parameters that are written to in more than one execution context. These types of parameters MUST use a MERGE block on the execution context diagram to consolidate these signals into one.

The MERGE is also used within a single execution context when the different modes of calculation for a single parameter are controlled from within different subsystem blocks.

Refer to the example below:

Here we have two triggered subsystems that will both write to variable. The pip context subsystem will add one to the variable each invocation and the bg context subsystem will clear the variable.

Below in Figure 25 is the top level diagram.

![Figure 25](merge/merge.png)

The naming convention used here is to label each signal line with the name of the actual parameter being written to (variable in this case) on the signals flowing into the merge. The Outport from each sourcing...
block is named the same name as the signal (variable in this case). The signal output from the merge is given the same name as the signals going into the merge (variable in this case).

In the below, the feat_bg process is shown.

![Figure 26](image)

Inside the feat_bg/pspec you should model the system as shown in the below:

![Figure 27](image)

In this execution context the variable is to be clear each time the event trig_feat_bg is received.
The internals of the feat_pip block are very similar as shown in the below

![Diagram of feat_pip block](image)

**Figure 28**

And finally the contents of the feat_pip/pspec block are shown below.
Notice the naming convention that must be followed for the case above when the same parameter is both an input and an output from the same subsystem pspec. Simulink will not allow two block with the same name so we choose to name the Outport the name of the variable being written to and the Inport has the characters '_in' appended to the name of the variable. The signal is, however, consistently named ‘variable’ throughout the model.

There shall be at most one Merge block in a particular signal topology. That is, you cannot pass a previously merged input into a merge. This rule applies within a feature execution context, within a complete feature, and even at the application level where a signal is written to in more than one feature.

In order to adhere to this rule, the Mux block must be used at lower hierarchical layers to make signal routing easy and to convey data flow. To fully convey data connectivity, the signal shall be routed from Merge output to input port when the merged signal is needed to source a Simulink or Stateflow block. When a signal is written by more than one feature, those features needing the signal as a source must have the signal as an input at the context diagram. Of course, each feature writing to the signal must show the signal as a output on the context diagram.

The following model illustrates these concepts:
Example to illustrate use of MUX in the signal topology requiring multiple merge sources.
In the simple example, one need not have actually used the MUX blocks. They only become mandatory when the signals to be merged cross subsystem boundaries (and hence, through output ports) to prevent additional port naming confusion. When Simulink is corrected to allow cascading of merged signals, the muxes can be quickly replaced by merges without further signal topology changes.

When one wishes to monitor an input to the merge block, special modeling actions need to be taken. Keep in mind, that when a merge is used, even though distinct signal lines feed into the block, those signal lines actually represent a SINGLE PARAMETER. One can not simple pull the signal that is input to the merge block into a scope. In fact, Simulink will issue an error message if this is attempted. The input signals and the output signal from the merge are really the same signal, so to try to intercept the signal at the input has no physical meaning.

In order to use (which is not typically recommended) or to monitor the input to a merge, a new signal must be created. Refer to the diagram below:

And inside the 'alt_data_calc' block the following is seen:
Notice, when the signal going into the merge is needed elsewhere, either for data visualization or for other calculations (USE WITH CAUTION), a new signal is created and a unity gain block is inserted to copy the intermediate signal to the final signal.

5.6.1 Vector Merging

Quoting from the Simulink help for the merge block:

“Simulink restricts the kinds of connections you can make to the inputs of a Merge block. In particular, it permits only connections that establish a one-to-one mapping from the outputs of non-virtual blocks to the inputs of a Merge block. For example, you can use a Go To/From block pair to connect the scalar or vector output of a non-virtual block in one part of a diagram to the input of a Merge block in another part of the diagram.

You cannot use a Merge block to connect multiple non-virtual outputs to a single input on a Merge block.

Simulink checks for invalid connections in a block diagram at the start of a simulation. If it detects an invalid connection, it stops and displays an error message.”

As an example of how this affects the algorithm modeler, consider the following diagram:
This diagram is UNACCEPTABLE because the inputs to the merge do not originate from the same non-virtual block. In this case, the signals X[0] and X[1] are not originating from the same block. In this case the following (somewhat cryptic) message will be received when starting a simulation with this model:
In fact, in order to merge vectors given adherence to this modeling style one of the following approaches must be taken.

The first option is to only run source signals originating from a non-virtual block capable of creating a vector. A Stateflow diagram or the Multiport DE-switch block are two candidates. Refer to the example below:
In this example, vector line widths have been turned on to show that the two blocks sourcing the merge inputs can, in fact, source vectors. In most cases that will be encountered by the modeler, the vector signals will not be sourced from the same non-virtual block as in the example above. In this case, one must create by hand a vector merge block. The block shall be masked and the text 'Vector Merge' be placed on the block icon. Refer the diagram below:
In this way, an element-by-element merging of the contents of the vector will be performed and Simulink will not produce any error messages.

5.7 Signal Labeling

Flows (or signals) of significant final or intermediate values shall be labeled. (These flow names appear in the Real Time Workshop automatically generated ‘C’ code. The signal label SPECIFIES the name to the software designer that should exist in the production code for that signal.

Input and output flows at the feature context level must be labeled. It is envisioned the application level interface consistency checking and application level model builds will use signal labels at the feature context level.

Signals that are to represent tool specific variables (e.g. switch inputs) shall be labeled and the naming convention specified earlier in this document is to prefix the name with the three characters ‘ml_’. This naming convention specifies a signal that has been introduced only for the purposes of modeling and should not necessarily exist in the final embedded software product.

Constants are not considered data flow and therefore the signal coming from a constant block shall not be labeled. This is true for calibratable and non-calibratable constants.

In Simulink process specifications, constant blocks are used to reference the Workspace variables. Constant names automatically appear in the block icons so there is no additional information conveyed by adding labels to constant block signal flows.

All signals in or out of a masked subsystem must be labeled or, for events only, have a Goto/From connector located adjacent to the block with the name of the event.

Only alphanumeric characters and the underscore,’_’ characters are allowed in signal (and port and block) names. Spaces, dashes, dots, carriage returns, slashes, & and other similar characters should not be used.

5.8 Data Scoping Rules

All constants, calibratable or non-calibratable, are modeled as global workspace variables within Simulink. Simulink signal labels and Stateflow data and event elements may be scoped as global or local. When two or more features are combined to form an application, global variables are promoted to application level global entities. Local scoping is preserved when applications are built from feature components.

5.9 Simulink Ports

Ports are the input and output connectors used to route data into and out of Simulink subsystems.

5.9.1 Naming Conventions

If the subsystem block is a component with a single instantiation (that is a subsystem that is not a reusable component from a library), port names should match the connected signal label to aid in the modeling effort and to support the downstream activities of software unit testing and model consistency checking.
However, it is not necessary to display the port name on the finished model as the displayed port and label names present redundant information at the expense of diagram clutter. The user documentation tool box will automatically turn off all port labels in the PDF form of the model only. The model itself will be left unaltered, with the port names ON.

If the subsystem block is a reusable component then the port names and attached signal names will necessarily be different. The subsystem port names of reusable components (e.g. filters, limiters, fox functions, tables) should be shown for representation purposes.

In the case where a parameter is both an input and an output from the same subsystem block, the output port should match the signal name and the input port should have the characters ‘in’ appended to the signal name.

Only alphanumeric characters and the underscore, ‘_’ characters are allowed in port (and signal and block) names. Spaces, dashes, dots, carriage returns, slashes, & and other similar characters should not be used.

### 5.10 Goto/From Tags

In general, one should attempt to form diagrams that have no crossing signal lines. The use of the Goto/From connectors can aid in this.

As an example of the tag or sometimes called the connector, refer to the model diagram below. In this case, connectors are used to prevent crossing signal lines in the diagram. They are a convenient mechanism for un-cluttering larger, more complicated diagrams.

The tag need only have an arbitrary and descriptive name to allow connecting of the goto and from connectors together. Good judgment must be used to determine if the names are descriptive. The use of color can also aid the identification of matched sets of Goto/From tags.

In Figure 25 below, the actual signal name was used as the connector name also. This is as descriptive as possible, but not necessary in every circumstance. One should weigh the tag size, diagram density, diagram complexity, maintenance, etc… when deciding how to name the tags. Tags should always be scoped to be visible only locally (the default) to prevent confusion in the diagrams and to allow for explicit diagramming of the data and control flows within the feature level system models. (Some exceptions to this rule may be necessary to implement some utility functions. Each deviation must be analyzed on a case by case with exceptions being granted only after thorough review of the surrounding issues and consultation with the architecture control activity).
The tags can also be used to break the signal flow of data signals, though not shown in this example (Figure 31 above).

In the case where a function call event signal path is broken by Goto/From connectors, only the connectors and not necessarily the signals need be labeled with the event name (i.e., do_pspec). This is done for readability purposes in the final user documentation. In other words, the event flow must either be labeled directly or have the name of the event embedded in the connector or both (but not mandatory on both).

When breaking a data flow signal with a connector, the signal propagation must be preserved through the connector. This is accomplished by either directly labeling or automatically propagating the signal on (mandatory) BOTH sides of the connector. This is because of the different nature of the signal labels on data flows. They represent parameter names to embed in the software, whereas event names need not be present in the software. Other issues will dictate specific software designs that may or may not actually create software functional decomposition and invocation that match the model structures.

There shall be no unattached goto/from connectors in the model.

5.11 Vectors

It is sometimes convenient to treat a number of signals as a single signal in a model. This allows for more compact models and allows for the logical grouping of related signals. One can also represent arrays in the simulink environment through the use of vectors.

Vectors are created through the use of the simulink/Connections/Mux and Demux blocks. As an example of the use of the Mux block, refer to Figures 32 and 33 below. Figure 32 is a true array in the software implementation. Figure 33 is a grouping of related signals that is not intended to be a software array.
ARRAY Example and Naming Conventions

NOTE: Use a BUS SELECTOR, not a DEMUX here. This will provide proper signal propagation.

Label the output of the MUX with the name of the array as it appears in the data dictionary.
Please note a few things with respect to the use of vectors:

- Label each signal into the Mux block or ensure a properly inherited signal label.
- In the case of the array, individual signal labels are named `array[n]` where `n` is the array index number for the particular element and `array` is the array base name and can be found in the data dictionary.
- In the case of the vector of related signals, each element should carry its actual name.
- When the vector is made of related signals, not an array, the vectorized signal shall be labeled with a name beginning with `ml_`.
- When the vector is an array, the vectorized signal shall be labeled with the name of the array.
- Vectors can be inputs to Mux blocks, thus creating vectors of vectors. See the following example and directions.
- All vectorized data must first be Demuxed or selected via a bus selector or some other mechanism before being input to a Stateflow data input port.
- Only vectors which represent true arrays in the embedded software with declared names in the data dictionary can be shown entering through the context diagram. In this case, the signal must be labeled with the base name of the array.
- Vectors of related parameters that should not be grouped as arrays in the software should be brought into the context diagram as properly labeled scalars.
- The Outport or Inport passing a vector through should be labeled with the base name of the array or in the case when the data is not meant to become an array in the embedded software, a descriptive name representing the collection of data that has been vectorized.
• The mixing of function calls and data on a single vector is prohibited.
• Please note that if an array is a direct output from a Stateflow block, the BUS SELECTOR block will not be able to resolve the array element names. A DEMUX must be used in this case.

