



# MOTOHAWK

# RESOURCE GUIDE



# MotoHawk

## TRAINING SUPPLEMENT

Agenda	2	Vardec Parameters	9
Software Requirements	3	Calibration Management	12
Training Project System Overview	4	Analog Input	14
Training Project Block Diagram	5	PWM Output	15
Controller I/O Acronyms	6	Fault Management	16
Controller Hardware Layout	8		

### CONTACT INFORMATION

#### Let us know how we can help!

To contact us by telephone, please call 877.234.1410

To request a quote, please contact [quotes@neweagle.net](mailto:quotes@neweagle.net)

For New Eagle orders, please contact [orders@neweagle.net](mailto:orders@neweagle.net)

#### Product and tools support

<http://www.neweagle.net/support/wiki/>

[support@neweagle.net](mailto:support@neweagle.net)



## Basic MotoHawk Training Agenda

	DAY 1	DAY 2	DAY 3
Morning	Introduction Software installation Basics of MATLAB/Simulink & MotoHawk Simple Simulink model First build, kit setup, and flash Basics of MotoTune	Fault management Throttle project: Faults Throttle project: PID control	Throttle project: CAN Libraries Components
Afternoon	Triggers I/O Calibrations, probes, & overrides Throttle project: I/O	Data storage Throttle project: Data storage CAN	New Eagle hardware & software offerings Questions Evaluation

# Software Requirements



Welcome to MotoHawk training! Presented by New Eagle, this three day course will train you to create a real-world application with Simulink and MotoHawk, program a module, and calibrate in real-time using MotoTune. But first things first, let's make sure you have the correct software on your laptop.

The training requires several software installations involving a somewhat complex compatibility maze. The following lists the software requirements and any relevant compatibility notes.

**Please install software in the order listed PRIOR to training.**



## Windows

If Windows installation is 64bit, MotoHawk 2010aSP0 or later is required



## MathWorks

Required installations\*

- MATLAB\*\*
- Simulink
- Real Time Workshop
- Real Time Workshop Embedded Coder
- Stateflow (optional, but highly recommended)
- Stateflow Coder (optional, but highly recommended)

\*Required before training (MathWorks distributes trial downloads and licenses)

\*\*If MATLAB installation is 64bit, MotoHawk 2010bSP0 or later is required

With the installations above complete, you're system is prepared to work with the MotoHawk Tool Suite.

The necessary software for the MotoHawk tools and a temporary license will be provided at training.

The following software will be installed DURING training.



## MotoHawk

MATLAB Version	MotoHawk Version						
	0.8.3	2008a	2008b	2009a	2009b	2010a	2010b
6.5.1	Y	Y	Y	N	N	N	N
7.0	Y	Y	Y	N	N	N	N
7.0.1	Y	Y	Y	N	N	N	N
7.0.4	Y	Y	Y	N	N	N	N
7.1	Y	Y	Y	N	N	N	N
7.2 (R2006a)	Y	Y	Y	N	N	N	N
7.3 (R2006b)	Y	Y	Y	N	N	N	N
7.4 (R2007a)	Y	Y	Y	Y	N	N	N
7.5 (R2007b)	Y	Y	Y	Y	Y	Y	N
7.6 (R2008a)	N	Y	Y	Y	Y	Y	N
7.7 (R2008b)	N	Y	Y	Y	Y	Y	Y (A)
7.8 (R2009a)	N	N	Y	Y	Y	Y	Y (A)
7.9 (R2009b)	N	N	N	Y	Y	Y	Y (A)
7.9.1 (R2009bSP1)	N	N	N	N	Y (B)	Y (B)	Y (B)
7.10 (R2010a)	N	N	N	N	Y (B)	Y	Y (A)
7.11 (R2010b)	N	N	N	N	N	Y (B)	Y (B)

A -- in current beta release, B -- planned in future beta or service pack



## MotoServer/MotoTune\*

\*With MotoHawk 2010aBeta6 and later, MotoServer/MotoTune 8.13.7.120 or later is required

\*With MotoHawk 2009bBeta1 and later, MotoServer/MotoTune 8.13.7.87 or later is required



