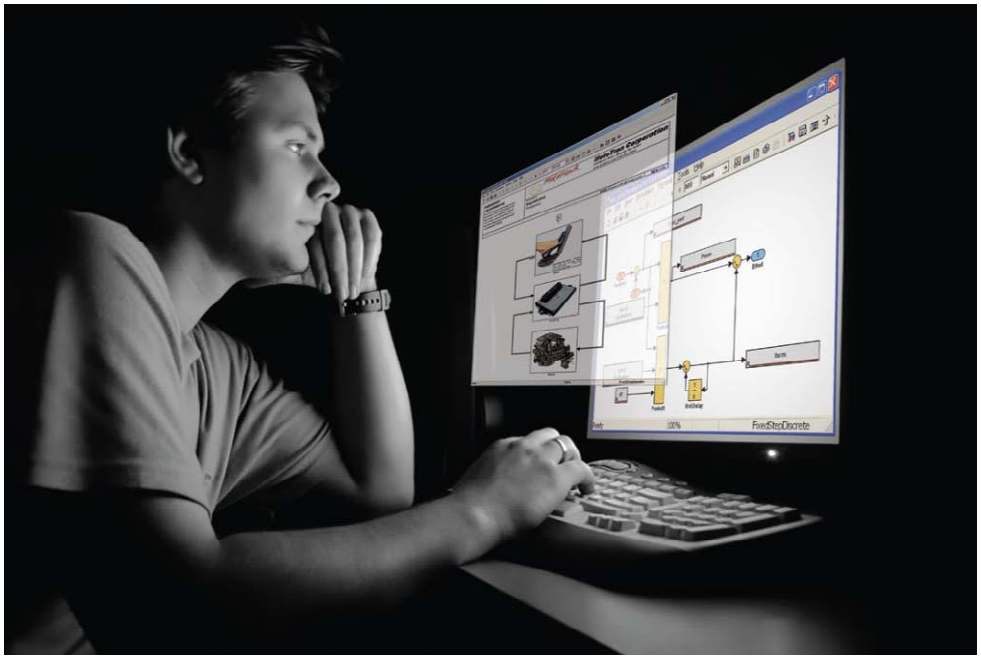
WOODWARD

Product Manual 36333
(Revision A)
Original Instructions



MotoHawk Development and Prototyping System

Resource Guide

IMPORTANT

This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

DEFINITIONS

- **DANGER**—Indicates a hazardous situation which, if not avoided, will result in death or serious injury.
- **WARNING**—Indicates a hazardous situation which, if not avoided, could result in death or serious injury.
- **CAUTION**—Indicates a hazardous situation which, if not avoided, could result in minor or moderate injury.
- **NOTICE**—Indicates a hazard that could result in property damage only (including damage to the control).
- **IMPORTANT**—Designates an operating tip or maintenance suggestion.

WARNING

The engine, turbine, or other type of prime mover should be equipped with an overspeed shutdown device to protect against runaway or damage to the prime mover with possible personal injury, loss of life, or property damage.

The overspeed shutdown device must be totally independent of the prime mover control system. An overtemperature or overpressure shutdown device may also be needed for safety, as appropriate.

Read this entire manual and all other publications pertaining to the work to be performed before installing, operating, or servicing this equipment. Practice all plant and safety instructions and precautions. Failure to follow instructions can cause personal injury and/or property damage.

This publication may have been revised or updated since this copy was produced. To verify that you have the latest revision, be sure to check the *publications page* on the Woodward website:

www.woodward.com/searchpublications.aspx

The current revision and distribution restriction of all publications are shown in manual *26311*.

The latest version of most publications is available on the *publications page*. If your publication is not there, please contact your customer service representative to get the latest copy.

Any unauthorized modifications to or use of this equipment outside its specified mechanical, electrical, or other operating limits may cause personal injury and/or property damage, including damage to the equipment. Any such unauthorized modifications: (i) constitute "misuse" and/or "negligence" within the meaning of the product warranty thereby excluding warranty coverage for any resulting damage, and (ii) invalidate product certifications or listings.

NOTICE

To prevent damage to a control system that uses an alternator or battery-charging device, make sure the charging device is turned off before disconnecting the battery from the system.

NOTICE

To prevent damage to electronic components caused by improper handling, read and observe the precautions in Woodward manual *82715*, *Guide for Handling and Protection of Electronic Controls, Printed Circuit Boards, and Modules*.

Revisions—Text changes are indicated by a black line alongside the text.

Contents

Contents (cont'd.)

Illustrations and Tables

Electrostatic Discharge Awareness

All electronic equipment is static-sensitive, some components more than others. To protect these components from static damage, you must take special precautions to minimize or eliminate electrostatic discharges.

Follow these precautions when working with or near the control.

1. Before doing maintenance on the electronic control, discharge the static electricity on your body to ground by touching and holding a grounded metal object (pipes, cabinets, equipment, etc.).

2. Avoid the build-up of static electricity on your body by not wearing clothing made of synthetic materials. Wear cotton or cotton-blend materials as much as possible because these do not store static electric charges as much as synthetics.

3. Keep plastic, vinyl, and Styrofoam materials (such as plastic or Styrofoam cups, cup holders, cigarette packages, cellophane wrappers, vinyl books or folders, plastic bottles, and plastic ash trays) away from the control, the modules, and the work area as much as possible.

4. Do not remove the printed circuit board (PCB) from the control cabinet unless absolutely necessary. If you must remove the PCB from the control cabinet, follow these precautions:

    - Do not touch any part of the PCB except the edges.

    - Do not touch the electrical conductors, the connectors, or the components with conductive devices or with your hands.

    - When replacing a PCB, keep the new PCB in the plastic antistatic protective bag it comes in until you are ready to install it. Immediately after removing the old PCB from the control cabinet, place it in the antistatic protective bag.

NOTICE

To prevent damage to electronic components caused by improper handling, read and observe the precautions in Woodward manual 82715, *Guide for Handling and Protection of Electronic Controls, Printed Circuit Boards, and Modules*.

Chapter 1. General Information

About MotoHawk

MotoHawk® is a controls system application development tool that allows the user to create Simulink® diagrams that run on rugged, automotive quality embedded control modules. The tool allows you to access the inputs and outputs of the modules, schedule when to execute tasks, manipulate the memory usage of the module, create a calibration interface, and most importantly, allows a single step build of the entire application. It extends Simulink and Real-Time Workshop Embedded Coder to generate code necessary to interface with the resources of the modules and control their behavior.

MotoHawk is built on Woodward's ControlCore® production software framework and supports a variety of applications using both single controller and distributed-by-wire implementations. It is intended for control feature development, vehicle calibration and fleet testing.

Features

- Auto-code generation of Simulink/Stateflow models using Embedded Coder/Stateflow Coder
- Rugged controllers for prototyping and production
- ControlCore enabled software
- Off-the-shelf engine control libraries
- Calibration using MotoTune or CCP based tools
- Responsive engineering and support services for a wide-range of applications
- Electronic control modules available for development, fleet and production

Benefits

- Simpler, faster development
- Better testing using real ECM hardware
- Quickly develop and enhance software features in Simulink
- Analyze and control real-time OS from Simulink/Stateflow
- Direct access to the production controller's I/O from Simulink
- Readable documentation of system design automatically created from models
- Lower cost for fleet testing; outfit an entire test fleet with rapid prototyping capability
- Custom block-set allows for integration of both handwritten and auto-code

® Simulink is a trademark of The MathWorks, Inc.

ECM565-128 Developer's Kit

Parts List

ITEM NO.	DESCRIPTION
1	ECM565-128 development module
2	ECM565-128 harness w/main power relay and fuse
3	Power switch assembly w/SmartCraft™ connector
4	SmartCraft to dual DB-9 adapter (GMLAM)
5	SmartCraft to dual J1939 adapter
6	10' SmartCraft cable w/terminating resistors
7	10' SmartCraft cable
8	SmartCraft terminating connector
9	6-port SmartCraft hub (2)
10	Optically isolated 4-port USB hub †
11	USB to dual CAN adapter
12	Green Hills Software MULTI2000™ compiler*
13	Software installation CD*
14	Security dongle*
15	Boot key

(TM) SmartCraft is a trademark of the Mercury Marine division of Brunswick Corporation

(*) Green Hills software, security dongle programming, and applications included on software CD are subject to your specific order and may not be included in shipment.

(†) USB hub may not included in kit as it is part of the Kvaser hardware and in future orders may not be included.

System Requirements

- Windows XP (any SP) or later (Windows 7 on MotoHawk 2010a and higher)
- Pentium III or IV, Xeon, Pentium M, AMD Athlon, Athlon XP, Athlon MP3.
- 345 MB disk space
- 512 MB RAM (1 GB or more recommended)
- 16, 24, or 32 bit OpenGL capable graphics adapter (strongly recommended)
- Microsoft Windows supported graphics accelerator card, printer, and sound card
- 1400x1050 display (min) (1600x1200 strongly recommended)

Note: User should also consider system requirements for MATLAB, Simulink, RealTime Workshop, and RealTime Workshop Embedded Coder.

MATLAB™ Installation Procedure

1. Insert CD in drive. If the installer does not start automatically, click Start/Run and double click on Autorun.exe.
2. Follow the instructions on the screen.

   Note: If you have a network license for your installation you will need to obtain a demo license from The Mathworks before training.
3. Install all of the following:
   - MATLAB
   - Simulink
   - Real Time Workshop
   - Realtime Workshop Embedded Coder
4. It is strongly recommended that you also install:
   - Stateflow
   - Stateflow Coder

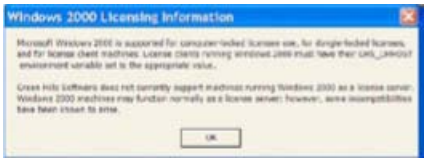
Green Hills Software Installation Procedure

Insert CD in drive. Click Start/Run and double click Setup.exe. Follow on-screen instructions.

Obtaining a License for Your MotoHawk Compiler

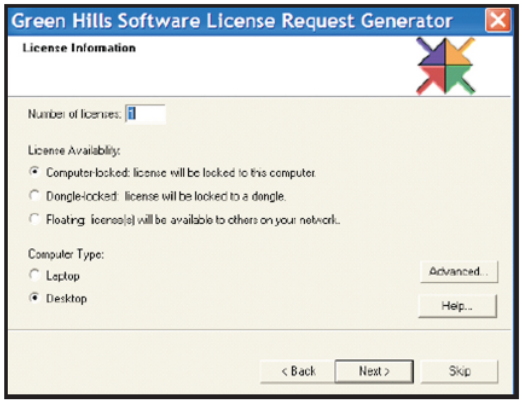
Once you have completed installation of the compiler on the unit that you will be using to develop your application, you must generate a request for a license.

1. Select Programs/MULTI2000,PowerPC v3.6/ Licensing/License Request Generator.
2. Select “OK” at the following screen.



   Each MotoHawk SDK includes one node locked license. Contact your sales representative if more are desired.

3. Indicate which type of computer you have installed the compiler on and select “Next.”



4. Select “Next.”

5. The next message window contains the License Agreement. Read it, then select “Yes” to continue.

6. You must accept License Agreement in order to use the compiler.
7. The next window contains the license request. Print or Save To File, then send it. An evaluation license will be sent to the e-mail address indicated in the Customer Information window, usually the same day.

8. Follow the instructions that accompany the license file. A hard copy of the License Agreement was included with your SDK.
9. FAX a signed copy to (805) 965-6343, Attn: Mickey Neal. Or email a copy to Mickey.neal@ghs.com

   A permanent license will be e-mailed to the address indicated in the Customer Information window (usually the next business day.)

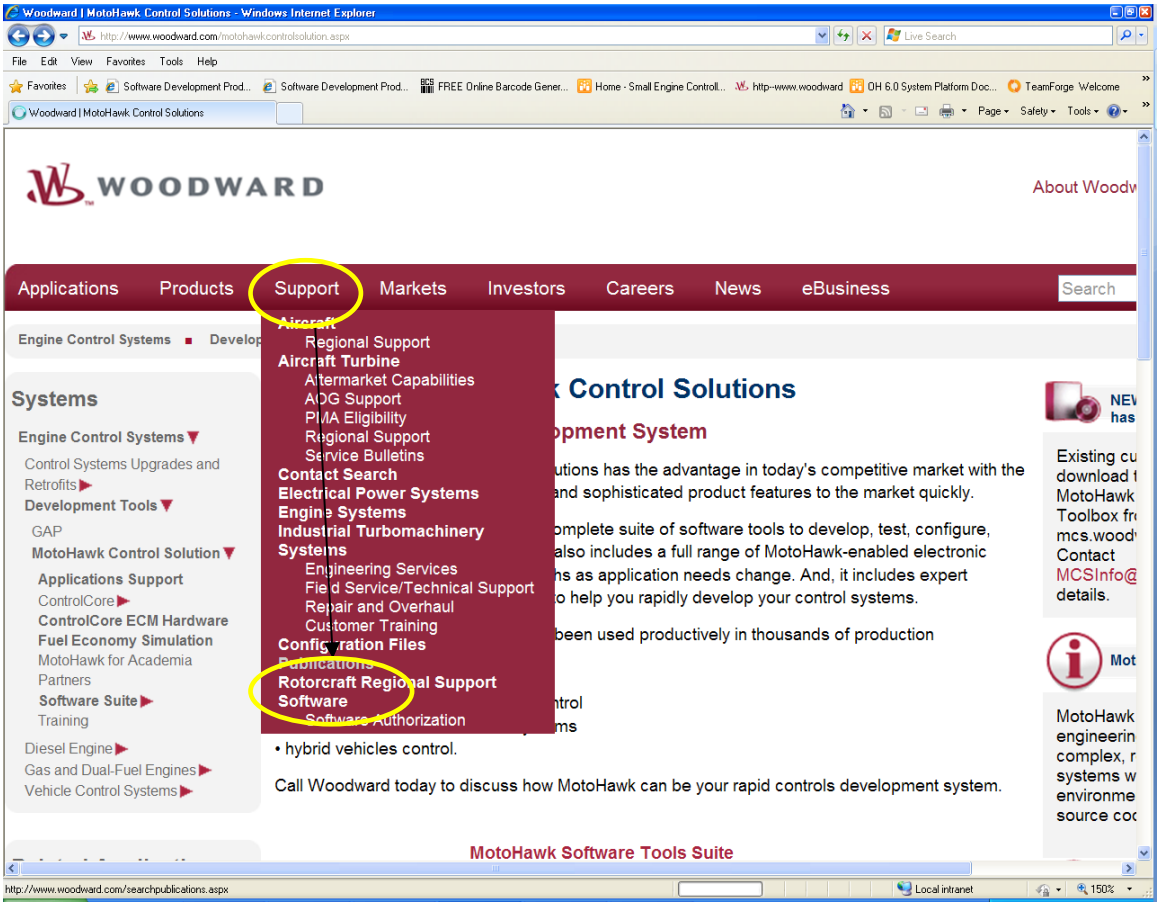
GCC Compiler Installation Procedure

MotoHawk 2009b and higher supports the use of the GCC compiler as Beta Trial. This is useful for getting up and running quickly or for initial development.

Note: GHS is the recommended compiler for production programs.

1. The GCC PowerPC eabi compilers are online at www.woodward.com.
2. Navigate to Support at the top pull down menu, and select Software.
3. The software is searchable by product name, key word, etc. Enter GCC in the search field.

   Note: There are two GCC compilers. The “SPE” version is for 55xx modules such as the ECM-5554-112. The “non-SPE” is for 5xx modules such as the ECM-565-128 or ECM-563-48.
4. Download the file to a temporary location.
5. Unzip the file. DO NOT run the installer from Winzip.
6. Run the installer.



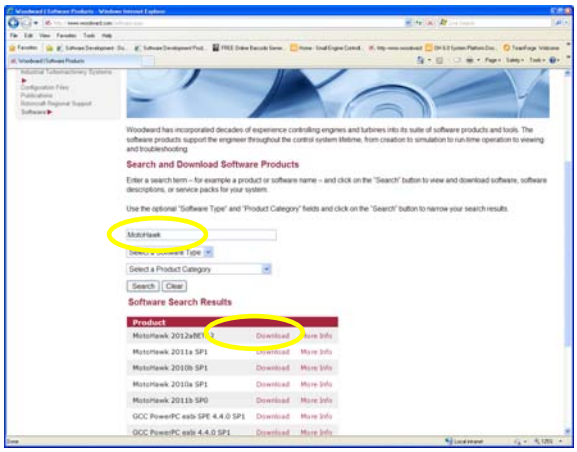
MotoHawk Installation Procedure

MotoHawk Software is also online at www.woodward.com.

1. Navigate to Support at the top pull down menu, and select Software.
2. Search the software products by name: MotoHawk, MotoTune, MotoServer Runtime
3. Download the files to a temporary directory, unzip the files, and then install downloads in the following order following the instructions for each one:
   - MotoServer Runtime
   - MotoTune
   - MotoHawk
4. Be sure your MotoHawk license dongle is in the USB port of your PC and run the MotoHawk Version Selector to associate your MotoHawk installation with your version of MATLAB.

   If your version of MotoHawk is shown in grey, the MATLAB and MotoHawk versions are not compatible.

   (Start→Programs→Woodward→MCS→MotoHawk)



NOTICE

You must use Windows Add/Remove programs to uninstall all previously installed versions of MotoServer and MotoTune prior to installation.

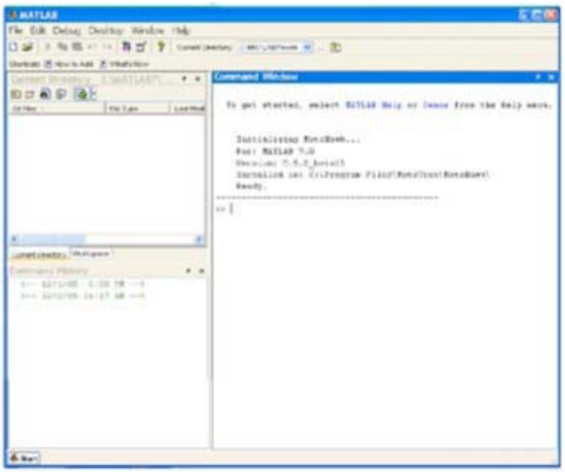
It is recommended that you DO NOT plug the adapter cable into the USB port *prior* to installing the MotoServer and MotoTune.

It is also recommended that you download the CAN King software — a useful tool when working with CAN networks.

Creating an Application in MATLAB™

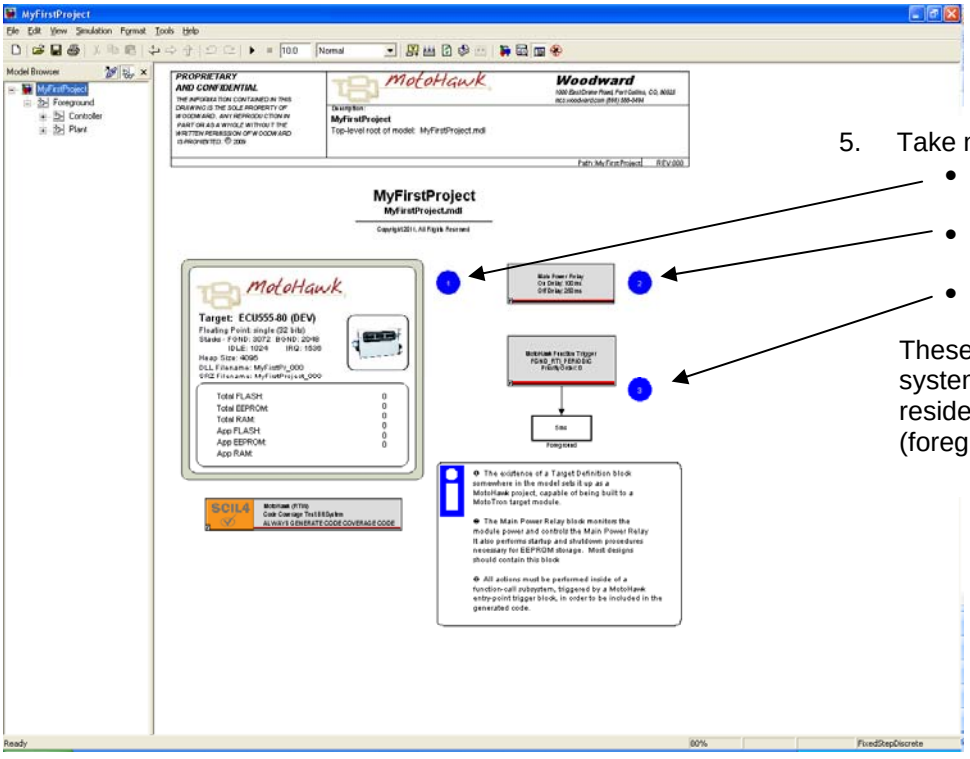
Once you have completed the installation of your software, create a model to verify operation.

1. Make sure your silver license dongle is in the USB port of your PC and start MATLAB: Double click on the MATLAB icon on your desktop or select from the programs menu.
2. The following screen will appear.



3. At the command line type: motohawk_project MyFirstProject.
4. Press the Enter key.

   The following window will open. (Allow 1-2 minutes for the application to complete.)



5. Take note of the:
   - Target Definition
   - Main Power Relay
   - Trigger blocks

   These comprise a rudimentary system. The executable algorithms reside in the Triggered Subsystem (foreground).

NOTICE

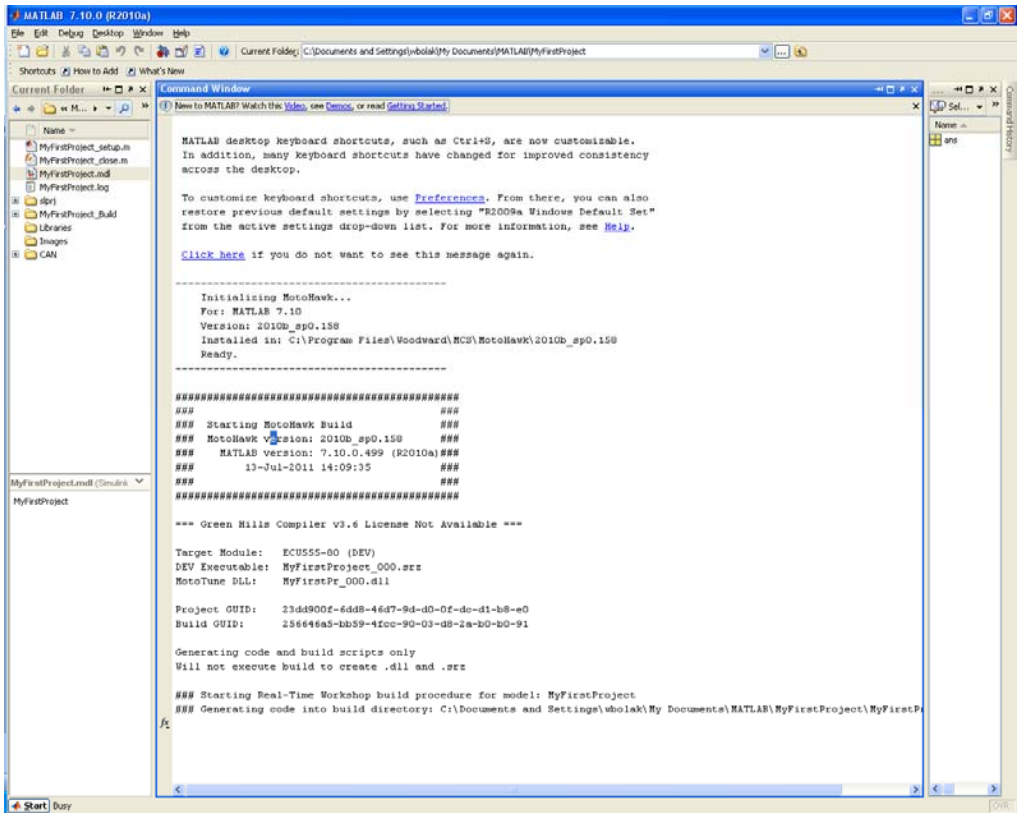
You must change your Target Definition Block to match your Hardware. Double click on the block and select your ECM model from the dropdown list.

It is recommended that the Target Definition Block be set to your hardware as the first step, as this defines the available resources for the model.

Building Your Application

1. Press CTRL+B

   The MATLAB window should look like this:



2. If the message says “Successful MotoHawk Generation (NoBuild)”:

   b) You may need to place a Tool Chain Definition Block in your model. Drag a Tool Chain Definition Block from MotoHawk/Build library in Simulink into your model and select your installed compiler.

   c) Check your Green Hills compiler installation: Type “motohawk_check_ghs” (a zero indicates that you have a problem with your Green Hills compiler installation).

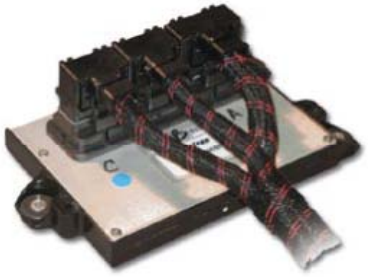
3. If you get an error, check with your instructor or email the log file (MyFirstProject.log in this example) to: MCSsupport@Woodward.com. A technical support representative will contact you.

4. Once you have successfully built your default application, open Windows Explorer and navigate to the C:\ECUFiles directory.

5. You will see a number of subdirectories including Programs and TDBDLL. These subdirectories contain, respectively, the .srz and .dll files which are used by MotoTune to program the ECU.

Assembling Your Kit

1. Install your isolated USB hub and apply power.
2. Insert your silver MotoTune dongle into the hub.
3. Connect the USB to CAN adapter and wait for Windows to auto-detect it. When the New Hardware window appears select “No, not this time” and click on “next.” Then, let Windows automatically install the drivers.
4. Connect the Development Harness to the module. (See datasheet for proper positioning.)

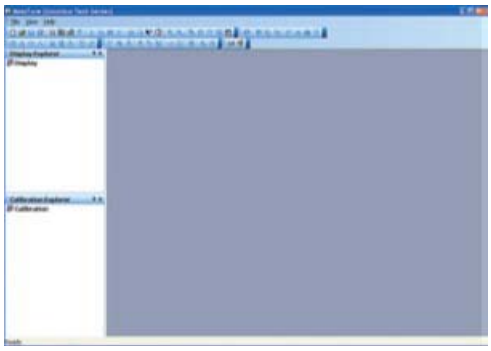


5. Connect Power branch to a 12 volt source (9V to 16 V, 3A min.) Attach the SmartCraft connector, USB to CAN adapter, and the power switch to the 6-position hub.

6. The boot key is not needed for normal programming or calibration so it can be set aside. Errors in configuration, logic and/or other programming made during program development (via .srz file) can cause a persistent loss of CAN communications with the module under development. If this happens, apply the boot key to force the module into reboot mode, reloading the module with functional program code (a known, valid .srz file) in order to allow resumption of module communication.