• Also note that BUS to BUS connections are not currently supported by the tool set.

• The muxing of function calls, except those that are directly input to a Stateflow block or trigger port of a subsystem block is prohibited. (Another way of stating this is that the DEMUXing of function calls or BUS selection of function calls is prohibited).

In order to create vectors of vectors, care must be taken when specifying the width of the input signals in the MUX block dialog. If the following conventions are not followed, signal name propagation and potentially simulation errors may result. Consider the following example:

When specifying the number of inputs in the dialog for the MUX block, even for simple 1 dimensional vectors, enter the information in the format shown. For a simple 1 dimensional vector of three scalars, one would enter \([1 \ 1 \ 1]\). For a vector of two scalars and a vector of 3 scalars, one would enter \([1 \ 1 \ 3]\). When this is done proper model operation can be expected.

The BUS SELECTOR dialog is also shown on the right hand side of the model block above. Notice how the vector structure is presented in the left hand pane of the BUS SELECTOR dialog.

5.12 S-Functions

S-Functions, or user written code, should only be included in a model under two circumstances:

Any library block from the \texttt{utils} library can have a S-Function implementation. These could be M or C code.
A non-library block specific to a feature level model implemented in C code. One should not use M code in a S-function due to compatibility with down stream processes like unit testing, code generation and for simulation speed efficiencies. The use of the S-Function should be a last resort when the functionality
cannot be modeled using Simulink and/or Stateflow. Individual cases will be examined during peer style review and will require concurrence of the senior modeling style inspector(s).

6. Stateflow Diagram Modeling

The following guidelines specify how the Stateflow modeling tool will be used to model both C-Specs and P-Specs.

Much of the existing strategy is specified as a series of If-Then-Else constructs. This type of logic is combinatorial that can be modeled without the use of states.

Some strategies use or could benefit from the use of finite state machine representations to represent the behavior of the strategy as the powertrain evolves from mode to mode. With a combination of flowcharting and state machine constructs, Stateflow is proficient at modeling both combinatorial and state based logic.

7. Flow Charts Using Stateflow

It is recommended that the flowchart style be used when no clear state information exists in the system being modeled. This style makes extensive use of the condition actions (those enclosed by ‘{‘ and ‘}’) and the connective junction Stateflow constructs.

7.1.1 If-Then

![Stateflow If_then endif](image-url)  

Figure 34
In the above, a very simple flowchart is represented. Execution begins at the top of the diagram at the default transition. The default transition is the one with the small dot at the end of the arrow shown at the top of this diagram. Each time the Stateflow diagram containing this flowchart is invoked, execution will begin at the default transition.

The evaluation priority in Stateflow will be used to determine which transition should be taken away from any connective junction. This priority states that first those transitions with event guards will be processed, then those with conditional guards and then those that are unguarded.

In this case, the [condition] is evaluated and if true, the actions specified inside the {}’s (called condition actions) are performed in the sequence shown. Stateflow will always evaluate transitions with conditions on them before evaluating any transition without a condition on it (termed ‘unguarded’).

Strictly speaking, the unguarded transition is not needed for this trivial example. It will become necessary if the flow is later extended to add additional, sequenced logic after this If-Then statement. Because of the potential for difficult to diagnose problems during simulation during later maintenance actions, it is required that the unguarded, no-action transition always appear, even when seemingly unneeded. Single entry-single exit to a flow chart is also required and will therefore mandate the use of these unguarded transitions.

The Mathworks has recommended including these unguarded transitions as the code generated will be more efficient as a result. An equivalent bit of C code would look like:

```c
if (condition1)
    {do_Pspec;
     trig_Cspec;}
else
    {}
```

An attempt to place the condition/action pairs consistently (to the right or below the transitions, for example) on the diagrams relative to transitions should be made. This may not be possible in all circumstances, however, any deviation from a standard approach should result in a model that is completely unambiguous when reading the model. It should be obvious which condition/action pairs belong to which transitions. Other conventions are possible (other than to the right and below) so long as they are consistent in their application and easily understandable.

All flow charts must have a single entry point as specified by the default transition and a single exit point as defined by a single connective junction within the chart where flow stops.

All flow charts must use the condition actions (enclosed in {}) rather than the transition action (preceded by /). This is done to ensure consistency from user to user and to ensure proper operation during simulation. There are subtle differences between the two types of actions. Please refer to the Stateflow user's guide for more details. Transition actions will be allowed in some limited case in design of state machines and will be outlined later in this document.

**7.1.2 If-Then-Elseif**
The next flowchart example shown in the above adds an else-if to the first example.

Since Stateflow will always evaluate the guarded transition first, it will take the [condition1] branch, if it evaluates to true, executing the do_Pspec1 action along the way.

If [condition1] does not evaluate to true, it will follow the unguarded transition to the second connective junction. This unguarded transition is necessary to establish that [condition1] should be checked first and only if it is false should the rest of the flow chart be executed. If this were not present, it becomes ambiguous which of [condition1] or [condition2] should be evaluated first. (The reality is that Stateflow has developed a convention to evaluate the transition of equal priority in a clockwise fashion starting at the 12:00 position of each connective junction. Our modeling style has been established so as to NEVER depend on this convention for clarity purposes.)

Again, it will evaluate the guarded transition first. If the guarding condition, [condition2] evaluates to true, it will traverse that transition and execute the action do_Pspec2.

If the condition is false, it will traverse the unguarded transition to the final connective junction and execution will return to the calling Stateflow diagram. The C code for this example would look like:

```c
if (condition)
    {do_Pspec1;}
else if (condition2)
    {do_Pspec2;}
else
```

Figure 35
### 7.1.3 Sequenced If-Then Statements

The next example in the figure above illustrates how to string more than one if-endif together.

The value of the unguarded transition from the first connective junction is now visible. Without it, [condition2] would only be checked if [condition1] evaluated to true. With it, there is always a path to the second connective junction, independent of [condition1]. The resulting C code looks like:

```c
if (condition1)
{do_Pspec1;}
else
{}

if (condition2)
{do_Pspec2;}
else
{}  
```

Figure 36
7.1.4 Sequenced If-Then-Elseif

In a similar manner as shown above, in order to string more than one if-else-end-if together, the flowchart would appear as in the figure above. The resulting C code looks like:

```c
if (condition1)
    {do_Pspec1;}
else if (condition2)
    {do_Pspec2;}
else
    {} 

if (condition3)
    {do_Pspec3;}
else if (condition4)
    {do_Pspec4;}
else
    {} 
```
7.1.5 Compound Conditionals

Conditions with a mixture of AND and OR logic are referred to as compound conditionals. Stateflow provides the opportunity to illustrate complicated compound conditional logic in a graphical format that can be easy to interpret.

The first type of compound conditional is the compound OR logic shown in the figure below.

![Compound OR Logic](image)

Figure 38

The three parallel paths through the flowchart without any actions represent a compound OR gate. If any one of the three branches through the logic is found to be true, the action will be taken. If none of the branches evaluates to true, the unguarded transition around the statement will be taken.

In C code this would be represented like the following:

```c
if   ( (x==1 && y==1) ||  (a==1 && b==1) || (e==1 && f==1) )
{action;}
else
{}
```

Obviously, the graphical form is more quickly understandable than the pseudo code.
One shall not embed actions to be taken in the middle of a compound conditional on one of the purely conditional transitions for the sake of diagram thriftiness. In fact, the simplest way to recognize a compound conditional embedded in a larger flow chart is to look for conditions without actions linked either serially or in parallel.

In the figure below, one of the parallel branches is replaced by an embedded compound AND gate.

C code for this example looks like:

```c
if   ( (x==1 && y==1) || ((a==1 || c == 1) && b==1) || (e==1 && f==1) )
{action;}
else
{}
```

Figure 39
Following the same pattern if another of the original OR parallel branches were replaced by an embedded compound AND gate, the diagram would look like that shown in the figure below:

![Diagram of Stateflow Compound Conditionals](image)

Figure 40

C code for this example looks like:

```c
if ( (x==1 && y==1) || ((a==1 || c == 1) && b==1) || ((e==1 || g==1) && f==1) )
{action;}
```
else
{}

The most simple form of the compound AND looks like that shown in the figure below:

Notice that a single unguarded transition around the compound conditional is required should the condition be FALSE. This will prevent the flow chart from exiting in the middle and allow for easy extension of the flow chart at a later time. Equivalent C code would look like:

```c
if   ( (x==1 || y==1) &&  (a==1 || b==1) && (e==1 || f==1) )
{action;}
```

Figure 41
else
{
}

If the middle condition were replaced by an embedded compound OR gate, the diagram would be modified as shown in the figure below:

![Diagram showing compound conditionals](image)

**Figure 42**

One can see that it is far more simple to show the embedded compound OR logic inside the compound AND logic than vice versa. The equivalent C code for this example looks like:

```c
if   ( (x==1 || y==1) &&  ((a==1 && c==1) || b==1) && (e==1 || f==1) )
{action;}
else
{}
```
Similarly, if the final leg of the compound AND gate were also replaced by an embedded compound OR gate, the diagram would be easily extended to look like that shown in the figure below:

![Figure 43](image_url)

The equivalent C code in this case is:
if ( (x==1 || y==1) && ((a==1 && c==1) || b==1) && ((e==1 && g==1) || f==1) )
{ action; }
else
{} 

7.1.6 While Loop

Another useful construct is the While Loop. Graphically, it will be shown as in the figure below in Stateflow:

![Stateflow While loop](image)

The loop will execute as long as the [cond] evaluates to true. Obviously, in order to prevent formation of an infinite loop, the actions executed during the loop must somehow affect the [cond] in a way to terminate the loop execution.

The diagram must not end on the last connective junction of a loop. This appears to be a bug in the product at this time (V1.0.5.0 of Stateflow). Once the operation has moved to a version of the product without this problem, this guide will be updated to eliminate the need for the unguarded exit from the loop.

7.1.7 For loops

The For loop is a common structural element in control flow specifications. The Stateflow flow chart editor can be used to realize these constructs. Refer to the diagram below:

The While Loop and For Loop graphically appear very much the same. This is because the For Loop is merely a special case of the While Loop. The While Loop can have any generalized conditional statement controlling loop execution. In contrast, the For Loop uses a counter parameter with start and stop conditions specified within the For statement itself (in C, that is). Graphically it becomes easier to understand why these two statements result in essentially the same behavior. Refer to the figure below:
The equivalent piece of C code to perform the same function looks like:

```c
for (i = 0; i < 10; i++)
do_Pspec;
```

Obviously, this example does not make any sense unless the data flow process specification, do_Pspec is correctly defined and interfaced to the surrounding model. In this example, the do_Pspec would merely be executed 10 times. It would yield the same result each time. This type of construct may be useful in establishing some parameter that is different for each engine cylinder, but the cylinder number must be made available to the Pspec through the interface from the Stateflow diagram to the surrounding Simulink environment.