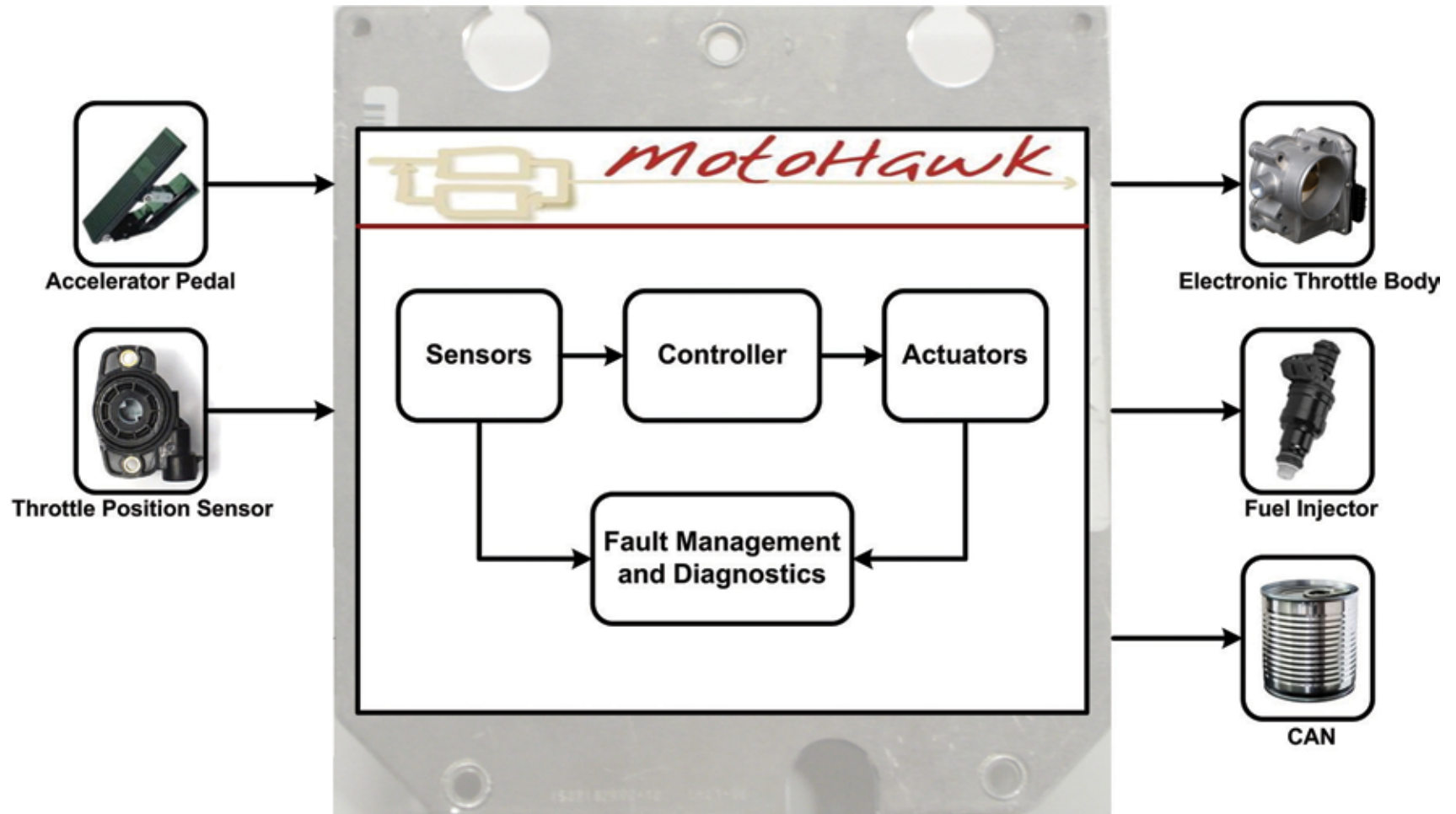
## GCC (compiler)\*

\*MotoHawk 2009bBeta2 or later is required

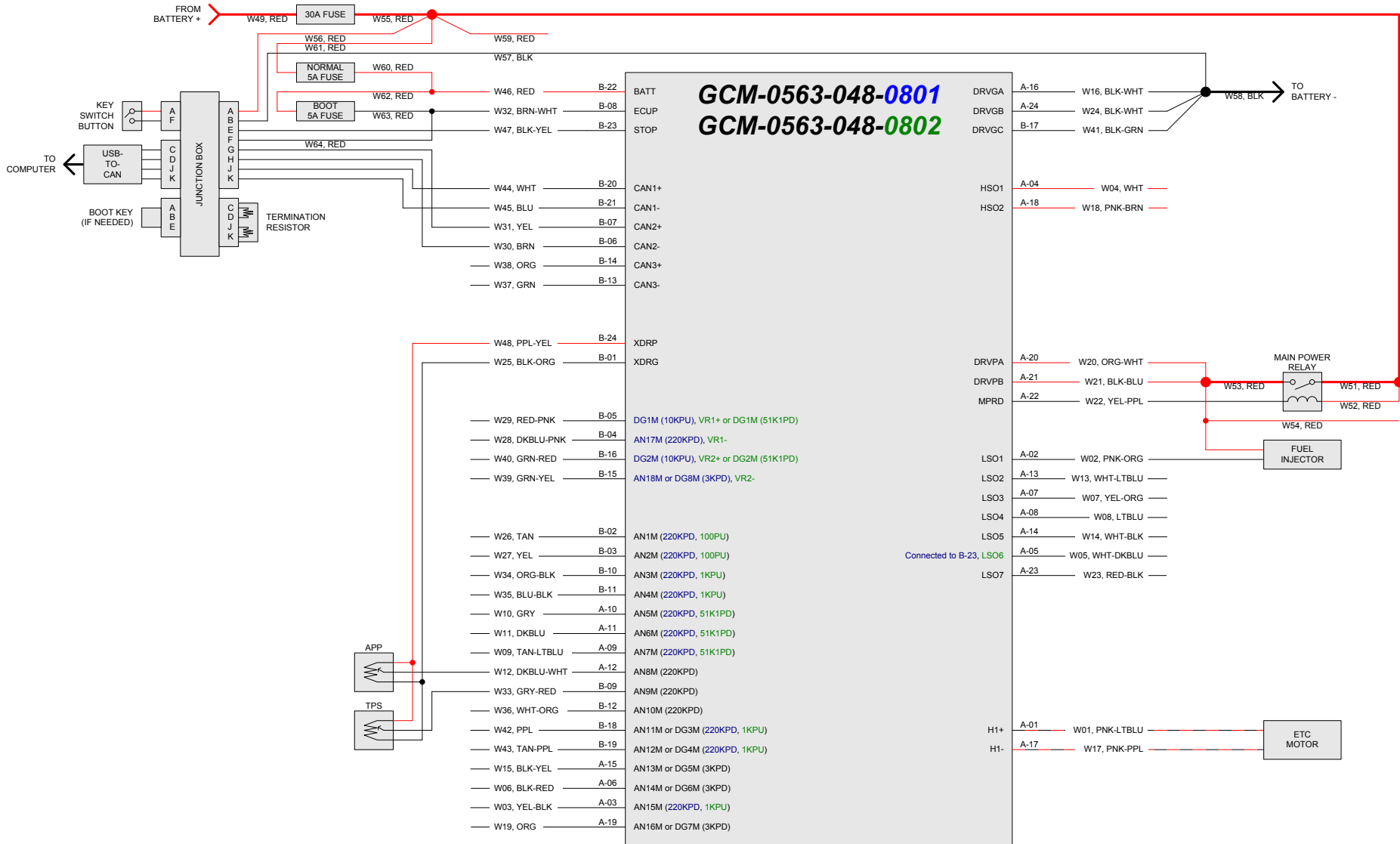


## CANKing

# Training Project System Overview



# Training Project Block Diagram



# Controller I/O Acronyms



GCM-0563-048-0801 GCM-0563-048-0802			
Acronym :	...Literally Means :	Other Notations :	General Notes :
<b>BATT</b>	<b>BATT</b> ery		~180mA at 13.8V (module only, no external loads); connected directly to battery+; low-current power to analog/digital core of module; allows controlled shutdown
<b>ECUP</b>	<b>ECU</b> Power	KEYSW, WAKE	~5mA at 13.8V; derived (through switch) from battery+; "wake-up" signal to module to initiate execution of software algorithm (when power removed, operations continue until software commands a shutdown)
<b>STOP</b>	Emergency <b>STOP</b>	ESTOP	when asserted, disables the main power relay through hardware (on other modules, may disable engine-related outputs such as EST and FUELP); input for boot mode signal
<b>DRVPx</b>	<b>DRiVer</b> Power	DRVPWRx	through MPR, provides battery+ to actuators; internally provides power to H-bridges (allows controlled shutdown on modules without BATT)
<b>DRVGx</b>	<b>DRiVer</b> <b>G</b> round	GNDx, PWRGRNDx	connected directly to battery-
<b>XDRPx</b>	<b>TRANSD</b> uce <b>R</b> Power	XDRPWRx	300mA maximum; 5V reference for sensors
<b>XDRGx</b>	<b>TRANSD</b> uce <b>R</b> <b>G</b> round	XDRGNDx	internal connection to DRVG; low reference for sensors
<b>ANxM</b>	<b>AN</b> alog sensor input <b>M</b> onitor	ANx	has a pull-up and/or pull-down internal resistor; time constant through an internal capacitor and additional internal resistor
<b>DGxM</b>	<b>Di</b> gital sensor input <b>M</b> onitor	DGx	same as analog sensor input monitor, but may also resolve frequency
<b>VRx+, VRx-</b>	<b>V</b> ariable <b>R</b> eluctance sensor input		typically used to resolve frequency (on other modules, used to resolve engine crankshaft position)
<b>MPRD</b>	<b>M</b> ain <b>P</b> ower <b>R</b> elay <b>D</b> river		controls the main power relay
<b>LSOx</b>	<b>L</b> ow- <b>S</b> ide driver <b>O</b> utput	LSDx	connection to DRVG through transistor; PWM capable
<b>HSOx</b>	<b>H</b> igh- <b>S</b> ide <b>O</b> utput	HSDx	connection to DRVP through power switch; PWM capable
<b>Hx+, Hx-</b>	<b>H</b> - <b>B</b> ridge output	HBxA, HBxB, ETCx, HBRIDGExA, HBRIDGExB	on some modules, can also be operated independently as low-side or high-side driver outputs
<b>CANx+, CANx-</b>	<b>C</b> ontroller <b>A</b> rea <b>N</b> etwork communication		CAN 2.0B protocol

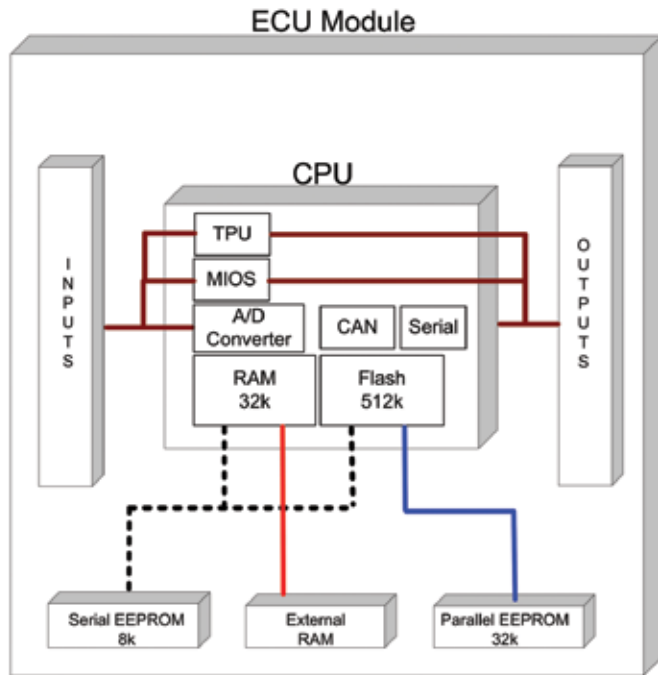
# Controller I/O Acronyms (continued)



Common Acronyms Found On Other Modules:			
Acronym :	...Literally Means :	Other Notations :	General Notes :
<b>GNDREF</b>	<b>GrouND REFerence</b>		ground reference signal
<b>CNK+, CNK-</b>	<b>CraNK</b> shaft encoder sensor input	CNKVR+, CNKVR-, CNKDG	typically used to resolve engine crankshaft position (with variable reluctance and/or digital sensor inputs)
<b>CAM+, CAM-</b>	<b>CAM</b> shaft encoder sensor input	CAMDG	typically used to resolve engine camshaft position (with variable reluctance and/or digital sensor inputs)
<b>SPDx, SPD-</b>	<b>SPeeD</b> (frequency) sensor input	SPEEDx	typically used to resolve frequency (with variable reluctance and/or digital sensor inputs)
<b>EKxP, EKxN</b>	<b>Engine K</b> nock sensor input	KNKx+, KNKx-	used with wide-range piezoelectric knock sensors
<b>EGOxP, EGOxN</b>	<b>Exhaust Gas O</b> xxygen sensor input		dual differential amplifier targeted at lambda oxygen sensor signal processing
<b>HEGOx</b>	<b>Heated Exhaust Gas O</b> xxygen sensor input	O2x+, O2x-	used with switching type oxygen sensor (heated or unheated)
<b>LSUxUN, LSUxIA, LSUxIP, LSUxVM</b>	<b>Lambda S</b> ensing <b>U</b> nit sensor input		used with the Bosch CJ125 exhaust gas oxygen sensor
<b>INJx</b>	fuel <b>INJ</b> ector driver output	FINJx, Flx	low-side output to drive high-impedance fuel injector
<b>ESTx</b>	<b>Electronic Spark T</b> iming output		5V signal to drive logic level (“smart”) ignition coil; on some modules, may be used as additional analog inputs
<b>EST RTN</b>	<b>EST ReTurN</b>		reference level for logic level (“smart”) ignition coils
<b>FUELPR</b>	<b>FUEL P</b> ump <b>R</b> elay	FUELP	low-side output to drive fuel pump relay
<b>BATT_OUT</b>	<b>BATT</b> ery <b>OUT</b>		supply voltage for external input devices
<b>TACH_LINK</b>	<b>TACH</b> ometer output or <b>LINK</b> interface		tachometer output or LINK serial interface
<b>SCL+, SCL-</b>	<b>S</b> erial <b>C</b> ommunication <b>L</b> ink	RS-485A, RS-485B	RS485 serial communication link
<b>ISO9141K, ISO9141L</b>	<b>ISO9141</b> communication link		ISO9141 communication link



# Controller Hardware Layout



## Component Descriptions

<b>Inputs</b>	Analog, discrete, frequency, crank/cam and corresponding resource circuitry
<b>A/D Converter</b>	For analog inputs, converts voltage into ADC's
<b>MIOS</b>	Modular Input/Output System, asynchronous (non-angle-based) operations only
<b>TPU</b>	Time Processing Unit, synchronous (angle-based) or asynchronous operations
<b>CAN</b>	Controller Area Network communication
<b>Serial</b>	RS485 serial communication (RS232 on some modules)
<b>RAM</b>	Random Access Memory, volatile variable, stack, and heap storage
<b>Flash</b>	Constant data and executable code
<b>Serial EEPROM</b>	Electrically-Erasable Programmable Read-Only Memory; nonvolatile variable storage
<b>External RAM</b>	Select modules only; same as RAM above (but slower)
<b>Parallel EEPROM</b>	constant variable storage, calibration in development units only
<b>Outputs</b>	Circuitry to drive the following outputs: discrete, PWM, engine-specific (fuel injector, EST), H-bridge, low-side output, high-side output

# Vardec Parameters



## Vardec – a global variable declaration.

Vardec's are unique by name and are created by numerous MotoHawk blocks, such as calibrations, probes, look-up tables, and data definitions. Below are common mask parameters found with MotoHawk vardec blocks.

Note that not all of the mask parameters below are available for each vardec.

**Name** — The name of the vardec.

Must be unique among all other vardec's in a MotoHawk model.

**Initial Value** — The initial value of the variable. The dimensions of the initial value set the size of the variable. For instance, if the initial value is set as zeros(5), the variable is a 5x5 matrix. If the initial value is set as [0 1 2 3], the variable is a vector of length 4. Or, if the initial value is set as 1/2, the variable is a scalar.

**Behavior** — The display and storage behavior of the variable.

**Storage Class** — The storage class of the variable.

**Output Data Reference** — Option to output a reference signal pointing to the variable. For instance, this outport signal could be connected to a MotoHawk Data Read block to expose that variable for use downstream in the application.

**MotoTune Window** — Option to set the variable as a calibration (i.e., appear in a .cal file) or a display (i.e., appear in a .dis file).

**Name Source** — Option to define the vardec name by the Parameter (using Name mask parameter above) or Output Wire Name (using the name of the output wire).

**Data Source** — Option on how to define the data source. Lookup By Name uses the Data Name mask parameter, Input Reference Signal provides a reference inport, Lookup By Name In Structure uses the Data Name mask parameter in conjunction with the name of the structure, and Lookup By Name In Structure Via Reference uses a reference inport in conjunction with the name of the structure.

**Data Structure** — The dimensional structure of the variable.

If Vector or Matrix is selected, additional options are provided to Read/write scalar from location by index (zero-based) or Read entire data structure at once; if the latter is selected, the dimensions of the vector/matrix must be explicitly specified.

Behavior	Description
Calibration	On a development module, stored in parallel EEPROM. On a production module, stored as a constant in flash (can only be changed through reflash). Viewed as calibration in MotoTune.
Display	Stored in RAM. Viewed as display variable in MotoTune.
Calibration NV	Stored in serial EEPROM (shadowed in RAM). Viewed as calibration in MotoTune.
Display NV	Stored in serial EEPROM (shadowed in RAM). Viewed as display variable in MotoTune.

Storage Class	Description
Constant	On a development module, stored in parallel EEPROM. On a production module, stored as a constant in flash (only can be changed through reflash).
Volatile	Stored in RAM.
NonVolatile	Stored in serial EEPROM (shadowed in RAM).
Fixed NonVolatile	Stored in serial EEPROM (shadowed in RAM) (attempts to maintain over programming cycle).



# Vardec Parameters (continued)



**Output Data Type** — The data type of the signal originating from the output port. The Inherit from ‘Default Value’ option evaluates the data type of the default value to set the block’s data type (e.g., the data type can be explicitly specified as uint8 by entering the default value as uint8(100)).

The Inherit via back propagation sets the data type to that governed (where applicable) by other connected Simulink/MotoHawk blocks. Otherwise, the data type can be explicitly specified as an integer or floating point data type.

**Read and Write Access Level** — Option to set security level on read/write access from MotoTune (1 is the lowest, 4 is the highest security level). The user’s security level is the minimum of the MotoServer communication port setting and the license dongle setting. A security level of 4 is required to create a new MotoTune online calibration; however, any security level will allow the creating of a new MotoTune online display.

After opening (previously created) MotoTune calibrations/displays, individual calibration/display variables will allow read/write access if the security level set in the MotoHawk block is equal to or less than the user’s security level.

*Note that, for variables with both read and write access, the read access level must be equal to or less than the write access level.*

Use uploaded calibration values from MotoTune  
Option to use or ignore values input from MotoTune.