Starting MotoTune

1. From the Start menu (or desktop shortcut) select All Programs/MotoTools/MotoTune.
2. The following window appears.



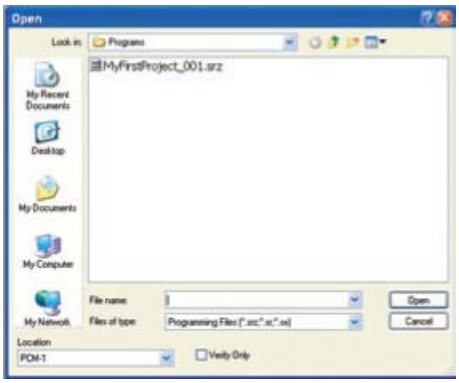
3. The name that was used to order your kit should appear at the top of the window. If it indicates [Unlicensed,] then you need to insert/reinsert the silver dongle.

Checking MotoServer

1. Right-click on the Satellite Dish icon for MotoServer (located on the system tray).
2. Select “Ports”.
3. If not already listed, add location PCM-1 as a CAN type port with Access Level 4; check the box on the list; and click on “Apply”.
4. You are now ready to connect to the module.

Programming the Module

1. Turn power on and apply ECUP signal via power switch.
2. Select File/Program, in the MotoTune window. The following pop-up appears:



3. This is the file created when you pressed CTRL+B.
4. Double-click on the .srz file in the window.
5. The Program ECU status pop up appears.

   If the Program ECU status pop-up doesn't advance to "Connecting," check your CAN to USB and SmartCraft connections. If they are operational, turn power off.

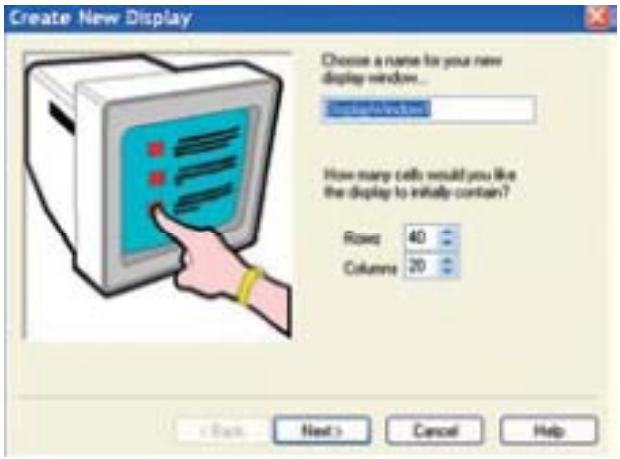
6. Install the BOOT KEY from your kit onto the SmartCraft hub.

   ECU555-128 users will also need to move the fuse from the Normal socket to the BOOT socket to insure boot loader is invoked.
7. Double-click on the .srz file and apply power.
8. If this does not work, check with your instructor or send an e-mail to: MCSsupport@woodward.com
9. When you see the "Programming Successful" message you are ready to create a display for your application.

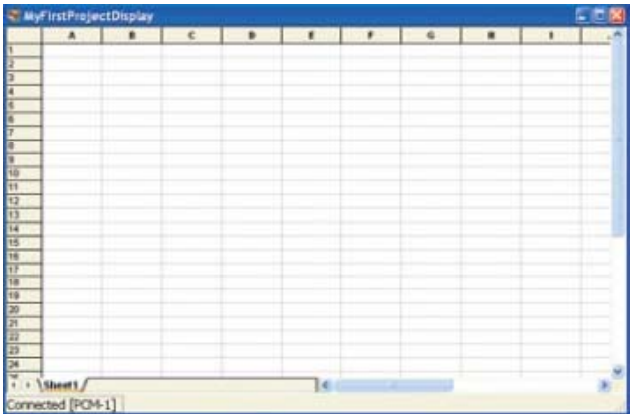
Creating a Display

1. In the MotoTune Window, select File/New/Online Display/Calibration.
2. Select Display on the pop-up and click on “OK.”

   The Create New Display window appears.



3. Give your display a meaningful name (ie. MyFirstProjectDisplay).
4. Select “Next” for default Row and Column settings.
5. Select “Next” for default Status Bar and Tab Control settings.
6. Use default Sheet1 by clicking on “Finish.” The following should appear:



7. Click on the “+” next to the MyFirstProject folder (listed on left side of the MotoTune window).
8. Open the following folders:
   - Foreground folder
   - Controller folder
   - Plant folder
9. Double-click on the Foreground block in your Simulink model.

   Note the one-to-one correspondence between the MotoTune folders and the subsystems in your model.

Checking Operation

1. Open the System folder, then the Performance folder.
2. Drag each of the display variables onto the spreadsheet.

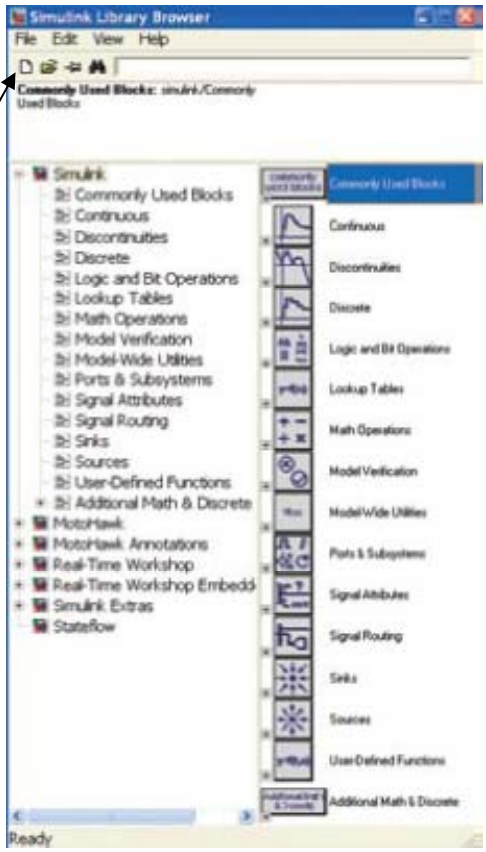
   Note that your system is running — these are its vital statistics.
3. Cycle the power switch off, then back on.

   Note that the display values briefly disappear, then return.
4. The Main Power Relay can be heard releasing and engaging.
5. Close this model by clicking on the red “X” in the upper right-hand corner.
6. You will be prompted to save the model. We are done with this one — you may save or not.

First Application

1. Click the Simulink icon  located on the top of the MATLAB window.
2. Simulink’s Library Browser appears

   These are the Simulink and MotoHawk blocks which are used for creating your application models.
3. In the MATLAB window, move up one level to the “work” directory. Create a new directory “MySecondProject” and double-click on it.
4. In the library browser, click here. A new model window opens.
5. Note the status window in the lower left hand corner. It indicates ODE45 which stands for Ordinary Differential Equation 4th and 5th derivative (Dormand-Prince method,) which is the type of solver that will be used for simulations.

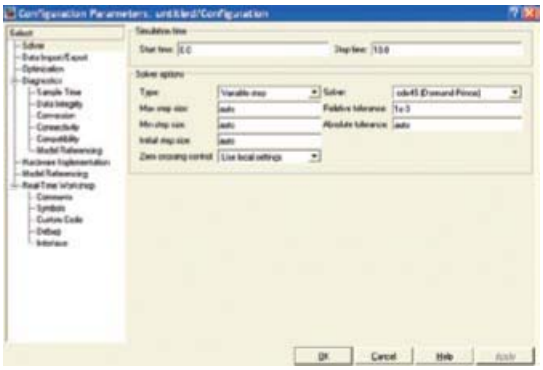


Generating Embedded Code

In order to generate embedded code we must change to a fixed-step discrete solver as follows.

1. Select “Simulation” at the top of the window, then “Configuration (or Simulation) Parameters”.

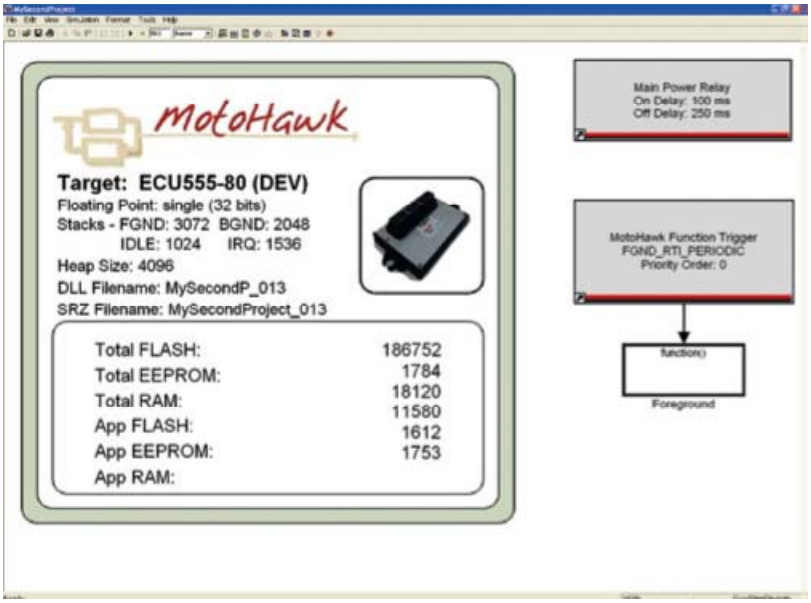
   The following window appears.



2. Using the pull downs, change Type to Fixed-step, and Solver to Discrete (no continuous states.)
3. Click on “Apply” and “OK”.
4. In the library browser, click on MotoHawk. Drag the MotoHawk Target Definition block from the bottom of the list into your model.

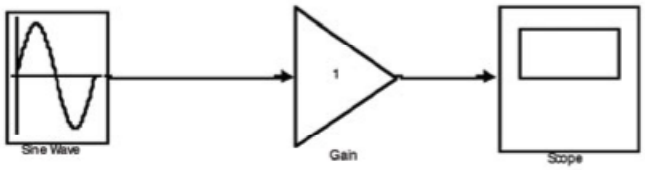
   Double-click on the block and verify that the target module is correct for your kit (80 pin, 128 pin, etc).

   The Memory Layout should be DEV.
5. Click on “Apply” and “OK”.
6. From the Trigger Blocks library, drag a MotoHawk Trigger block into your model. Double-click on the block to open the dialog box and set the pull-down to FGND_RTI_PERIODIC.



7. Click “Apply” and “OK”.
8. From the Extra Development Blocks library, drag a Main Power Relay block into your model.

   (Default settings will serve our purposes for now.)
9. From the Ports & Subsystems library drag a Function-Call Subsystem block into your model. Double-click on this block and a new window appears.
10. From the Sources library, drag the Sine Wave block from the bottom of the list into your model.

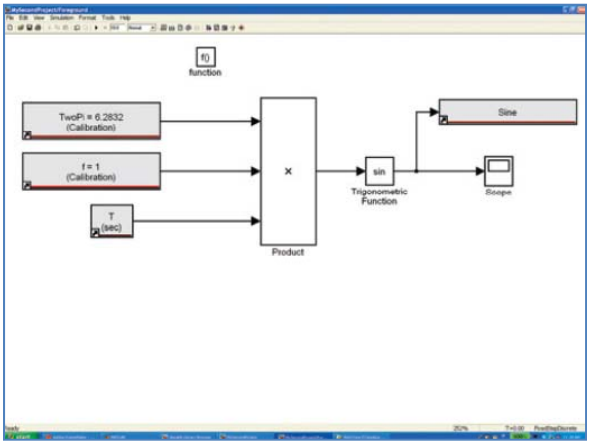


11. Click on Sinks and drag a Scope block into your model.
12. Click on Math Operations and drag in a Gain block.
13. Note the greater than (>) symbols on each block. These are Simulink ports that are used to control the signal flow through your model. The Sine Wave block, being a signal source, has only one (output) port. Likewise the Scope block, being a sink, has only one (input) port, while the Gain block has one of each.

    More complex blocks will have more input or output ports or both.
14. Select the Sine Wave block, hold down the CTRL key and click on the Gain block.

    Notice how Simulink connects the two blocks. This technique can be used to “wire” the blocks to one another and is especially useful when wiring signals to or from consecutive ports on a block. Simulink will start at the top and work down either side (in or out) of the block.
15. At the top level of your model, connect the trigger block to the subsystem block. Select File/Save As. Give your model a meaningful name (ie. MySecondProject) and click Save.
16. Press CTRL + D.

    Notice that Simulink has generated an error message and highlighted the offending subsystem and block — informing us that “only constant or inherited (-1) sample times are allowed in triggered subsystems.”



17. Double-click on the Sine Wave block to open its dialog box.

    At the bottom of the dialog box the Sample Time is zero. As you may have guessed this means continuous.

18. Change it to -1 (inherited.)

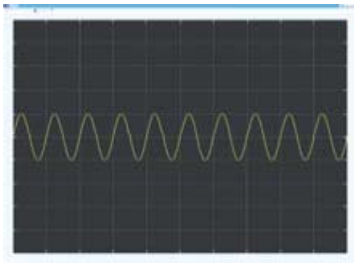
    The subsystem will now inherit its sample time from the parent (level above) which is FGND_RTI_PERIODIC or 5 milliseconds.

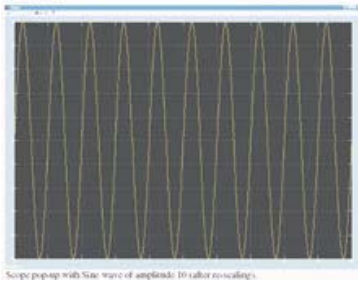
19. Press CTRL + D again.

    No error messages are generated.

20. Double-click on the scope — a pop-up window appears complete with grid and axis markings.

21. Select Simulation/Start — a Sine wave appears.





22. Double-click on the Gain block, change to 100.

    The small triangle in the middle of the window at the top can be used to start the simulation. Note that the Sine wave has changed.

23. Click on the binoculars icon.

    This will scale the display for your input automatically. Clicking on the name of the subsystem (Function-Call Subsystem) opens it for editing.

24. Change the name to “Foreground.”

25. Press CTRL + B.

    MotoHawk builds your application.

26. In the MotoTune Display Explorer pane, right-click on Display1 on [PCM-1.]

27. Select “Save As” and give it a meaningful name (ie. ”MyFirstProjectDisplays”). Use pulldown to specify the folder.

    Note that while MyFirstProjectDisplays contains only MyFirstProjectDisplay, it may contain others that provide different views into the system.

28. Right-click on MyFirstProjectDisplays and select Close.

    Currently, this is the only way to close one display and open another in MotoTune.

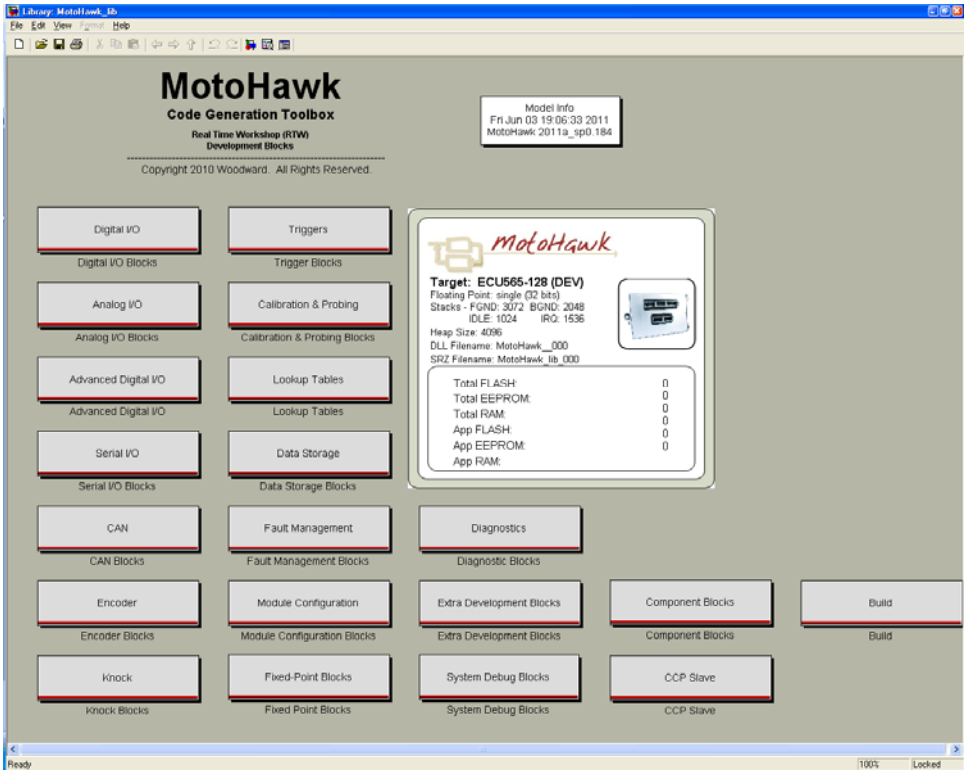
29. Select File/Program and download MySecondProject into the module.

30. Create a new display as above. (ie. “MySecondProjectDisplay”).

31. Drag in your System Performance variables and observe via your display and the Main Power Relay that your application is running.

Modifying the Application

This procedure allows you to gain some control over its operation.



1. Double-click on the Foreground block in your model; select the Sine wave generator and the gain block.

   Press the delete key to remove these blocks.

2. From the Calibration & Probing Blocks library, drag a motohawk_calibration block and a motohawk_probe block into your model.
3. From the Extra Development Blocks library, drag in a motohawk_abs_time block.
4. Double-click on the Calibration block and change the name to ‘TwoPi’ and the value to 6.28318.

   The single quotes must be used.

5. From the Math Operations library, drag in a Product block.
6. Double-click on it and change the number of inputs to 3.
7. Right-click on the TwoPi block and drag down.

   A duplicate block is added to your model.

8. Double-click on the new block and change its name to “f” and its value to 1.
9. Wire these 3 blocks to the inputs on the Product block.
10. From the Math Operations library, drag in a Trigonometric Function block.

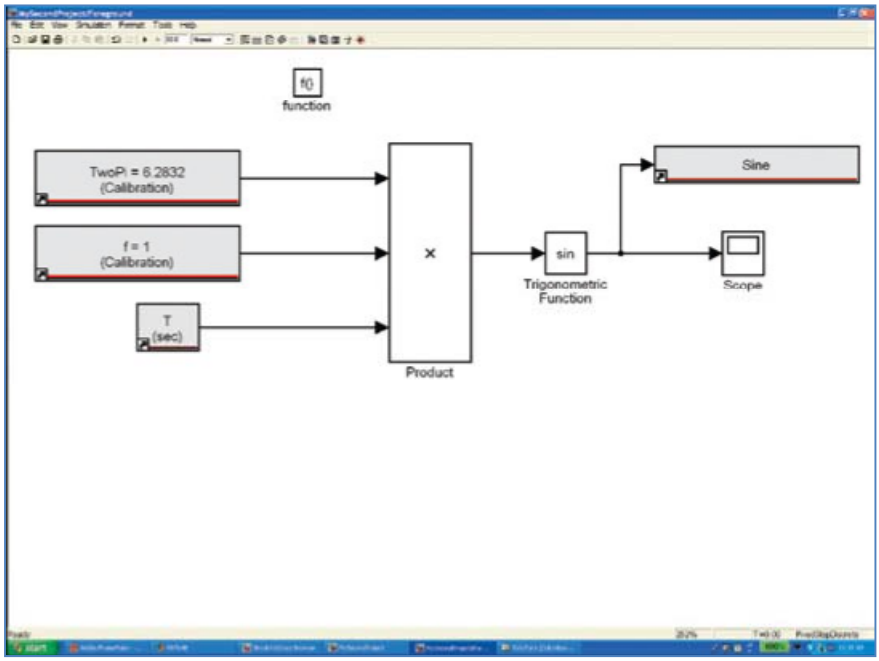
    If it is not already set to Sine, change it.

11. Wire the output of the Product block to the input of the Trigonometric Function block.
12. Wire the output of the Trigonometric Function block to the Scope block.

13. Double-click on the Probe block and change its name to Sine.
14. Place the cursor over the input port of the “Sine” Probe block.

    Notice that the cursor changes into a cross-hairs.
15. Click on the port and drag to the wire connecting the Trigonometric Function block and the Scope.

    A connection dot appears on the wire and a wire connects to the Sine Probe block.
16. Your model should look similar to this



17. Press CTRL + D (see that there are no errors).
18. Press CTRL + B (verify that the build is successful).
19. Close the display in the MotoTune Display Explorer pane as above and program the module with your modified application.
20. Select File/New and create a new calibration.
21. In the Calibration Explorer pane, Click on the “+” next to the MySecondProject folder.
22. Double-click on Foreground.

    A Calibration sheet opens in the right hand pane of the MotoTune window.
23. Create another display sheet and drag it down or to the side such that both are visible.

    You should be able to see the Sine value changing.
24. Right-click on the cell containing the Sine value and select Properties. Click on Set Fast and verify that the Add to chart/log box is checked. Click OK.
25. Select Chart/Open Chart.

    A pop up appears displaying your Sine wave.

26. In the Foreground sheet change the “f” value to 2.

    Note the frequency changes when the Enter key is pressed.

27. Change “f” to 0.5 – observe change in chart.

    Occasionally, flat spots will appear on the chart – a result of Windows OS “garbage collection” and other operations, and is no cause for concern.

Introducing a Gain Stage

There are two methods for introducing a gain stage.

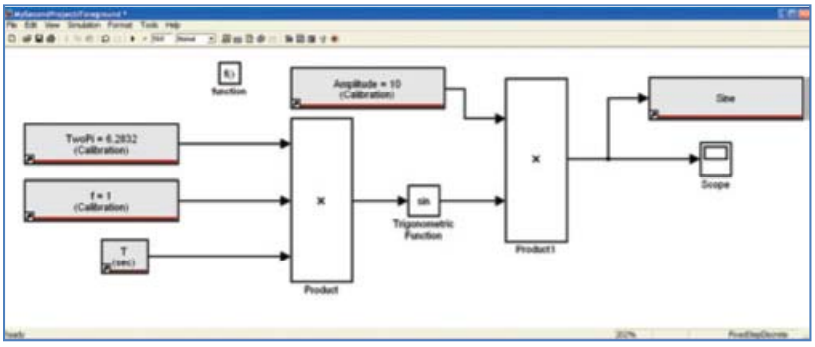
Method 1: Add a Gain block from the Math Operations library.

Method 2: Add a Product block from the same library and a Calibration block from the MotoHawk library.

In the case of a Gain block; Real Time Workshop will allow us to change the Gain value during simulation but our objective is to generate embedded code.

The RTW Embedded Coder treats a Gain block as a hard-coded constant which, precludes changes at run-time. Therefore, we will use the second approach; an “Amplitude” calibration block and a product block.

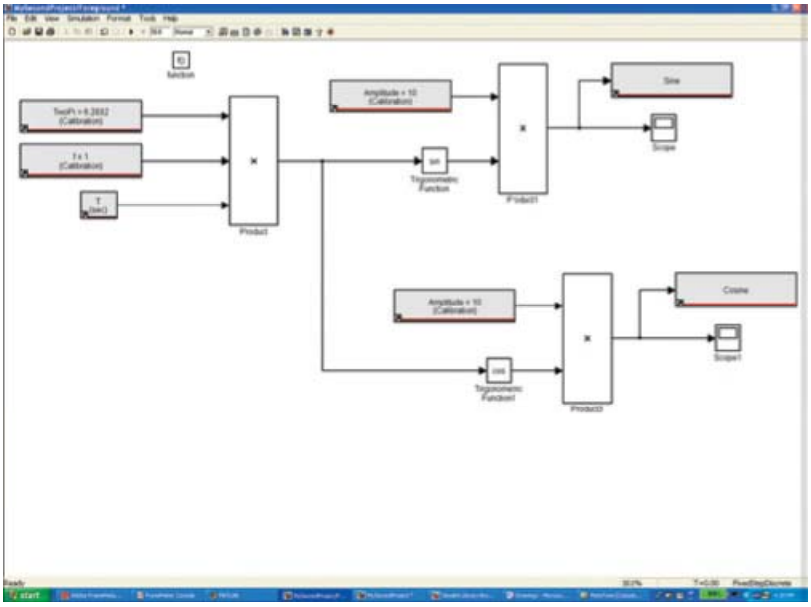
1. Select the wire connecting the Trigonometric Function block and the Scope and press the Delete key.
2. Right-click on the TwoPi block and drag a copy to one side.
3. Double-click on the new block and change the name to ‘Amplitude’ and the value to 10.
4. Likewise, copy over the product block and change its Number of inputs to 2.
5. Connect the new calibration block and the Sine block to the product block inputs.
6. Wire the product block output to the Scope and Sine probe block.
7. Your model should look similar to this:



8. Press CTRL + D and verify that there are no errors.
9. Then press CTRL + B to build it.
10. Program the module with the new application. Set up your display and calibration windows in MotoTune as before.
11. Open a chart for the Sine probe and verify the amplitude value.
12. Now change the amplitude to 100.

    Note that the display is rescaled for the new value.

    If a Cosine signal of the same amplitude is also needed, hold down the Shift key and select the Amplitude, Trigonometric Function, Product, and Sine Probe blocks from the Right side of the drawing.
13. Right-click and drag down to copy them.
14. Wire the blocks together as before, connecting the input of the Trigonometric Function to the output of the Product block on the Left.
15. Change the Trigonometric Function to Cosine and rename the Probe block accordingly.
16. Your model should look similar to this:



17. Press CTRL + D.
18. Read the error message. Simulink is complaining that the name ‘Amplitude’ is not unique. We could rename this, but we know that the value is important and it would be convenient to be able to re-use it. The way to do this is to use the MotoHawk Data Storage blocks.

MotoHawk Data Storage Blocks

1. From the library, drag a motohawk_data_def block and a motohawk_data_read block into your model.
2. Double-click on the motohawk_data_def block, change the name to ‘Amplitude’, change the Storage Class to constant, and verify that “Attach a VarDec for Visibility from MotoTune” is checked.
3. Highlight the two calibration blocks called “Amplitude” and delete them.
4. Double-click on the motohawk_data_read block, change the name to ‘Amplitude’, and drag it over to one of the loose wires left by the previous deletion.
5. Right-click on the motohawk_data_read block and drag a copy over to the other loose wire.
6. Press CTRL + D again.

   No errors should be generated.
7. Build your model, program the module, and set up your display and calibration windows as before
8. Right-click on either the Sine or the Cosine value and set the properties to:
   - Fast
   - Add to chart/log
   - Apply to all
9. Click OK.
10. Select Chart, Open Chart and observe your signals.
11. In the calibration pane change the Amplitude value and observe the changes in your signals.

    For calibration values that are used in only one place in the model, the motohawk_calibration block is a convenient means of introducing the variable.

    When a calibration is to be used in more than one place, a motohawk_data_def block with motohawk_data_read blocks is best.
12. Double-click on the motohawk_data_def block.

    A brief description of the block’s parameters appears at the top of the dialog box. In addition to the variable’s name, initial value, and storage class, we can specify a data type (click on the pull down to see them), and an Output Reference Data type (for pointer based operations.)

    Storage Class Parameter allows us to specify the type of resource that will be allocated for the variable.

    Constant, as the name implies, does not change unless a tool changes it.

    Volatile will be re-initialized at power up.

    Non-volatile will be preserved across a controlled shut-down/power-up cycle (when MPRD block or similar construct is included in the model).