The use of the transition back onto the same or earlier connective junction can be generalized to form any of the conventional looping constructs (repeat-until, do-while, for, etc...).

The style expressly prohibits unstructured looping (even though the Stateflow tool itself does not). The following figure shows a nested pair of for loops that are allowed and an unstructured for loop (labeled /*BAD */) that attempts to enter in the middle of one of the others. This is an unstructured loop and can only be implemented with gotos in the code. This is prohibited.
7.1.8 Multi-Way Branching

There is a special case logical structure that compares a single parameter or calculated quantity against a number of alternatives, taking different action in each case. This is similar to the functionality the CASE-SWITCH statements provide in the C programming language. This functionality can also be implemented graphically in the Stateflow editor using the constructs shown in the figure:
It is important to understand the priority with which the transitions are evaluated. First those transitions with events guarding them are evaluated, then those with no event but with conditions, then finally the unguarded transition.

It becomes obvious that if a number of transitions are coming from the same junction and they have equal priority (as in the example above), the simulation engine MUST have some sort of mechanism for deciding which to execute or evaluate first. In fact it does, but this style guide is written so as not to depend on it. This is why such limiting restrictions as those outlined below are required when using the multi-way branch construct.

Some things to note:
1. The first transition after the default transition at the top of the chart is optional.
2. If the variable to be tested against already has the value to be tested, the transition assigning a value to \( x \) should not be included.
3. All branches MUST compare against the same parameter (\( x \) in this example).
4. If the calculation of \( x \) is of a temporary nature, be sure to declare the variable as local or temporary in the Stateflow data dictionary.
5. Because comparisons of equality are used, (\( \{ x == 1 \} \)) extreme care must be taken to ensure that the variable \( x \) is typed such that it can exactly equal an integer value. The variable may need to be defined as an integer in the data dictionary to allow this. Remember that all variables in Simulink are real doubles.
6. One does not have to use comparisons to equality, as long as it can be shown that all the cases are mutually exclusive, independent of the actual data being compared.
7. There must be an unguarded transition to provide for the default case where none of the other cases evaluated to true (even if no actions are taken on the branch).
8. The action on any of these branches is optional and does not have to be a function call event as is shown in this example.
9. By definition, the sequence in which the cases are evaluated should be of no importance (even though Stateflow would actually evaluate the conditions from right to left in this example).
10. This construct should not be used in place of an if-elseif-...-endif statement. It should only be used when sequence of evaluation of the conditions, against exactly the same variable, is irrelevant.

Please refer to the Matlab help information for a more thorough description of the capabilities in the Stateflow action language. It is a very powerful language. We have decided on a standardized subset of the entire capability because it covers all the needs we currently have to model our controllers. If however, there is the need for some capability not specifically shown here, it should be discussed and added to this style guide as an accepted practice.

8. State Machines Using Stateflow

State machine representations should be used when modeling state based strategies. It is very natural to model the modes and the state transition paths of complex systems such as the powertrain. In this style actions are typically restricted to execute when entering, exiting, or when remaining in a particular state. A number of examples follow to illustrate conventions to be used when modeling using state machines.

8.1.1 Two State Machine

In the figure above, a simple two state machine is represented.

Upon initialization, the OFF state is entered via the default transition.
The machine remains in the OFF state until event2 occurs, at which time the trig_Cspec function call is performed and when that is complete, the ON state is entered. The event is specified as a function call by the algorithm designer to allow a more accurate representation of system behavior. All events specified in the control/diagnostic algorithm will be specified as function calls with the exception of events used purely for the purposes of simulation of the physical plant or the input to the operating system.

The machine waits in the ON state until the event1 occurs, at which time the do_Pspec function call is performed and when that is complete, the OFF state is entered. The machine continues in this cycle forever.

An attempt to place the condition/action pairs consistently (to the right or below the transitions, for example) on the diagrams relative to transitions should be made. This may not be possible in all circumstances, however, any deviation from a standard approach should result in a model that is completely unambiguous when reading the model. It should be obvious which condition/action pairs belong to which transitions.

### 8.1.2 Guarded Transitions and The Inner Flow

In the figure above two things have been added to the simple state machine of the first example.

First, guarding conditions have been placed on the transitions. Now, in order to move from the OFF state to the ON state, not only does the event2 event need to be received, but the [condition2] must evaluate to true.
Second, a flow chart has been added to the interior of each of the two states. While the state is active, the flowchart will be executed and the \textit{do\_Pspec3 or 4} will be invoked, depending on which state the machine is in.

When the machine is in the OFF state and the \textit{event2} is received, the state machine will be invoked. It will first check \textit{[condition2]} and if true, \textit{trig\_Cspec2} will be invoked. When control returns back to this state machine, the state ON will be entered and execution will always begin at the default transition. Since the flow chart inside the state ON does not have any guarding conditions, the event \textit{do\_Pspec3} will be generated and execution control will be returned to the caller that originally sent out the \textit{event2} event. The state machine then goes to sleep.

On the next invocation of the state machine, assuming it is either not \textit{event1} which has awoken the machine or \textit{[condition1]} is false, the unguarded \textbf{inner flow} will be taken (this is the flow from the inside edge of the state ON). This will result in the execution of the \textit{do\_Pspec3} event after which the state machine goes back to sleep.

Basically, the inner flow transition is used to specify action to be taken while resident in a state. Remember, the inner flow is the lowest priority transition and the state machine will always try to leave the state if possible before executing the inner flow path.

\subsection*{8.1.3 Transition Actions and Mixed State/Flow Charts}
The figure shown above illustrates two more features of the Stateflow action language.

First, note the flow chart inserted on the path between the OFF and the ON state. In order to make it all the way to the ON state, event2 needs to occur and both condition2 and (condition3 or condition4) need to be true.

Second, the condition action do_Pspec2 will be executed if event2 and condition2 are true. Alternatively, the transition action do_Pspec6 will only be executed if a state transition takes place from OFF to ON. The sequence that will occur if event2 occurs and both condition2 and 3 are true is do_Pspec2, do_Pspec5, d_Pspec6, do_Pspec3.

Care must be taken when using the transition action. It should be restricted to use when the same action is required on more than one transition path from a state so as to prevent having to write the actions more than once on the diagram. It should not be used as a replacement for condition actions. Each case will be reviewed during peer style review for acceptability.

If event2 and [condition2] is true and both condition3 and condition4 are false, then no state change will occur. In fact, the OFF state will never actually be exited, even though the action do_Pspec2 will be
executed. In this case, the action *do_Pspec4* will also be executed as a result of the inner flow path being executed.

### 8.1.4 History Junction and State Machine Reset

The the figure above illustrates the use of the **History Junction** and two alternative methods of performing a *reset* action.

The first reset action **does not** require the use of the History Junction. When the *reset* event occurs, the state machine is pulled out of whichever state it is currently in and the event *trig_reset_action* is executed and the OFF state is entered. This type of reset action is used if the designer always wants to enter the same state upon receipt of the *reset* event.

The alternative reset approach **does** require the use of the History Junction. In this case, when the *alternate_reset* event occurs, the state machine is pulled out of whichever state it is in and the event *trig_alt_reset_act* is executed and control is returned to whichever state was last active. This type of reset should be used when normal control should be immediately resumed after the reset actions have been taken. This is used more infrequently than the previous type of reset behavior.

### 8.1.5 State Hierarchy
Figure 52

The figure above illustrates the concept of \textbf{state hierarchy}. In this case, if the \textit{disable\_event} is received, the machine is pulled out of whatever state it is in and enters the \textsc{supervisor} state, performing the \textit{do\_Pspec4} along the way. It stays there until the \textit{enable\_event} is received, at which time the \textit{do\_Pspec3} is executed and the \textsc{off} state is entered through the default transition.

\section*{8.1.6 State-to-State Transitions}

When more than one transition from a state is needed, it is important that the sequence in which the transitions out of the state be explicitly modeled. For example, consider the portion of a state diagram below:
In the (incomplete) diagram shown in the figure, it is not obvious which transition should be tested first. In order to eliminate this problem, the same rules that apply to flow charting also apply to flow charts constructed between states. The diagram above should be redrawn to be more like the one below in the figure:

**Figure 53**
In this way the transitions will be tested in the sequence indicated on the diagram with the number embedded in comments. Placing the numbers in comments is not required in the model, but is shown here only to make the sequence of execution visible. Do not place these in the model.

Of course, the rules for multi-way branching can be invoked and a diagram like the one shown in the figure above can be used. In this case each transition must compare against the same parameter and all conditions must be mutually exclusive.

8.1.7 State Maintenance Action

It is sometimes desirable to perform a periodic action and use the result of that action to control a state machine. For example, a system can be either hot or cold based on an A-to-D value that must be read every second. Two distinct things need to happen in the model of such behavior. First the A-to-D needs to be read and second the decision of Hot versus Cold needs to be made based on the temperature read. There are two states the system can be in, Hot or Cold. A number of stylistic options exist to perform this type of action.

The first would be to draw a flowchart to first perform the periodic action to read the A-to-D and second to use an If-Then construct to set a flag, 1 for Hot or 0 for Cold. This has the graphical disadvantage of not showing the two states explicitly and the complicating hysteresis branch. The state information is stored in a flag. Note also that this approach specifies that the test for COLD ((temp < A)) is done before the test for HOT. This could be considered over specification as the hysteresis would work equally as well.
if HOT was tested for first. This is a side effect of using the very specific mechanism of flow charting to describe what is inherently state behavior. Additionally, it is difficult to specify that the variable flag be refreshed each execution cycle while within the hysteresis zone, which is common practice in embedded controller specifications. Refer to the the figure below.

![Diagram](image-url)

**Figure 55**

One can recognize that in the flowchart above, there is a branch through the logic where the value of the 'flag' parameter does not change. This is a clear indication that there is STATE information that can be modeled here. In this case, the hysteresis associated with the comparisons to the 'temp' parameter is what is providing the hysteresis.

A better alternative, to preserve the graphical view of the states and to still centralize the A-to-D read on the same diagram would be to introduce an inner flow, and state hierarchy as in the the figure below:
You can clearly visualize the two states the system needs to be in, Hot and Cold. The actions to move from state to state are clearly and simply marked on the transitions between the states. The inner flow provides the maintenance (or graphical during function) action of reading the A-to-D without pulling the machine out of any of the states. Of course, when code is written for this state machine or any other, some variable must be used to store the state information.