**View Value As** — Option as to how the engineering value is displayed in MotoTune. The Number option displays the engineering value numerically. The Enumeration option displays an associated enumeration; for example, if the Enumeration (Cell String, or Struct) is {‘State 1’, ‘State 2’, ‘State 3’} and the numeric engineering value is 1, then “State 2” will be displayed in MotoTune.

*Note that this option is only applicable with Boolean or integer data types and that the enumeration definition indices are zero-based.*

The Text option displays the ASCII character interpretation of the value; for example, if the engineering value is a vector with values [35 63 106], MotoTune will display “#?j”.

*This option may only be used with a uint8 data type.*

**Show Vectors As** — Option to display (in MotoTune) the vector as a Wide Row or a Tall Column.

*This is only applicable if the variable is a vector.*

Data Type	Size (bytes)	Min.	Max.	Resolution
double	8	-inf	inf	depends on magnitude
single	4	-inf	inf	depends on magnitude
int8	1	-128	127	1
uint8	1	0	254	1
int16	2	-32768	32767	1
uint16	2	0	65535	1
int32	4	-2147483648	2147483647	1
uint32	4	0	4294967295	1
boolean	1	0	1	1
reference	2 for S12, 4 for other	n/a	n/a	n/a
struct	inherits data types from fields in structure declaration			
struct container	dependent on number of structure instances associated with container; inherits data types from fields in structure declaration			
struct reference	2 for S12, 4 for other	n/a	n/a	n/a

# Vardec Parameters (continued)



**Help Text** — A description of the variable to be displayed in MotoTune. Enclose the help text between single quotation marks.

**Units** — The engineering units of the variable to be displayed in MotoTune. Enclose the units between single quotation marks. For example, the help text and units for a group of calibrations may appear in MotoTune as:

Name	Value	Units	Help
Calibration1	1.10	Units1	This is help text for the first calibration
Calibration2	2.20	Units2	This is help text for the second calibration
Calibration3	3.30	Units3	This is help text for the third calibration

**Row Header Enumeration (Cell String, or Struct)** — Headings for the rows of the variable to be displayed in MotoTune. A row header is only applicable if the variable is a column vector or a matrix (i.e., the heading is for the rows). *Specify the headings in a cell string format.*

**Column Header Enumeration (Cell String, or Struct)** — Headings for the columns of the variable to be displayed in MotoTune. A column header is only applicable if the variable is a row vector or a matrix (i.e., the heading is for the columns). *Specify the headings in a cell string format.*

For example, if the row header enumeration is specified as {'Row 1', 'Row 2', 'Row 3'} and the column header enumeration is specified as {'Col 1', 'Col 2', 'Col 3'}, the variable will be seen in MotoTune as the following:

	Col 1	Col 2	Col 3
Row 1	1	1	1
Row 2	1	1	1
Row 3	1	1	1

**Minimum Value** — The minimum allowable value for the variable. This will prevent a user from entering an out-of-range value from MotoTune.

*This value is in engineering units (i.e., after the gain, offset, and exponent have been applied).*

**Maximum Value** — The maximum allowable value for the variable. This will prevent a user from entering an out-of-range value from MotoTune.

*This value is in engineering units (i.e., after the gain, offset, and exponent have been applied).*

**Precision** — The precision of the variable as displayed in MotoTune. Enclose the precision between single quotation marks using the syntax '0.DecimalPlaces'. For instance, if the variable has a value of 98.76543 and the precision is '0.3', the value would appear in MotoTune as 98.765. The default precision of '' will display 2 decimal places.

**Gain** — The gain applied to the raw value to calculate the engineering value as observed in MotoTune.

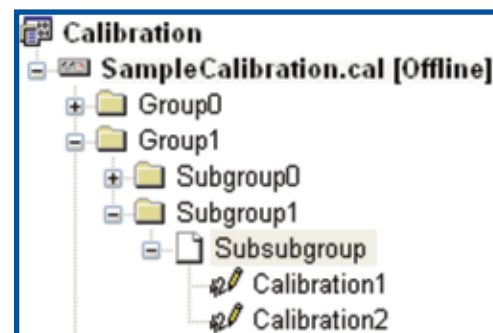
**Offset** — The offset applied to the raw value to calculate the engineering value as observed in MotoTune.

**Exponent** — The exponent applied to the raw value to calculate the engineering value as observed in MotoTune. *The MotoTune gain, offset, and exponent are applied according to the following:*

$$(\text{engineering value as displayed in MotoTune}) = (\text{Gain} * (\text{raw value}) )^{\text{Exponent}} + \text{Offset}$$

**MotoTune Group String** — The folder name and hierarchy that contains the variable in MotoTune. Enclose the MotoTune group string between single quotation marks, and use the | character to delineate subfolder structure.

For example, if the MotoTune group string for Calibration1 is specified as 'Group1 | Subgroup1 | Subsubgroup', the variable will be found in MotoTune in the following:



# Calibration Management



## Calibration Management

There are several methods of calibration management within the MotoHawk and MotoTune software.

### A. Maintain the calibrations in the model. (Optional)

The engineer inserts correct calibrations into the default values of the vardec blocks. Benefits of this approach are:

- the engineer can easily run the model in simulation and perform software-in-the-loop development.
- maintaining the calibrations in the model means the application is ready to run with simply a model build; no additional steps are necessary.

The downsides to this approach are:

- online calibration changes need to be manually and tediously transferred into the MotoHawk model.
- this approach neglects the useful calibration management tools that MotoTune provides.

### B. Maintain the calibrations in a .cal file. (Recommended)

The proper and recommended approach is to maintain the calibrations in a .cal calibration file. MotoTune offers several tools for calibration management. Although there are several different procedures that can be used to update calibrations from one build to another, the recommended approach is to merge the desired calibration values into the new .srz build. Then, upon programming, the application begins execution immediately with the correct calibration values.

This merge process is described below:

#### 1. Build the new software.

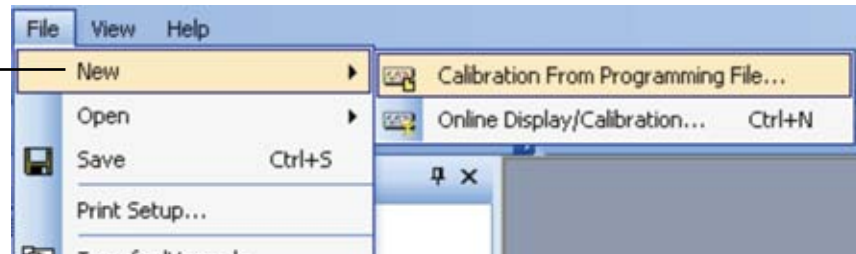
From the MotoHawk model, press CTRL-B to build the new software.  
The result is a New.srz and New.dll.

#### 2. Create a calibration file.

In MotoTune, select File/New/Calibration From Programming File. Select the latest build New.srz, and save as New\_000.cal.  
This calibration file is created offline and contains the default values from the model, as it was created from the .srz build file.

Close the .cal file for the next step.

MotoHawk/MotoTune Files	
<b>.mdl</b>	<b>The Simulink model file containing the MotoHawk application.</b> This typically exists in a project directory, which also has other MATLAB files to complement the application, such as function files, .m files, library files, etc.
<b>.srz</b>	<b>The compiled executable file.</b> This is the file created during a model build (CTRL-B) and programmed onto the module. By default, saved in C:\ECUFiles\Programs.
<b>.dll</b>	<b>A dynamic link library file also created during the model build (CTRL-B).</b> In summary, a vardec memory mapping; the correct .dll file is needed to view a calibration or display file. By default, saved in C:\ECUFiles\TDBDLL.
<b>.cal</b>	<b>A calibration file created in MotoTune.</b> Contains calibration values, help descriptions, units, etc. By default, saved in C:\ECUFiles\Cals.
<b>.dis</b>	<b>A display file created in MotoTune.</b> Contains an Excel-like display layout, help descriptions, units, etc. By default, saved in C:\ECUFiles\Displays.
<b>.log</b>	<b>A log file created in MotoTune.</b> Contains logged data with date, time, and value names. By default, saved in C:\ECUFiles\Logging.



# Calibration Management (continued)



## 3. Transfer/upgrade calibrations.

The Transfer/Upgrade tool transfers calibration values from an old .cal file to a new .cal file.

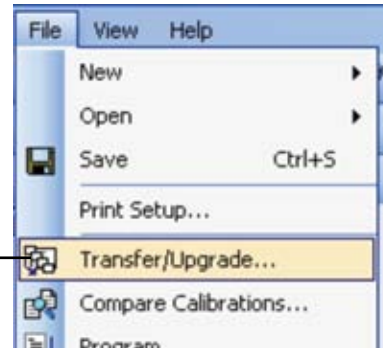
In MotoTune, select File/Transfer/Upgrade, or press the Transfer/Upgrade icon.

For Source, select the file which contains your desired calibrations (e.g., Old.cal).

For Target, select New\_000.cal.

Press Start and take note of any differences outlined in the report.

Then, save the new calibration file, overwriting as New\_000.cal (or select a new descriptive name).



## 4. Merge into the new build file.

Open the New\_000.cal file offline.

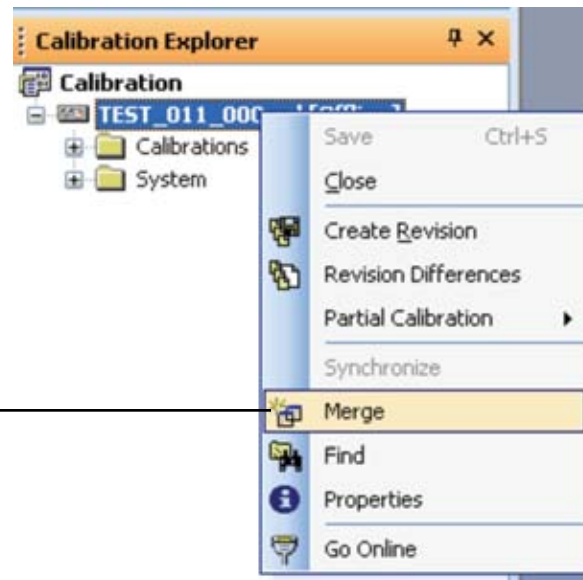
Right click, and select Merge. Then select the New.srz.