MotoTune Options

Attach VarDec for Visibility

Selecting ‘Attach VarDec for Visibility’ from MotoTune expands the dialog box, giving us more options:

- a choice of which pane to view it in: Calibration or Display
- the option to restrict Read and Write access level
- whether to use uploaded calibration values from MotoTune
- how to view the value: Number, Enumeration (on, off, running, stopped,) or Text

Select the Help button at the bottom of the dialog box to view remaining options.

If the MPRD block is not used, a motohawk_store_nvmem must be included in a background subsystem in order to execute the transfer to EEPROM (with the caveat that there are a limited number of write cycles for the EEPROM devices.)

Also, when a revised model is downloaded to the module, the values stored in EEPROM will be loaded into RAM unless the structure has changed or the RestoreNVFactoryDefaults function is invoked from the System\NonVolatile Storage folder in the Display pane.

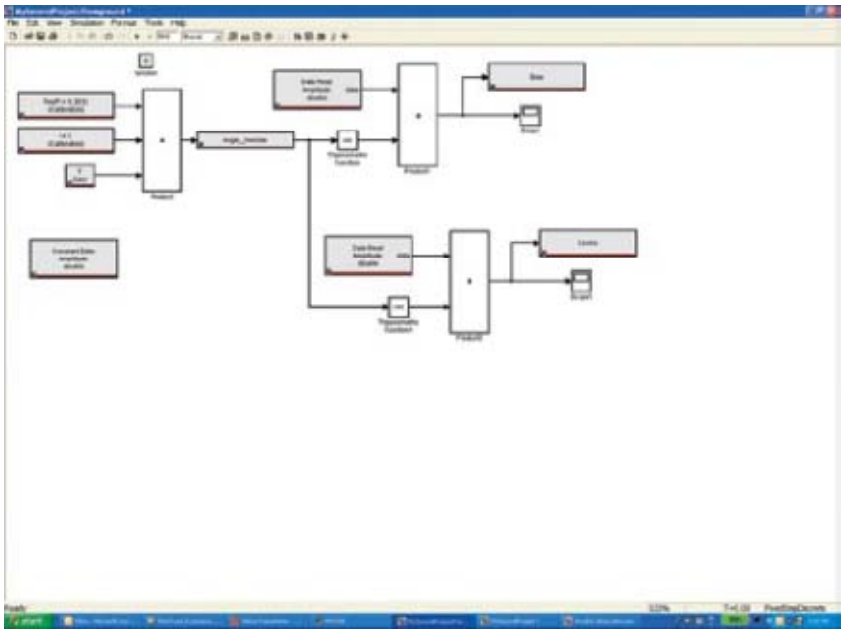
Example: You are adjusting calibration values and you decide to change the logic in your module (ie. change a greater-than to a greater-than-or-equal-to.) You can rebuild the application, reprogram the module, and pick up where you left off, without having to up-load the calibration.

Calibration and Probing Blocks

Another useful block is the motohawk_override_abs block from the Calibration and Probing library.

1. Drag one into your model and place it over the wire connecting the first product block to the trigonometric function blocks.

   Note: Simulink breaks the wire, making the necessary connections.
2. Double-click on the block and give it a meaningful name (ie. “Angle_Override”).
3. Click on Apply and OK.
4. Press CTRL + D and CTRL + B.
5. Program the module and set up your Display and Calibration panes as before.
6. Your model should look similar to this:



7. Drag the two new parameters from the Foreground\Angle_Override folder into the Display spreadsheet.
8. Start a chart for your Sine and Cosine waves.
9. Set Angle_Override_new to 3.14.
10. Click on the value for Angle_Override_ovr.

    A pull-down arrow appears next to the cell.
11. Click on the pulldown and select override.
12. Look at your chart to see the effect of this change after pressing Enter. As expected, the Sine value goes to 0 while the Cosine value goes to -1.

    The override is a display, not a calibration.

    Display or Calibration... What’s the difference?

    Displays allow the engineer or technician to monitor or manipulate signals in the system to establish conditions necessary for testing or calibration.

The changes made via Display variables are not saved in the .dis file and so do not persist past the MotoTune session.

On the other hand, Calibration changes are saved in a .cal file and can be Merged with or Transfer Upgraded into another calibration (or .srz) file to create a new .cal (or .srz) file which contains the desired changes.

Gathering Data

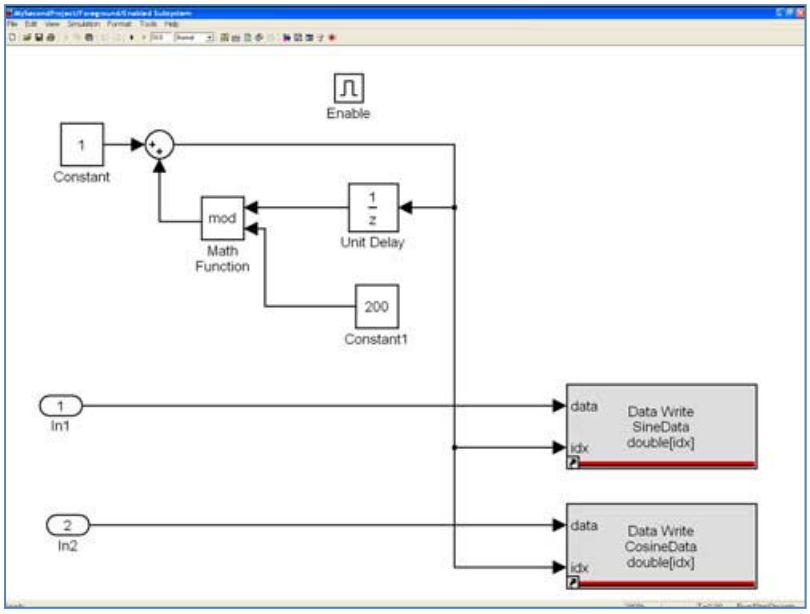
We have seen how a data definition block is used to introduce a constant into the system. Now, look at how it can be used to gather data from our system.

1. From the Ports & Subsystems library, drag in an enabled subsystem and delete the scopes.

   Double-click on the enabled subsystem and a new window opens up.
2. Delete the output port and copy the input port by right-clicking on port 1.
3. From the commonly used blocks library, drag in a constant block and a sum block.
4. From the math operations library, drag in a math function block.
5. From the discrete library drag, in a unit delay block.
6. Right click to copy the constant block. Set the value of the new (constant1) block to 200.
7. Double-click on the math function block and use the pull-down to select mod (modulo) function.
8. Click on Apply and OK.
9. Right click on the mod block and select format and flip block.

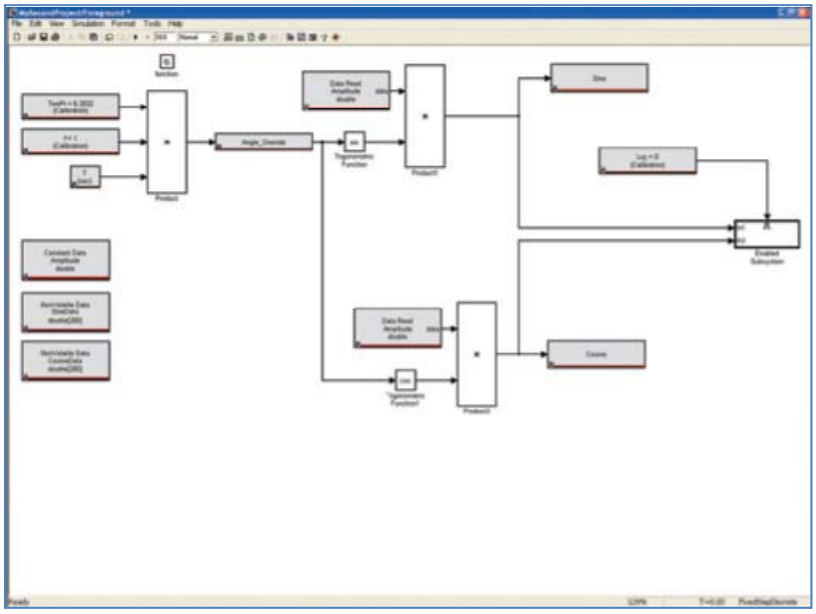
   Likewise flip the unit delay and constant1 blocks.
10. Wire the constant and mod blocks to the sum block inputs.
11. Wire the output of the sum block to the input of the unit delay block and the outputs of the unit delay and constant1 blocks to the inputs of the mod block.
12. From the data storage blocks library, drag in a motohawk_data_write block and make a copy of it.
13. Double-click on the first data write blocks. Name it SineData.
14. Using the pull down, set data structure to vector.
15. Name the second data write block CosineData and make it a vector as well.
16. Wire the idx input of each data write block to the output of the sum block.
17. Wire input1 to the data input of the SineData block and input2 to the CosineData block.

18. Your enabled subsystem should look like this:



19. Save file and close this window.
20. In the Foreground window, right-click on the Amplitude data definition block and make two copies.
21. Double-click on the first copy, change the name to SineData, change the Storage Class to NonVolatile, and change MotoTune Window to Display.
22. Place the following in the Initial Value box: zeros (1,200).
23. Click Apply and OK.
24. Double-click on the second copy, change the name to CosineData, Storage Class to NonVolatile, and MotoTune Window to Calibration.
25. Click Apply and OK.
26. Place the following in the Initial Value box: ones (1,200).
27. Copy the ‘f’ calibration block and rename it.
28. Log and set the initial value to zero.
29. Wire the Sine signal to In1 and the Cosine signal to In2 of the enabled subsystem.
30. Wire the Log block to the input at the top of the enabled subsystem.

31. Your model should look like this:



32. Press CTRL - D. If there are no errors, press CTRL - B.
33. Start MotoTune and create a new display and a new calibration.
34. In the display pane expand MySecondProject and Foreground.
35. Drag SineData into the worksheet.

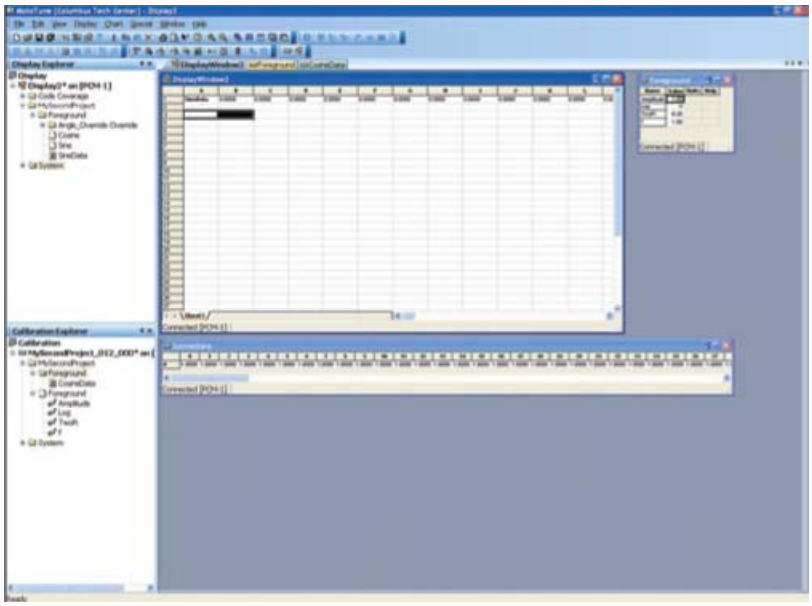
    Note that all of the values have been set to zero.
36. In the calibration pane, expand MySecondProject.

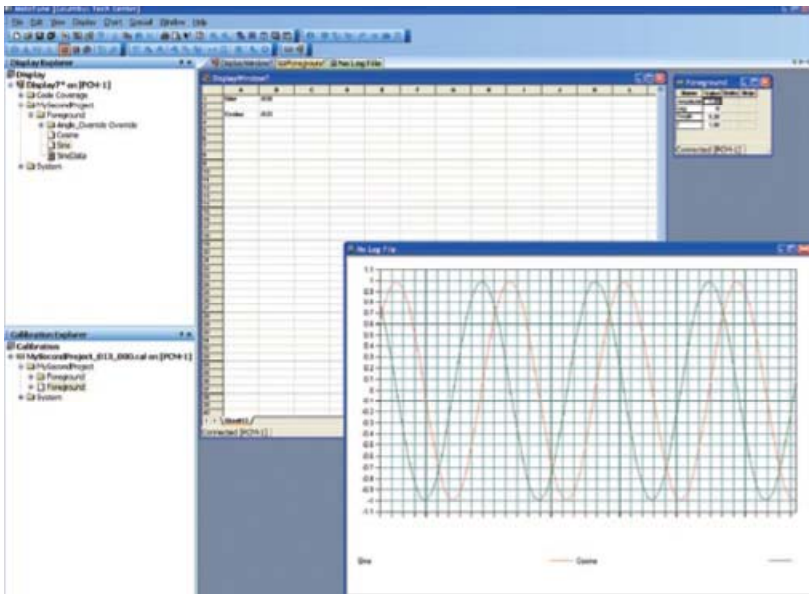
    Note the folder and sheet of paper, both named Foreground.

    Expand each to see their contents.

    The folder contains the CosineData vector array (another sheet of paper). The sheet of paper contains the scalar variables. Both have been defined in the Foreground layer of the model and the default group string was used.
37. Double-click on the Foreground and the CosineData sheets of paper and arrange them in the window.

38. Your window should look like this:





39. Note that the CosineData array contains all 1s.

    Changing the Log variable to 1 enables the subsystem that logs the data.

    The SineData array changes immediately, but the CosineData does not.

40. Select Calibration-Refresh Volatile Map (or press F5) and the CosineData array is updated.

    The Sine Data array may be used to examine the Sine values and can be copied and pasted into a spreadsheet for analysis.

    If there is no need to edit the values offline (factory defaults are a good starting point for an adaptive algorithm,) the Display variable will suffice. If, however, the values are best customized based on which variety of installations it will be used on, then the Calibration variable is the one to use.

Helpful Tips

Here are two ways to help minimize confusion:

1. Utilize the Show MotoTune Group check box and explicitly name the MotoTune Group String.
2. Place the data definition blocks in the enabled subsystem.

The system designer needs to decide the best way to organize these data structures.

A CTRL - B is required to generate a new DLL.

Throttle Control Challenge

The following example uses a slider potentiometer and an electronically controlled throttle assembly: (Woodward P/N: 6945-5001 40MM BOSCH ETC [A 289 000 464-999]).

Table 1 lists the signals and their corresponding connector pin numbers.

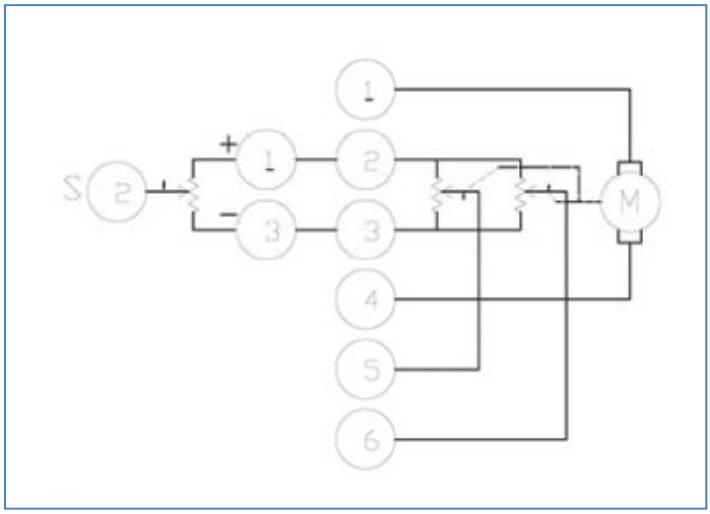


Figure 1. Electronic Throttle/Slider Potentiometer Schematic

Table 1. Electronic Throttle Connector Pinout

PIN NUMBER	SIGNAL NAME
1	Motor-
2	XDRG
3	XDRP
4	Motor+
5	POT2
6	POT1

1. The Slider pot should be connected to XDRP, XDRG, and AN1M. POT1 and POT2 should be connected to AN2M and AN3M respectively.
2. Consult the datasheet for your module to determine the appropriate wire number for each of the signals.
3. At the Simulink command line, use the motohawk_project instruction to open a new project. Name it ThrottleControl.
4. Double-click on the Foreground block and delete the Controller and Plant blocks.
5. From the MotoHawk Analog I/O Blocks library, drag in a motohawk_ain (Analog Input) block.
6. Select “Allow I/O pin to be calibrated from MotoTune,” and name the block ThrottlePedal.
7. Select AN1M from the pull down and click on “Apply” then “OK.”
8. Drag in a Gain block and a motohawk_probe block.
9. Wire the ThrottlePedal block to the Gain block and the Gain block to the motohawk_probe block.
10. Set the Gain block Gain to 100/1023.
11. Name your probe SetPoint.
12. Press CTRL - D.

    In some versions of MATLAB, you may get a warning regarding datatypes.

    In this instance, the A/D on the 555 is 10 bits, which fits into a unit16.

    Other resources have the following data types:

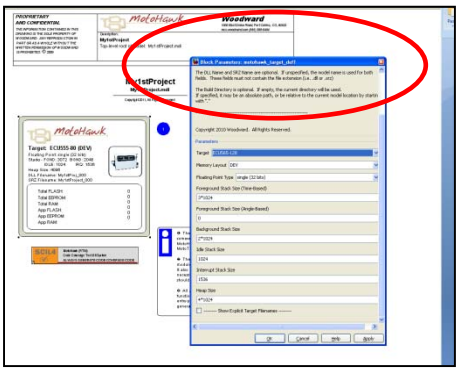
    - Digital Inputs and Outputs are Boolean.
    - Frequency Inputs and Outputs are uint32 (scaled by 0.01Hz).
    - Duty Cycle Inputs and Outputs are int16.

13. Go to the top level of your model, double-click on the Target Definition block and click on the “Floating Point Data Type” pull down.

    The choices are:

    - single (32 bits)
    - double (64 bits)
    - disabled

    These determine the way that memory will be allocated during code generation. The default is single (32 bits) and should not be changed unless greater resolution is required or the target processor does not support floating point operations.



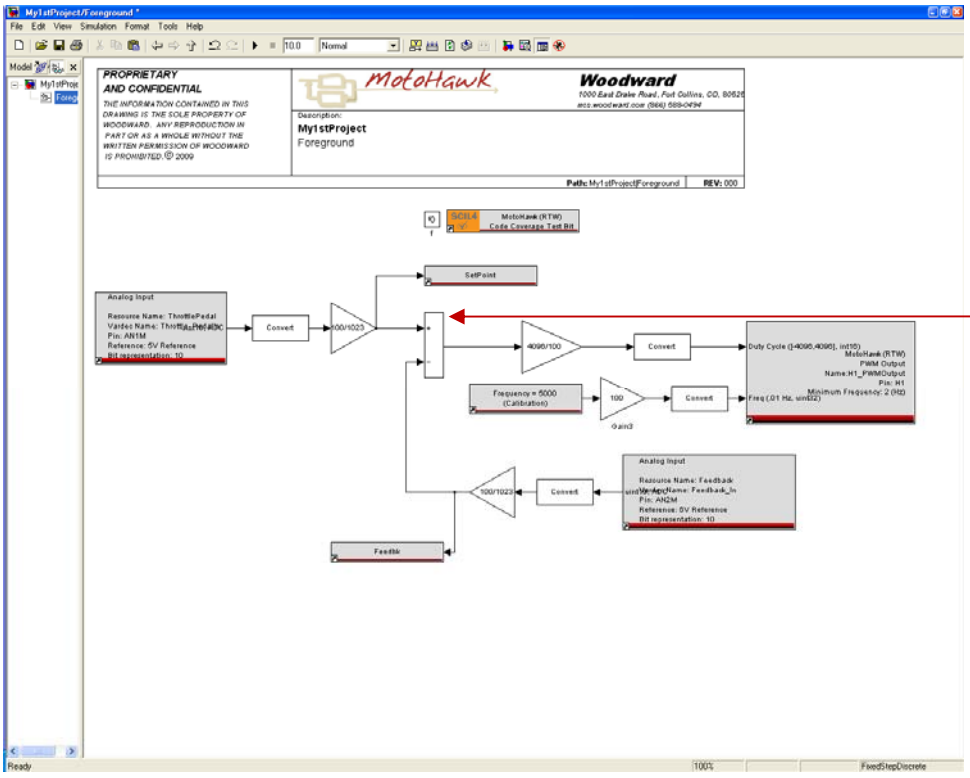
14. Return to the Foreground level of your model and drag a Data Type Conversion block in from the Signal Attributes library. Place it between the ThrottlePedal block and the Gain block.
15. Press CTRL - D again.
16. There should be no errors reported.
17. From the Format menu select Port/Signal Displays and check Port Data Types.
18. The data type appears adjacent to each wire. This is a convenient way to verify that your data types are consistent in your model.
19. Make copies of the analog input, data type conversion, gain, and probe blocks.
20. Highlight them and select Format-Flip Block (or CTRL - I).
21. Select AN2M for the analog input, name the probe Feedback.
22. Drag in a motohawk_pwm block from the Analog I/O Blocks library and select H1 as the resource.
23. Drag in a motohawk_calibration block. Name it ETC_ Frequency and set the Default Value to 5000.

Proportional Control Example

1. To make a proportional control like the one shown below:

   Add a summing block. Copy and modify the gain block and data conversion blocks.

   When you first wire in your blocks, the data type adjacent to each wire will indicate double (MATLAB's default), but when you press CTRL - D they are updated to indicate the appropriate data type.



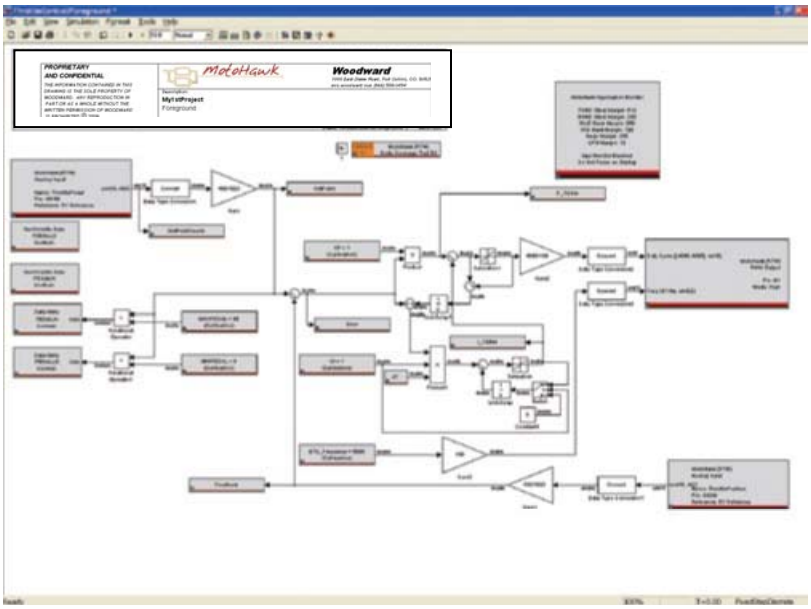
2. Press CTRL - B to build your model and use MotoTune to download it to the module.
3. Operate the Throttle Pedal slider and observe the behavior.

   This model is a simple proportional control. Realistically, a more complex control is required.

This value represents the difference, or error, between the Throttle Pedal Value and the actual Throttle Pedal Position.

Fault Detection on Throttle Pedal

The next model introduces rudimentary fault detection on the Throttle Pedal Position sensor and adds an integrating term to the command signal. It also includes diagnostic probes and calibration for Proportional and Integral gains.



1. Modify your drawing to look like the one shown above.
2. Press CTRL - D to check your model. Then build it using CTRL-B.
3. Open a display and a calibration in MotoTune. Set up your probes and adjust the ETC_Frequency value until the high pitched sound can no longer be heard.
4. Set the Integral Gain to zero and increase the proportional gain until the throttle plate exhibits ringing when operated.
5. Open a chart and increase the Integral Gain until the traces for SetPoint and Feedback come together.

   The Error trace should be zero.

Chapter 2. Faults

Introduction

This chapter covers the basics of faults within MotoHawk.

Faults are used to indicate failures within a system. For instance, if a sensor becomes disconnected, the application can detect this out of range condition and signal the issue via a fault.

Fault diagnosis usually accounts for 50-70% of the code within any production application. In other words, when you have the control logic done but not the fault detection, you are only about 1/3 to 1/2 done with your application.

MotoHawk provides a nice set of blocks to help you signal faults and take actions as a result of faults.

Faults are nothing more than signals that some logic has found an issue within the system.

Fault diagnosis and identification is a complex subject that changes based on the application. However, you will find that all good applications at least diagnose sensor failure, and should diagnose actuator failures if possible. Why? Because wiring harnesses fail, sensors fail, and actuators fail.

Ideally, your application will do these things well:

- *Fault Containment.* The act of keeping a fault from propagating to other parts of the system.
- *Fault Identification.* The act of determining, as precisely as possible, the source of the fault.
- *Fault Annunciation*. The act of reporting the fault to someone who can fix it.
- Fault Action.The act of adjusting system operation in response to the fault.

Some faults are easy to detect — like a signal being out of range. Others can be terribly difficult — like a signal stuck in range. Unfortunately, MotoHawk does not help you with containment or identification problems. That is the job of the application designer. MotoHawk will however, allow you to record the faults, help annunciate them and help interface to action code.

MotoHawk Fault Theory of Operation

MotoHawk contains a series of blocks that allow you to signal a fault, read the fault status, change the fault status, and take fault actions. The easiest way to think about this — you have fault signals and fault actions.

- *Fault Signals* are an indication that a fault has occurred.
- *Fault Actions* are what the application should do when various faults occur.

Routing Multiple Faults to a Single Fault Action

MotoHawk allows you to route multiple faults to a single fault action. This is a powerful idiom that will simplify the designer's job. Because fault actions are independent of faults, there is no need to define various levels of seriousness to the faults. The seriousness is contained within the application.

For instance, an engine designer may design a fault that detects low oil pressure and an action that is capable of shutting down the engine. The designer can then decide if low oil pressure is worthy of shutting down the engine.

Often times, this decision cannot be made at design time. You may be building an engine that can be installed in a trash truck and a fire truck. Shutting down a trash truck because of low oil pressure is probably very desirable so that the engine can be repaired. However, most fire departments would just as soon pump water onto the fire until the engine is reduced to a pile of molten metal rather than shut the engine down.

MotoHawk respects this and allows you to calibrate faults to fault actions, rather than requiring the routing be set at design time. This allows a single code build to handle both of the example cases with just a change in the calibration.

Fault Filtering

Faults also need to have filtering. MotoHawk faults provide an X out of Y test which basically says that the fault must be present X times out of Y samples to be declared active.

Faults are considered "Suspected" whenever any of the Y number of samples have detected the fault but the number is less than X. Faults are "Active" when at least X out of Y have occurred.