8.1.8 Concurrency
One final concept to illustrate is state concurrency or parallelism. There are cases when true concurrency should be modeled, as is shown in the the figure.

In other cases, there is no data dependencies between the two states and it does not matter what sequence they get executed. This is a don’t care situation that can also be modeled with concurrency.

It is **extremely** important that guards are placed on all flow paths in the state machines contained in the parallel machines. This is because Stateflow will wake up the State machines with any of the input events wired to it. If there are any unguarded transitions that can be taken, they will be. This is why the `trig_bg_mgr` and the `trig_pip_isr` events are specified at the start of each flow chart inside each of the concurrent states. This would prevent the `trig_pip_isr` event from causing execution of the `bg_mgr` flow chart and vice versa.

### 8.1.9 Sequenced State Machines

In order to produce compact models, it is sometimes convenient to place multiple state machines on a single diagram and specify the sequence in which to execute each one.

At this time the only way to specify the sequence is graphically (one can verify the sequence by looking into the Stateflow Explorer) or through the use of a main flowchart and subroutine flowcharts with local event guards (refer to the figure).
Consider the figure above. Three independent flow charts have been constructed and, for modeling convenience and compactness, they are shown as concurrent states in the same Stateflow diagram. They will be executed in the sequence task1, task2, task3, as shown in the Explorer.

They will all respond to the same enabling event, in this case given our naming convention, trig_sequenced.

In this example all the conditions and actions have been left off the flow charts because the detail is not important.

Another slight variation to this construct is when the three flow charts are actually triggered by different events. Again, the modeler may choose to show the charts together in one Stateflow diagram for convenience or compactness. This is especially true if the flow charts are all computing the same data in different ways.

The diagram is modified as shown in the figure to illustrate the three incoming events. If more than one event is wired to a single Stateflow diagram, it is critical that ALL transition paths be guarded at their starting point so they are only activated by the one(or more) event(s) that should activate them.
Please also note in the diagram above that one of the three enabling events has been given a name to follow the naming conventions as described earlier in this document. The other two events need merely start with 'trig_'.

The Simulink diagram which holds this Stateflow diagram must first MUX these three events together before wiring it in.

Again, all additional detail within the flowcharts has been omitted as it is not relevant to the point being illustrated.
Shown above in the figure is a third use of concurrent states. In this case, there is one master flow chart which makes local event calls to other flow charts within the same diagram. In this way, graphical subroutines can be shown.

It is important to note that these graphical subroutines must be single entry, single exit flowcharts, like all other flow charts.

A couple of things to note in this example:

The 'main' flow chart must be last to execute on the page graphically. If this is not the case, the first execution of the subroutine flow charts may not occur properly due to the initialization sequence of the charts. Essentially, the subroutine charts must have had a chance to be activated before they will respond.
the first time to the local enabling event. By locating them higher on the diagram, they will be activated first and be ready for the first subroutine enabling event.

If the local events, sub1 & sub2 are scoped to the diagram and not to the receiving state they are in, the 'main' chart needs to guard its inner flow transition to prevent a run time infinite loop. When the sub1 or sub2 events are sent locally, they will also wake up the entire state machine, sequenced2_ctl, again. This will cause all unguarded transitions to be taken, including the one that broadcast the event in the first place. This is an infinite loop and is only detected at run time.

If the local events are scoped to the receiving states rather than to the diagram, directed event broadcasting can be used. Refer to the figure below for this alternative. In this case, the local event is sent as state_name.event_name.

The subroutine flow charts do not have default transitions or the default transitions should take no actions. This is required to prevent initialization issues.

Figure 61
Going back to the State Maintenance Example once again, we can illustrate the use of the concurrent states with a slight re-design of the Hot-Cold hysteresis example. Refer to the model below in the figure.

In the example shown in the figure, we have pulled the reading of the ADC to its own flow chart. In this case, the state TEMP will be executed first and the A-to-D will be read. Then the state a will be executed. This is subtly different than the earlier example in that the A-to-D would not be read on the very first execution cycle in the previous example. It will be in this example. This is because, in the previous example, the do_read_adc event was on the graphical during transition. It would not be executed on the first activation of the state unless it was on the default transition path.

8.2 Keywords and Built-in Functions

Stateflow provides a number of built in Keywords and functions available for use. We will restrict usage to the following:
**in(state_name)**: A function that returns a Boolean that can be used in a conditional.

**change(data_name)**: A function that returns a Boolean that can be used in a conditional.

**t**: Time since the start of simulation.

**exit**: The keyword attached to a state label used to specify actions to be taken just before leaving a state.

The remainder of the keywords are not to be used because they have graphical equivalents which favor a more readable document. These include **during**, **entry(state_name)**, **exit(state_name)**, **on event_name**, **send(event_name, state_name)**, **matlab()**, and **matlab.MATLAB_workspace_data**. The diagram below illustrates the graphical **during**, **on**, and **entry** alternatives. The textural **exit** action is also shown (no graphical equivalent).

There are a number of compiler defined functions such as **abs()**, **fabs()**, **min()**, **max()**, etc… that are available and can be used. These should be used with extreme care because their behavior is compiler (for Stateflow - Microsoft VisualC++ in the case of PCSE) dependent. Care should also be taken to ensure that the functionality can be implemented eventually in an embedded controller. Only the following C library functions are allowed to be used:

- **abs()**
- **fabs()**
- **min()**
- **max()**

---

Figure 63
log()

The use of built in matlab name space operators and functions (specified by ml.func_name() or ml('func_name') ) is prohibited in algorithm modeling.

The use of Literals in Stateflow (action language you want the parser to ignore but you want to appear as entered in the generated code) shall not be used.

8.3 Annotations

Stateflow transitions should be annotated with standard C-language type comments (i.e., enclosed in /* and */ characters) to provide descriptive language describing the operations being performed. In addition, the document hyperlink for the Stateflow diagram can be used to point to a separate document describing behavior or requirements for the diagram. This is accessed as data or event interface items are added to the Stateflow diagram via the Stateflow Explorer and can be a separate document for each parameter/event entered in the interface.

Additionally, the stateflow BOX can be used to float comments on a stateflow diagram. Stateflow allows you to use graphical entities called boxes to organize your diagram visually.

Boxes are primarily graphical entities. They do not correspond to any real element of a state machine. However, boxes do affect the activation order of a diagram's parallel states. In particular, a box wakes up before any parallel states or boxes that are lower down or to the right of the box in the diagram. Within a box, parallel states still wake up in top down, right-to-left order.
8.4 Additional Stateflow Considerations

Certain additional restrictions are placed on the internal structures of state charts and flow charts that would result in an ambiguous or unstructured specification.

There shall be no unreachable or dead specification elements within a stateflow c-spec. That is, all branches shall be in an execution path that can be reached by at least one combination of inputs/calibration constants/local parameter values.

In the first example, a transition is shown leaving the state and returning. The transition with condition1 on it will override the normal clockwise evaluation of the transitions in this multi-way branch statement and lead to confusion over execution sequence. It is not allowed.
In general, transitions that exit and then re-enter a nested flowchart should not be used. They provide little additional specification flexibility that cannot be handled with other, easier to understand constructs. In the example above, the EXIT action is NOT invoked when the [condition1] transition is taken, so the transition exiting the state and then re-entering provides no benefit whatsoever.

In the next example, a similar situation exists, except a transition to a new state is introduced from the embedded flow chart. Again, confusion can result in the reader over order of evaluation and the construct is not allowed:
If a construct like the ones shown above is needed, then order of execution must be explicitly called out as in the example below:
Another potential poorly structured specification involves backtracking in state transition logic. Consider the following:
The way Stateflow works, the action \texttt{a3} would get executed two times. Once on the first try when the \texttt{true} branch is taken. When the \texttt{false} condition is encountered, execution will backtrack up the logic and try to find a path through the \texttt{a2} action branch, executing \texttt{a3} again before stopping execution. As a general rule, specifications that take advantage of this backtracking in Stateflow that result in the multiple execution of any actions are prohibited. This will be avoided if one adheres to the same rules in state transitions as must be followed in flow charts (i.e., unguarded paths around all logical constructs). This example can be modified with any one of the transitions (numbered 1-5) that have been added in the figure below:
8.5 Event Scoping

Events, like data, can be scoped within the Matlab tool. Events can be scoped locally to a single Stateflow diagram or a state within a diagram, exported to the interfacing Simulink environment, or imported from the interfacing Simulink environment.

The Stateflow Add menu pick in the Stateflow editor is the mechanism for defining data and events and establishing their scope. (Event and data attributes can later be viewed and modified directly from the Explore pick from the Tools menu pick in the Stateflow editor).

Events scoped within a state chart (local) do not have a type. The internals of the Stateflow tool will deal with the typing and implementation of such events.

8.6 Event Typing
All events external to a feature level Stateflow block will be of *function call* type. This allows simulation control to be completely specified in the model and not at all dependent on the built in simulation scheduler in Simulink. This will allow for a realistic representation of our own scheduling systems.

All Simulink subsystem p-spec blocks are to be triggered subsystems and must be triggered by a function call event generated from within a Stateflow diagram.

**Note:** Scheduler type statecharts with an interface to the physical world may be of an edge (rising/falling/either) type to accurately mimic the interrupt behavior of the Scheduling system. For example, the rotating engine model in the continuous time domain will generate a PIP event every so many crank angle degrees. This interrupt will invoke a series of control strategy models, just as in the final software implementation.

### 8.7 Wired Event Connectives

Stateflow output function call events are explicitly wired to receiving Simulink subsystem or Stateflow blocks.

The function call event is a single point to point event. It is scoped via the Stateflow data dictionary to be visible at the point in the system hierarchy where it is initially defined and all levels below that. The process execution order is explicitly under the designers control in this case. This is the recommended approach for most applications. Refer to the diagram below.

The signal carrying an output event from a Stateflow diagram shall be labeled with the event name for direct connections to receiving Pspecs or Cspecs.

In the case where a function call event signal path is broken by Goto/From connectors, only the connectors and not necessarily the signals need be labeled with the event name (i.e., do_pspec). This is done for readability purposes in the final user documentation.

Refer to the example in the figure below.
8.8 Broadcast Events

Broadcast events communicate between Stateflow blocks without wired connections. They are intended to trigger a dedicated function with multiple instantiations across several features. The order of execution of each destination is not specified. Broadcast events are not to be used.

8.9 Data Dictionary

The names of all Stateflow data and event elements must be entered in the State flow data dictionary. This is the only way to create input/output ports to interface with Simulink diagrams.

Data dictionary elements may be referenced directly in the Stateflow diagram. Unless the data element is new to the Ford Parameter Dictionary, do not fill in any data element attributes.

To view the contents of the Stateflow data dictionary, a model explorer is available under the Tool menu on the Stateflow diagrams (Explore option).
The Explorer shows the hierarchy of the currently opened models and the defined events and data, as well as their scope for the selected Stateflow chart. Refer to the Matlab help for detailed directions on how to operate the Explorer.