*Note the differences from the default calibration values.*

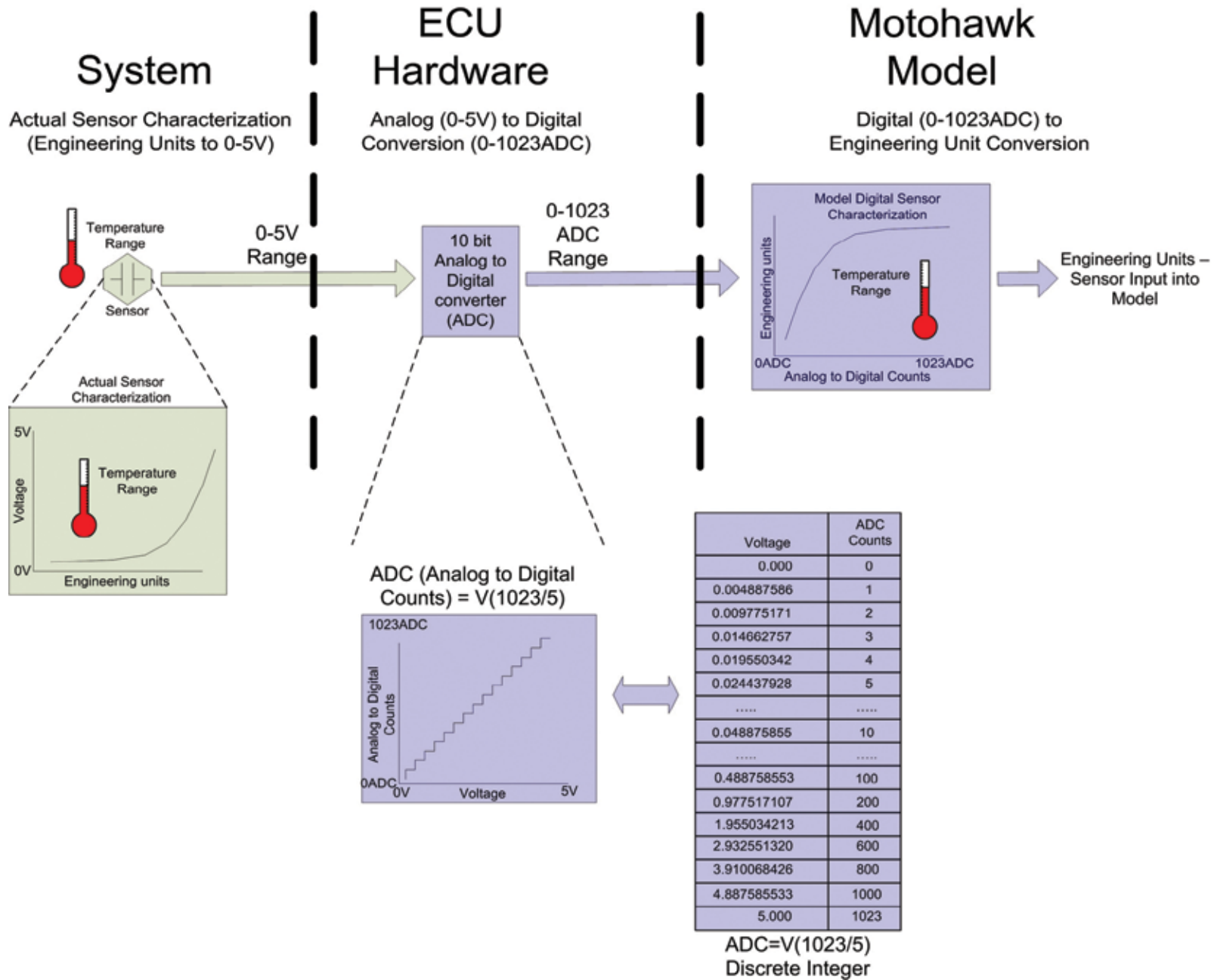
Change the name to indicate a merged .srz (e.g., NewMrg000.srz), and save.

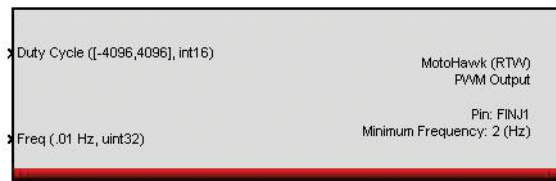
## 5. Program the module.

Program the module with NewMrg000.srz, which has the newest software functionality with the proper calibration values.



# Analog Input





**PWM Block**

## PWM stands for Pulse Width Modulation

A PWM signal is a square signal that has 3 defining properties:

Amplitude (A), in Volts. Amplitude is set by the voltage source (e.g. DRVP), which is typically fairly constant.

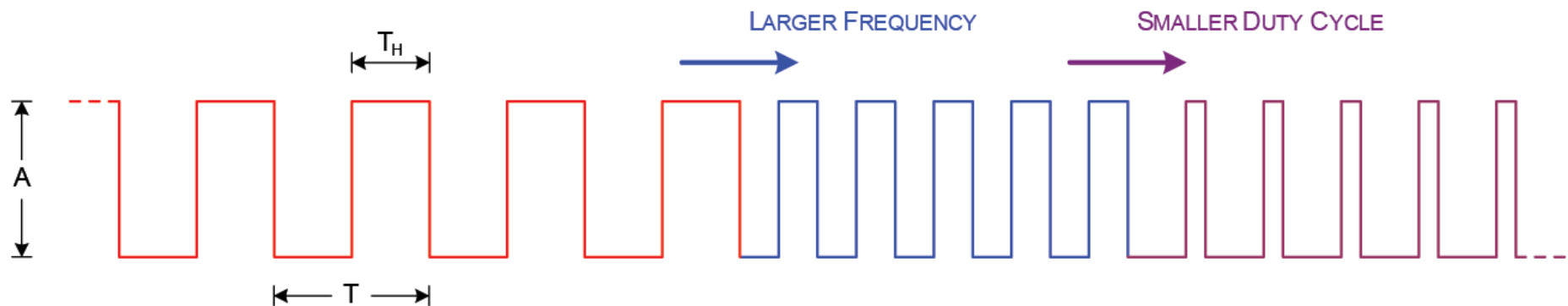
Frequency ( $f = 1/T$ ), in Hertz.

Duty cycle ( $DC = T_H / T$ ), in %. The duty cycle is the percentage of the signal that is non-zero (when analyzed over 1 cycle.)

When used to control an output, the combined electrical and mechanical response of the actuator results in an effective average voltage (this is a more efficient method of controlling power than with resistive methods).

The PWM frequency is matched to the actuator to minimize oscillations.

A PWM-driven output can also be used in conjunction with a low-pass electrical filter to produce an analog voltage, where the duty cycle is proportional to the voltage.





# Fault Management



		Fault x/y setting																	
		3/5																	
		cycle number ->	1	2	3	4	5	1	2	3	4	5	key cycle	1	2	3	4	5	
fault behavior	Fault input ->	0	1	1	1	0	0	0	0	0	1	1			0	1	0	0	0
enabled	Fault status ->	...	S	S	A	A	A	A	A	A	0	0			...	S	...	...	...
sticky	Fault status ->	...	S	S	A	A	A	A	A	A	A	A			...	S	...	...	...
save-occurred	Fault status ->	...	S	S	A	A	A	A	A	A	0	0			0	S	0	0	0
sticky-persistent	Fault status ->	...	S	S	A	A	A	A	A	A	A	A			A	A	A	A	A

**S = Suspected**

**A = Acted**

**0 = Occurred**



## CHAPTER 1 : Intro to MotoHawk

About MotoHawk	1
ECM565-128 Developer's Kit	2
System Requirements	3
MATLAB™ Installation Procedure	3
Green Hills Software	4
Obtaining A License For Your MotoHawk Compiler	4
MotoHawk™ Installation Procedure	5
Creating an application in MATLAB™	6
Building Your Application	7
Assembling Your Kit	8
Starting MotoTune	8
Checking MotoServer	8
Programming the Module	8
The Program ECU status pop up appears	9
Creating A Display	9
Checking Operation	10
First Application	10
Generating Embedded Code	10
Introducing a Gain Stage	14
MotoHawk Data Storage Blocks	15
MotoTune Options	16
Calibration and Probing Blocks	17
Gathering data	17
Throttle Control Challenge	21
Pin Number & Signal Name	21
Fault Detection on Throttle Pedal	23



### ... About MotoHawk

MotoHawk makes it possible to run a Simulink model on a Woodward module.

MotoHawk 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.

MotoHawk extends Simulink and Real-Time Workshop Embedded Coder to generate code necessary to interface with the resources of the modules and control their behavior.

**The goal of MotoHawk is to let the user concentrate on solving the control problem rather than solving the programming problem.** Programming an embedded module is notoriously difficult both in terms of coding as well as actually transporting the application into the module during reprogramming. MotoHawk addresses all of this to make the stuff that should be easy actually easy. Unfortunately, the difficult part of conjuring up the magic to control your engine or vehicle is still complex. MotoHawk just makes it simpler to implement the magic.

## ECM565-128 Developer's Kit

1. ECM565-128 Development Module
2. ECM565-128 MotoHawk™ Harness w/Main Power Relay and fuse
3. Power Switch Asm. 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
16. MotoHawk™ Resource Guide (this manual)

\*Green Hills Software, Security Dongle programming, and applications included on Software CD are subject to your specific order and may not be included in this shipment.



## System Requirements

1. Windows XP (any SP,) Windows 2000 (SP3 or SP4)  
Windows NT (SP5 or SP6a)
2. Pentium III or IV, Xeon, Pentium M, AMD Athlon,  
Athlon XP, Athlon MP
3. 345 MB disk space
4. 512 MB RAM (1 GB or more recommended)
5. 16, 24, or 32 bit OpenGL capable graphics adapter  
(strongly recommended)
6. Microsoft Windows supported graphics accelerator card,  
printer, and sound card
7. 1400x1050 display (min)  
(1600x1200 strongly recommended)

## MATLAB™ Installation Procedure

Insert CD in drive. If the installer does not start automatically, click Start/Run and double click on Autorun.exe. 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 arriving for training.*

### Install all of the following:

MATLAB \_\_\_\_\_  
\_\_\_\_\_ Simulink  
\_\_\_\_\_ Real Time Workshop  
\_\_\_\_\_ Realtime Workshop Embedded Coder

### It is strongly recommended that you also install:

Stateflow \_\_\_\_\_  
\_\_\_\_\_ Stateflow Coder



## Green Hills Software

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.

Select Programs/MULTI2000,PowerPC v3.6/Licensing/License Request Generator.

Select OK at the following screen.

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

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

Initially, you will want to request an evaluation license — this will get you up-and-running quickly.

Select Next.

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

You must accept License Agreement in order to use the compiler.

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.)

Follow the instructions that accompany the license file.

A hard copy of the License Agreement was included with your SDK.

FAX a signed copy to (805)965-6343, Attn: Mickey Neal.

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



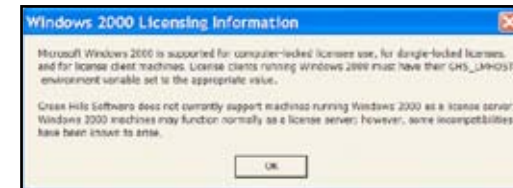
**Green Hills Software License Request Generator**

To request a license, provide the information below and press Next. Press Skip if you do not want to request a license at this time.

Name: \_\_\_\_\_ Address 1: \_\_\_\_\_  
 e-mail: \_\_\_\_\_ Address 2: \_\_\_\_\_  
 Position: \_\_\_\_\_ City: \_\_\_\_\_  
 Company Name: \_\_\_\_\_ State/Province: \_\_\_\_\_  
 Telephone: \_\_\_\_\_ Postal Code: \_\_\_\_\_  
 Fax: \_\_\_\_\_ Country: USA

GHS User ID (if known): \_\_\_\_\_

Next > Skip



**Windows 2000 Licensing Information**

Microsoft Windows 2000 is supported for computer-locked licenses use, for single-locked licenses, and for license client machines. License clients running Windows 2000 must have their GHS\_LICENSE environment variable set to the appropriate value.

Green Hills Software does not currently support machines running Windows 2000 as a license server. Windows 2000 machines may function normally as a license server; however, some incompatibilities have been known to arise.

OK



**Green Hills Software License Request Generator**

**License Information**

Number of licenses: 1

License Availability:

- Computer-locked: license will be locked to this computer.
- Single-locked: license will be locked to a single.
- Floating: license(s) will be available to others on your network.

Computer Type:

- Laptop
- Desktop

Advanced Help

< Back Next > Skip



**License Type**

License Type:  Evaluation  
 Permanent

Product Name: MULTI 2000, PowerPC v3.6

(Purchase Order Required)  
 Purchase Order Number: \_\_\_\_\_

Where did you get the software?  
 Green Hills Pattern

Advanced Help

< Back Next > Skip



**Green Hills Software License Request Generator**

**License Agreement**

Please read the following license agreement in its entirety. If you agree to the terms of the license agreement, please click, "Yes."