In addition to filtering, MotoHawk faults have some different behaviors. A Fault can be:

- *Disabled*. It will not signal a fault even if the X out of Y condition is satisfied.
- *Sticky.* Once set it will remain set until the next power down or until it is explicitly cleared. This setting is handy for detecting transient or intermittent faults that may appear and disappear before they can be observed in MotoTune.
- *Persistent.* A fault that acts like the "Sticky" fault, in that it will remain set once the fault conditions occur. But it will remain set across a power cycle. A persistent fault once set will remain set until it is explicitly cleared.

Fault Actions can be initiated by one or more faults. Any given fault can drive up to four fault actions based on various states of the fault (i.e. Suspected or Active). The fault action block will report a high Boolean signal when any of the associated faults are set. The application designer is then responsible to define the proper system response.

Fault Blocks

MotoHawk provides several blocks to define and interact with faults within your system. These are located in the MotoHawk Library under Fault Management.

Fault Manager

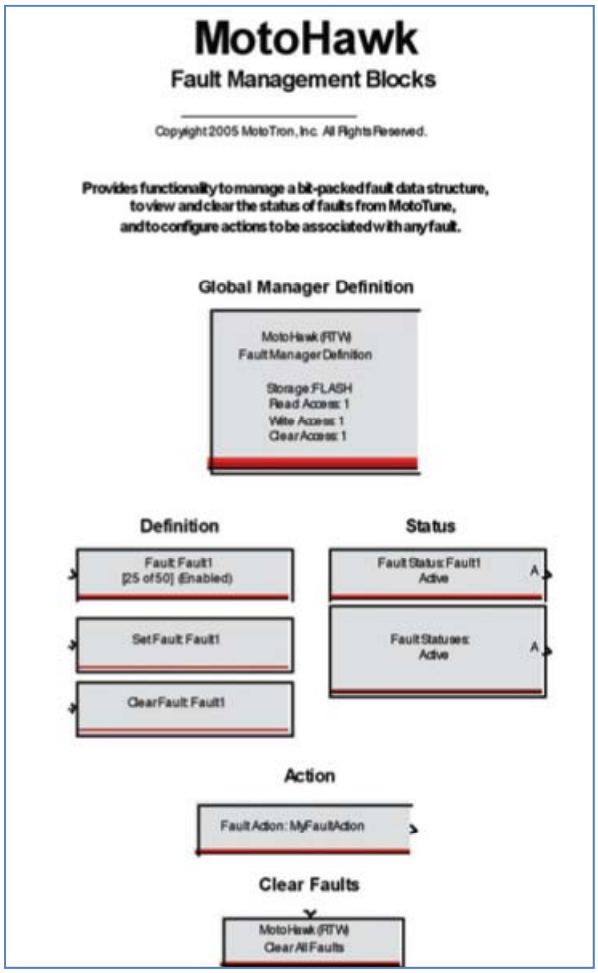
This block can exist anywhere in your model. You will need only one for the model. The storage for the fault manager allows you to control where the fault calibration is stored.

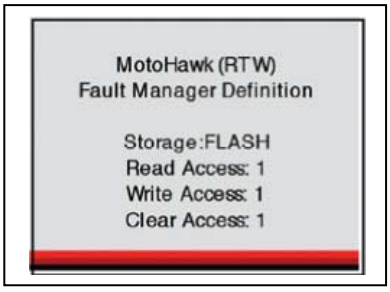
If set to FLASH — the faults can only be calibrated on a development module or offline.

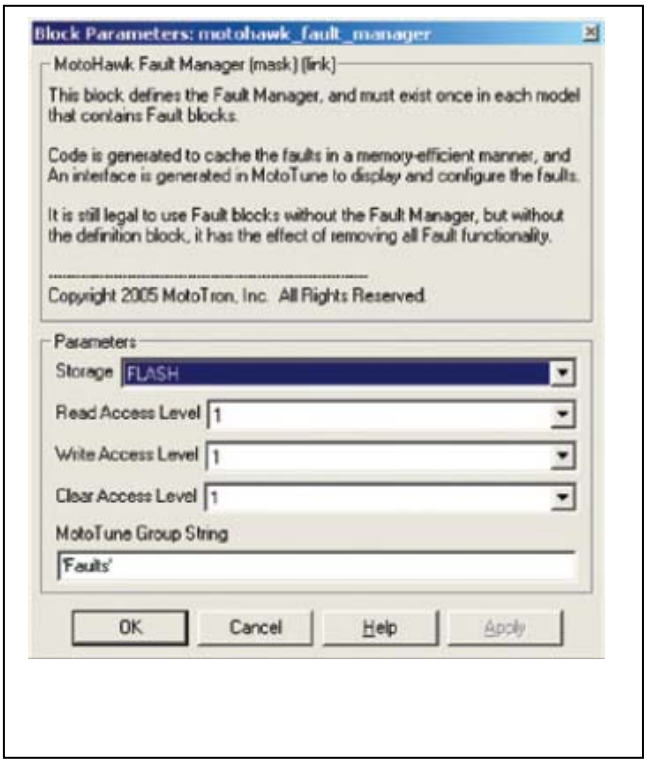
If set to EEPROM — the calibration can be adjusted on any module.

The access level refers to the security level required of the MotoTune user to perform the action.

The MotoTune group string controls where the Fault calibration will be shown in the MotoTune Calibration Tree.

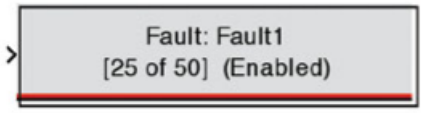


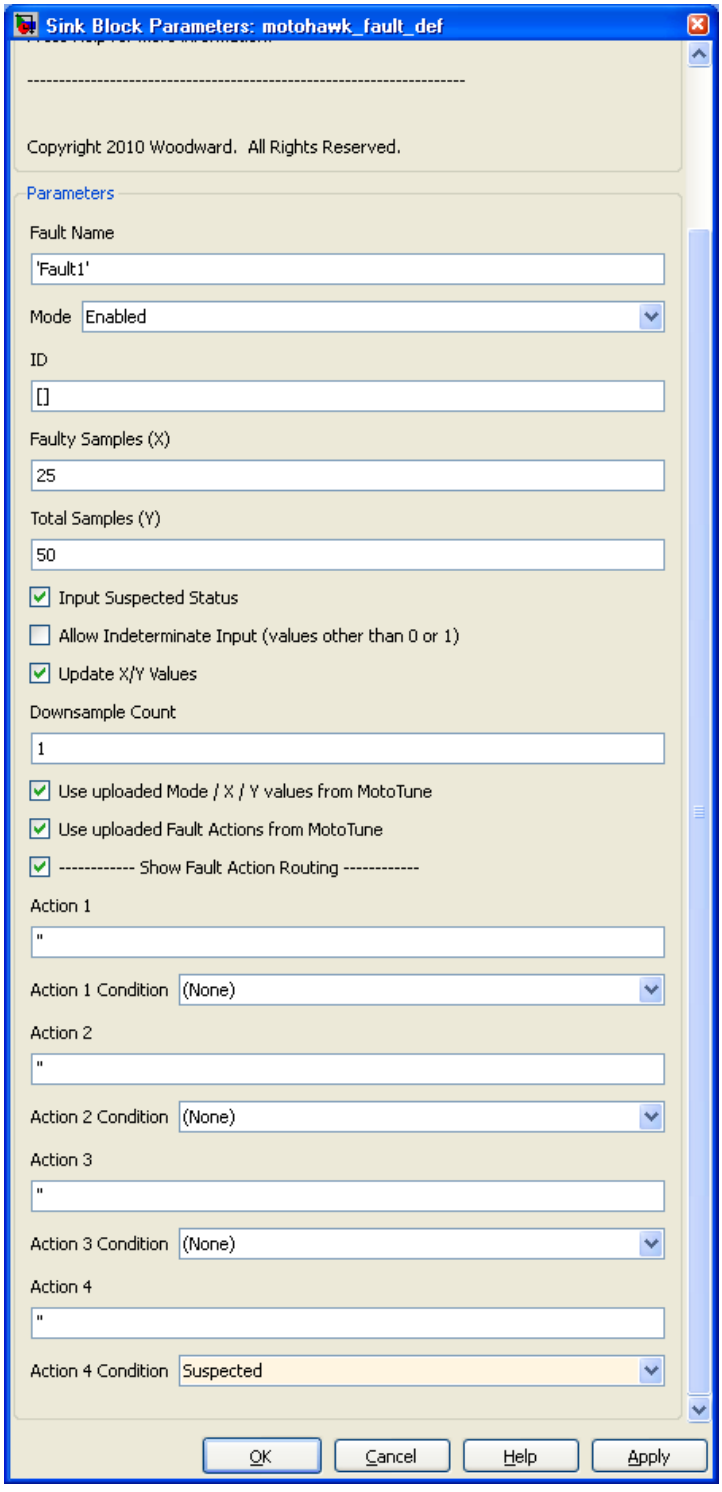




Fault Definition

This block defines a fault in your system. Faults must have unique names throughout the system.

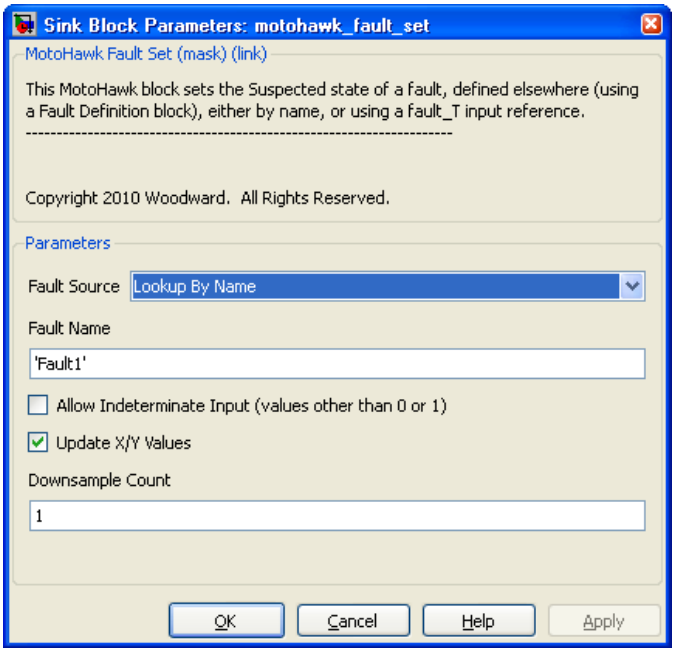


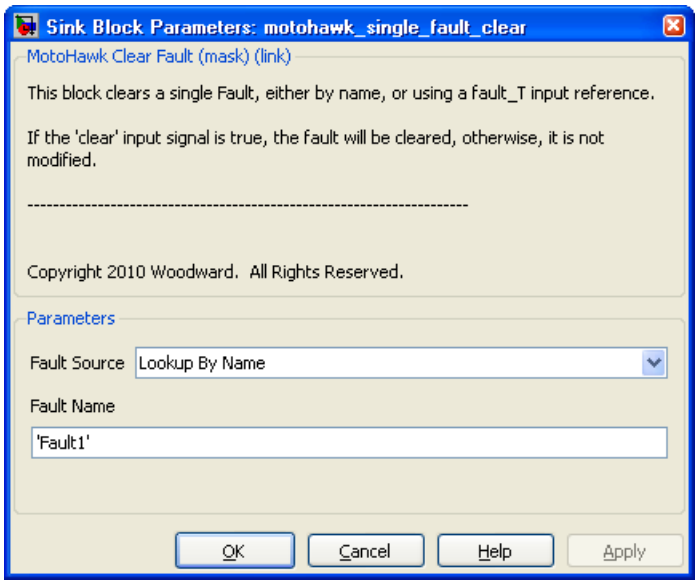


Set Fault & Clear Fault

These blocks will set or clear a fault that has been defined elsewhere.

The application is responsible for coordinating when these blocks run— there is no coordination done by the Fault Manager.

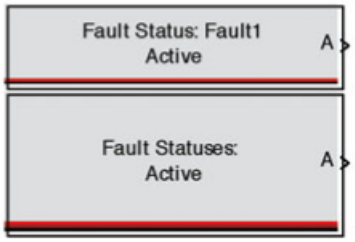


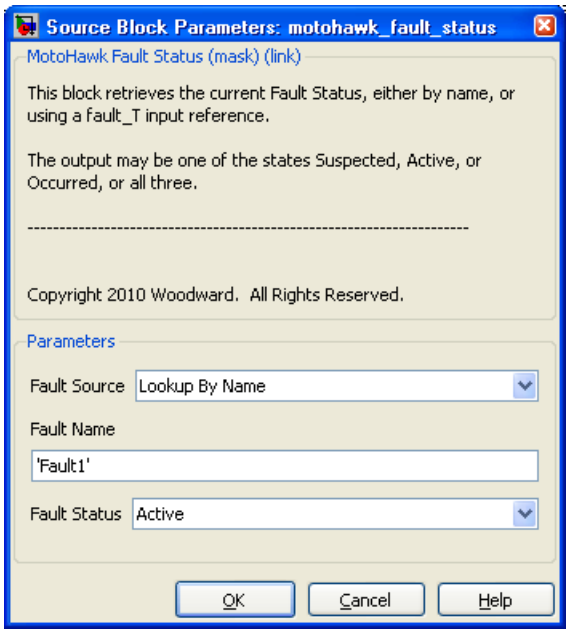


Fault Status

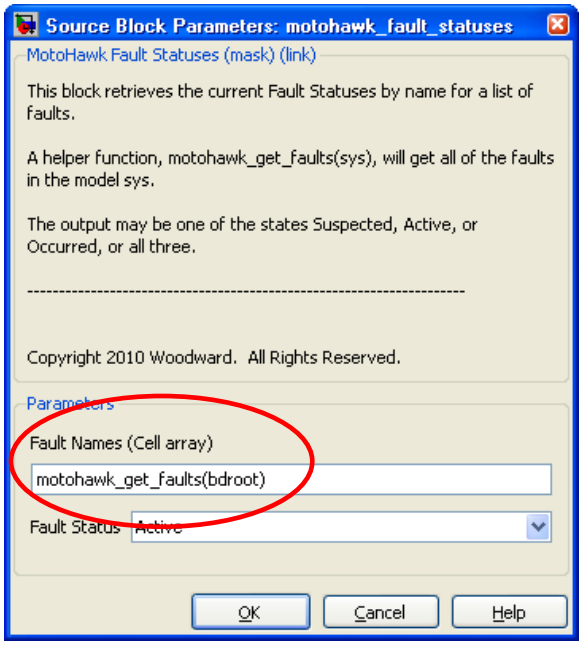
These blocks allow you to read the status of a single fault or a group of faults.

When reading multiple statuses, the output will be a vector of Boolean values corresponding to the fault list.



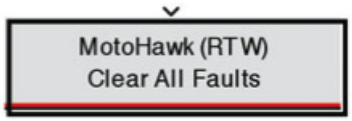


Motohawk_get_faults(system)
This is a utility function that will retrieve all of the faults located in the system and its children. Use bdroot to find all faults within the model. The fault list returned by this function will be alphabetized.



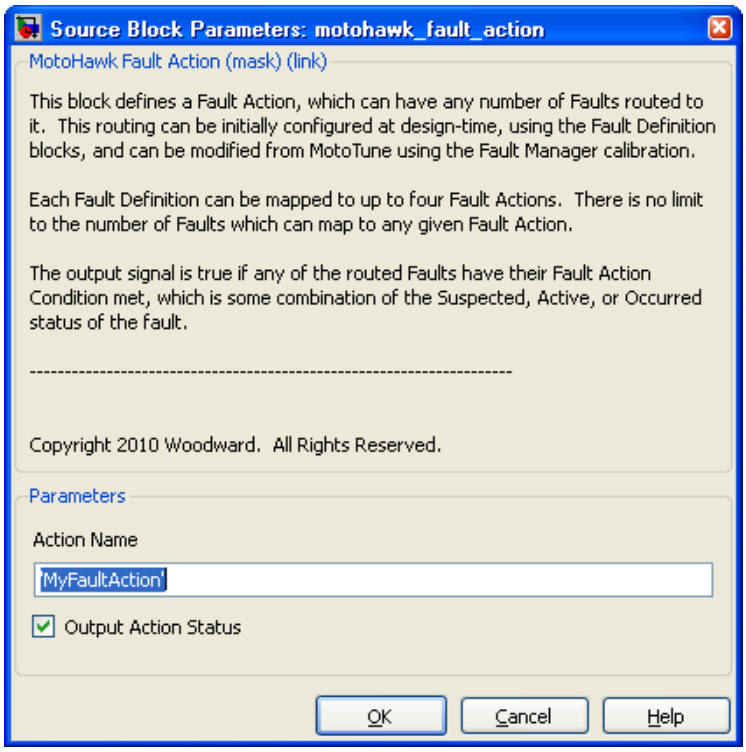
Clear All Faults

This block, when triggered will clear all of the faults. If the fault conditions still exist, once the X of Y filters are satisfied, the faults will re-activate.

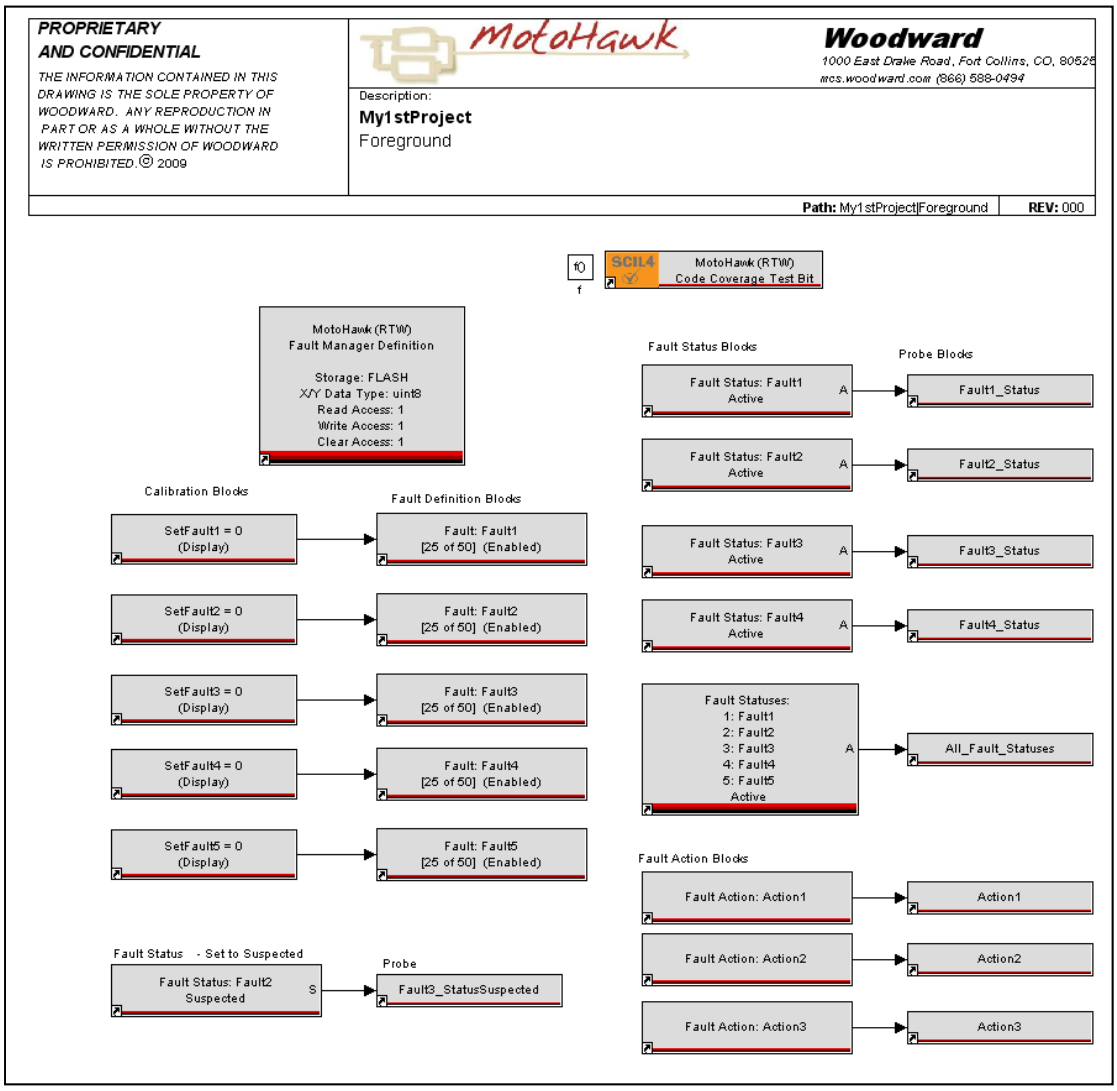


Fault Action

This block defines a fault action. The fault action name must be unique within a model. The action will become active when a fault is routed to it either via the design or via calibration. The application designer then needs to create the code that will execute when the fault action is active.



Fault Blocks Example



Fault Block Exercise Steps

1. Start with motohawk_project('Fault1.')
2. Remove the existing contents of the Foreground subsystem
3. Create the model as shown.
4. Build the model.
5. Run MotoTune, program the module, and open a display or open the FaultExample.dis file.
6. Notice in MotoTune display explorer, there is a category for Faults that contains the display variables for:
    - Active Faults
    - Occurred Faults
    - Suspected Faults
    - Fault Command (to clear faults

    Also, for every Action there is a reason display variable that will tell you all of the faults that are causing the particular action.

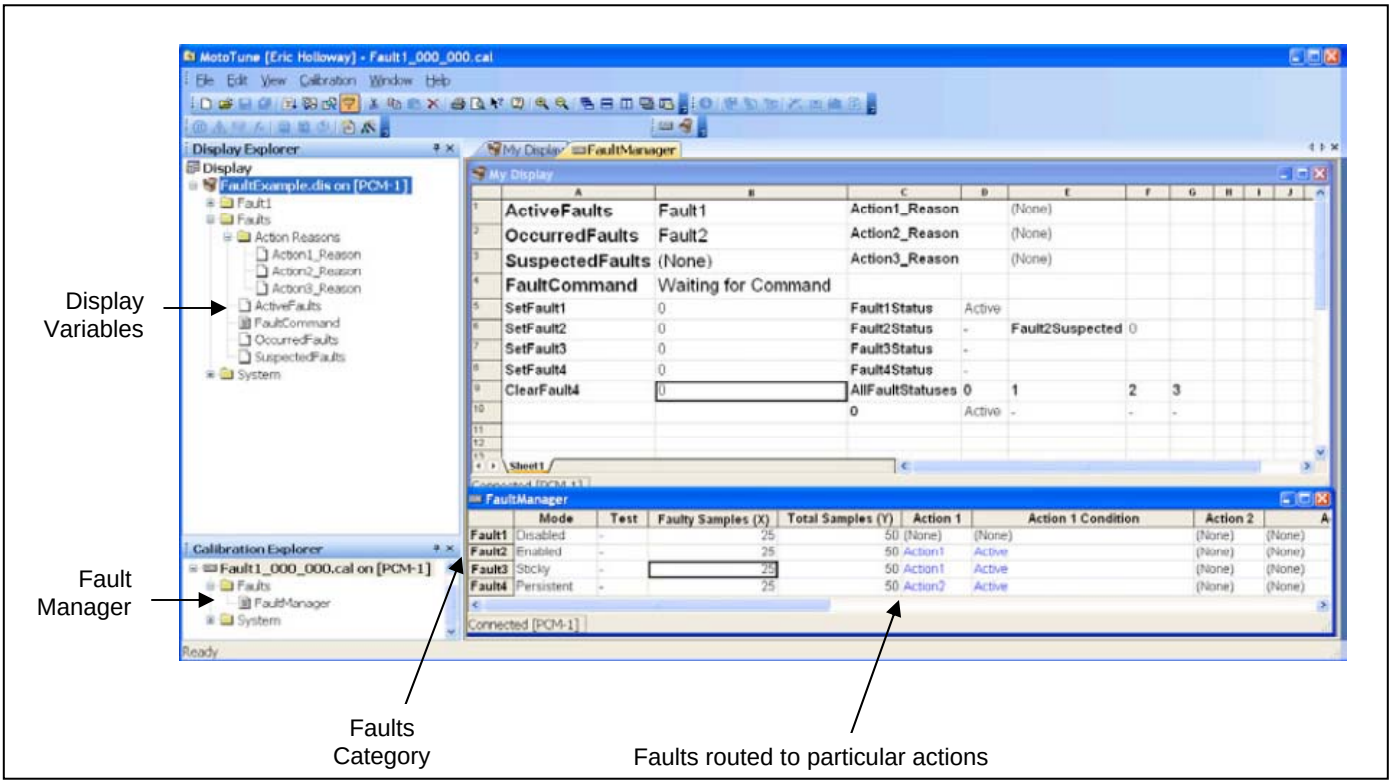
All of the displays are marquee type displays that will roll through the faults and display the fault names.

7. Open a calibration and notice the Faults category in the Calibration Explorer. The Fault Manager is located here.
8. Open it up.

   The fault manager contains fields that can be set in the Simulink Fault Definition Block. They are found here and can be adjusted at run time.

   There is also an extra field, “Test,” that will allow you to force the fault active without the input conditions being set.

   Note how the calibration has been adjusted to route some of the faults to particular actions.



Chapter 3.
CAN

Introduction

The CAN standard was developed to facilitate the communication of data between devices in a vehicle. CAN literally means Controller Area Network.

The current standard for CAN is CAN 2.0B with which all Woodward MotoHawk controllers are compatible. Most of our modules have at least one CAN port, while a few have as many as three.

Sending and receiving messages via a CAN port is incredibly simple. It is far easier to send or receive a message via CAN than it is a RS232. However, there are a couple of issues that can make it seem daunting — especially when talking about CAN protocols like J1939 or SmartCraft.

CAN Bus Basics

First, a CAN bus requires at least two participants in order to be a bus. The physical connection between devices is a 2 wire cable. The wires are often labeled CAN-H and CAN-L. There must be a 120 ohm resistor between CAN-H and CAN-L somewhere on the bus called a terminator. The terminator resistor can be placed physically anywhere in the bus, but ideally is located at one end or the other. You can have more than one terminator, but remember that too many cause the bus to stop working.

CAN bits are transmitted across the bus as either dominant or recessive. This means that a dominant bit (a 0) will win over a recessive bit (a 1). All of the transceivers on the bus must be operating at the same bit rate (aka the baud rate.) All of the transceivers on the bus synchronize to one another by detecting the edges between 1s and 0s. Luckily, the transceivers do much of the hard work of transmitting and receiving messages. The software needs only load messages to be sent and react to incoming messages. The transceivers make sure that a message gets out on the bus if possible. Commonly, busses are operated at 250K baud but can run as fast as 1M baud. The length of the bus is directly related to how fast you can run the bus. For reliable communications, the maximum range at 250K baud is 100 feet; at 1M baud it is 30 feet.

All CAN messages are comprised of:

- An ID of either 11 bits (aka a standard ID) or 29 bits (aka an extended ID),
- A Data Length Field saying how many payload bytes there are. This number can be from 0 to 8, and a payload of 0 to 8 bytes.

Notice that the payload can change sizes. Yes, a perfectly valid message can contain no payload at all. You might ask, why you would ever transmit a message with no data? Usually to indicate that a module is alive by sending a heartbeat to other modules in the system or to represent the occurrence of an event.