### 8.10 Port Names

The Stateflow data dictionary entries for data inputs and outputs are managed by port name. In turn, the Stateflow port names are referenced in the Stateflow graphical specification of the function.

For Stateflow diagrams of a single instantiation (i.e., not a reusable library block), the Stateflow port name should match the attached Simulink signal flow to aid in modeling and representation. This will also facilitate data consistency when the integrated Simulink/Stateflow data dictionary is provided.

When a parameter is both an input and output from a single Stateflow diagram, the following conventions should be followed.

If the parameter is only written to within the single Stateflow diagram and not written to in any other diagram (Simulink or Stateflow) within the system, the parameter should be declared in the Stateflow Explorer as an Output To Simulink from the Stateflow diagram. By doing this, it can be referenced both as an input and written to as an output from within the Stateflow diagram.

If the parameter is to be written to in more blocks than just the one Stateflow block, the parameter needs to be brought in as an input with a different name. The characters '_in' should be appended to the signal name when declaring the input to the Stateflow block. This creates a potential issue with the user documentation since the port labels are turned off in the final user documentation. Inside the Stateflow diagram, the signal labeled as `variable` will be referenced as a parameter named `variable_in`. One shall make an unconditional assignment as the first executable statement in the Stateflow diagram to graphically map the parameter `variable_in` to the signal `variable`. Refer to the example Stateflow diagram below:
And inside the Chart diagram:
For reusable Stateflow diagrams, the port names should be descriptive to aid in connectivity and the Stateflow port names will most likely be different than the Simulink flow names.

Refer to the example in the Figure below of a re-usable Stateflow diagram for a flip-flop and note the generic port names used.
8.10.1 Data Types

There are two distinct categories of data in the model. Those that have to do with control flow specifications and those that have to do with data flow specifications.

Typically, control flow variables are Discrete-Enumerated: Flags, Enumerated Types, Counts, Events and other tool specific control flow parameters used for switch control. Data elements used purely to facilitate modeling (control input to multi-port switch, for example) and should not be in the final software should be given a name that begins with the characters 'ml_'. This convention will be used to show model only parameters that should not necessarily be coded and do not appear in the Ford Parameter Dictionary.

Data flow variables are continuous variables that hold values associated with continuous data (i.e., Temperature, RPM, Load, Spark, etc).

Calibration data can be either Discrete-Enumerated (i.e., hardware present switches) or Continuous (i.e., enabling/disabling thresholds, controller setpoints, open loop commands for output control, etc...).

Again, care must be taken within the Stateflow diagrams that data types are properly selected to support the type of conditional statements used. It can be problematic if floating point numbers are compared to integer values in an equality comparison. Select the data type in the Stateflow Explorer carefully. One should not modify data type for any parameter within the Stateflow explorer from the default (REAL-DOUBLE) unless bit operations or other data type operation is necessary.

Note that within Stateflow LOCAL variables are the equivalent of STATIC C function variables. They keep their value from one invocation to the next within Stateflow.

TEMPORARY variables behave more like AUTOMATIC C function variables in that they are initialized each invocation of the Stateflow diagram they are owned by.

WORKSPACE variables are used to hold all calibratable constants.

CONSTANT variables are used to hold literal constants (typically defined by #define in C). If these are needed globally throughout the model, they can be promoted in scope using the Stateflow explorer to the Model (M) level.

8.10.2 Data Assignments

Stateflow diagrams can assign or transform Discrete-Enumerated variables and Tool specific variables as needed. That is, variables like counters can be incremented, decremented, reset, etc… Any calculation necessary to control logical flow within a Stateflow C-spec can be performed on discrete enumerated data.

Additionally, any calculation, on either discrete enumerated or continuous variable, can be performed in a Stateflow C-spec if that variable is LOCAL or TEMPORARY to the Stateflow block.

Stateflow C-spec diagrams should not be used to assign continuous variables values except as outlined below.
To ensure good separation of control and data flow processes, Stateflow C-spec diagrams can only assign continuous variables a constant value if the Stateflow block serves as the (or one of the) original source(s) of the variable. In other words, the Stateflow block should not transform a signal. Stateflow C-specs can assign one continuous variable signal to another. It can initialize a signal to a constant or calibration parameter value.

For example, setting \( ECT=0 \) or \( ECT = ACT \) in a Stateflow C-spec diagram is permissible, but setting \( ECT = ACT/5 + 32 \) is not. Care must be taken when using Stateflow C-specs for this purpose. Initialization actions like \( ECT=0 \) may result in the need for the merge block with the output of a Simulink block which calculates the non-initialization values for ECT which can complicate the data flow of the algorithm model. It is far better form to consolidate all modes of writing to a particular variable to a single architectural element within the model. This makes the model easier to de-bug and easier to understand as the reader of the model.

Another option is to introduce a model only data switch variable to control selection of the correct data flow path for a particular variables calculation.

Switches can be use to select from multiple data flow sources. The switch control is via the Stateflow block. Control logic can not be built into the Simulink block diagram. The following example illustrates the proper use of the switch.

### 8.10.2.1 The Data Switch

The example below illustrates the preferred method of using the modeling switch variable. This switch variable is used to control which of a number of different equations should be used in a P-Spec to calculate a particular variable (when more than one equation can be used under different conditions).

This modeling variable switch will not necessarily appear in the implemented code that will be written from these models. It is purely a convention used in the algorithm models to allow for proper separation of data and control flow information.

In the example, a gain value can take on one of two different values. The first when the engine is hot and the second when the engine is cold.

The C-Spec makes the decision of whether the engine is hot or cold and sets a modeling switch variable (it must contain the characters `ml_` at the beginning of the name). The modeling switch variable will be passed to the P-Spec and used as the controlling input to a multiport switch (from the simulink/nonlinear block palette). The value of the switch will decide which of two different constants is used as the value of the gain parameter.

First is the top level diagram in Simulink for this example shown in the figure:
Next, the contents of the Cspec Stateflow diagram in the figure:

Example to illustrate the use of the switch. The Cspec selects from one of two calibrated gains based on ect. It is called through the trig_Cspec event input.
And finally, the contents of the Pspec Simulink block in the figure.

Figure 74

In this example, input1 (ml_1) was given a value in a Stateflow diagram based on some combinatorial logic and passed to this Pspec as a control flow parameter. All conditional logic MUST be located in the Stateflow diagram.

Figure 75

Please note that the multi-port switch should be used in this case, not the threshold switch. This is primarily because the threshold switch 'hides' the threshold value in the mask dialogue and can lead to
difficult to diagnose problems during simulation. When reading the model in PDF, it is also impossible to
determine what value the threshold is set to. The threshold switch also unnecessarily mixes control flow
logic into Simulink data flow diagrams. There are cases when a true threshold function is desired, but this
should always be below a masked subsystem to prevent confusion when reading the model in user
documentation form.

An alternate (and in this case, better) form of the same functionality would place the Stateflow diagram on
the triggered P-Spec diagram. In this case the triggering function, trig_Cspec would be changed to a
do_Pspec directly. The Stateflow diagram would be executed as part of the execution of the P-Spec.
Simulink would schedule it to run based on data flow analysis of the P-Spec body. This is allowed because
the Stateflow diagram does not output any function calls, therefore it can 'free float' on the Pspec diagram.
The model would then look like that shown in the figure:

![Diagram](image)

Figure 76

Notice the P-Spec has been masked in this example. When the model P-Spec block is double clicked, the
popup shown in the figure is displayed. The rest of the model structure is hidden.

![Popup](image)

Figure 77
Below the mask the model looks like that shown in the figure:

![Diagram](image)

In this example, ml_1 is given a value in the Cspec Stateflow diagram based on some combinatorial logic and passed directly to the switch as a control flow parameter. All conditional logic MUST be located in the Stateflow diagram.

**Figure 78**

The Stateflow diagram has been slightly modified to remove the output function call event and the input function call event. It is shown below in the figure:
An alternate (and less desirable, in this case) to using the data switch in this example would be to employ the MERGE block. Instead of encoding the `ect` condition into the `ml_1` data switch and passing it to a single Pspec, two Pspecs could be formed and their outputs merged. The controlling Stateflow Cspec would trigger one or the other of these two Pspecs and the result would pass through the MERGE to create the final output. Refer to the the figures 68-70 below:
In this example, the MERGE block is used instead of the data switch. The Stateflow diagram decides which of low_gain or hi_gain to run and then merges the result.

**Figure 80**

Inside the Stateflow diagram the diagram would be the following:
Inside the individual gain calculation blocks (one shown here - they are similar) is the following:

\[
\begin{cases}
\text{do\_hi\_gain;}
\end{cases}
\]

\[
\begin{cases}
\text{do\_low\_gain;}
\end{cases}
\]

\[
\text{[ect < WARM\_ECT]}
\]

---

Copyright Ford Motor Company 1999
Both of the above approaches, one using the data switch and one using the merge are acceptable. It is purely a matter of style which is chosen.

In the case of the data switch, all the different ways the output can be calculated are shown in one Simulink subsystem, but one must introduce the data switch control variable, ml_1.

In the case of the merge, the different ways the output can be calculated are spread out over multiple subsystems and the merge block must be introduced, but no additional ‘model only’ parameters are needed.

For this trivial example, one would not actually use any of the approaches shown above. Remembering Stateflow blocks can write to continuous variables, so long as only assignments to constants or simple assignments to other continuous variables are taking place within the Stateflow diagram, a single Stateflow diagram would suffice. The single diagram, strictly speaking is a Pspec (it does not produce any function calls), however it is a combination of data and control flow. Refer to the figure below:

![Figure 83](image_url)

Example to illustrate the production implementation. The Pspec performs all the actions necessary to carry out the function. No Mask is needed.

And inside the Pspec Stateflow block:
When a Stateflow diagram is acting as a P-spec, any calculation can occur inside the diagram. A Stateflow diagram that does not send out any function calls and is either free-floating on a Simulink diagram or triggered with a ‘do_’ type function call is a P-spec.

8.10.3 Bit Operations

All built in bit operations are allowed within the Stateflow diagrams. Refer to the user documentation for the full set of allowed operations. Stateflow allows enablement and disablement of the bit operations through a configuration switch in the Coder Commands field. When enabled, certain operators change their behavior from logical operations to binary operations. All Stateflow models should be constructed assuming bit operations have been ENABLED, even if they are not for a particular model. This is necessary to support integration testing of models built independently.

For example, the | operator is a logical OR without bit operations and a bitwise OR when bit operations are enabled. If a logical OR is wanted, use the || which is has behavior which is bit operations mode independent.

9. Reusable Model Libraries

Standard block diagram utilities such as FOX functions, tables, limiters and filters will be managed as a library. This library will be managed by a central MATLAB librarian.
These library models are true references, rather than copies, to the central model used to perform the utility function. As the library models evolve, individual feature models will be automatically updated to reference the new model as soon as the new library is moved to your local PC.