Green Hills Software 30-day Evaluation License Agreement (v3.6) Green Hills Software, Inc. (Green Hills) agrees to grant and user ("Licensee") agrees to accept a non-transferable and non-exclusive license ("the License") to use the executable programs, Green Hills C/C++ Compiler and MULTI ("the Licensed Programs"), on the following terms and conditions:

1. Consideration

For and in consideration of receiving the License, Licensee agrees to evaluate the Licensed Programs for the sole purpose of making a purchase decision, and to report to Green Hills any problems encountered in their use. Licensee also agrees to prevent the disclosure of any and all results of Licensee's use of Licensed Programs, including but not limited to, space performance, the performance, compile time performance, new features, bug fixes, defects, bugs, documentation requirements, dissemination errors, user interface, competitive analysis, comparison with competing products, etc.

Do you accept the terms of the preceding license agreement?

< Back Yes > No



**Green Hills Software License Request Generator**

**License Request**

Submit this request via e-mail: license@ghs.com or fax: (805) 965-6343

This is a request for a license to use MULTI 2000 and the other Green Hills tools.

Product: MULTI 2000, PowerPC v3.6  
 Debug Server: see PO  
 CD Obtained From: Green Hills Pattern  
 License Availability: Computer-Locked  
 License Type: Evaluation  
 System Name: MFGJUL  
 OS: Windows NT  
 Server Code: 33720/5057 7474 7677 616

I HAVE READ AND AGREE TO ABIDE BY THE TERMS OF THE ENCLOSED GREEN HILLS

Print Send Save To File < Back Skip

## MotoHawk™ Installation Procedure

Go to <http://mcs.woodward.com/> web site and register.

Have your instructor or sales representative upgrade your access level.

(Log out and back in, once your access level has been updated.)

Navigate to the “Support/Downloads” section.

Install downloads in the following order:

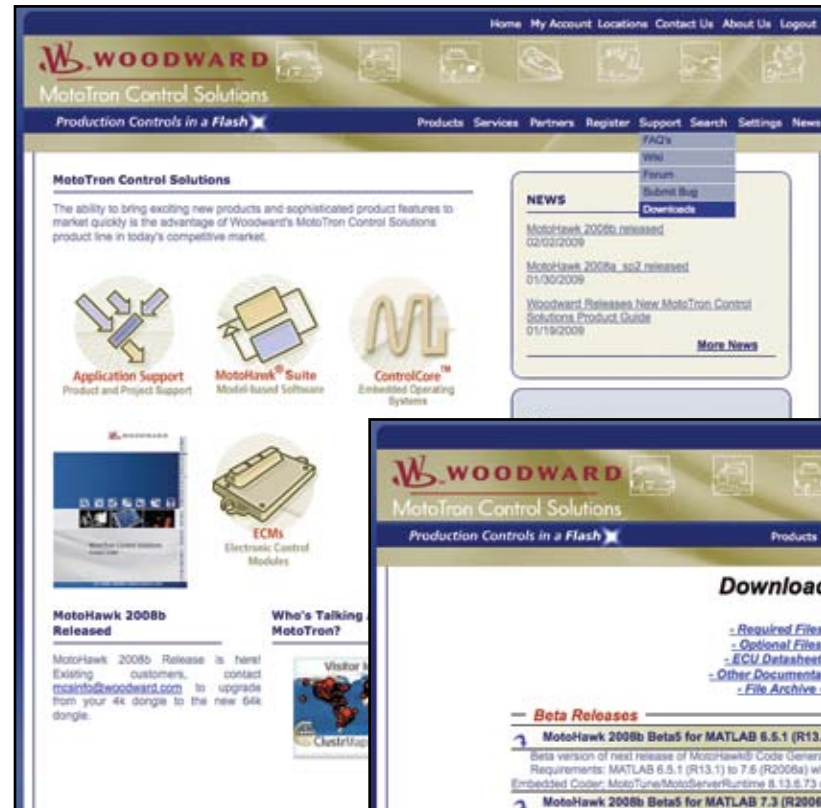
- Kvaser Drivers \_\_\_\_\_
- MotoServer \_\_\_\_\_
- MotoTune \_\_\_\_\_
- MotoHawk \_\_\_\_\_

Follow installation instructions for each one.

Note: It is recommended that you do not plug the adapter cable into the USB port prior to installing the Kvaser drivers.

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

## Website Resources <http://mcs.woodward.com>



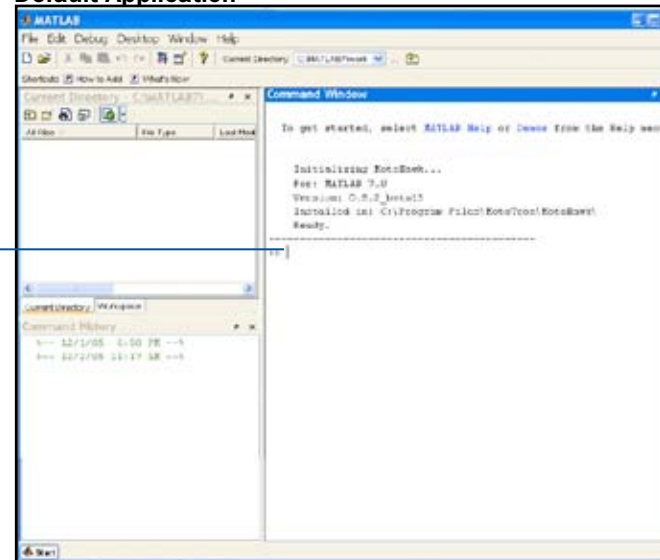
## Creating an application in MATLAB™

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

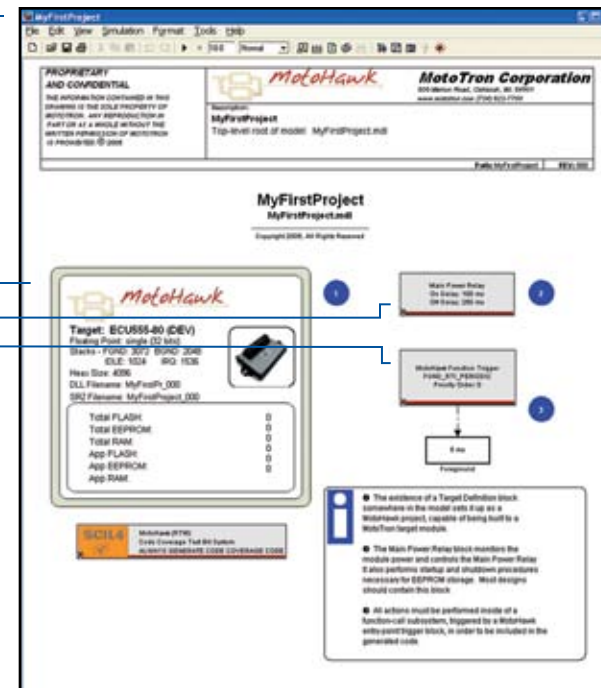
Start MATLAB: Double click on the MATLAB icon on your desktop or select from the programs menu.

The following screen will appear. \_\_\_\_\_  
 At the command line \_\_\_\_\_  
 type: motohawk\_project ('MyFirstProject')

### Default Application



Press the Enter key.  
 the following window will open \_\_\_\_\_  
 (Allow 1-2 minutes for the application to complete.)



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.)

The MATLAB window will look like this

## Building Your Application

Press CTRL+B

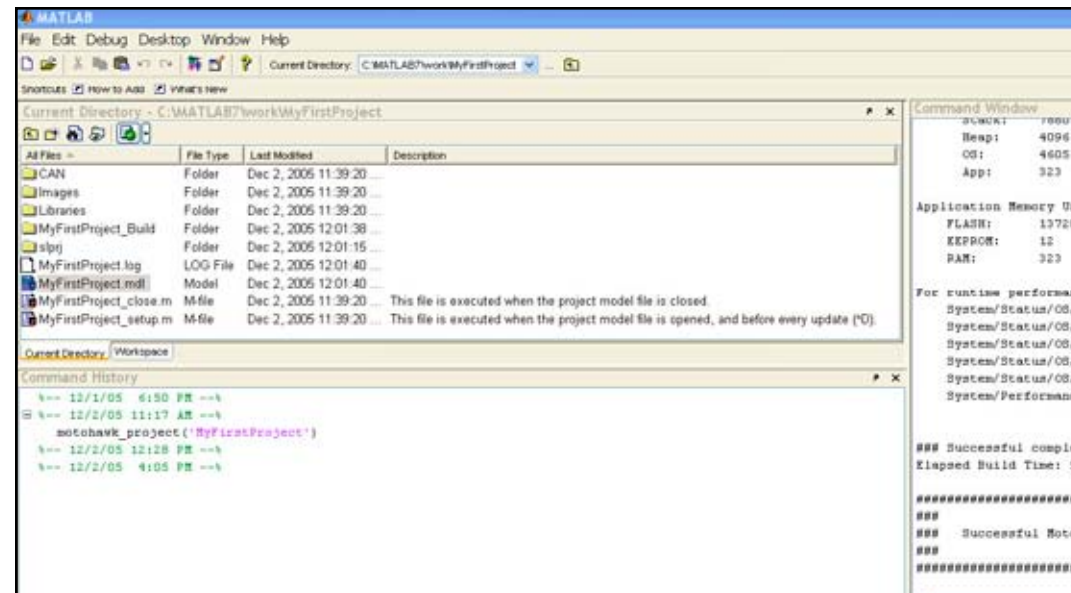
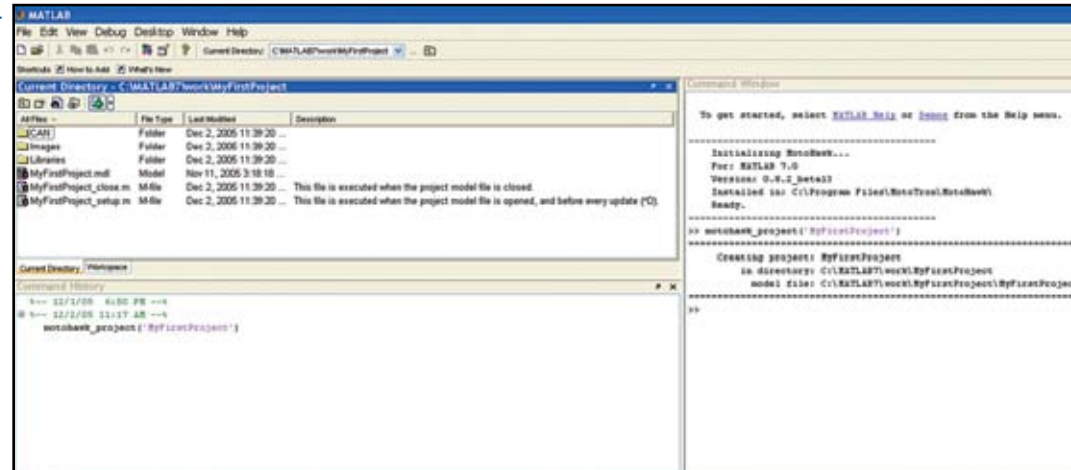
The MATLAB window should look like this

If the message says “Successful MotoHawk Generation (No Build,)” check your Greens Hills Compiler installation: Type “motohawk\_check\_ghs” (a zero indicates that you have a problem with your Green Hills Compiler installation.)

If you get an error, check with your instructor or e-mail the log file (MyFirstProject.log in this example) to: [MCSSupport@woodward.com](mailto:MCSSupport@woodward.com). A Technical Support Representative will contact you.

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

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

Install your isolated USB hub and apply power.

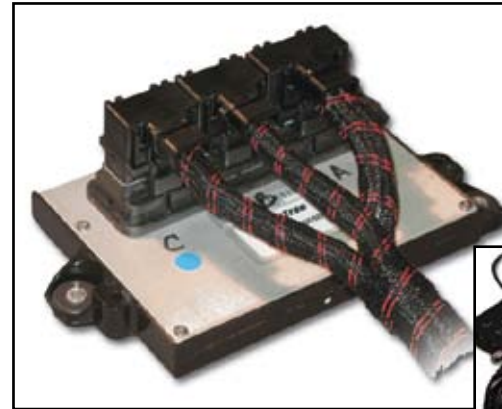
Insert your silver MotoTune dongle into the hub.