Notice as well that IDs can be of two different types. Is it permissible to have both types on the bus at the same time? Absolutely. The bus will perform just fine with both types of IDs and variable length payloads running across it. Also, messages with IDs of the same value but different type are considered totally different messages.

So, how are the inevitable bus collisions (times when two modules want to transmit at the same time) handled in CAN? Very nicely.

Remember that all transceivers are synchronized. The two transmitting modules will start clocking out their ID bits at the same time starting with the most significant bit. As soon as the ID bits differ, the device that is transmitting the 0 wins the bus (because 0s are dominant) and continues clocking its bits out. The device with the 1 in the ID bit, automatically detects that it lost the bus and stops trying to transmit, and it will automatically wait until the next transmission slot to try again.

So, this brings us to a couple of rules:

1. Lower ID values have higher priority on the bus
   (and standard IDs are higher priority than extended IDs)
2. No two devices can transmit the same ID.

The first rule is fairly obvious. 0s are dominant, so lower IDs will make it on the bus first. The fact that standard IDs are higher priority than extended is caused by the transmitting of a 1-in-1 of the early messaged header bits to indicate that the following ID is extended.

The second rule is not as obvious, but will bite you. If two modules tried to send the same ID at the same time, neither would know that it did not win the bus. The failure would not occur until they had a different bit in the payload. Unfortunately, each module will only be informed that its message failed a parity test (due to the payload bits being clobbered). Each module will then dutifully retry to transmit. Since they are synchronized, they will once again clobber each other. So, never, ever have two modules potentially sending the same ID. Of course, never is a strong word. And, you will see that some protocols actually will break this rule to do address claiming — but more on that later.

Payloads

Recall that payloads can have between 0 and 8 bytes of data.

Those 8 bytes can mean anything you want them to mean. The CAN 2.0B specification does not have an opinion about the contents of the payload. Of course, choosing IDs and defining payload contents can be a daunting task. If you own the entire bus design, you can simply choose IDs and data packing. However, if you need to coordinate bus usage, then a protocol needs to be chosen so that IDs are unique and multiple developers can interface to one another. Luckily for you, there are plenty of protocols to choose from like J1939, GMLan, SmartCraft, CANopen, etc. You can also run multiple protocols at the same time across the bus — just make sure the IDs do not clash and there is sufficient bandwidth and you are good to go.

How much data can a CAN bus transfer?

The maximum performance is about:

- 2000 messages per second at 250K Baud
  (or 16000 bytes per second of payload.)
- 4000 messages per second at 500K Baud.
- 8000 messages per second at 1000K Baud.

Good network design requires that you plan for no greater than 70% bus utilization or about 1400 messages per second at 250K. Protocols will often require you to pace messages at a minimum interval between messages so that the instantaneous message rate adheres to these limits. For instance, J1939 paces messages at 50 milliseconds for large data transfers. In other words, they are limiting a block transfer to about 1% (1/0.05/2000) of the available bandwidth.

Protocols

Protocols are where CAN gets thorny.

Because CAN has a limited number of ID bits and only 8 bytes of payload, defining ways to transport all types of data can be difficult. Often times we hear questions like, “Do you support CAN?” The answer is, of course, yes. What they are probably asking is, “Do you support [something like] J1939 running across CAN?” The answer is maybe.

We usually consider protocols to be application specific. That is, the application is responsible for implementing the protocol. MotoHawk, Control Core, and the module hardware provide all the necessary infrastructure to implement protocols, but it is rare for protocols to be implemented in these layers. The exception to this is the reprogramming protocol for the module via CAN. The boot loader needs to communicate with MotoTune to reprogram a module. Since the application is not running during reprogramming, the boot loader then becomes responsible for the reprogramming protocol.

Protocol Specifications

Most protocol specifications will define Message Definitions which include:

- ID (including whether it is extended or standard)
- Description of any of the meaning of any ID bits
- Description of any ID bits that are “don’t care,” commonly called the mask
- Frequency of the message, or the event that will cause it to transmit
- Device responsible for transmitting the message
- Expected number of bytes in the payload
- Contents of the payload
- Size of each content item in bits
- Location of each content item in the payload
- Data type of each of the content items
- Byte packing order of each of the content items
- Translation of each content item into “real world” units
- If the protocol has states, then a list of all states and transitions

Unfortunately, 95% of all protocol specifications are incomplete because they assume certain facts (like byte order) without specifying them. The missing information is often the reason that you cannot connect your application to an existing CAN network without problems.

Examples of Protocols

- *J1939*. Recommended practice for a serial control and communications vehicle network.

  This is the network found on many heavy duty trucks. Communication is defined for a very large number of devices like engines, transmissions, dashes, anti-lock brakes, etc.
- *NMEA2000*. This is the protocol published for marine vessels. The protocol is similar to J1939.
- *SmartCraft*. This is the drive-by-wire protocol on Mercury Marine powered vessels.
- *GMLan.* This is the protocol running in your favorite Chevy.
- *CCP.* This is the CAN calibration protocol used by many controllers for calibration and service tool interaction.

MotoHawk CAN Theory of Operation

MotoHawk provides several blocks to make interfacing to any CAN bus and protocol relatively easy.

Transmitting Messages

When transmitting, all messages are transmitted via a single hardware buffer (usually buffer 0) from a software queue. As the application executes, each message that is to be transmitted is loaded into the software queue. The OS then monitors the buffer and transmits messages from the queue as quickly as possible. (Remember at 250K baud, it takes about 500 ns per message to transmit if the bus is not otherwise busy.)

Two different forms of transmit blocks are available.
One will transmit a raw message — meaning a message with the ID and payload computed by another part of the application. The other block will form the message from individual signals being fed to the block and a message specification. The latter block is generally used for broadcast, fixed content messages. The former is generally used to handle protocols in which the payload changes based on the state of the protocol.

Receiving Messages

The receiving of messages is conceptually simple, but terribly complex because the CAN hardware does not provide much assistance. MotoHawk has abstracted much of the complexity away by automatically generating a sophisticated software message dispatcher. As you create message receive blocks, each block will require a message ID and a message ID Mask that describe the message ID that you want to receive. The ID mask is simply a description of which bits of the ID must match in order for the message to be accepted.

For instance, if the Message ID is set to 0x7ff and the ID Mask is set to 0x7f0, then all messages from 0x7f0, 0x7f1, through 0x7ff will be received by this block.

At code generation time, the entire model is surveyed for all of the various IDs and masks and a software dispatcher is generated to handle this combination. The dispatcher will adjust the hardware to filter as many messages as possible from the bus and then filter the rest in software so that only the desired messages are passed up to the application.

Each CAN receive block can optionally provide a slot name that allows other blocks to access and adjust the defined slot.

Receive Block Slots are Like Post Offices

The way to think about this mechanism is like a post office. Your slot is where you expect to get mail (or messages) destined for you.

The mail sorter (or the software dispatcher) grabs all of the mail and sorts it into various slots. Sometimes you may want to adjust the rules for your slot; maybe you are going on vacation, so that the mailman changes what shows up in your slot.

For MotoHawk CAN receive blocks, you can create a slot by name that can be adjusted elsewhere in your application. In the previous example, we decided at design time that we needed to receive all messages between 0x7f0 and 0x7ff. But perhaps at run time some logic decides that you really only need to receive 0x7F1, because the module now knows what engine it is installed on.

There is a **slot properties block** that allows you to adjust the slot to tighten the ID mask — so only message 0x7F1 shows up at the receive block.

In other words, the mailman will deliver all of the mail that you requested when the code was built. But you have the ability to ask him to throw away some of the messages prior to placing them in your slot.

There is also a slot trigger block that can be used to notify that a slot has received a message via a function call trigger. In other words, the mailman will ring your doorbell when he puts mail in your slot.

Just to make matters more interesting, you may want to censor some of your mail so that only messages with certain contents are placed in your slot. Each of the CAN receive blocks has the ability to filter based on the payload contents via a **payload value** and a **payload mask set of values**. Like the ID, the payload mask simply indicates which bits of the received payload must match the given payload value.

For instance, say that you want to receive messages 0x7f1 whenever the first byte of its payload is exactly 0x8f and when the last bit of the payload is set. The payload value would be set to [0x8f 0x00 0x00 0x00 0x00 0x00 0x00 0x01] and the payload mask would be set to [0xff 0x00 0x00 0x00 0x000x00 0x00 0x01.] In other words, the first byte must match all 8 bits and the last bit must be set in order for this message to be put into this particular slot.

So now, the mailman reads our mail for us and obeys our content requirements before shoving the mail into the slot. As with IDs, the payload requirements can be adjusted at run time via the slot properties block.

Like the transmit blocks, there are two flavors of receive blocks, one for raw messages and one that will unpack the payload and the ID into their respective data fields, providing them as signals to the rest of the application.

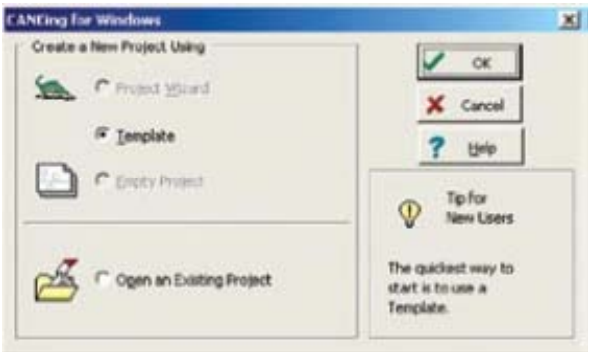
Using CANKing to Observe the Bus

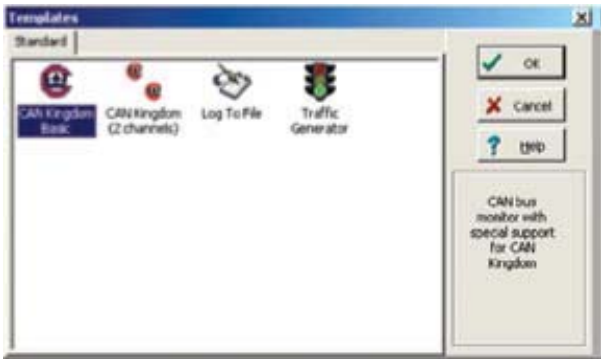
Your MotoHawk kit included an interface for your PC that allows the PC to communicate to two CAN ports via the USB interface. These devices are made by Kvaser (www.kvaser.com.)

Kvaser publishes a free CAN tool called CANKing that will allow you to observe the bus and even send messages. Download this tool from the MotoHawk web site (www.motohawk.info.) You will need this tool routinely.

Initially running CANKing, you will get dire warnings about safety the first time you run the program. Acknowledge their warning and check the box to prevent the warning in the future.

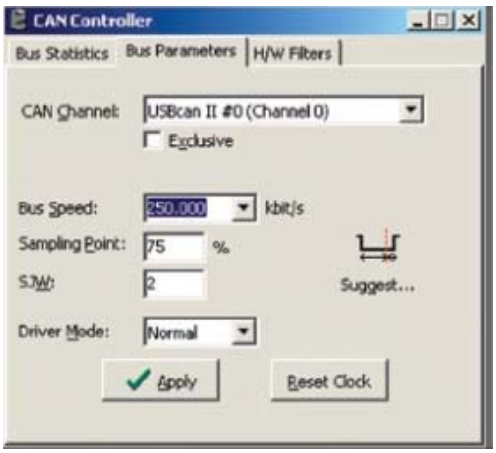
CANKing will launch with the following window.





1. Choose “Template” to start a new project.
2. Choose “CAN Kingdom Basic” from the templates dialog.

   You will then have several windows scattered about your desktop.First, look at the “CAN Controller” window.
3. Choose the “Bus Parameters” tab. Choose the channel that you want (Channel 0 is the typical choice for the MotoHawk kits.) Set the Bus speed to 250 Kbits/s.



IMPORTANT

Uncheck the Exclusive box or MotoTune will not be able to communicate to the module while CANKing is running. Unfortunately, this setting is not saved in the CANKing project file so you will need to browse to this window and uncheck the Exclusive box each time you run the program — even if you reopen a saved project rather than start again from a Template.

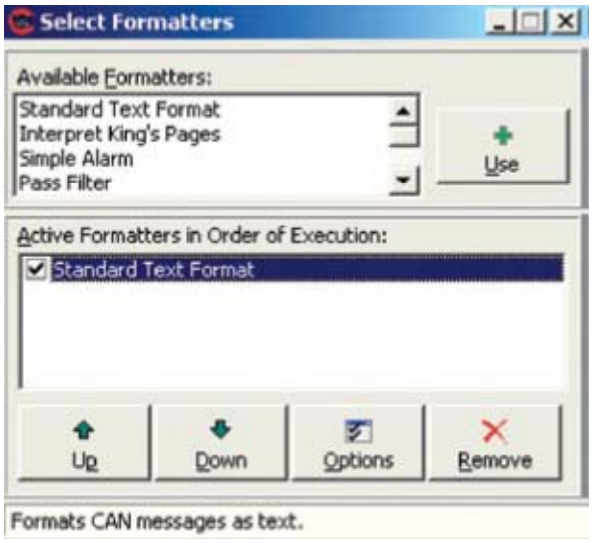
4. Switch to the “Bus Statistics” Tab and press the “Go On Bus”button.

If there is traffic on the bus, the “Bus Load” bar will give you an idea of how much bandwidth is being consumed. If the “Error Passive” indicator illuminates, there are 3 possible reasons:

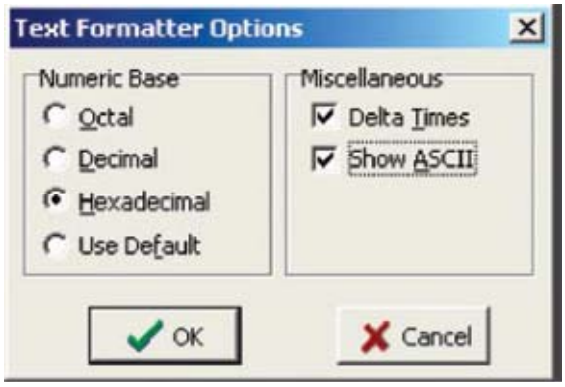
- No bus terminator
- Incorrect BAUD rate
- No other modules on the bus (because the modules are not operating or there is a wiring problem).

Just because there is a green light for “On Bus” does not mean that the bus is actually connected properly. An “Error Passive” will not occur until a message is sent from CANKing which cannot reach a receiver, or a bad message is received. If nothing is received and nothing is sent, then CANKing stays in the “On Bus” state, which can be confusing.

5. Open the Messages menu and select the Universal page to get a window that will allow you to test transmission of messages.
6. Transmit anything and you will either see the state “Error Passive” or the message will appear in the “Output Window.”
7. To display the messages in a useful form, find the “Select Formatters” window, select the “Standard Text Format” in the “Active Formatters in Order of Execution” list, and press the “Options” button.

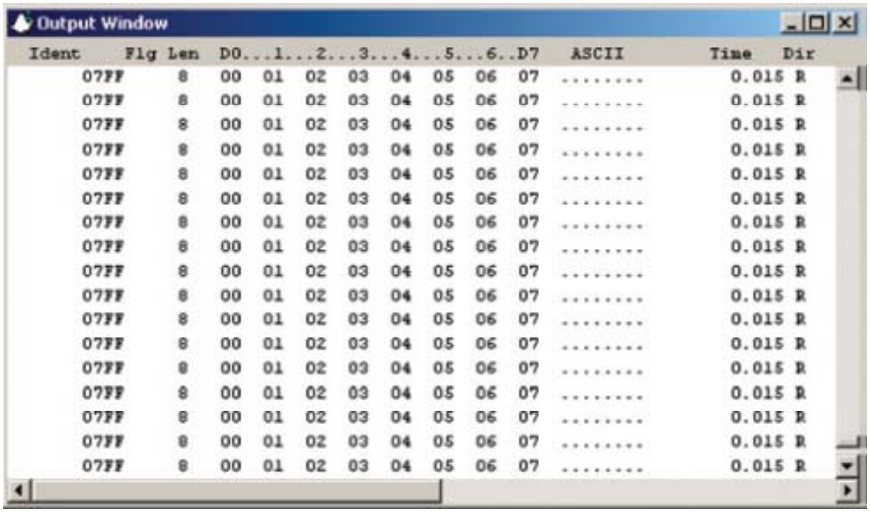


The window “Text Formatter Options” will appear.



8. Choose the setting shown.

   These settings will cause the data to be displayed as shown.

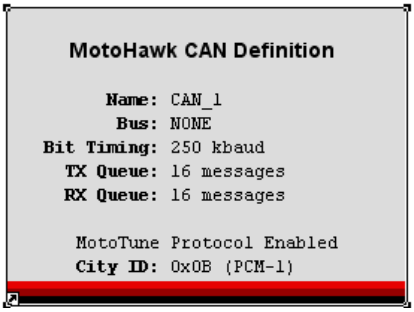


A handy option in the “Output Window” is available via the right-click mouse button. This will fix the positions of the messages into lines of the display rather than showing the bus trace.

Basic CAN Blocks

MotoHawk provides a number of different CAN blocks that you will need to use for different circumstances. **The most important block is the CAN definition block** that will set up a channel's BAUD rate, configure the transmit queue size, and allow the installation of the MotoTune protocol. This block must exist in order for any CAN transmission or reception to take place.

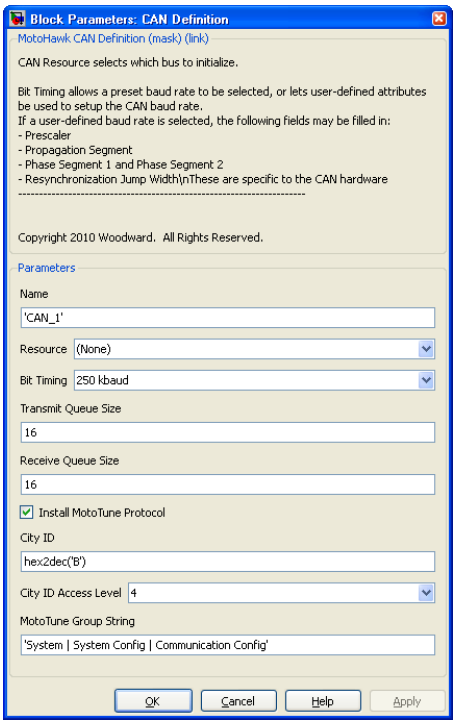
The next two basic blocks are the "CAN Send Raw" and the "CAN Receive Raw" blocks. These blocks simply transmit or receive messages without any payload manipulation.



CAN Channel Definition

This block can exist anywhere in your model. You will need one for each CAN channel.

- *Bit Timing* sets the bus speed or baud rate.
- *Transmit Queue Size* defines the size of the transmit queue.



NOTE: If you set your Resource to None, you will not be able to communicate with your module and will need to use a boot key or boot harness to recover the module.

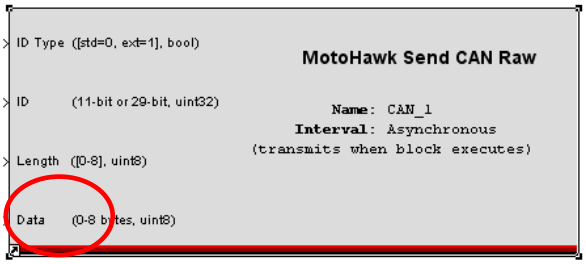
If you are changing the Target in the Target Definition Block, you may need to redefine the CAN resource.

MotoTune can be automatically installed along with defining the City ID and calibration details for the City ID.

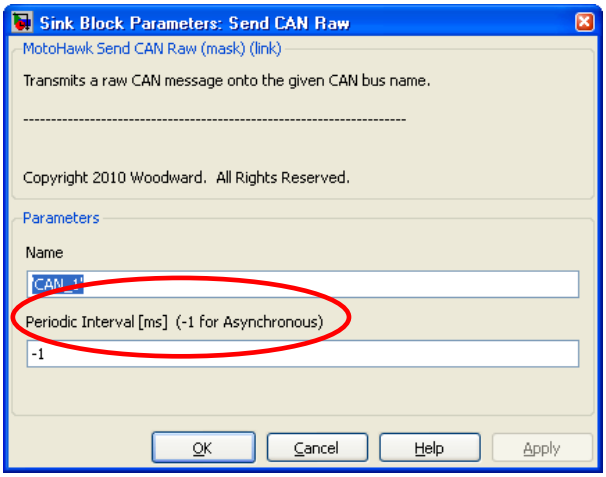
The City ID is a MotoTune protocol value that essentially identifies the device. City ID 11 (0x0b) is the default for all of our modules. City ID 2 (0x02) is the ID for MotoTune. If you monitor the CAN bus while MotoTune is active, you will see extended message IDSs like 0x00000b02 and 0x0000020b. The MotoTune Protocol uses a scheme where messages are transmitted with IDs of the form 0x0000DDSS where DD is the Destination City ID and SS is the Source City ID. You can simultaneously connect MotoTune to several modules. Each module must have a different City ID.

CAN Transmit Raw

This block can have multiple instances within your model. The bus that you want to transmit on and the interval of transmission are defined. The inputs to the block are the ID and its type, the length of the data to send and the data itself.



Data (0-8 bytes uint8): This block is designed to take a vector on the Data port of any size of up to 8 bytes. If you feed the port with a vector of only 3 bytes, but set the Length port to be 8, then the block will pad the extra bytes with the value 0.

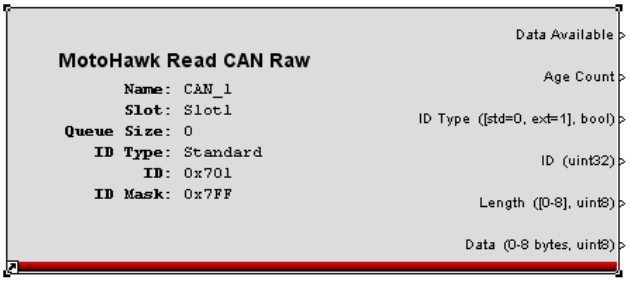


Periodic Interval [ms]: If this value is set to -1, then the message will be sent every time this block is executed. If the value is set to a positive value, then the block will attempt to transmit the message at the requested rate. However, this check is only done whenever the block is executed. So, if the block is running at 5 ms and the Periodic Interval is set to 12 ms, you will see the message on the bus at a 15 ms period.

CAN Receive Raw

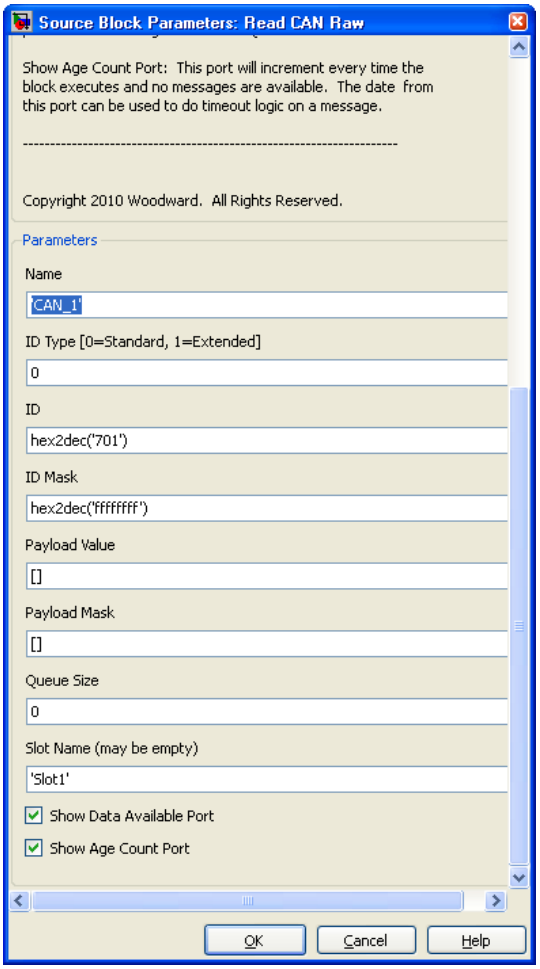
This block can have multiple instances in a model. If the slot name is defined, it must be unique.

The parameters define the CAN bus, message ID, ID mask, Payload and Payload Mask, along with the receive queue size and the slot name. A data available port (1 whenever the queue has any messages) and an Age Count port (increments whenever a message is not available and resets when a message is available) are also present.



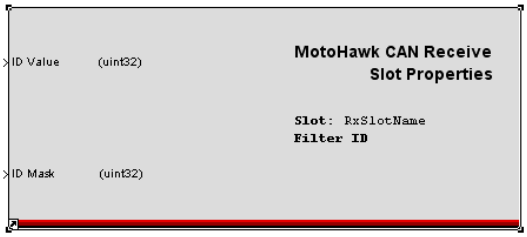
Masks

Masks define which bits must match. A bit value of 1 within a mask means that the corresponding bit in the ID or payload must match the incoming message to be received by this block. A bit value of 0 in a mask positions means that you do not care what value is in that position.



Slot Properties

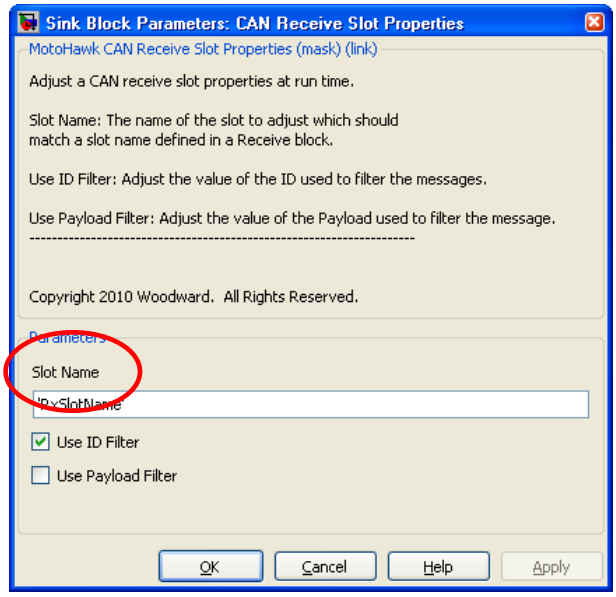
This block can have multiple instances in a model.



The slot name is used to match to the slot defined in the receive block. The choice of adjusting ether the ID filter or the Payload filter is set here.

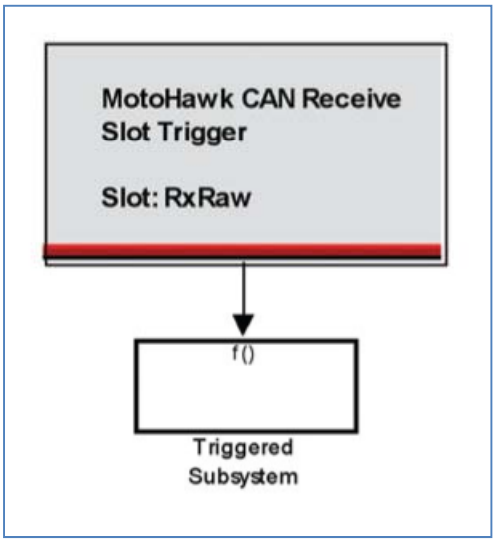
Remember that slots can only be tightened, so only mask bits that were 0 in the corresponding receive block can now be set to 1.