Stateflow diagrams are not yet supported for library management.

It is envisioned that feature models will be managed as library components. This will facilitate feature model sharing among PCSE developers and architecture control and development.

9.1 Standard Utilities

The following standard utilities will be available for use on Simulink diagrams. Their functionality mimics that of standard PCSE utilities. The standard utilities are located in the model library named utils. This library shall be mounted on a shared drive accessible by all algorithm modelers.
9.1.1 Calibration Constants

A calibration constant is a parameter that represents a single numeric value. Calibration constant values will be managed through the Matlab workspace. A special version of the constant block has been placed in the utils library that displays the constant name and value on the block face. Please use it instead of the simulink/sources library constant block.
The generic definition of a calibration constant, for use in Matlab, is as follows.

\[ \text{<constant\_name>} = \text{<value>}; \]

where \text{<constant\_name>} is the name which identifies the constant within the Matlab model diagrams (show all calibration constants in upper case), \text{<value>} is a decimal numeric value.

Examples:

\[
\text{EGC\_CONFIG} = 1; \\
\text{EGC\_TH\_I\_GAIN} = 0.03027;
\]

When defined in this manner in the Matlab workspace, a constant can be referenced by name in dialog boxes like the one shown here in the figure for the \text{utils Constant} block. One shall not enter any mathematical expression (though allowed by Simulink) in these blocks. This would only complicate unit testing and code generation later.

![Constant Block](image-url)  
**Figure 86**

In a Simulink model, the constant then appears as shown below in the figure.

![Simulation Model with Constant](image-url)  
**Figure 87**
9.1.2 FOX (library component)

A calibration function is a collection of paired input/output values. A single calibration function is represented by a single structured variable in the Matlab workspace containing two elements (.X and .Y). Each element is a one-dimensional vector.

The generic definition of a single calibration function, for use in Matlab, is as follows.

```matlab
<function_name>.X = [<x_value> <x_value> <x_value> <x_value> <x_value> <x_value> ];
<function_name>.Y = [<y_value> <y_value> <y_value> <y_value> <y_value> <y_value> ];
```

*Where:*

- `<function_name>` is the name which identifies the function within the Matlab model diagrams (show all calibration values in upper case),
- `<x_value>` is a decimal numeric input value,
- `<y_value>` is a decimal numeric output value.

".X" and ".Y" portions of the element names are uppercase.

Example:

```matlab
FN123.X = [ -39 -20 -10 0 30 50 80 ];
FN123.Y = [ 0.80 0.75 0.60 0.40 0.60 0.90 1.3 ];
```

When defined in this manner in the Matlab workspace, a function can be referenced by name in dialog boxes like the one shown here for the fox block in the figure.

![Figure 88](image)

In a Simulink model, the f(x) function then appears as shown below in the figure.
Intermediate variables may be used when calibrating in the MATLAB workspace. An intermediate variable might be useful for several reasons. It allows the easy reuse of a commonly occurring set of breakpoints, or it may help to maintain or document a link to the actual software implementation of the calibration function (not recommended however).

Examples:

Reuse:

all_ones = [1 1 1 1 1];
FN123.X = [-40 -20 0 20 60];
FN123.Y = all_ones;
FN123.X = [-39 -10 0 50 100];
FN123.Y = all_ones;

Documenting implementation details:

FN123x = [-40 -20 0 20 60];
FN123y = [1 0 0 0.5 1];
FN123.X = FN123x;
FN123.Y = FN123y;

The fox block is available in the **utils** library and should be used for all F(x) lookups rather than the built in simulink/Nonlinear/Look Up 2-D block.

### 9.1.3 TABLE LOOKUP (library component)

A calibration table is a two-dimensional array of output values, indexed by means of two input parameters.

The generic definition of a single calibration table, for use in Matlab, is as follows.

```plaintext
<table_name>.X = [ <x_value> <x_value> <x_value> <x_value> ];
<table_name>.Y = [ <y_value> <y_value> <y_value> ];
<table_name>.Z = [ ...
    <z_value> <z_value> <z_value> <z_value>; ... ]
```

Copyright Ford Motor Company 1999
Structured Analysis Using Matlab/Simulink/Stateflow  
Modeling Style Guidelines

\[
\begin{align*}
&<z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}>; ... \\
&<z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}>; ... \\
&<z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}> <z_{-value}> \\
\end{align*}
\]

where  
\( \langle \text{table_name} \rangle \) is the name of the complete data structure representing the table (using upper case),  
\( \langle x_{-value} \rangle \) is a decimal numeric input value,  
\( \langle y_{-value} \rangle \) is a decimal numeric input value,  
\( \langle z_{-value} \rangle \) is a decimal numeric output value.

The \( \langle x_{-value} \rangle \)'s of the ".X" element correspond directly to the columns of the table array and are integers 0 - (#columns - 1).  
The \( \langle y_{-value} \rangle \)'s of the ".Y" element correspond directly to the rows of the table array and are the integers 0 - (#rows - 1).  Values for the indices are inputs to the table block and usually come from the output of a f(x) normalizing function.  
Z-array indexing in MATLAB starts at (1,1) in the upper left corner of the array.  
Each row within the array is concluded with a semicolon, except the last row.

The x-vector, y-vector, and z-array are therefore related to each other as indicated in the table below within the MATLAB workspace.

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;y_{-value}(1)&gt;</td>
<td>&lt;x_{-value}(1)&gt; &lt;x_{-value}(2)&gt; &lt;x_{-value}(3)&gt; &lt;x_{-value}(4)&gt; &lt;x_{-value}(5)&gt;</td>
</tr>
<tr>
<td>&lt;y_{-value}(2)&gt;</td>
<td>&lt;z_{-value}(1,1)&gt; &lt;z_{-value}(1,2)&gt; &lt;z_{-value}(1,3)&gt; &lt;z_{-value}(1,4)&gt; &lt;z_{-value}(1,5)&gt;</td>
</tr>
<tr>
<td>&lt;y_{-value}(3)&gt;</td>
<td>&lt;z_{-value}(2,1)&gt; &lt;z_{-value}(2,2)&gt; &lt;z_{-value}(2,3)&gt; &lt;z_{-value}(2,4)&gt; &lt;z_{-value}(2,5)&gt;</td>
</tr>
<tr>
<td>&lt;y_{-value}(4)&gt;</td>
<td>&lt;z_{-value}(3,1)&gt; &lt;z_{-value}(3,2)&gt; &lt;z_{-value}(3,3)&gt; &lt;z_{-value}(3,4)&gt; &lt;z_{-value}(3,5)&gt;</td>
</tr>
</tbody>
</table>

The x-vector, y-vector, and z-array are therefore related to each other as indicated in the table below within the Ford Parameter Dictionary.  Notice that the MATLAB array element \( z(1,1) \) corresponds to the PARAMETER DICTIONARY table element FNxxxx(0,0).

<table>
<thead>
<tr>
<th>PE</th>
<th>Columns = FN(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows = FN(Y)</td>
<td>&lt;z_{-value}(1,1)&gt; &lt;z_{-value}(1,2)&gt; &lt;z_{-value}(1,3)&gt; &lt;z_{-value}(1,4)&gt; &lt;z_{-value}(1,5)&gt;</td>
</tr>
<tr>
<td>&lt;z_{-value}(2,1)&gt; &lt;z_{-value}(2,2)&gt; &lt;z_{-value}(2,3)&gt; &lt;z_{-value}(2,4)&gt; &lt;z_{-value}(2,5)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;z_{-value}(3,1)&gt; &lt;z_{-value}(3,2)&gt; &lt;z_{-value}(3,3)&gt; &lt;z_{-value}(3,4)&gt; &lt;z_{-value}(3,5)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;z_{-value}(4,1)&gt; &lt;z_{-value}(4,2)&gt; &lt;z_{-value}(4,3)&gt; &lt;z_{-value}(4,4)&gt; &lt;z_{-value}(4,5)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Example:

\[
\begin{align*}
FN1234.X & = [ \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 ]; \\
FN1234.Y & = [ \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 ]; \\
FN1234.Z & = [ ... \ 64 \ 64 \ 64 \ 54 \ 98 \ 98 \ 70 \ 51 \ 32 \ 23; ... \\
& \ 28 \ 28 \ 28 \ 28 \ 48 \ 48 \ 48 \ 40 \ 29 \ 20; ... \\
& \ 20 \ 20 \ 20 \ 20 \ 38 \ 32 \ 38 \ 32 \ 26 \ 18; ... \\
& \ 12 \ 12 \ 12 \ 15 \ 23 \ 23 \ 23 \ 27 \ 23 \ 15; ... \\
& \ 6 \ 6 \ 6 \ 12 \ 19 \ 20 \ 23 \ 22 \ 20 \ 13; ... \\
& \ 0 \ 0 \ 0 \ 0 \ 13 \ 16 \ 16 \ 18 \ 15 \ 10; ... \\
\end{align*}
\]

Copyright Ford Motor Company 1999
When defined in this manner in the Matlab workspace, a table can be referenced by name in dialog boxes like the one shown here for the table block shown in the figure.

**Figure 90**

In a Simulink model, the table then appears as shown below in the figure.

**Figure 91**

**NOTE:**

In order to properly use the table block for the PCSE applications, the normalizing functions should also be explicitly shown as separate FOX blocks in your models. See the example below in the figure:
A nice short hand notation to use when defining the data in the workspace for the table is to set the .X and .Y input vectors to the output of the normalizing functions. For example, if the table FN1234 uses FN123 for the X normalizing function and FN124 for the Y normalizing function, declare the table as follows:

```matlab
FN1234.X = FN123.Y;
FN1234.Y = FN124.Y;
```

The fox block is available in the `utils` library and should be used for all F(x) lookups rather than the built in Simulink/Nonlinear/Look Up 2-D block.