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.

Connect the Development Harness to the module (see datasheet for proper positioning.)

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.

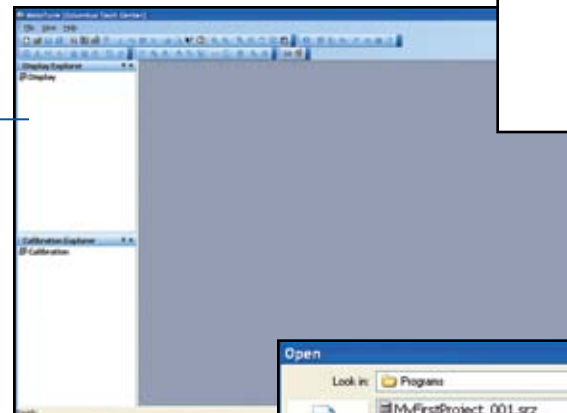


## Starting MotoTune

From the Start menu (or desktop shortcut) select All Programs/MotoTools/MotoTune.

The following window appears

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

Right-click on the Satellite Dish icon for MotoServer. (Located on the system tray.)

Select “Ports”.

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”.

You are now ready to connect to the module.

## Programming the Module

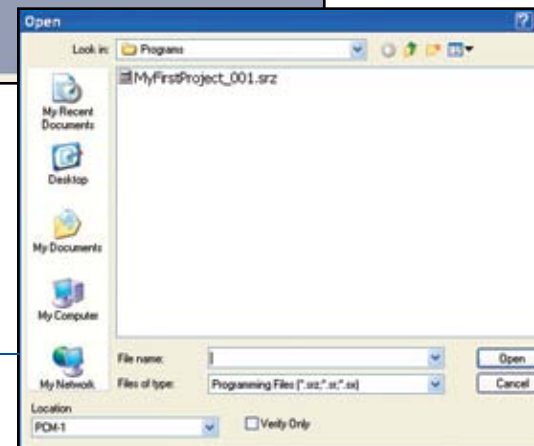
Turn power on and apply ECUP signal via power switch.

Select File/Program, in the MotoTune window.

The following pop-up appears

This is the file created when you pressed CTRL+B.

Double-click on the .srz file in the window.



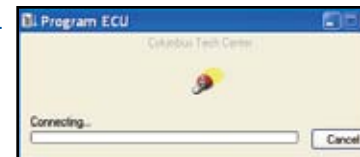
### The Program ECU status pop up appears

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. 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.)

Double-click on the .srz file and apply power.  
If this does not work, check with your instructor or send an e-mail to: MCSsupport@woodward.com.

When you see the Programming Successful message, you are ready to create a display for your application.



### Creating A Display

In the MotoTune Window, select File/New/Online Display/Calibration.

Select Display on the pop-up and click on OK.

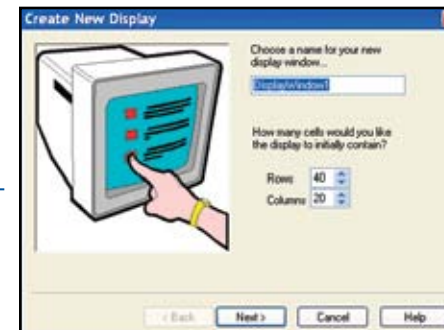
### The Create New Display window appears

Give your display a meaningful name (ie. MyFirstProjectDisplay.)

Select "Next" for default Row and Column settings.

Select "Next" for default Status Bar and Tab Control settings.

Use default Sheet1 by clicking on "Finish."



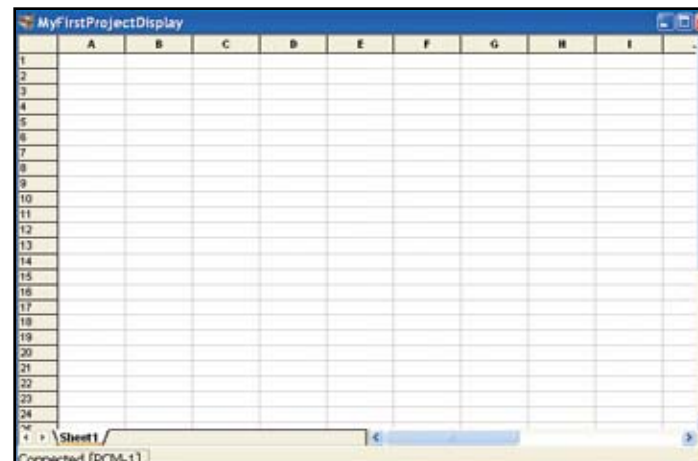
### The following should appear

Click on the "+" next to the MyFirstProject folder (listed on left side of the MotoTune window.)

Open the: >Foreground folder  
>Controller folder  
>Plant folder

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

Open the System folder, then the Performance folder.

Drag each of the display variables onto the spreadsheet.  
*Note that your system is running – these are its vital statistics.*

Cycle the power switch off, then back on.  
*Note that the display values briefly disappear, then return.*  
 The Main Power Relay can be heard releasing and engaging.

Close this model by clicking on the red “X” in the upper right hand corner.

*You will be prompted to save the model. We are done with this one – you may save or not.*

## First Application

Click the Simulink icon \_\_\_\_\_

Simulink’s Library Browser appears \_\_\_\_\_

*These are the Simulink and MotoHawk blocks which, are used for creating your application models.*

In the MATLAB window, move up one level to the “work” directory. Create a new directory “MySecondProject” and double-click on it.

In the library browser, click here \_\_\_\_\_

A new model window opens.

*Note the status window in the lower left hand corner. It indicates ODE45 this 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:

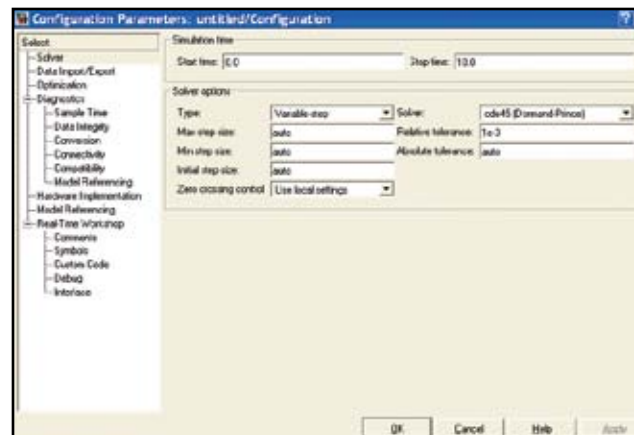
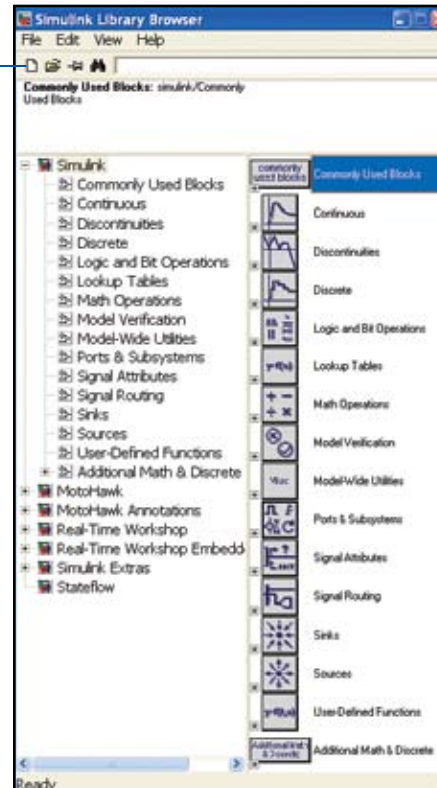
Select “Simulation” at the top of the window, then “Configuration (or Simulation) Parameters.”

The following window appears \_\_\_\_\_

Using the pull downs, change Type to Fixed-step, and Solver to Discrete (no continuous states.)



Simulink Icon:  
located top of the MATLAB window.



Click on Apply and OK.

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.

Click on Apply and OK.

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.

Click Apply and OK.

From the Extra Development Blocks library, drag a Main Power Relay block into your model.

(Default settings will serve our purposes for now)

From the Ports & Subsystems library drag a Function-Call Subsystem block into your model. Double-click on this block and a new window appears.

From the Sources library, drag the Sine Wave block from the bottom of the list into your model.

Click on Sinks and drag a Scope block into your model. Click on Math Operations and drag in a Gain block.

Note the greater than (>) symbols on each block. These are Simulink ports, which are used to control the signal flow through your model.

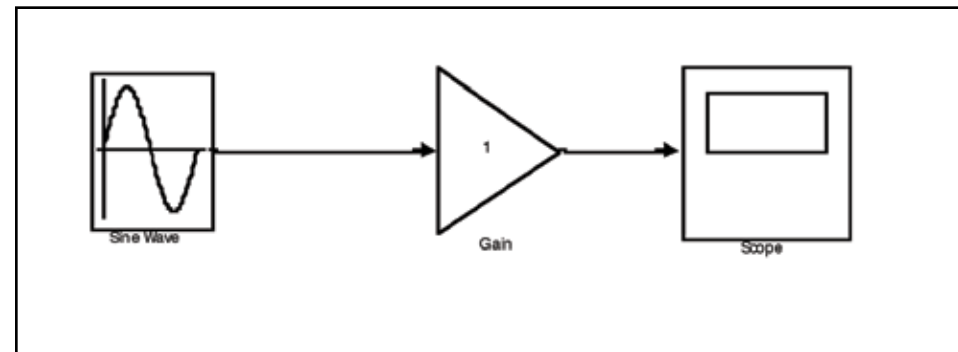
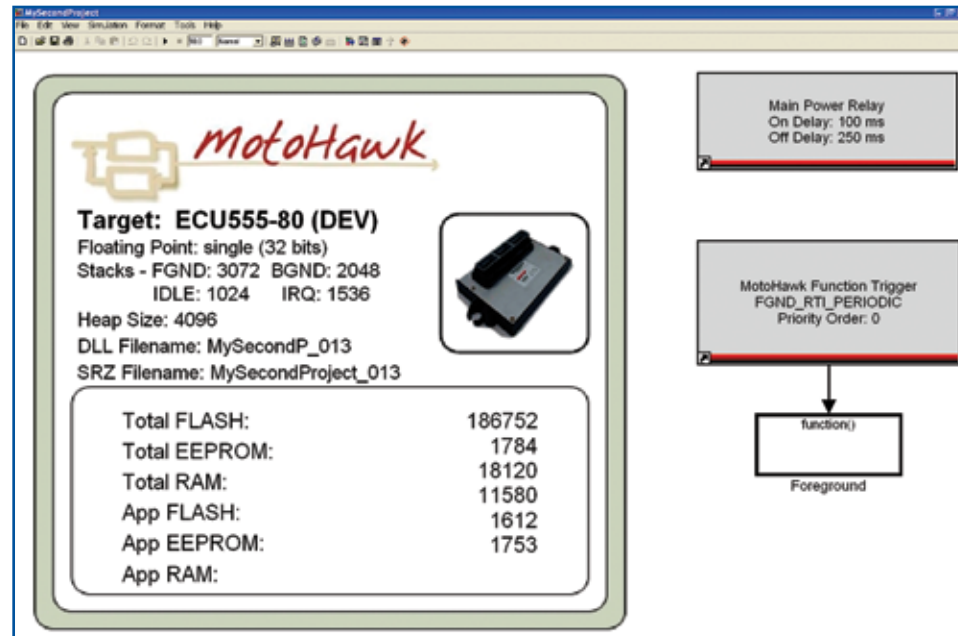
The Sine Wave block, being a signal source, has only one (output) port. Likewise the Scope block, being a sink, has only one (input) port, while the Gain block has one of each.