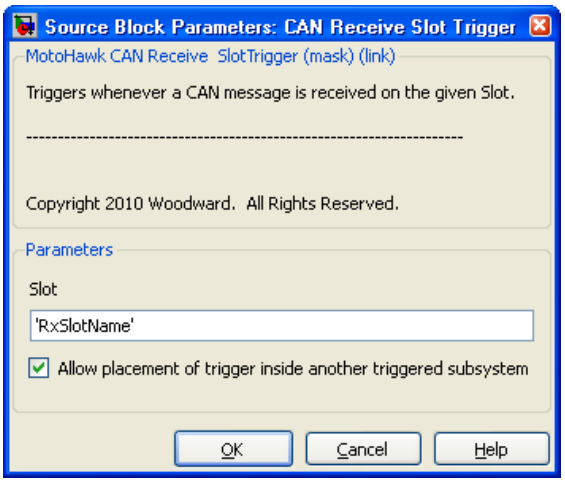
Usually, this block is placed in a triggered subsystem, so that the slot properties are adjusted only on some conditions — such as at startup or on change of some state.



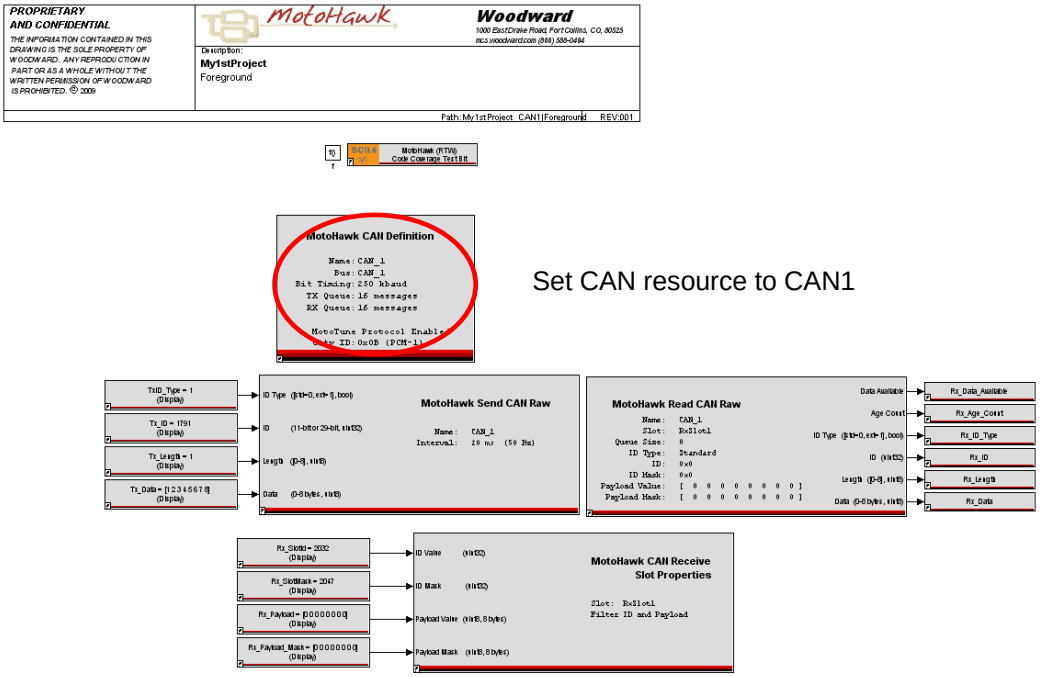
Slot Receive Trigger

This block provides a function call trigger whenever the specified slot receives a message. This trigger is high priority and will interrupt any other executing periodic task.





Example of Basic CAN Blocks



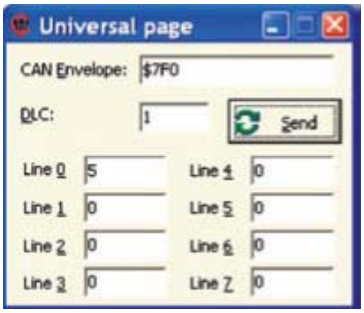
Set CAN resource to CAN1

1. Start with motohawk_project Can1.
2. Remove the existing contents of the Foreground subsystem.
3. Create the model as shown using the Can Send/Receive Raw blocks and Probes/Calibrations as block input and output.— Build the model.

4. Note: If you leave the resource set to “none” in the CAN Definition Block, you will not be able to communicate with the module and will need to recover module with a boot-key.
5. Run MotoTune, program the module, and open a display.
6. Run CANKing.
7. Right-click on the CANKing output window and select “Fixed Positions”.
8. In your MotoTune display — change the formatting of RX_slotID, RX_slotIDmask, RX_ID, and TX_ID to display hex.

   Notice in CANKing that the message 0x6ff is being transmitted every 20 ms.

9. In CanKing, transmit a message on address $7f0 with any data.

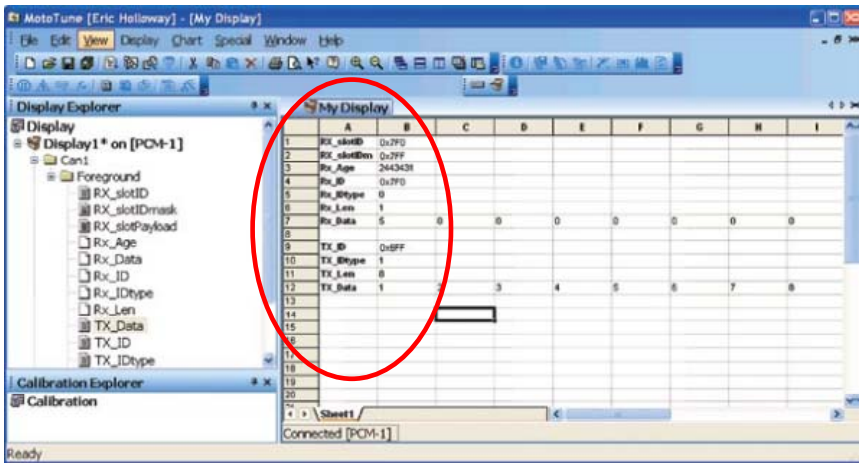


You should see the data in your MotoTune display and the Rx_Age value should reset and start counting from 0.

Adjust the slot ID and mask as well as the payload values to see how the messages are affected.

In CANKing, a value starting with $, like $7F0, means that the value is in hexadecimal rather than decimal.

Ending the ID value with an x, like $7f0x, would mean make the ID extended rather than standard.



10. The individual bytes of the payload may also be set using the $ notation for hexadecimal.
11. DLC is the number of bytes to transmit in the payload.

Advanced CAN Blocks

Payload Bit Numbering

Critical to the definition of messages is the location of the least significant bit within the possible payload positions.

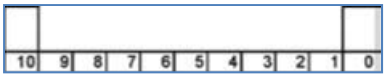
MotoHawk defines the bit numbering as shown below. This bit numbering is different than most protocol specifications.

You ALWAYS specify the location using the LSB of the field, regardless of the byte packing order.

You do NOT necessarily use the bit furthest to the right, which would be the positions. MotoHawk defines the bit numbering as shown to the right. This bit numbering is different than most protocol specifications.

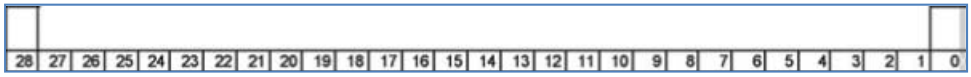
Standard ID Bit Numbering

Like payloads, IDs can be packed or unpacked. For standard IDs, the bit number is defined as shown below.



Extended ID Bit Numbering

For extended IDs, the bit numbering is defined as shown below.



The payloads contained within CAN messages often need to be packed or unpacked into their constituents for use by the rest of the model. MotoHawk provides a transmit and a receive block that incorporates the packing and unpacking of data elements into the messages. Additionally for transmission of messages, the block can pack multiple messages simultaneously and place them onto the bus at a specified period for the message group, as well as an inter-message pacing interval to conserve bus bandwidth.

Each of these blocks requires a message definition in order to properly pack or unpack the data. The message definition is nothing more than a MATLAB structure containing specific fields which we will cover below. In addition to unpacking the payload, it is also possible to unpack the ID fields. This becomes important for protocols, like J1939, where the bottom byte of the ID is the source address of the module transmitting the message. As with the Can Read Raw block, all of the ID mask and payload mask details still apply.

For transmitting CAN messages — setting the payload mask will cause the bits that are set to precisely have the value set in the payload value, regardless of the

value of any fields that might be defined on those bits. This allows you to set fixed elements of the payload to a value without needing to define fields for those values.

An m-file, motohawk_can_example, is provided with MotoHawk that defines a proper Matlab structure for defining a MotoHawk CAN message. We recommend copying this file and creating new CAN message definitions using the supplied structure as a template.

Message Definition Structure

Motohawk_can_example.m contains the details of the structure format needed to define a message.

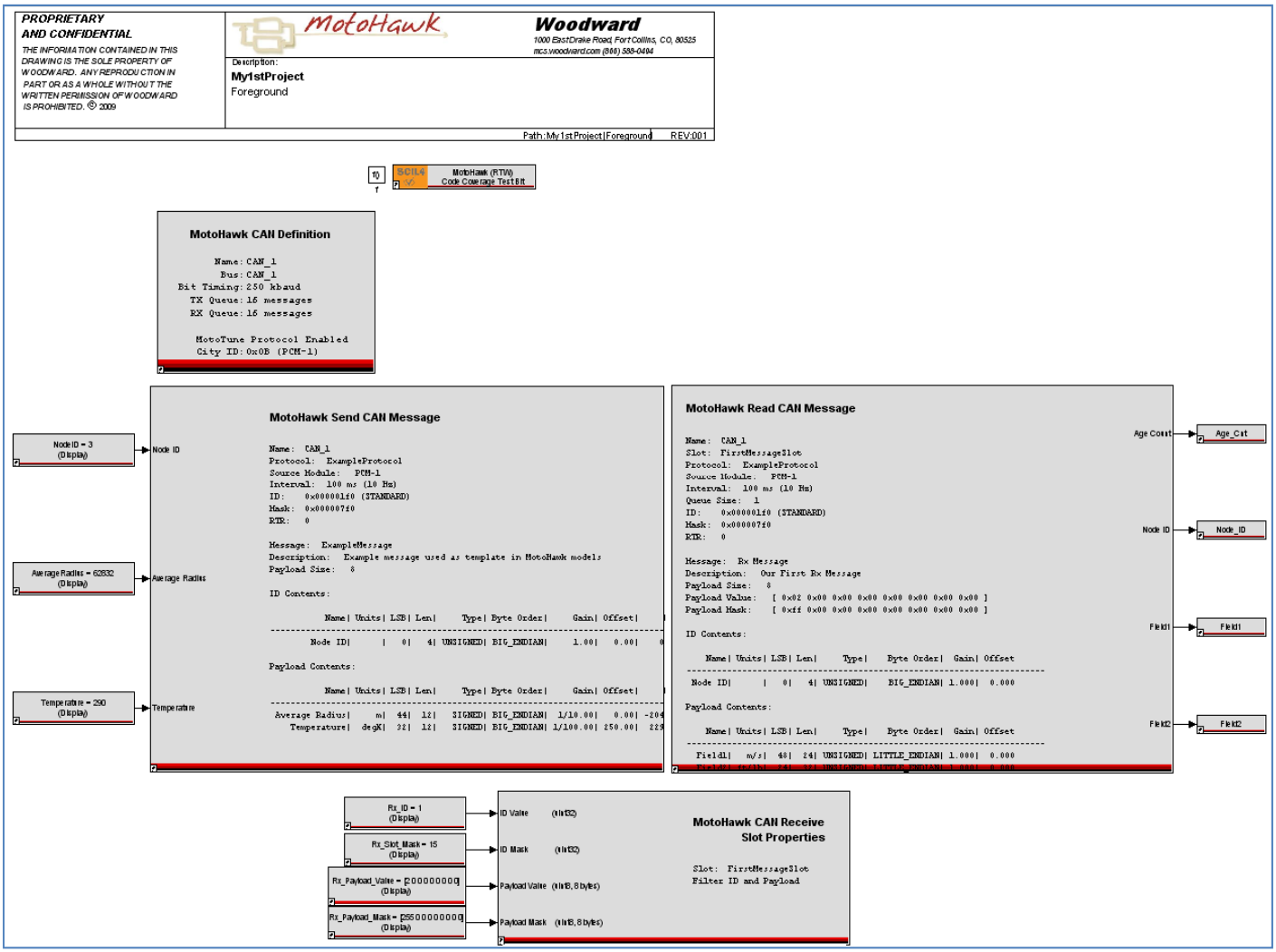
.name	name displayed on block	(default: empty string):
.description	brief text used to document message	(default: empty string)
.protocol	name of the protocol used	(default: empty string)
.module	name of the source module	(default: empty string)
.channel	number of the source CAN channel	(default: 1)
CAN ID Setup		
.id	may be either 11 or 29 bits (if undefined, uses .idinherit = 1)	
.idext	either ‘STANDARD’ (11-bit) or ‘EXTENDED’ (29-bit) (if undefined, uses .idinherit = 1)	
.idmask	indicates which bits are relevant for a receive slot (default: 0xffffffff)	
.idinherit	Default 0. When set to 1, causes the message to use the ID of the previous message in a list of messages (only applies for transmit messages)	
.idcontent{}	bit fields within message ID, as described below. (optional). Describes individual fields within the ID. May be undefined or empty, if no ID content is defined.	
Transmit Interval, Message Size, and Contents		
.interval	period in milliseconds, or -1 if sent asynchronously (default: -1)	
.payload_size	payload size may be from 0 to 8 bytes. (default: 8) Transmit: exact number of bytes to send. Receive: minimum number of bytes required.	
payload_value	Just as an ID has a value and mask, so can the (optional) payload. For receives, this will result in a software filter requiring the bits set in the payload mask to be equal to those in the payload value. For transmits, any bits set in the payload mask will be hard-coded to be the corresponding bits of the payload value, regardless of any payload fields that may overlap it. A typical use of this feature is to identify a specific message by the first byte of the payload. May be a vector of bytes or a hex string.	
.payload_mask	Indicates which bits of the payload are relevant for a receive slot, or which bits will be hardcoded for transmits. If the number of bytes is less than the size of the payload, the unset bytes are assumed to be 0, meaning do not care. May be a vector of bytes or a hex string.	
.fields{}	Fields within message payload, as described below. (optional). Describes individual fields within the payload. May be undefined or empty if no payload fields are defined.	

Structs in the .idcontent{} and .fields{} cell arrays may contain the following fields:

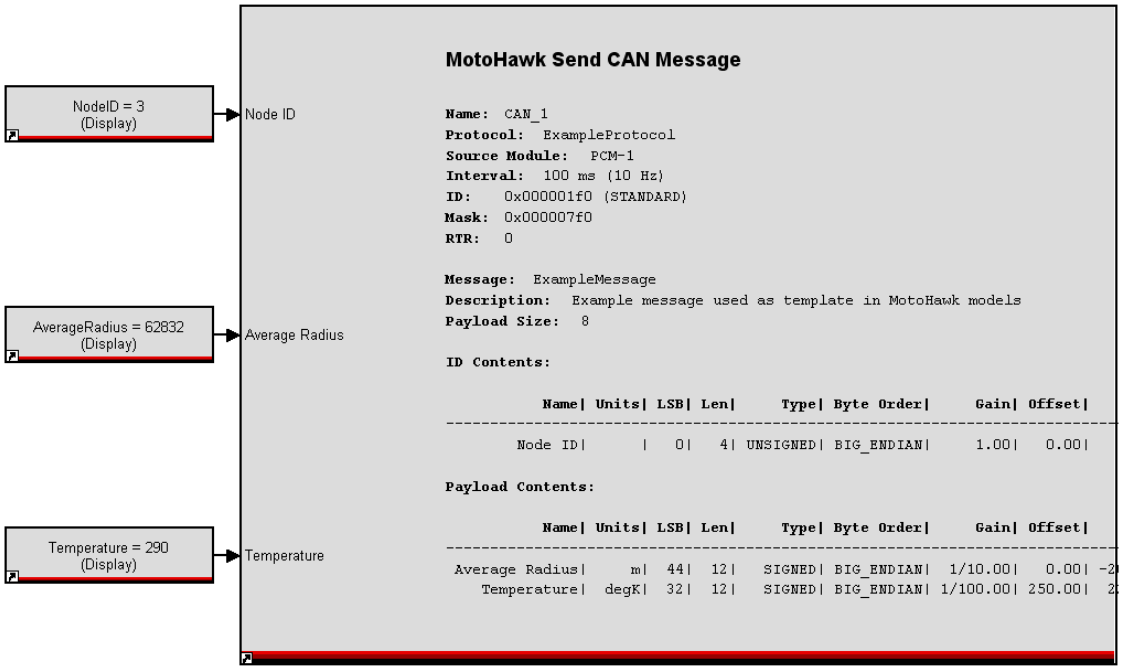
.name	name displayed on the block	(default: empty string):
.units	units (of Simulink-model value) used in mask display	(default: empty string)
.start_bit	indicates the least-significant bit of the field regardless of endian-ness (required)	
.bit_length	number of bits in the field may spill across bytes (required)	
.byte_order	may be 'BIG_ENDIAN' or 'LITTLE_ENDIAN'. (default: 'BIG_ENDIAN') (only 'BIG_ENDIAN' is valid for .idcontent{} fields)	
.data_type	may be 'UNSIGNED', 'SIGNED', 'FLOAT32', or 'FLOAT64' (default: 'UNSIGNED')	
.scale	scale factor. Since the same message description (default: 1.0) struct is used for both transmits and receives, the scale factor should not be thought of as a gain. Instead, think of it as the units of the signal in the payload on the CAN communication wire such as 1/100 of a degree for a signed integer representing degrees Kelvin where 1245 (in the payload on the CAN communication wire) represents 12.45 degK (in Simulink model units). See equation below.	
.offset	offset applied to the field in engineering units. (default: 0.0). This is sometimes used to represent high-resolution values in a range far from zero. To represent Simulink-model values from 230 to 270 Kelvin, a range of +/ - 20.47 degC with 0.01 degC resolution is available using a signed 12-bit value in the payload on the CAN communication wire with an offset of 250 Kelvin. See equation and example below.	

Advanced Example

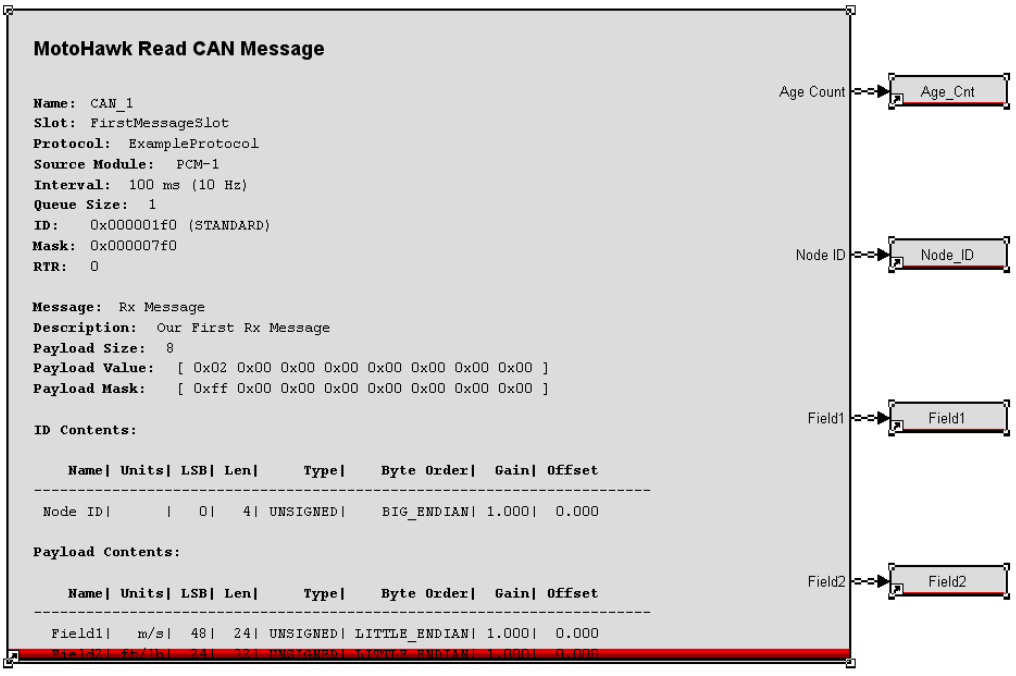
1. Create a new model using motohawk_project('can2')
2. Remove the existing contents of the Foreground subsystem
3. Create the model as shown using the default MotoHawk_CAN_example, CAN Send/CAN Read blocks with Probes and/or Calibrations on CAN block input/output. Build the model.
4. Run MotoTune, program the module, and open a display.
5. Run CANKing.



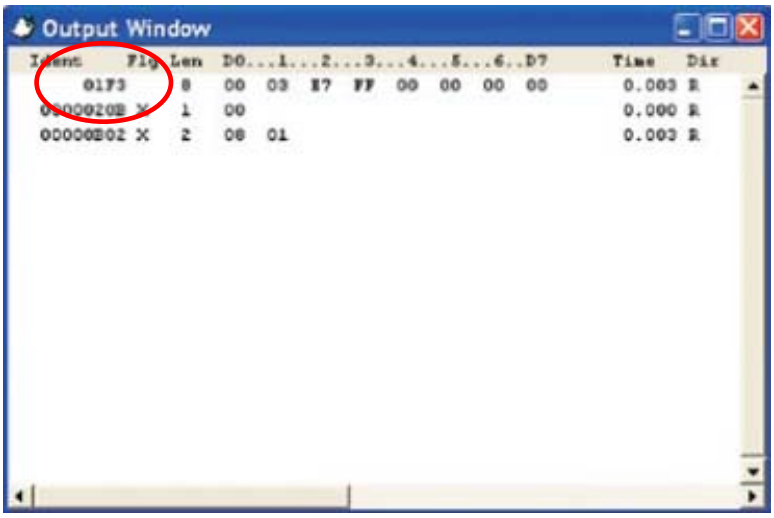
Expanded Image of Send CAN Message from Above:



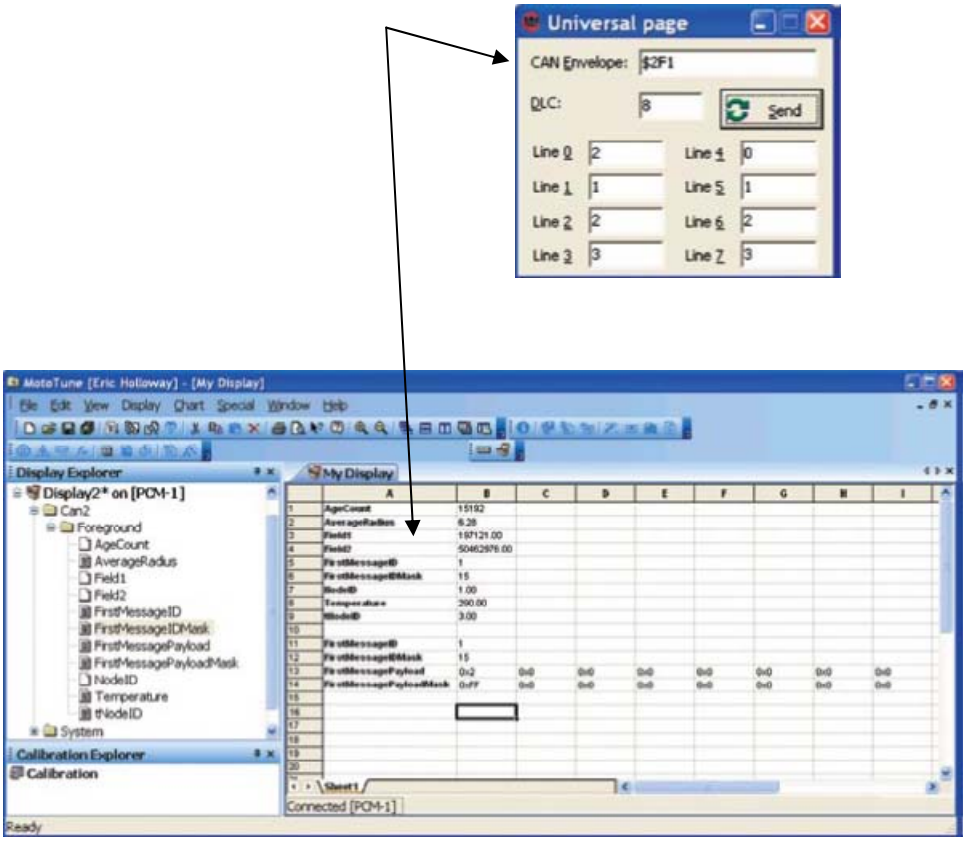
Expanded Image of Read CAN Message:



6. Note the 1F3 message being transmitted. The 3 comes from the Node ID input.



7. Transmit a message from CANKing and verify that the value is received by the module as shown by the probe values in MotoTune.



Chapter 4.
Memory Management

Introduction

MotoHawk is designed to be an integrated rapid prototyping control system solution out of the box. However, once a system starts growing into a larger control system, memory management becomes increasingly more important.

In this section, we will discuss the basic memory layout of Woodward's control modules and discuss in detail the blocks that have the most impact on memory usage and performance. Memory management of your MotoHawk control system requires the understanding of vardecs (Variable Declarations).

There are three types of variables available in MotoHawk; Constant (Const,) Non-Volatile, and Volatile Data.

Constant: is just that, constant, never changing data.

Non-Volatile data: can be changed and is saved between power cycles. Non-Volatile data is predictable during and between power cycles because it will always retain its last known value.

Volatile data: can be changed, but is not saved. After a power cycle it will return to its original default value.

Knowing your memory

Woodward control modules include three types of storage devices.

Flash is read only memory and retains its information between key cycles.

Control Core, the MotoHawk application, and constant data are stored in the flash region of the module.

EEPROM (Electrically Erasable Programmable Read Only Memory) is similar to flash, in that it will retain its information across key cycles.

However, EEPROM can be erased and written to. This section of the module becomes the most important when saving calibration changes and is responsible for saving and recalling the non-volatile data in a model. Read and write to the EEPROM as your control algorithm changes. We will discuss later when the EEPROM is written to and how to ensure that you safe guard your data.

There are two different types of EEPROM, serial and parallel. Parallel EEPROM is only available on a development module. This memory is what allows the user to change non-volatile display and calibration variables in real-time during testing and validation.

RAM (Random Access Memory) is only temporary memory space used for volatile data.

The contents of RAM are erased between key cycles. Any changes made in RAM will be lost once the module has been turned off.

Flash is used to write information that cannot be accidentally overwritten. This is why the program is stored in flash. If the program was stored in EEPROM, one wrong memory write and you may have overwritten a vital part of the control system.

Why so much different memory?

EEPROM is the work horse for memory management of your control system and offers the best of both worlds. It is capable of storing information, but is also capable of erasing and writing new information.