### 9.1.4 ROLAV_TC (library component)

The ROLAV_TC block implements the time based rolling average equation:

```matlab
classdef ROLAV_TC < Simulink.BlockDiagram
    % new_average = \[ \frac{\text{sample interval}}{\text{sample interval} + \text{time constant}} \] * 
    % \( \text{new_value} - \text{old_ave} \) + \text{old_ave}
    %
    % The time constant is entered through the dialogue block shown below in the figure:
end```

```matlab
d = new_average = \[ \frac{\text{sample interval}}{\text{sample interval} + \text{time constant}} \] * 
\( \text{new_value} - \text{old_ave} \) + \text{old_ave}
```

The time constant is entered through the dialogue block shown below in the figure:
When used in this manner, the block will appear in your model as shown below in the figure. The value for `old_ave` is usually the last pass value of `new_ave`. Under special circumstances, the ROLAV_TC block will, however, need to be seeded with a different, or reset, `old_ave` value and therefore the additional input is available rather than incorporation of this functionality within the block.

This block calculates the elapsed time between calls internally. Therefore it is very important that the block be invoked each execution cycle. If this is not done, a large change in the output of the block will be noticed when triggering the block after it has not been triggered for some time. If it is not possible to trigger the block each pass, then another version of the block has been placed in the library which allows one to pass the delta time between calls to the block as an input. It is shown below in the figure:
There are other cases where the time constant is a calculated, not a constant value. In this case, the time constant must be passed as an input to the block. Again, we have two versions of the block. The first computes delta time internally and must be called each pass and is shown below in the figure:

![Figure 95](image)

The other allows the designer to pass both the time constant and the delta time as inputs to the block. This is the most general form of the block and could be used in all cases, if desired. It is shown below in the figure:

![Figure 96](image)
One final version of the block exists which has the trigger built into it. In this case, the time constant is entered through the mask and must be a constant and the delta time is calculated internally and therefore the block must be called each pass. It is shown below in the figure:

9.1.5 ROLAV_FK (library component)

The ROLAV_FK block implements the event based rolling average equation:

\[
new\_ave = f \times new\_value + (1 - f) \times old\_ave
\]
Where $f$ is the filter constant of the equation. The filter constant is entered through the mask dialogue, as shown below in the figure:

**Figure 99**

When used in this manner, the block will appear in your model as shown below. The value for `old_ave` is usually the last pass value of `new_ave`. Under special circumstances, the ROLAV_FK block will, however, need to be seeded with a different, or reset, `old_ave` value and therefore the additional input is available rather than incorporation of this functionality within the block. Refer to the figure below:

**Figure 100**

There are times when the filter constant is a calculated value and must be passed as an input to the block. In this case use the block shown below in the figure:
Finally, another version of the block has been constructed which has the trigger built in. The filter constant must be entered through the mask and is therefore must be a constant value. Refer to the diagram below in the figure:

**Figure 101**

![Diagram of the block with trigger built in](image)

9.1.6 CLIP (library component)

A special CLIP block has been prepared that will perform a min and max clip and will also display the clip values on the icon of the block. The mask dialogue for this block is shown below in the figure:

**Figure 102**

![Diagram of the CLIP block](image)
The block will appear in your model as shown below in the figure with the clip values on the icon.

If a clip to only a maximum or minimum value is required, but not both, merely enter inf or -inf as the unwanted clip value. This will be displayed on the block face.

9.1.7 FAULT CODES (library component)

In order to specify control of a fault code, a block has been designed for the utils library. The block has two function call event input, one to specify a malfunction and the other to specify a clear_malf. Function calls starting with trig_ should be wired to these ports. The code number is entered through the block dialogue and appears on the block face. An output port is shown which output the fault code data vector. At this time, the fault code block is non-functional and is shown in your model only to specify interface to the diagnostic executive. The block will become functional at some point in the near future. NOTE:
Only ONE fault code block for a given fault code number may appear in any model. Refer to the figure below:

![Diagram of TIMER component](image)

**Figure 105**

### 9.1.8 TIMER (library component)

It is common practice to use a software timer to time events as they occur. A special block has been created that acts as a software timer. It can count up, down or hold. It can be reset to any arbitrary value. It will output its value at any arbitrary rate as defined by the user of the block.

The inputs to the block are made both through direct wiring of the block as well as through a mask dialogue. The dialogue prompts for the output resolution of the time value (in units of seconds) and for the initial value of the timer. Refer to the figure.

The hard wired inputs include the function call event to execute the timer and have it post a new value at its output, a function call event to reset the timer to the value input at the reset port of the timer and a pair of inputs used to control the direction the timer will count.

If the increment flag input is set to 1 and the decrement flag is not 1, then the timer will count up. If the decrement flag is 1 and the increment flag is not 1 then the timer will decrement. If the increment and decrement flag inputs are set to the same value, then the timer will hold its current value.
The first entry is used for block labeling purposes and should match exactly the name of the timer.

The initial value of the timer can also be specified through the second entry.

The timer output is also optionally clipped between the specified clip values if the check box ‘Clip Timer?’ is checked.

Finally, the timer units are selected as either SECONDS (with microsecond accuracy) or MILLISECONDS (with 1 millisecond accuracy).

The block will appear in your model as shown below in the figure:
9.1.9 VARIABLE SATURATION (library component)

It is sometimes desirable to clip a parameter to a calculated value. The CLIP block cannot perform this function. In this case the VARIABLE SATURATION block should be used. The block dialogue describes the block functionality as shown below in the figure:

Figure 108
The block will appear in your model as shown below in the figure.

![Diagram of Variable Saturation - Minimum Clip block](image)

**Figure 109**

### 9.1.10 VARIABLE SATURATION - MINIMUM CLIP (library component)

A similar block the basic Variable Saturation is available for those times when only a minimum clip is desired. Refer to the figure below:
9.1.11 VARIABLE SATURATION - MAXIMUM CLIP (library component)

A similar block to the basic Variable Saturation is available for those times when only a minimum clip is desired. Refer to the figure below:

![Figure 111](image1)

9.1.12 DOC-LINK BLOCK (library component)

There is a specially designed block to allow hypertext linkage within your models. The DOC Link Block will perform this function. Refer to the figure. If the block is clicked on from within the model, the
Netscape browser will be brought up and pointed to a user defined location. The default location supplied with the block is The Mathworks web site, www.mathworks.com.

9.1.13 CALIBRATION FILE SPECIFICATION (library component)

A special block has been created and included in the utils library to pre-calibrate a model when opening the model. The block, when dragged into a model and an M-file is specified through its dialogue, will set the PreLoadFcn of your model to execute the M-file specified. This can be used to load all the calibration values automatically when the model is opened. The block will appear in the model as shown below in the figure:

![Figure 112](image)

![Figure 113](image)
The mask dialogue looks like that shown in the figure1:

![Figure 114](image)

**9.1.14 PID BLOCK (library component)**

A special block has been created to allow definition and declaration of Parameter Ids (PIDs). There is one input to the block which represent the data to be mapped to the PID. Through the block dialogue, the PID mode, PID number, any bit field information and control parameter name are entered. Pull downs are used for PID mode and bit field information. Refer to the figure2.

![Figure 115](image)
And the dialogue looks like that shown in the figure3:

Figure 116

9.1.15 CRIB SHEET LINK BLOCK (library component)

A special block has been created to allow one to open one model from within another model. This technique has been used when a Stateflow block has been used as a documentation crib sheet drawing tool. In general it can be used to link any two models together. The models will NOT simulate at the same time. The block looks like the figure4:
When the block is open, the following instructions on the use of the block are displayed. Refer to the figure 5.

CRIB Sheet SAMPLE model.

To use this you need to:

1. Break the library link.

2. Edit the mask help text to be the name of the model you wish to open when double clicking on this block.
9.1.16 MULTI-PORT DE-SWITCH (library component)

A special block has been created to write a value to one particular element within an array or vector. This is like the multiport switch built in block, except it runs in reverse. The first input to the block is the array index (starting with the first element number '1') to which a value should be written. The second input is the data that should be written to the element specified by the index. All other elements within the array keep their current value. In order to do this, the block retains the last pass value of the output internally as a state.

The block implements the following:

Output vector @ time t = Internal State = \( X_t \)
Output vector @ time t+1 = \( Y_{t+1} \)
Index input = \( U(1) \)
Data input = \( U(2) \)

\( Y_{t+1}(U(1)) = U(2) \)
All other elements of \( Y_{t+1} = X_t \)
\( X_{t+1} = Y_{t+1} \)

Since the block keeps track of the last pass value (or State) internally, it is important that this block be the only source for the output vector. No merging of the output of this block and the output of another block is possible.

The number of elements in the output vector must be entered through the block dialogue.

Refer to the diagram below in the figure6:

![Diagram of Multiport Deswitch](image)

**Figure 119**
The block dialogue looks like the figure 7:

![Diagram of Multiport DE-switch](image)

\textbf{Multiport DE-switch - internal last pass val...}

- Multi-port DE-switch (mask) [link]

**MultiPort DE-switch:**

Opposite operation to the MultiPort Switch block in the simulink/nonlinear library.

The block will route the value on the signal connected to input port 2 to the output specified by the value of the signal connected to input port 1.

The output vector width is dynamically sized and specified through the mask dialogue.

ALL other elements of the output vector will remain UNCHANGED.

**Parameters**

- Width of Output Vector

If it is required to write to the output vector in more than one location, then the alternate form of the block should be used. In the form, the last pass value of the block must be fed into the third input of the block. Refer to the diagram below in the figure 8:
9.1.17 Delta_Time Block

There are cases when the elapsed delta time between execution cycles is needed exactly in the embedded controller. To specify this functionality is required within Simulink, the Delta_Time block is to be used. This blocks essentially outputs a constant, idealized execution cycle time and draws its value from the Matlab base workspace. Thus, moving a complete task to a new rate can be performed simply by altering the scheduler model and changing the parameter in the workspace corresponding to the task being moved. The mask dialog for the block is shown below:
One need only pick from the Task Number pop up a number 1-5 (there are only 5 fixed rates in the OS available - this block can be extended in the future if more become available). Then the standard feature abbreviation is to be typed in (case insensitive).

For example, if TASK 3 is chosen and the feature MYFEAT is entered, the block will appear as:

![Figure 124](image)

**Figure 124**

IMPORTANT: It is then the modelers responsibility to create the workspace parameter ML_MYFEAT_TASK3_DT in the base workspace. If this is not, the model will not simulate and the following error dialog box will appear whenever the block initialization commands are executed:
10. Data Dictionary Interaction

In the short term, there is no data dictionary to specify the Simulink data flow attributes. The Stateflow tool is currently equipped with a capable data dictionary tool called the Stateflow Explorer. An integrated Simulink / Stateflow data dictionary is planned for the future.

The data type, scope, persistence, ownership, range and resolution, initial or nominal values, and descriptions of the data shall continue to be specified in the Ford Parameter Dictionary. The Ford Parameter Dictionary will remain the master source of data management. Mechanisms are available to:
- generate a list of data entities used in a model.
- automatically assign calibration values from the Ford calibration management system (either default values from the Parameter Dictionary or actual values from a Symbols file.

To create an initial calibration file using the default values from the parameter dictionary, follow the following steps.

Open the model.
Run the M-file `ml2pd` by typing the following from the MATLAB command line:

```
ml2pd('model_name')
```

Ensure all duplicate entries or entries for parameters that should not exist in the Parameter Dictionary are removed from the 'ModelName_cals.txt' file.
FTP the resulting file named 'ModelName_cals.txt' to the PED2 system.
Run the unix `pd2cmd` command which is described below as follows:

```
pd2cmd -matlab -f1 -len 15 ModelName_cals.txt model_name_fcal.m
```

FTP the file `model_name_fcal.m` back to the PC where MATLAB is running.
Execute the M-file that was just created.

This procedure will load all constants in the model with the base values from the parameter dictionary.

10.1 Pd2cmd
The following text is what one would see if they typed:

>pd2cmd -h

from the UNIX command line on the PED2 system:

```
pd2cmd v2.4 Development built on Apr 16 1998 at 10:49:41
OSF/1 version

Parameter Dictionary to Control Model Data
Usage:
pd2cmd [options]  inputfile  outputfile

Options:

-h or -H  This help message
-a       Make sure parameters are authorized
-matrixx  Output MATRIXX control model data
-matlab   Output MATLAB control model data
-f0       Format for MATLAB (FNaaa -> Xaaa, Yaaa) (default)
-f1       Format for MATLAB (FNaaa -> FNaaa.X FNaaa.Y)
-s name   specify a symbols file to use
=req      use parameter dictionary required values
-imp      use parameter dictionary implementation values
-v        verbose mode
-ptece     symbols file is PTEC oriented
-806x     symbols file is 806X oriented
-model name  Overrides the default model name
-n        for tables output normalization function values
-len num  use num for the maximum length of parameter names

The default model for -matrixx is MATSAVE
```

11. Model Initialization

Model initialization is a complex issue and modeling techniques to represent behaviors under all initialization conditions have not been fully worked out and condensed. Modifications to the base tool set may be required to efficiently model initialization behaviors.

This section will describe a number of approaches for dealing with model initialization. These should be considered suggestions only until such time as consensus is reached in how to model initialization.

It is important to fully specify what values all parameters must take during system initialization. This can include both Keep Alive Memory (KAM) and Random Access Memory (RAM) initial values.

In general, parameters can be initialized to zero, a constant value or a calculated value. Refer to the diagram below in the figure:
Initialization can be performed as a separate, stand alone execution context invoked during system reset or can be performed the first time through the normal execution context where parameters are controlled. To complicate this further, some parameters can be written to in more than one execution context. In this case the initialization must occur prior to usage of the parameter in any context.

Consider the following model which contains examples for each type of initialization:

Above the context diagram level, refer to the example in the figure:
Inside the OS model, refer to the figure:
Here one can see that the power down state of the controller is explicitly modeled as the 'down' state. Within the powered up state, 'up', concurrent flow charts are used, one for each context.

Upon power up, the event `do_feature_ram_init` event is sent out. This will force execution of the RAM initialization context. A separate context can be used for RAM when the initialization is to a constant value. This is equivalent in the C code to specifying the initial value on the data declaration. In this case, the compiler will generate code that treats the initialization as a separate execution context and load the initialization values before any other code is executed. In order to model this so it matched exactly the behavior that is expected in the final product, the separate execution context is used.

After the 'up' state is entered, a first pass flag, one per execution context, is set. Only the following task rates should be used for the PPC applications. The following naming conventions must be followed in naming the first pass flags:

- `ml_fp_100ms`
- `ml_fp_50ms`
- `ml_fp_30ms`
- `ml_fp_16ms`
- `ml_fp_8ms`
- `ml_fp_pip`
The feature is then invoked and the first pass flag is cleared. This mechanism is used for initializing parameters to calculated values (not constants). In this case, the Cspecs throughout the system must recognize the first pass flag and take the initialization actions required to set parameters/states to calculated initial values.

Finally, the kam_error flag must be monitored in any context that uses KAM. When the flag is set, KAM initialization actions must be taken within the feature.

A further complication of these mechanisms occurs when parameters are written to in more than one execution context and they are initialized to calculated values. In this case a convention has been adopted (so that the code and model behaviors will match - the convention was first adopted by the PTEC/Black Oak coding staff) to perform the initialization in the SLOWEST non-interrupt context where the parameter is written. This also implies that the OS model must be set up properly to invoke the slower processes first upon power up. This is done by locating the slower process physically higher in the 'up' state as is shown above.

Inside the feature, first is the feature level execution context diagram as shown in the figure2:
Inside the stand alone RAM initialization context one would insert something similar to the following the figure3:
The output from this block must then be merged with the normal calculation of the parameter. Refer to the earlier execution context diagram.

If we proceed down inside the background context we will see the following the figure4:
Please note that the naming convention for the C-specs have not been strictly followed here. This is only done to clarify the flow-chart versus state machine type of C-spec.

In this context we are writing to three outputs. The first is initialized in the stand alone RAM initialization context and therefore no initialization actions will be taken in the background sequence on this parameter.

The second output is KAM and therefore is reset when the kam_error flag is found to be set.

The last output is initialized to a calculated value, and therefore when the context detects the first pass flag set, the initialization will take place. This parameter is also written to in the fast foreground context. Since the background is the slower context, by convention described earlier, the background will be the context where initialization will occur.

Inside the C-Spec for the flow chart one would see what is shown in the figure5:
In the case of the state machine, one would see the following what is shown in the figure:

Figure 132

In the case of the state machine, one would see the following what is shown in the figure:
Inside the Pspec, the multiport switch is used to pick the correct value. Refer to the diagram and associated comments below in the figure7:
Figure 134

Inside the foreground context is the following as shown in the figure:

[Diagram description goes here]
In this case, since the output parameter is also written to in the background and initialized to a calculated value, no additional initialization actions need be modeled in this context. The details of the Cspec and Pspec are not shown here as a result.

12. Unit Testing Considerations

Please refer to the following web site for details on Unit testing. Specifically the document, "MATLAB Coding guidelines for Unit Testing"

"http://www.pcse.poee.ford.com/cacsd/

13. Conclusion
We have presented a number of modeling conventions using The Mathworks products Matlab, Simulink and Stateflow in this document. More issues will certainly arise as more and more people begin to apply this tool and methodology to their particular features. Feedback from those individuals will be used to further refine these conventions and enhance the document.

The concepts presented in this guide are the result of work done by a large number of individuals within SRL and PCSE during early testing and development of the CACSD process. Special thanks to Bob Baskins, Al Beydoun, Ken Butts, Scott Fable, Kim Ferrell, Cory Ho, Bob Jentz, Barb Mattison, Kevin MacFarlane, Patrick Menter, Shankar Raman, Kevin Rzemien, Siva Shivashankar, Steve Toeppe, and Tony Tsakiris for their involvement in the definition of the original style guidelines. Without their commitment to the success of the CACSD process, these guidelines would not be possible.

Please contact Paul Smith (PSMITH13, x89448 or x41507), or Ken Butts (KIBUTTS1, x30814) with any questions or issues that may arise during the use of the tools or application of this style guide or enter a problem support through the WEB site:

http://www.darborn.ford.com/csed/cgi-bin/matlab.cgi
14. Index

I

1/z · 23, 24

A

abs() · 73
action language · 57, 60
annotation · 24
arrays · 38, 40
AUTOMATIC · 86

B

Black Oak · 129
Broadcast events · 82

C

calibration constants · 7, 99
calibration procedures · 19
case · 10, 12, 13, 25, 37, 38, 42, 43, 53, 55, 56, 57,
60, 61, 62, 64, 65, 67, 68, 70, 71, 81, 82, 89, 90,
95, 99, 100, 102, 106, 107, 108, 113, 126, 128,
129, 133, 136
change(data_name) · 73
clear_malf · 110
CLIP · 7, 109, 113
clockwise · 44
code generation · 4, 41, 99
code generation · 4
color · 37
combinatorial · 42
compiler · 73, 128
compound · 47, 48, 49, 50, 51, 52
concurrency · 67
concurrent · 67, 68, 70, 72, 128
condition actions · 42, 43, 60
connective junction · 42, 43, 44, 45, 53, 54
connectors · 36, 37, 38, 81
consistency · 4, 7, 36, 43, 58, 83
contant · 7, 23, 36, 86, 87, 98, 99, 104, 106, 107,
108, 109, 124, 125, 128, 129
Constants · 36
comment diagram · 5, 6, 7, 9, 10, 25, 40, 126, 129, 131
control flow · 4, 37, 53, 86, 87, 90
crib sheet · 118
C-spec · 10, 17, 132

d data dictionary · 56, 81, 82, 83, 124
data flow · 4, 10, 36, 54, 86, 87, 90, 124
data store memory · 25
Data Stores · 25
data type · 10, 86, 124
default transition · 43, 56, 57, 59, 62, 71, 136
delay · 23, 24
delta time · 105, 106, 107
Demux · 38, 40
directed event broadcasting · 71
DOC Link Block · 115
documentation · 4, 6, 12, 17, 19, 20, 24, 37, 38, 81,
90, 118
Drawing commands · 18
Drop shadows · 24

E
eMBEDDED SOFTWARE · 36, 40
entry(state_name) · 73
execution context diagram · 9, 10, 25, 129, 131
exit · 37, 43, 53, 57, 60, 70, 73
Explorer · 67, 68, 74, 83, 86, 124

F

fabs() · 73
fault code · 110
feat_vxx · 6
filter constant · 108, 109
first pass flag · 128, 132
flowchart · 42, 43, 44, 46, 47, 59, 64, 65, 69, 70
For loop · 53
FOX · 96, 103
fox block · 100, 101, 104
fprintf · 18
From · 36, 37, 38, 81
Function block · 24
function call · 10, 16, 38, 57, 58, 81, 82, 111

G
global · 36
Goto · 36, 37, 38, 81
Goto/From · 36, 37, 38, 81
guarded transition · 43, 44
Structured Analysis Using Matlab/Simulink/Stateflow
Modeling Style Guidelines

separation · 87
sequenced logic · 43
S-Functions · 41
shared constants · 7
signal labels · 5, 36, 124
simulation · 4, 41, 43, 56, 58, 73, 81, 90
Simulink · 4, 5, 9, 10, 16, 17, 23, 24, 36, 42, 54, 56,
   69, 80, 81, 82, 83, 85, 87, 89, 90, 95, 97, 99, 100,
   103, 104, 124, 136, 137
single entry · 43, 70
single exit · 43, 70
state hierarchy · 62, 65
state machine · 42, 43, 57, 58, 59, 61, 64, 66, 67, 71,
   132, 133
Stateflow · 4, 6, 7, 10, 12, 17, 36, 40, 42, 43, 44, 53,
   54, 55, 56, 57, 60, 67, 68, 69, 72, 73, 74, 80, 81,
   82, 83, 85, 86, 87, 88, 90, 91, 92, 93, 97, 118, 124,
   136, 137
subroutine · 70, 71

T

table block · 102, 103
tags · 37, 38
TEMPORARY · 86
threshold switch · 89
timer · 10, 111
timing · 24
tool specific variables · 36
transition action · 43, 60
trigger port · 10

U

unguarded · 43, 44, 45, 53, 56, 59, 67, 71
unit testing · 4, 36, 41, 99, 136
unstructured loop · 54
utilities · 96, 97
utils · 41, 97, 98, 99, 101, 104, 110, 116

V

VARIABLE SATURATION · 113
vector · 38, 40, 100, 102, 104, 110, 120, 121
vectors · 38, 40, 104
version · 4, 5, 6, 24, 53, 98, 105, 106, 107, 109, 125
visibility · 10

W

web site · 116, 136
While Loop · 53