More complex blocks will have more input or output ports or both.

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.

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.



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.”

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.

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.

Press CTRL + D again.

No error messages are generated.

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

Select Simulation/Start — a Sine wave appears.

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.

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.

Change the name to “Foreground.”

Press CTRL + B

MotoHawk builds your application.

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

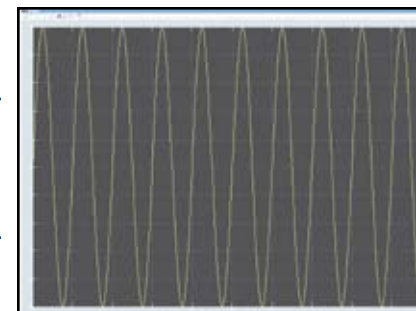
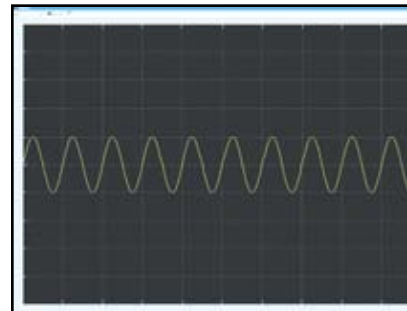
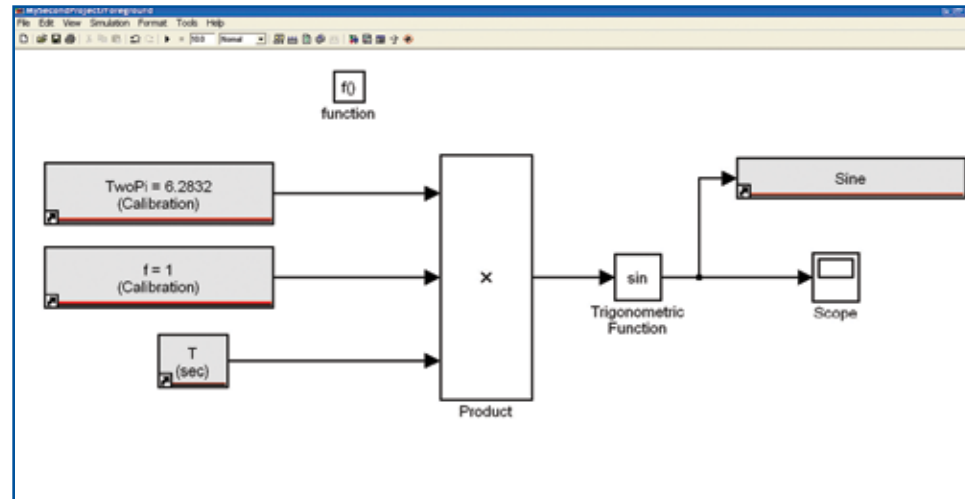
Right-click on MyFirstProjectDisplays and select Close.

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

Select File/Program and download MySecondProject into the module. Create a new display as above.

(ie. “MySecondProjectDisplay.”)

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



## Modifying the application

Allows you to gain some control over its operation.

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.

From the Calibration & Probing Blocks library, drag a motohawk\_calibration block and a motohawk\_probe block into your model.

From the Extra Development Blocks library, drag in a motohawk\_abs\_time block.

Double-click on the Calibration block and change the name to 'TwoPi' and value to 6.28318.

The single quotes must be used.

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

Right-click on the TwoPi block and drag down. A duplicate block is added to your model.

Double-click on the new block and change its name to "f" and its value to 1.

Wire these 3 blocks to the inputs on the Product block.

From the Math Operations library, drag in a Trigonometric Function block.

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

Wire the output of the Product block to the input of the Trigonometric Function block.

Wire the output of the Trigonometric Function block to the Scope block.

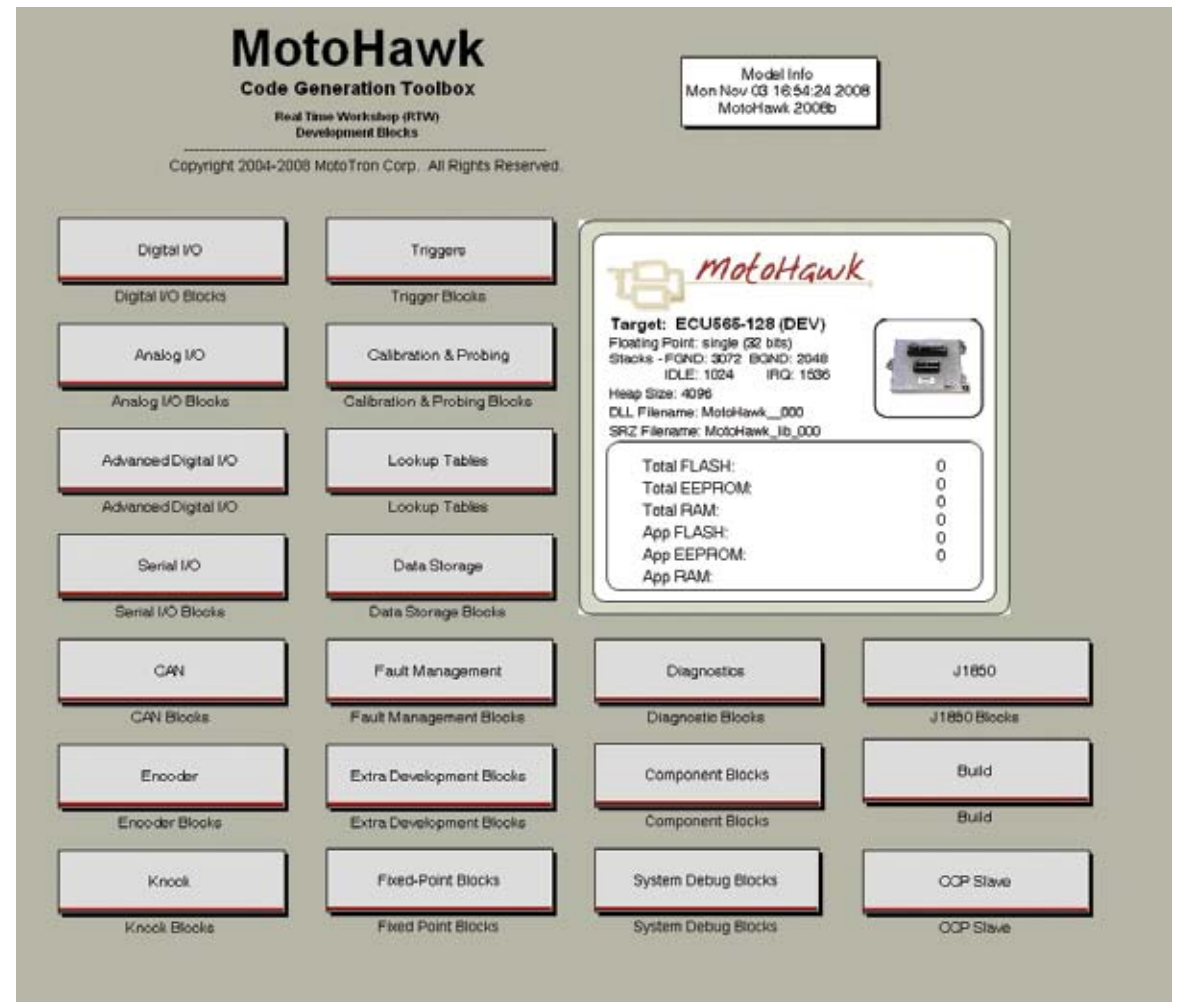
Double-click on the Probe block and change its name to Sine.

Place the cursor over the input port of the "Sine" Probe block.

Notice that the cursor changes into a cross-hairs.

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.



Your model should look similar to this

Press CTRL + D (see that there are no errors.)

Press CTRL + B (verify that the build is successful.)

Close the display in the MotoTune Display Explorer pane as above and program the module with your modified application.

Select File/New and create a new calibration.

In the Calibration Explorer pane, Click on the "+" next to the MySecondProject folder.

Double-click on Foreground.

A Calibration sheet opens in the right hand pane of the MotoTune window.

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.

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.

Select Chart/Open Chart.

A pop up appears displaying your Sine wave.

In the Foreground sheet change the "f" value to 2.

Note the frequency changes when the Enter key is pressed.

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.

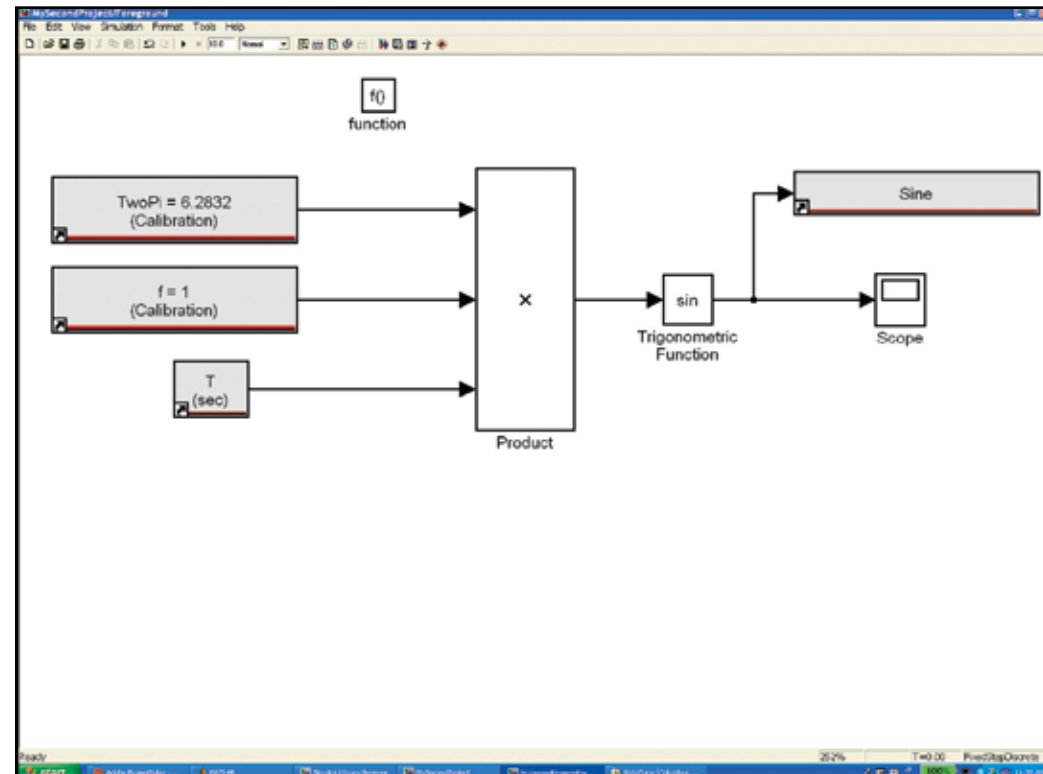
## Introducing a Gain Stage

Select the wire connecting the Trigonometric Function block and the Scope and press the Delete key.

Right-click on the TwoPi block and drag a copy to one side.

Double-click on the new block and change the name to 'Amplitude' and the value to 10.

Likewise, copy over the product block and change its Number of inputs to 2.



## .....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.

Connect the new calibration block and the Sine block to the product block inputs.

Wire the product block output to the Scope and Sine probe block.

Your model should look similar to this

Press CTRL + D and verify that there are no errors. Then press CTRL + B to build it.

Program the module with the new application. Set up your display and calibration windows in MotoTune as before.