There is one drawback to EEPROM — any given memory location can only be written to at most 100k times. So if you were saving a variable every 5ms, it would not take long to reach the 100k cycle and possibly burn out that location of the EEPROM.

To avoid this problem, the contents of the EEPROM are “shadowed” into RAM when the module is turned on. Changing a variable that will be saved across a key cycle is actually changed in the RAM copy and shadowed back in the EEPROM at shutdown. Later you will learn how to save the Nonvolatile data based on your own criteria.

Knowing the hardware

Woodward control modules come in two different versions.

The development version has an added parallel EEPROM region where vardecs are stored.

This extra memory region allows the user to view and change calibration and display variables using MotoTune and is typically used for testing and calibration.

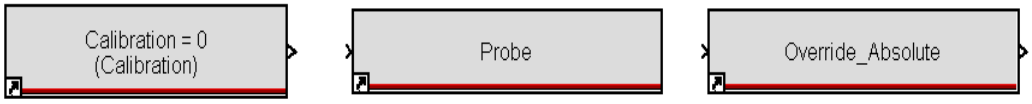
A production module contains only the serial EEPROM.

No real-time calibrations can be performed with this module without explicitly assigning the variable to be stored in the non-volatile region, which we will discuss in the next section. In this way, the cost of production modules is kept down relative to their development counterparts.

Familiarize yourself with the interface

MotoHawk has three basic blocks that allow viewable variables to appear in MotoTune: calibrations, probes, and overrides. Before we discuss each individual block in-depth, let’s look at the similarities you may find when looking at their masks.

Mask parameters are accessed by double clicking on the block.

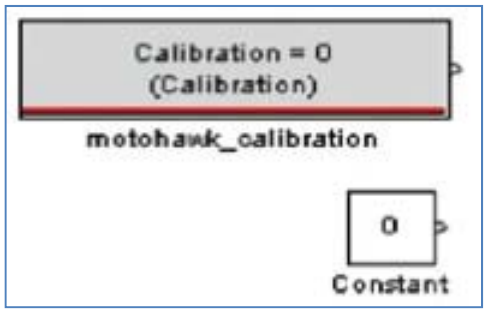


We will use a calibration block to illustrate the parameters. Drag a calibration block into your model and double click on the block and a separate window will appear listing that block's mask parameters. Anything said about this block's mask parameters can be applied to any block with similar fields.

Calibrations

Calibrations can be described as a MotoTune accessible Simulink constant block.

Unlike Simulink constant blocks, the MotoHawk Calibration block can only be used once per declared vardec in the model. However, because a Calibration is composed of a Data Storage block, you can use a Data_Read block to access it in other parts of the model.

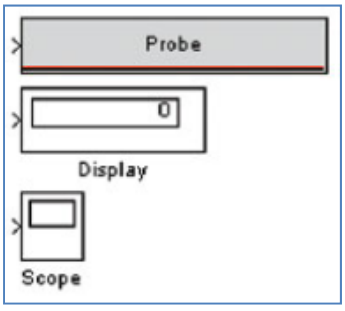


Be careful – it is easy to drop calibrations all over the model, but if you allow them all to be doubled, you may be wasting memory.

Probes

Probes are read-only displays stored in RAM. A MotoHawk probe is similar to Simulink's native display block and scope block. Probes can be very helpful when testing and debugging, but if used carelessly, they can use more RAM than necessary.

Probes will require extra memory when the wire it is placed on is not already being kept around between execution cycles in the system, so if the control system design requires the value of a particular wire to retain its value between cycles, the value will be allocated memory space. Probing such a wire will not add any further memory because the optimizer recognizes the two values to be the same and they both reference the same memory location. However, if the signal is not kept by the control system, then adding a probe will require more memory.



Overrides

Overrides are inline calibrations and have both an input and an output. There are two different types of override blocks. Both blocks create two vardecs that can be manipulated within the display window of MotoTune.

The override relative block is a way to lock the output and apply an offset. It was designed around some legacy software and is typically not a block that a control system will use.

The override absolute enables the MotoTune user to ignore the input and use a specified value.

Block Parameters

Name: This field can be any MATLAB expression (such as those in the Motohawk_can_example.m file above) or a string, so that it can be called from other MATLAB functions. If a string is used, make sure to enclose the string in single quotes.

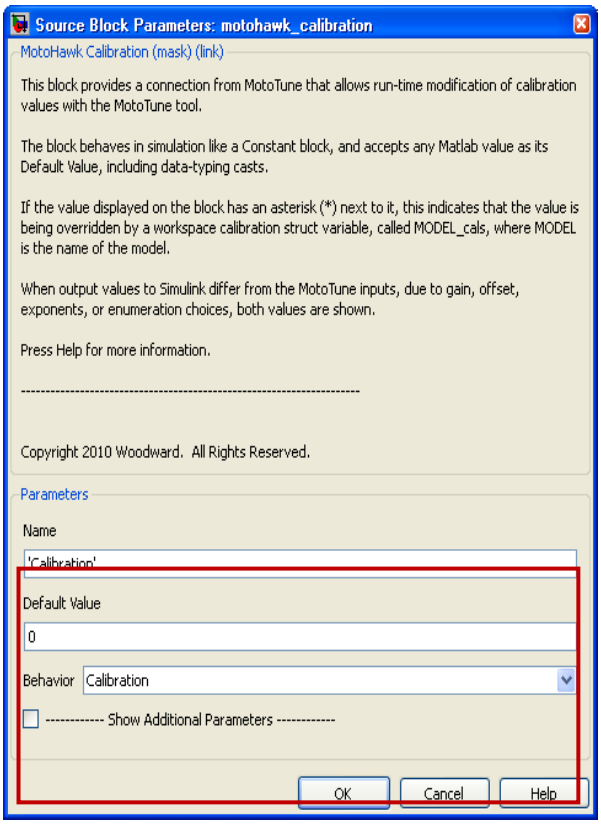
What are valid MATLAB expressions?

MATLAB expressions can be workspace variables or MATLAB functions that return a number. Commonly, an m-script is made with calibration values stored by name. This m-script is called and all the calibrations are loaded in the workspace and the default values reference those values.

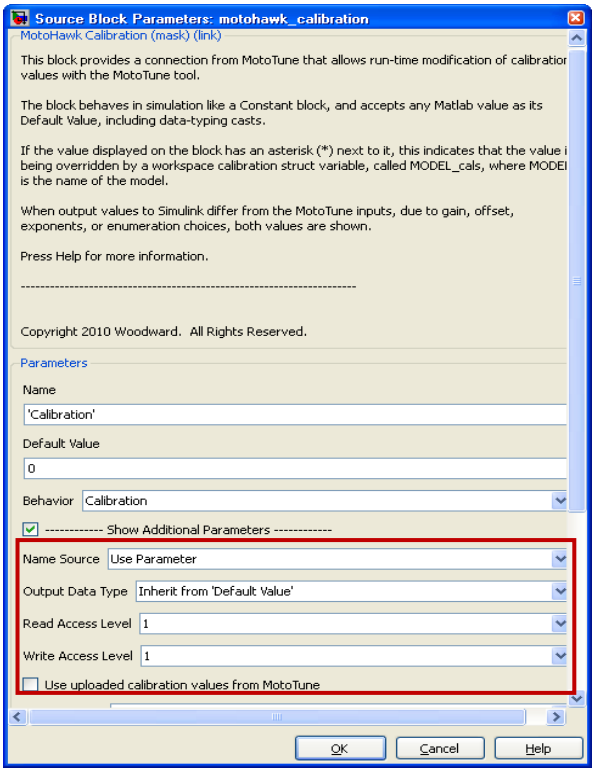
Default Value: This is the value that takes effect from the first time of programming. It remains in effect until it is changed using MotoTune.

Behavior: This is where you decided what type of memory this variable will be stored in.

- Calibration
  - Flash (prod)
  - Parallel EEPROM (dev)
- Display – RAM
- Calibration NV – Serial EEPROM
- Display NV – Serial EEPROM



Show Additional Parameters: Click the check box to show a list of additional parameters to modify for this block.



Name Source: "Use Parameter" is the default for this field, which requires the name field above to be entered. Other choices include "Use Output Wire Name", or "Use Input Wire Name". This will gray out the "Name" field and reference the wire name attached to the block.

For a calibration, if you select "Use Output Wire Name", then double click on the wire attached to the output and provide a name for the wire, update the model, then the calibration block will take the name provided for the wire.

Output Data Type: By default, MATLAB makes all data types double. By not making a selection or specifying it in the "Default Value" field, then the output will default to double. Otherwise, you have two ways of specifying the data type. You can leave this field to "Inherit" from "Default Value" and enter the data type along with the "Default Value" field. (For instance, unit 16(0)). This indicates that the default value will be a 16 bit unsigned integer with the value of zero, or you can use the pull down selection of this field to explicitly identify the output data type, such as uint16. You can then leave the default value to be just the number zero.

Access Levels: Access levels handle the security of the control system and relate to the access level of the MotoTune security dongle, as well as the port access level specified on the PC connecting to the control module. Access Levels range from a value of 1 thru 4. By default, all the blocks that have access levels are set to "1." Anyone with a MotoTune security dongle with access level 1 or above may view and/or change this vardec.

Since 1 is the lowest access level, everyone has access to this vardec. However, if the access level was set to a 2, and your MotoTune security dongle only had access level 1, you would not have permission to view or change this vardec.

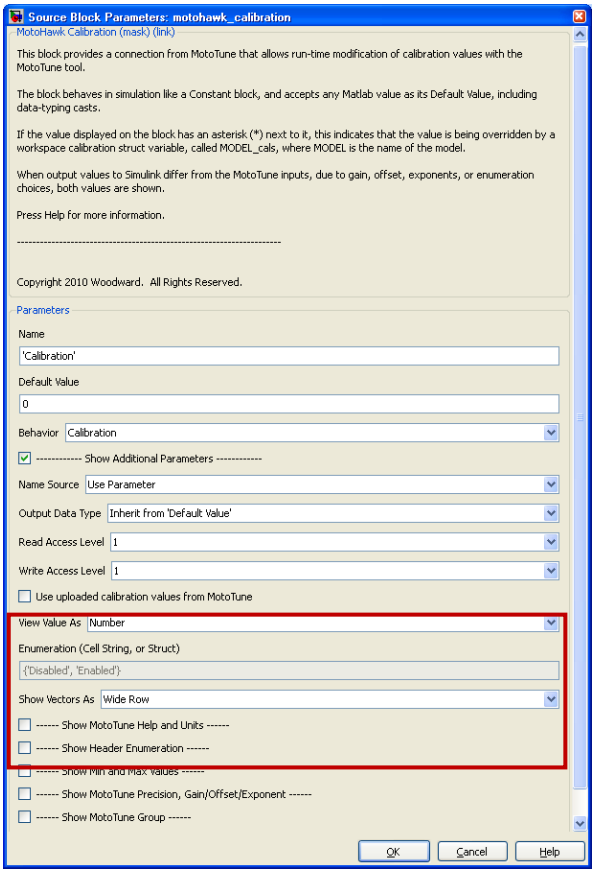
By default all MotoHawk kit dongles have access level number 4. Since level 4 is the highest, those dongles have access to everything within the control system. Lower level dongles are available from Woodward.

Show Additional Parameters (cont'd)

Use uploaded calibrations values from MotoTune:

This selection indicates if you want the source of this variable to be a separate MATLAB file or if it may come from a different source.

The MotoHawk upload calibration feature will make a MATLAB m-script for every defined vardec, but if a vardec is generated from a separate piece of software, you want your model to ignore the value located in the m-script file. For example, if you were constructing an autonomous vehicle that included GPS coordinates and the coordinates are generated from a mapping program, you would deselect this option.



View Value As: MotoTune has been designed to show data one of three different ways: *number*, *enumeration*, or *text*.

If enumeration is selected, then the Enumeration field may be used to specify the text associated with the enumeration. What you see in MotoTune will be the text in the Enumeration field (On/Off, Start/Run/ Stop, etc.) instead of a number. Be careful to make sure the enumeration text and numbers align properly.

Enumeration (Cell String, or Struct): Enumeration associated with the input when the "View Value As" field is selected to Enumeration.

Show MotoTune Help and Units: Select to show help text and units.

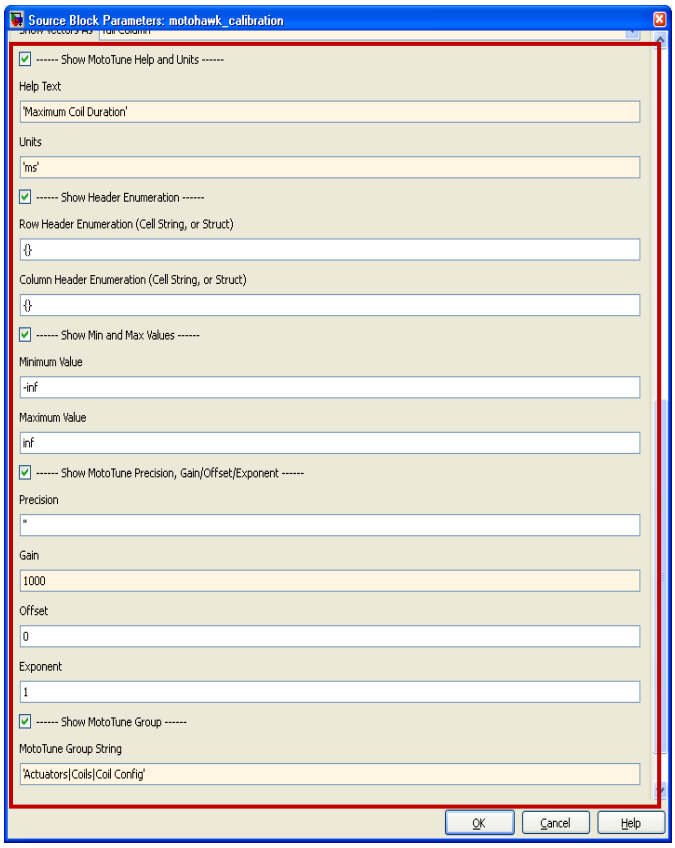
Show Additional Parameters (cont'd)

Help Text and Units:

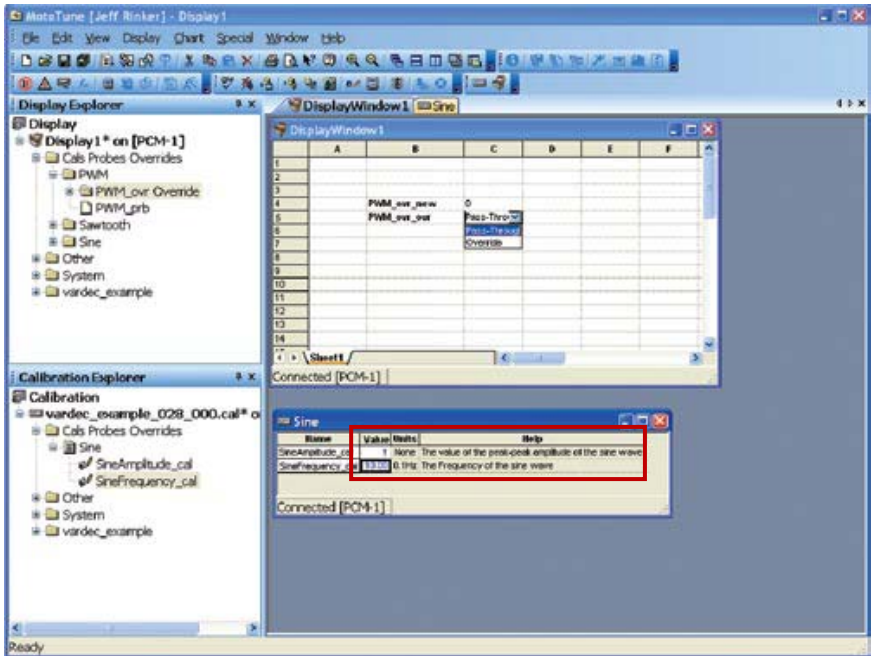
Help Text: Text to aid the MotoTune users what this vardec does and what it might effect if changed.

The text shows up automatically with calibrations. The help text and units automatically display with calibration values.

For displays, right click on a variable and select its properties to view the associated information including the help text for that variable.



Notice the help text and unit information that is displayed next to the calibration. If the vardec is specified as a display, you must right click on the value and go to Properties/More to view its help and unit information.



Units: Indicates to the MotoTune users what units this vardec is specified in for clarification during testing and calibration.

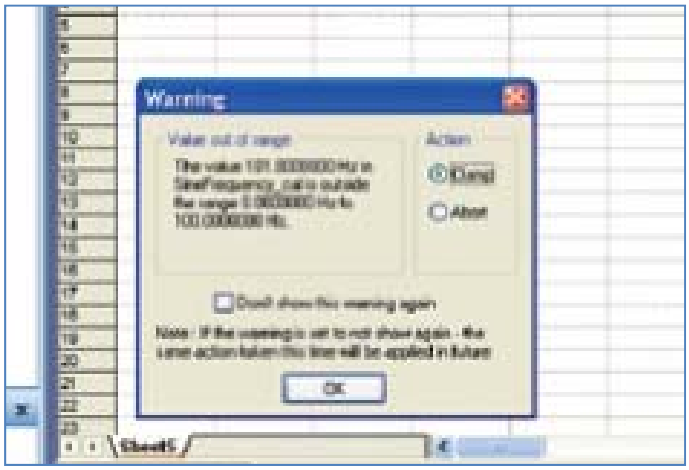
Show Min and Max Values: Select to show min and max values.

Minimum Value and Maximum Value: Minimum or Maximum Value for this vardec. This will clamp the signal in MotoTune.

If an attempt is made to go below (above) this value, then MotoTune will display a clamp value message and will force the value to this minimum (maximum) value.

This is useful to ensure a calibration is not accidently changed outside of a specified range. This min and max takes into consideration any gain, offset, or exponent that was applied to the value.

By default the minimum value is –infinity (-inf) to prevent MotoTune from clamping the value if it is changed.



Show MotoTune Precision, Gain/Offset/Exponent: Select to show MotoTune Precision information. The Precision, Gain, Offset, and exponent information is for MotoTune use only.

This is not to be used to convert analog/digital counts (ADC) to engineering units. These values are typically used to allow the designer of the system to use proper system units, but display the value in more convenient units in MotoTune (ie. English units, SI units).

Show Additional Parameters

Show MotoTune Precision, Gain/Offset/Exponent (cont'd.):

Precision: Sets the default precision for the variable. The format is: 'width.decimal'.

For instance, if you wanted the entire width of the variable to display 6 digits with 4 decimal places of precision, you would enter '6.4'. The width takes precedence, so if your variable is six digits, there will be one decimal place applied. However, if your variable becomes a seven digit number, then the precision would expand.

Gain, Offset, Exponent : These values only apply to how the variable will be displayed in MotoTune.

These values are not to be used to apply a gain, offset, or exponent for ADC to Engineering Unit conversion.

The equation is as follows :

MotoTuneValue = (*value * gain*)*exponent + offset*

This determines how MotoTune will organize the data within its messages and how it will be displayed. So, if MotoTune were to display a value in 1000's of RPM, a 1 would appear in the cell in your display window for a value of 1000RPM.

Show MotoTune Group: Select this to specify the MotoTune group. This entry allows customizing of the group structure in MotoTune.

Just like the Name field, this value can be an expression, which means it can be a function call, just as the default value is. The default value "motohawk_vardec_path(gcb)" returns the path structure of your model.

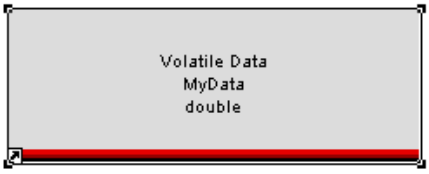
By default this value runs a function called motohawk_vardec_path(gcb,) thus the location of each vardec will be the same as the model.

For instance, if you have a calibration in a model just under the foreground task in a model named example, then by default the calibration will be located under example/foreground/ calibration in MotoTune.

To specify your own directory structure, use the vertical bar (pipe) to separate the paths. So, to put the calibration in a folder called calibrations under controller, you would type: *'controller | calibrations'* in the MotoTune group field. *Remember the single quotes.* MotoTune's directory structure consists of folders, pages, and values.

Data Storage Blocks

The MotoHawk Data Definition Block



The MotoHawk Data Definition Block defines data to be accessed via a MotoHawk Data Read or Write block.

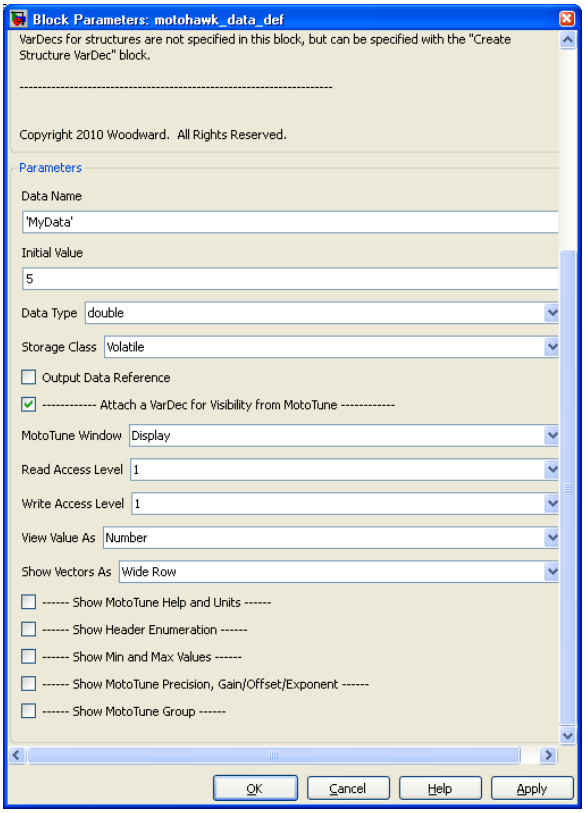
The 'Data Name' provides a globally unique name, accessible from anywhere in the model. It is illegal to have duplicate names.

The data defined in the Data Definition block is accessed in the Read and Write blocks by the Data Name

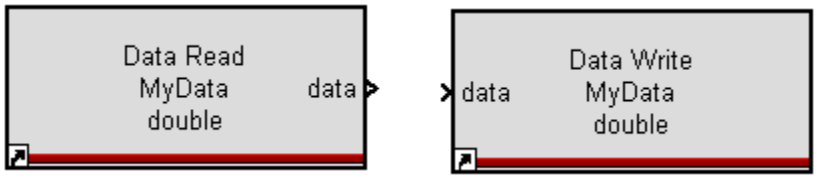
The initial value can be either a single value, a vector, or a matrix. For example, setting the initial value to [1,2,3,4,5] defines a 5 element vector of data. Matlab expressions (ex. zeros(1,50)) can also be used to define the initial value.

'Storage Class' identifies the behavior of the data on the target. Volatile data will return to its 'Initial Value' on every startup. NonVolatile data will be saved in EEPROM, and return to the last written value on startup.

Checking 'Attach a VarDec for visibility in MotoTune' will show similar parameters as the Calibration block. If unchecked, the variable will not be available from MotoTune.



Global Data Read and Write Blocks



MotoHawk Data Read Block

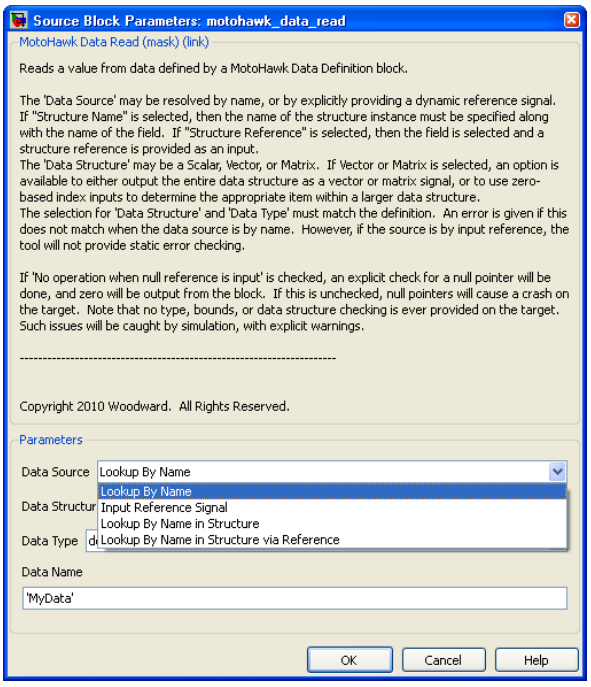
The MotoHawk Data Read Block reads a value from data defined by a MotoHawk Data Definition block.

The 'Data Source' may be resolved by name, or by explicitly providing a dynamic reference signal.

The Data Structure defines whether the input is a scalor, vector, or matrix as defined in the Data Definition Block.

The data type must match the Data Type selected in the Data Definition block

The Data Name field appears if the Data Source is defined to be ‘Lookup By Name’. In this case, the name is set to match the Name field defined in the corresponding Data Definition Block



If Vector or Matrix is selected, the array can be read or written all at once, or by specified index.

The indexing is zero based (indices range from 0 to size-1).

The selection for 'Data Structure' and 'Data Type' must match the definition.

MotoHawk Data Write

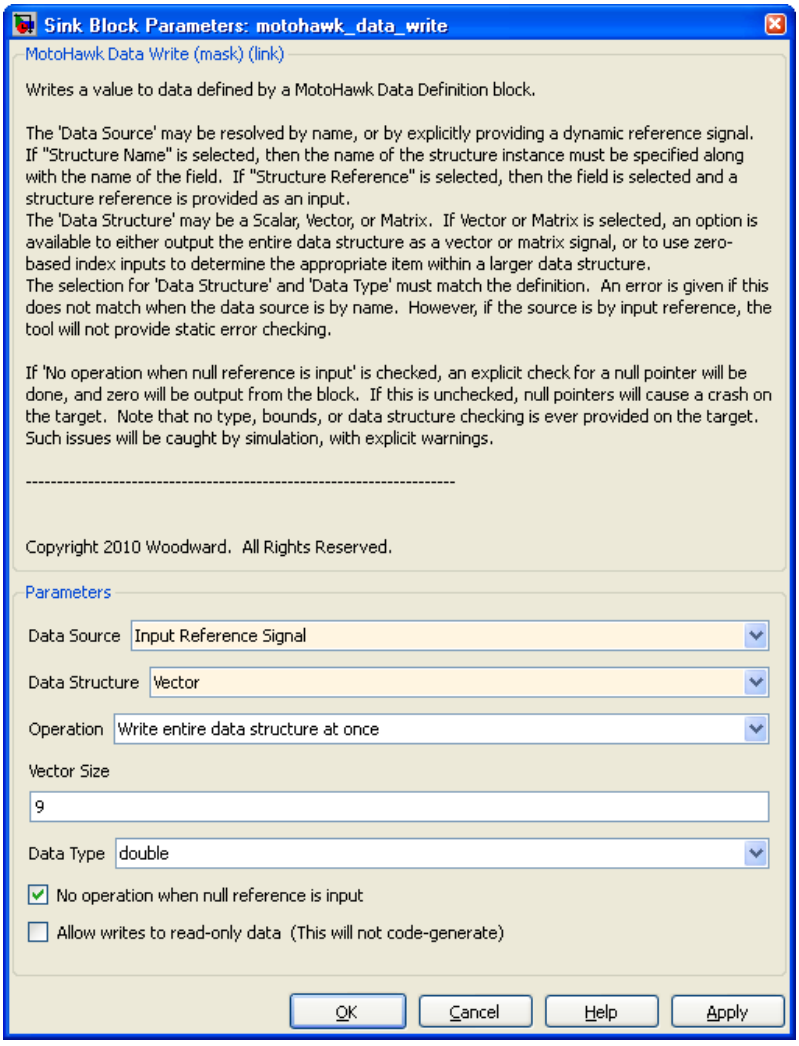
The MotoHawk Data Write block writes a value to data defined by a MotoHawk Data Definition block. The fields are similar to the Data Read Block.

The 'Data Source' may be resolved by name, or by explicitly providing a dynamic reference signal.

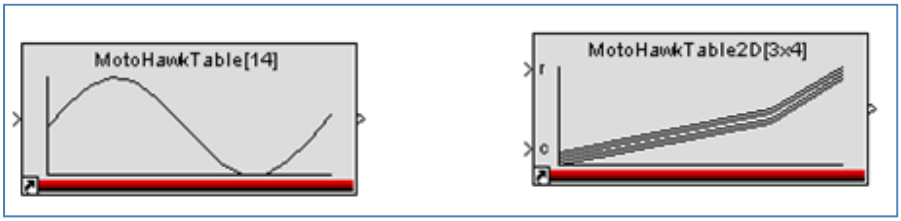
The Data Structure defines whether the input is a scalor, vector, or matrix as defined in the Data Definition Block.

The data type must match the Data Type selected in the Data Definition block.

The Data Name field appears if the Data Source is defined to be 'Lookup By Name.' In this case, the name is set to match the Name field defined in the corresponding Data Definition Block



MotoHawk Lookup Tables



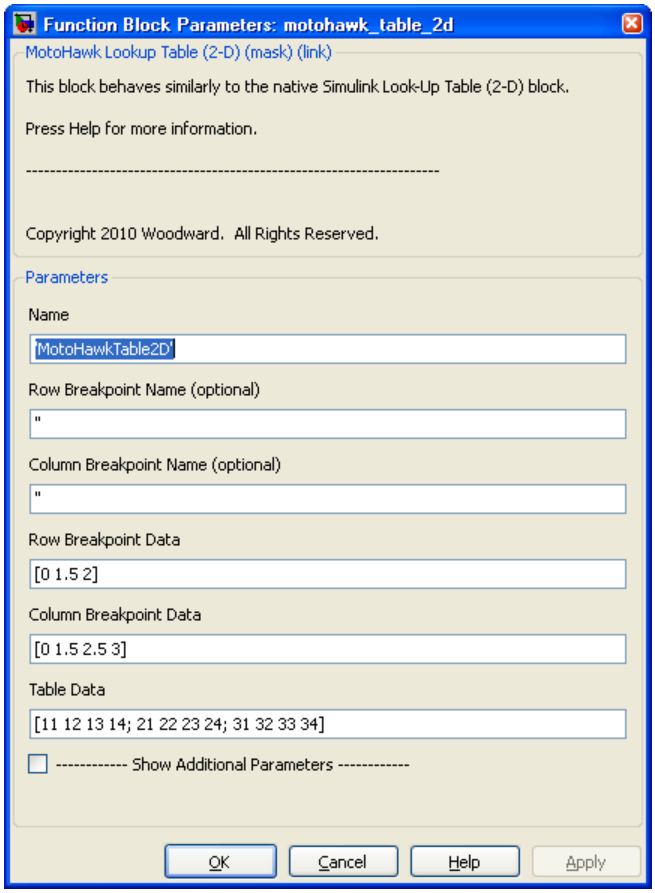
The lookup table block performs 1-D or 2-D linear interpolation of input values using the Breakpoint and Table Data. The table data will saturate at the endpoints for inputs above or below the breakpoint data range.

The Name field is the Vardec used in MotoTune. The output is normally assigned to be a display variable with this name.

Output Name is an optional name for the output of the table. If left blank, a VarDec will be generated called Name. If non-empty, a downstream probe must be provided called output name, which allows customization of the output VarDec from MotoTune. An error will result if the VarDec does not exist.

The ‘Show Additional Parameters’ option can be checked to allow entry of read/write access levels, to Enable/Disable to use or ignore uploaded values from MotoTune, as well as set Breakpoint/Table Data maximum and minimum values, and show MotoTune units and help.

See block help for additional details.

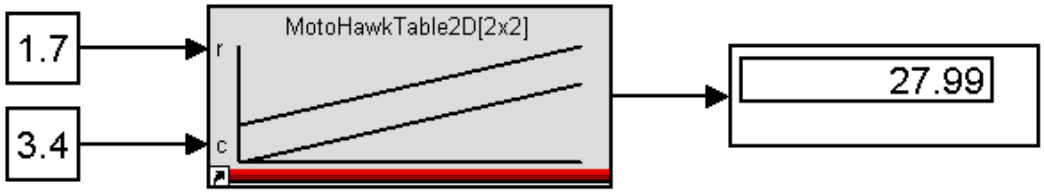


MotoHawk 2-D Lookup Table Example

Build a simple MotoHawk 2-D lookup table with the following data:

- row breakpoint data is [1,2]
- column breakpoint data is [3,4]
- table data is [10,20; 30,40]

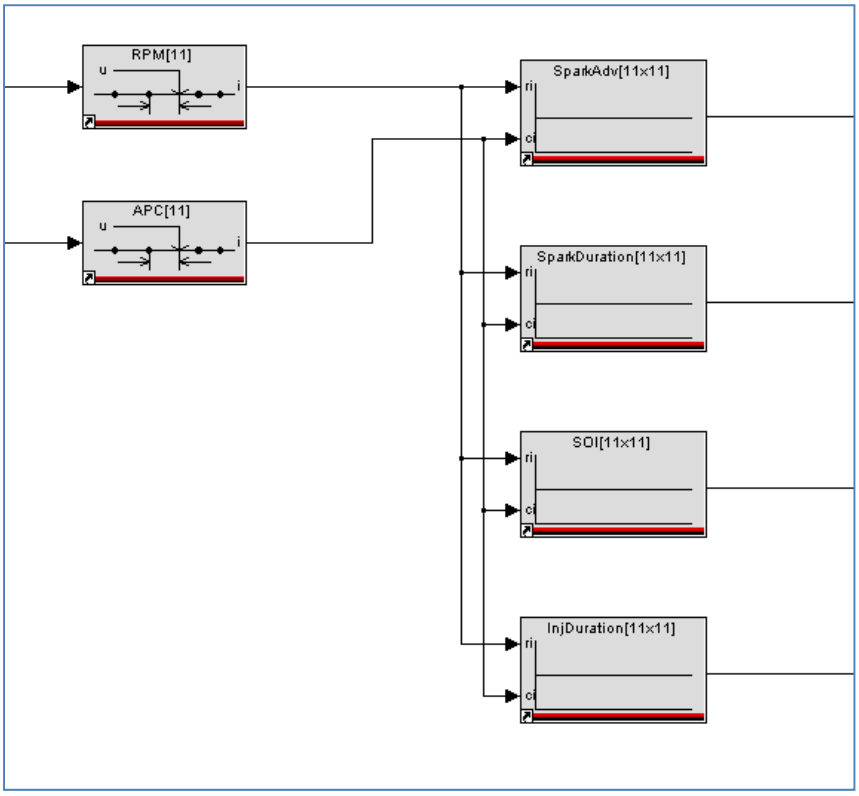
	3	3.4	4
1	10	14	20
1.7	24	28	34
2	30	34	40



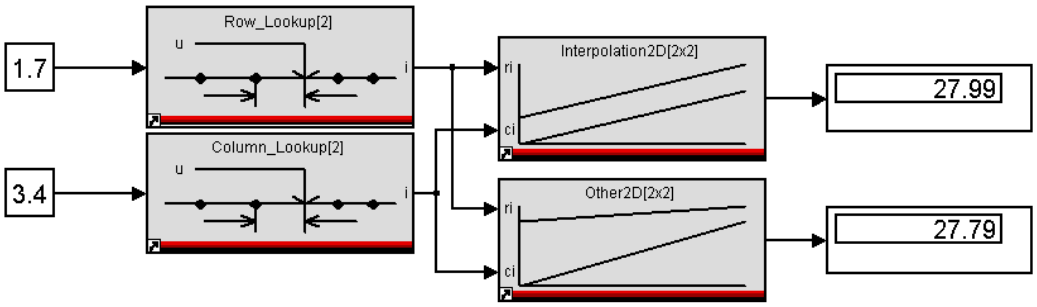
MotoHawk PreLookup and Interpolation

The MotoHawk Interpolation tables are actually constructed from two blocks, the MotoHawk Prelookup and MotoHawk Interpolation Blocks. This can be seen by right clicking on the MotoHawk Interpolation block and selecting “Look Under Mask.” The MotoHawk Prelookup and Interpolation tables can be used to share the prelookup (interpolation between two points on the axis) with multiple tables of data. The output of the prelookup can only be connected to the input port of the interpolation block.

As shown below, separate interpolation tables (which are set to zeros and can be calibrated later in MotoTune) use the same prelookup.



Recreate the first lookup table example, except this time with prelookup and Interpolation:



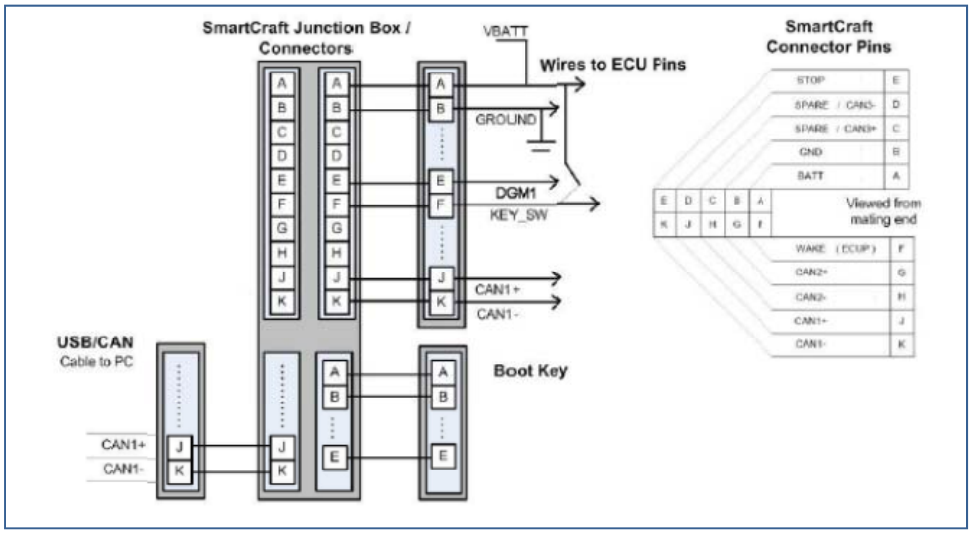
Chapter 5.
Boot Key Recovery

Errors in configuration, logic and/or other programming made during program development when programmed into a module (via .srz file), can cause a persistent loss of CAN communications with the module under development. If this happens, apply the boot key (or boot harness) to force the module into reboot mode, reloading the module with functional program code (a known, valid .srz file) in order to allow resumption of module communication. Follow the steps listed in this section.

WARNING **Remove the ECU from direct control connections before performing the reboot procedure, as outputs are set to defaults or undefined states, with unpredictable and possibly hazardous results if applied.**

NOTICE **Remove other ECUs from CANbus for this procedure.**

Refer to diagram below for connections.



Use Boot Key and Reprogram the Module

1. Connect the module for programming via necessary cables, CAN converter, etc.
2. Select a known, valid .srz file for programming.
3. With key off, disconnect battery power from module.
4. With module power off, initiate programming of the module using MotoTune.
5. When the “Looking for an ECU” prompt appears in the dialog, reconnect Battery, and then turn key on, to power up and “wake-up” ECU. The module must “wake-up” (KEYSW on) with the boot key or cable connections applied as described in order to initiate a reboot and to absorb the selected program.

Chapter 6.
MotoHawk Acronyms and Terms

TERM OR ABBREV.	MEANING	NOTES
General Terms		
CC	Control Core	MotoHawk's Operating System
City-ID	MotoTune's Address ID for communication port	
GCC		Open Source Compiler that can be used with MotoHawk
GHS	Greenhills Compiler	Production compiler for use with MotoHawk supplied by Greenhills Corp.
MCS	MotoHawk Control Solutions	Product Line containing Modules, MotoHawk and MotoTools Software, and related components
MH	MotoHawk	Model Based Software Tool
MS	MotoService	A scaled down version of MotoTune for programming and calibration in the field
MT	MotoTune	A tool for programming and calibration of MCS modules
NV	Non-Volatile memory	Memory that is stored across key cycles, as opposed to Volatile memory which is not stored across key cycles.
PWM	Pulse Width Modulation	Square Wave Signal that varies pulse width at a certain frequency
RTI	Real Time Interrupt	Periodic rate for application code
RTW	Real Time Workshop	Software from the Mathworks that is used along with MATLAB and Simulink for embedded code generation
TLC	Target Language Compiler	Mathwork's proprietary compiler, used by MotoHawk in code generation
Vardec	Variable Declaration	Term is generally used for items that can be viewed or calibrated in MotoTune
File Extensions		
.dll	windows dll file, created by build	
.mdl	Extension of a MotoHawk Model file	
.srz	Extension of the programming file created by the MH build	

MotoHawk Acronyms and Terms (cont'd.)

TERM OR ABBREV.	MEANING	NOTES
Licensing Terms		
.acf	Activation File	A file that is returned from a license update transaction containing license activation
dongle	Silver USB token containing software licensing	A silver token that a user purchases that contains the license for MotoHawk, MotoTune, and/or MotoService. The token is placed in the USB drive of a computer to run the software.
.tkf	Transaction File	A file that can be sent by the customer to update MotoHawk license by email, rather than shipping a new dongle
Some MH Related Industry Terms		
CCP	CAN Communication Protocol	Standard communication protocol used for calibration and data acquisition from electronic control units. Modules communicate with MotoTune on a Woodward proprietary protocol, but can be calibrated by industry standard tools such as INCA with CCP.
HIL	Hardware in the Loop testing	
ISO15765/UDS		A standard protocol for communicating diagnostic information with a scan tool. MotoHawk has an ISO15765/UDS blockset available for purchase separately
J1939		A vehicle bus standard for CAN communication protocol and hardware used for communication and diagnostics among vehicle components. MotoHawk is developing a J1939 library to facilitate the implementation of J1939 communication protocols
OBD	On-Board Diagnostics	A term referring to a vehicle's self-diagnostics and reporting capability. This usually refer to emission requirements and regulations
OBD-FM	On Board Diagnostics Fault Manager	A library of MotoHawk blocks that can be used by an application team to develop software to meet OBD emissions requirements. The use of the OBD-FM blocks in itself does not make a system emission compliant.
SIL	Software in the Loop testing	

MotoHawk Acronyms and Terms (cont'd.)

TERM OR ABBREV.	MEANING	NOTES
Naming Method for ECUs		
ECM-0565-128-1001C	Engine Control Module - Processor Family - Number of Pins - Model Year- Version Number - Type	ECM- Engine Control Module; 0565: Refers to the Microprocessor; 128: Number of pins on the Module; 1001: The model year and revision - so first module of 2010; C: Calibratable (DEV) module
Hardware Related Terms		
ADC	Analog to Digital Converter	Takes a 0-5V voltage and converts to binary to feed into the CPU
AIN	Analog Input	
Bedrock		The ECM-0S12-70, 70 pin module
CAN	Controller Area Network	CAN 2.0B Communication Bus
CAM	CAM input	Signal input from the CAM sensor
CNK	Crank input	Signal input from the Crank Wheel Sensor, can be Vr or Hall effect
CPU	Central Processing Unit	The main microprocessor unit of the module.
DVRG	Driver Ground	
DVRP	Driver Power -	Provides battery power to the actuators through the Main Power Relay
DEV	Development Module	Noted by a 'C' for calibratable in the part name. The modules have additional parallel EEPROM memory for on the fly calibrations
ECM	Engine Control Module	
ECM-OH	Engine Control Module - OH	New module being developed for the OH-6 program
ECU	Electronic Control Unit	
ECUP	ECU Power	Also called Wake or Key - provides the 'wake-up' signal to the microcontroller
EGO	Exhaust Gas Oxygen sensor	
EKP, EKN	Engine Knock (sensor) Positive/Negative	
EST	Electronic Spark Timing Output	Spark Output to drive a smart coil, logic level (0-5V) output
GCM	General Control Module	
HCM	Hydraulic Control Module	
HEGO	Heated Exhaust Gas Oxygen	
HSO	High Side Output	
INJ	Injector	Fuel Injector Output
I/O	Input/output	

MotoHawk Acronyms and Terms (cont'd.)

TERM OR ABBREV.	MEANING	NOTES
Hardware Related Terms (continued)		
LECM	Large Engine Control Module	New module being developed for large engines
LSO	Low Side Output	Connection to DVRP through a transistor - ground is switched for power to flow from DVRP to the LSO output
MPRD	Main Power Relay Driver	Controls the Main Power Relay, MPRD block in MotoHawk provides for a controlled module shutdown
PCM09	Powertrain Control Module 09	The ECM-5554-112, 112-pin module
PCMHD	Powertrain Control Module - Heavy Duty	The 128-pin control module
PROD	Production Module	Noted by an 'F' for flash in the part name. The modules do not have the additional memory for on the fly calibrations.
RTC	Real Time Clock	
SCL+/SCL-	Serial Communications Link	RS232 or RS485 communication channel
SECM-48	Small Engine Control Module	The ECM-0563-048 module used in the OH-4 system
SPD+/SPD-	Speed Input	Input for measuring frequency with Vr or Hall Effect sensor
STOP/ESTOP	Emergency Stop -	when asserted signal disables the main power relay and may also disable engine related outputs such as injection and spark
TPU	Time Processing Unit	Very fast section of the microprocessor used to make angle based calculation for engine position and high resolution outputs
uChi		The GCM-0S12-024-0401 Module
VARCAM	Variable CAM	CAM phasing
XDRP	5V 300mA power source for sensors	

Chapter 7.
Service Options

Product Service Options

If you are experiencing problems with the installation, or unsatisfactory performance of a Woodward product, the following options are available:

- Consult the troubleshooting guide in the manual.
- Contact the manufacturer or packager of your system.
- Contact the Woodward Full Service Distributor serving your area.
- Contact Woodward technical assistance (see "How to Contact Woodward" later in this chapter) and discuss your problem. In many cases, your problem can be resolved over the phone. If not, you can select which course of action to pursue based on the available services listed in this chapter.

OEM and Packager Support: Many Woodward controls and control devices are installed into the equipment system and programmed by an Original Equipment Manufacturer (OEM) or Equipment Packager at their factory. In some cases, the programming is password-protected by the OEM or packager, and they are the best source for product service and support. Warranty service for Woodward products shipped with an equipment system should also be handled through the OEM or Packager. Please review your equipment system documentation for details.

Woodward Business Partner Support: Woodward works with and supports a global network of independent business partners whose mission is to serve the users of Woodward controls, as described here:

- A **Full Service Distributor** has the primary responsibility for sales, service, system integration solutions, technical desk support, and aftermarket marketing of standard Woodward products within a specific geographic area and market segment.
- An **Authorized Independent Service Facility (AISF)** provides authorized service that includes repairs, repair parts, and warranty service on Woodward's behalf. Service (not new unit sales) is an AISF's primary mission.
- A **Recognized Engine Retrofitter (RER)** is an independent company that does retrofits and upgrades on reciprocating gas engines and dual-fuel conversions, and can provide the full line of Woodward systems and components for the retrofits and overhauls, emission compliance upgrades, long term service contracts, emergency repairs, etc.
- A **Recognized Turbine Retrofitter (RTR)** is an independent company that does both steam and gas turbine control retrofits and upgrades globally, and can provide the full line of Woodward systems and components for the retrofits and overhauls, long term service contracts, emergency repairs, etc.

You can locate your nearest Woodward distributor, AISF, RER, or RTR on our website at:

www.woodward.com/directory.aspx

Woodward Factory Servicing Options

The following factory options for servicing Woodward products are available through your local Full-Service Distributor or the OEM or Packager of the equipment system, based on the standard Woodward Product and Service Warranty (5-01-1205) that is in effect at the time the product is originally shipped from Woodward or a service is performed:

- Replacement/Exchange (24-hour service)
- Flat Rate Repair
- Flat Rate Remanufacture

Replacement/Exchange: Replacement/Exchange is a premium program designed for the user who is in need of immediate service. It allows you to request and receive a like-new replacement unit in minimum time (usually within 24 hours of the request), providing a suitable unit is available at the time of the request, thereby minimizing costly downtime. This is a flat-rate program and includes the full standard Woodward product warranty (Woodward Product and Service Warranty 5-01-1205).

This option allows you to call your Full-Service Distributor in the event of an unexpected outage, or in advance of a scheduled outage, to request a replacement control unit. If the unit is available at the time of the call, it can usually be shipped out within 24 hours. You replace your field control unit with the like-new replacement and return the field unit to the Full-Service Distributor.

Charges for the Replacement/Exchange service are based on a flat rate plus shipping expenses. You are invoiced the flat rate replacement/exchange charge plus a core charge at the time the replacement unit is shipped. If the core (field unit) is returned within 60 days, a credit for the core charge will be issued.

Flat Rate Repair: Flat Rate Repair is available for the majority of standard products in the field. This program offers you repair service for your products with the advantage of knowing in advance what the cost will be. All repair work carries the standard Woodward service warranty (Woodward Product and Service Warranty 5-01-1205) on replaced parts and labor.

Flat Rate Remanufacture: Flat Rate Remanufacture is very similar to the Flat Rate Repair option with the exception that the unit will be returned to you in “like-new” condition and carry with it the full standard Woodward product warranty (Woodward Product and Service Warranty 5-01-1205). This option is applicable to mechanical products only.

Returning Equipment for Repair

If a control (or any part of an electronic control) is to be returned for repair, please contact your Full-Service Distributor in advance to obtain Return Authorization and shipping instructions.

When shipping the item(s), attach a tag with the following information:

- return authorization number;
- name and location where the control is installed;
- name and phone number of contact person;
- complete Woodward part number(s) and serial number(s);
- description of the problem;
- instructions describing the desired type of repair.

Packing a Control

Use the following materials when returning a complete control:

- protective caps on any connectors;
- antistatic protective bags on all electronic modules;
- packing materials that will not damage the surface of the unit;
- at least 100 mm (4 inches) of tightly packed, industry-approved packing material;
- a packing carton with double walls;
- a strong tape around the outside of the carton for increased strength.

NOTICE

To prevent damage to electronic components caused by improper handling, read and observe the precautions in Woodward manual 82715, *Guide for Handling and Protection of Electronic Controls, Printed Circuit Boards, and Modules*.

Replacement Parts

When ordering replacement parts for controls, include the following information:

- the part number(s) (XXXX-XXXX) that is on the enclosure nameplate;
- the unit serial number, which is also on the nameplate.

Engineering Services

Woodward offers various Engineering Services for our products. For these services, you can contact us by telephone, by email, or through the Woodward website.

- Technical Support
- Product Training
- Field Service

Technical Support is available from your equipment system supplier, your local Full-Service Distributor, or from many of Woodward's worldwide locations, depending upon the product and application. This service can assist you with technical questions or problem solving during the normal business hours of the Woodward location you contact. Emergency assistance is also available during non-business hours by phoning Woodward and stating the urgency of your problem.

Product Training is available as standard classes at many of our worldwide locations. We also offer customized classes, which can be tailored to your needs and can be held at one of our locations or at your site. This training, conducted by experienced personnel, will assure that you will be able to maintain system reliability and availability.

Field Service engineering on-site support is available, depending on the product and location, from many of our worldwide locations or from one of our Full-Service Distributors. The field engineers are experienced both on Woodward products as well as on much of the non-Woodward equipment with which our products interface.

For information on these services, please contact us via telephone, email us, or use our website: **www.woodward.com**.

How to Contact Woodward

For assistance, call one of the following Woodward facilities to obtain the address and phone number of the facility nearest your location where you will be able to get information and service.

Electrical Power Systems

Facility	Phone Number
Brazil	+55 (19) 3708 4800
China	+86 (512) 6762 6727
Germany	+49 (0) 21 52 14 51
India	+91 (129) 4097100
Japan	+81 (43) 213-2191
Korea	+82 (51) 636-7080
Poland	+48 12 295 13 00
United States	+1 (970) 482-5811

Engine Systems

Facility	Phone Number
Brazil	+55 (19) 3708 4800
China	+86 (512) 6762 6727
Germany	+49 (711) 78954-510
India	+91 (129) 4097100
Japan	+81 (43) 213-2191
Korea	+82 (51) 636-7080
The Netherlands	+31 (23) 5661111
United States	+1 (970) 482-5811

Turbine Systems

Facility	Phone Number
Brazil	+55 (19) 3708 4800
China	+86 (512) 6762 6727
India	+91 (129) 4097100
Japan	+81 (43) 213-2191
Korea	+82 (51) 636-7080
The Netherlands	+31 (23) 5661111
Poland	+48 12 295 13 00
United States	+1 (970) 482-5811

You can also locate your nearest Woodward distributor or service facility on our website at:

www.woodward.com/directory.aspx

Technical Assistance

If you need to telephone for technical assistance, you will need to provide the following information. Please write it down here before phoning:

Your Name	
Site Location	
Phone Number	
Fax Number	
Engine/Turbine Model Number	
Manufacturer	
Number of Cylinders (if applicable)	
Type of Fuel (gas, gaseous, steam, etc)	
Rating	
Application	
Control/Governor #1	
Woodward Part Number & Rev. Letter	
Control Description or Governor Type	
Serial Number	
Control/Governor #2	
Woodward Part Number & Rev. Letter	
Control Description or Governor Type	
Serial Number	
Control/Governor #3	
Woodward Part Number & Rev. Letter	
Control Description or Governor Type	
Serial Number	

If you have an electronic or programmable control, please have the adjustment setting positions or the menu settings written down and with you at the time of the call.

Revision History

Changes in Revision A—

- Convert data from MotoHawk brochure format to Woodward technical manual format

We appreciate your comments about the content of our publications.

Send comments to: icinfo@woodward.com

Please reference publication 36333.

PO Box 1519, Fort Collins CO 80522-1519, USA
1000 East Drake Road, Fort Collins CO 80525, USA
Phone +1 (970) 482-5811 • Fax +1 (970) 498-3058

Email and Website—www.woodward.com

**Woodward has company-owned plants, subsidiaries, and branches,
as well as authorized distributors and other authorized service and sales facilities throughout the world.**

Complete address / phone / fax / email information for all locations is available on our website.

2012/5/Skokie