Open a chart for the Sine probe and verify the amplitude value. 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.

Right-click and drag down to copy them.

Wire the blocks together as before, connecting the input of the Trigonometric Function to the output of the Product block on the Left.

Change the Trigonometric Function to Cosine and rename the Probe block accordingly.

Your model should look similar to this

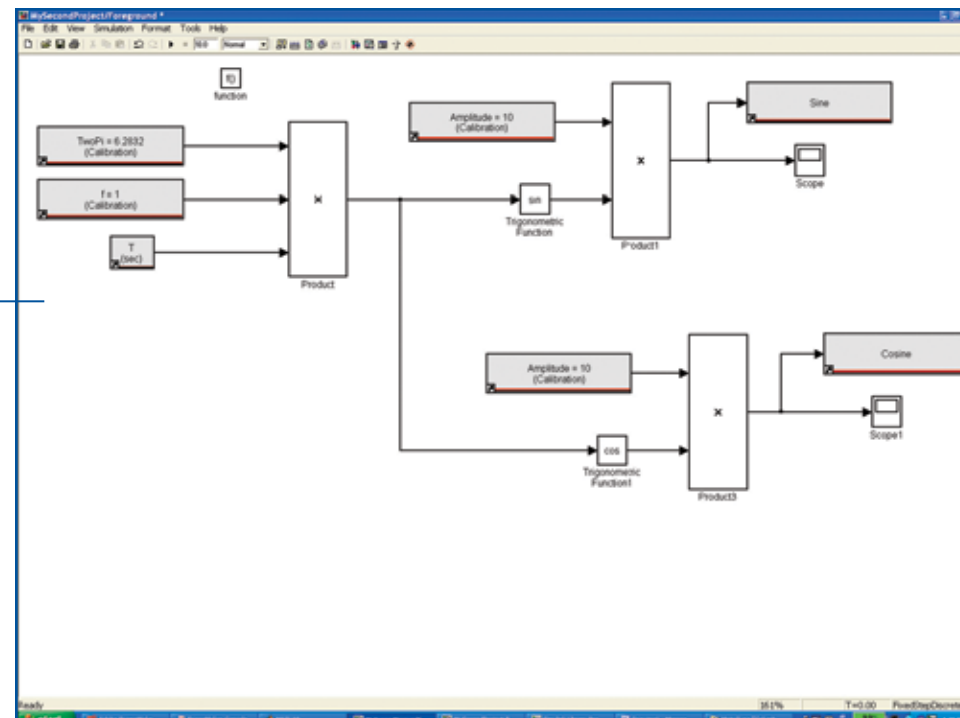
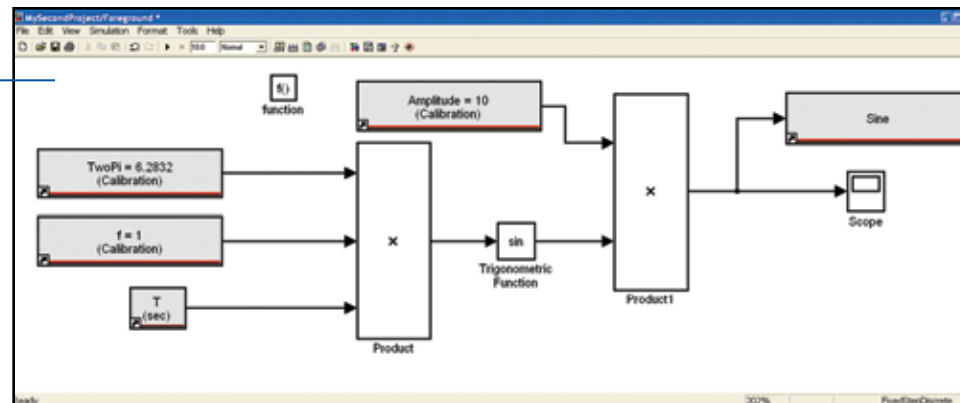
Press CTRL + D.

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

From the library, drag a motohawk\_data\_def block and a motohawk\_data\_read block into your model.

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.





Highlight the two calibration blocks called “Amplitude” and delete them.

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.

Right-click on the `motohawk_data_read` block and drag a copy over to the other loose wire.

Press CTRL + D again.

*No errors should be generated.*

Build your model, program the module, and set up your display and calibration windows as before.

[\(using the MotoHawk Data Storage blocks continued\)](#)

Right-click on either the Sine or the Cosine value and set the properties to:

- >Fast
- >Add to chart/log
- >Apply to all

Click OK.

Select Chart, Open Chart and observe your signals.

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.*

## Read More .....

### MotoTune Options .....

Selecting [Attach VarDec for Visibility](#) from MotoTune expands the dialog box giving us more options.

## ..... Data Storage Blocks: A closer look

### **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.)

### **The 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).

### ..... **Attach VarDec for Visibility offers:**

- 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.



enabled subsystem and a new window opens up.

Delete the output port and copy the input port by right-clicking on port 1.

From the commonly used blocks library, drag in a constant block and a sum block.

From the math operations library, drag in a math function block.

From the discrete library drag, in a unit delay block.

Right click to copy the constant block. Set the value of the new (constant1) block to 200.

Double-click on the math function block and use the pull-down to select mod (modulo) function. Click on Apply and OK.

Right click on the mod block and select format and flip block. Likewise flip the unit delay and constant1 blocks.

Wire the constant and mod blocks to the sum block inputs.

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.

From the data storage blocks library, drag in a motohawk\_data\_write block and make a copy of it.

Double-click on the first data write blocks. Name it SineData. Using the pull down, set data structure to vector.\*

Name the second data write block CosineData and make it a vector\* as well.

\*When you set up the Data Write Blocks as Vector, select 'Write Scalar into element by index'.

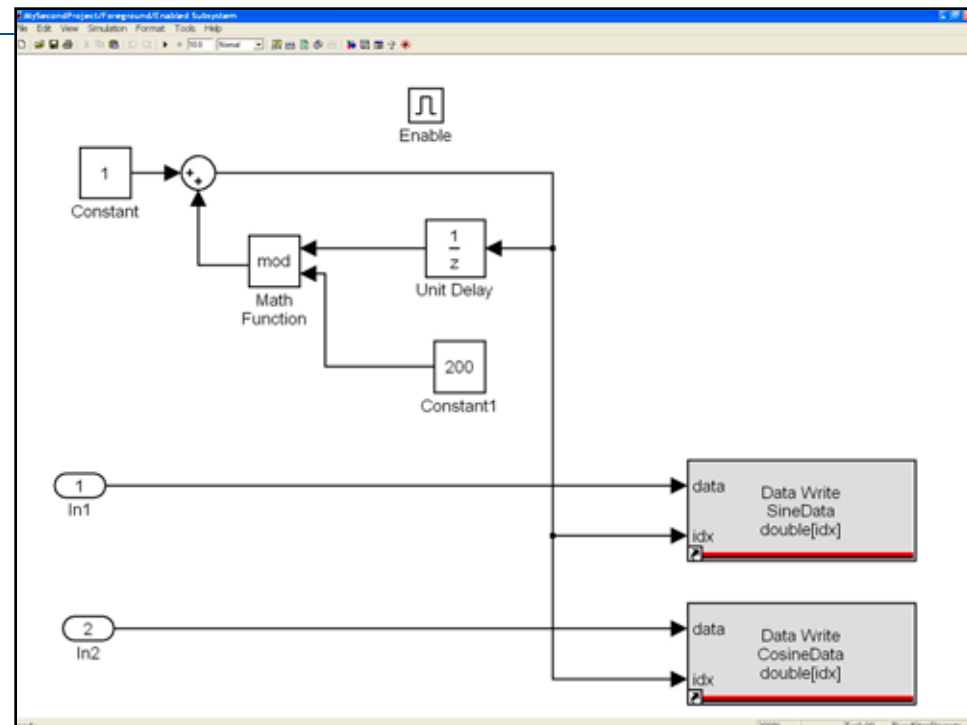
Wire the idx input of each data write block to the output of the sum block.

Wire input1 to the data input of the SineData block and input2 to the CosineData block.

Your enabled subsystem should look like this

Save file and close this window.

In the Foreground window, right-click on the



Amplitude data definition block and make two copies.

Double-click on the first copy, change the name to SineData, change the Storage Class to NonVolatile, and change MotoTune Window to Display.

Place the following in the Initial Value box: zeros (1,200.)

Click Apply and OK.

Double-click on the second copy, change the name to CosineData, Storage Class to NonVolatile, and MotoTune Window to Calibration.

Click Apply and OK.

Place the following in the Initial Value box: ones (1,200.)

Copy the 'f' calibration block and rename it.

Log and set the initial value to zero.

Wire the Sine signal to In1 and the Cosine signal to In2 of the enabled subsystem.

Wire the Log block to the input at the top of the enabled subsystem.

Your model should look like this

Press CTRL - D. If there are no errors, press CTRL - B.

Start MotoTune and create a new display and a new calibration. In the display pane expand MySecondProject and Foreground. Drag SineData into the worksheet.

Note that all of the values have been set to zero.

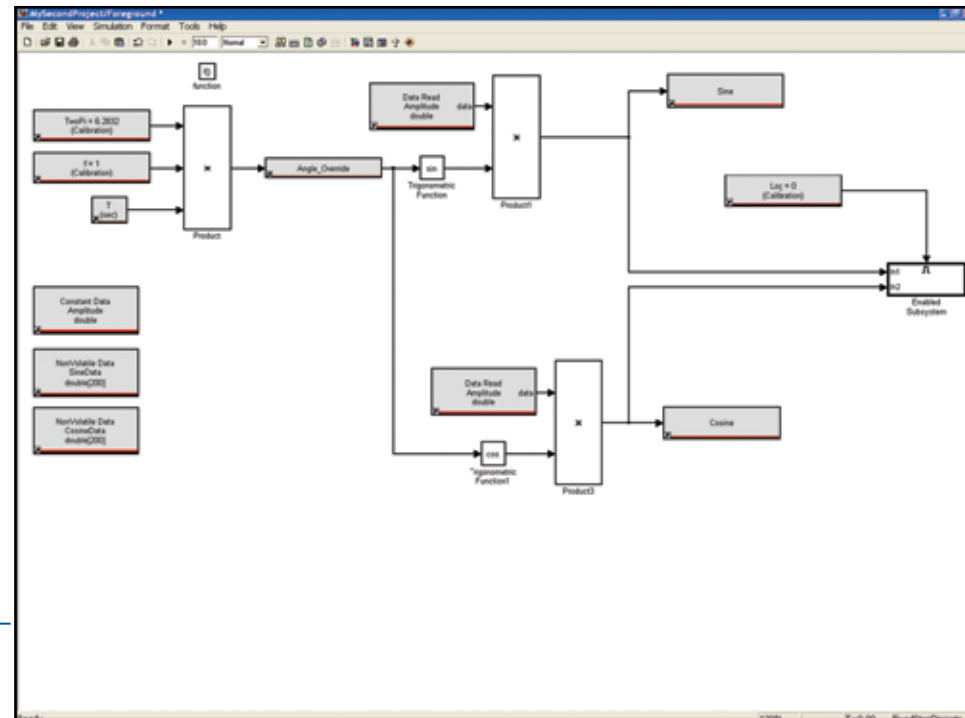
In the calibration pane, expand MySecondProject.

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.

**Confusing? Read more** .....



**Two ways to get around the confusion:**

The first would be to utilize the Show MotoTune Group check box and explicitly name the MotoTune Group String.

The other would be to place the data definition blocks in the enabled subsystem.

The system designer needs to decide what is the best way to organize these data structures, a CTRL - B is required to generate a new DLL.

For now, double-click on the Foreground and the CosineData sheets of paper and arrange them in the window.

Copy the “f” box and label it “Log”; set the value to 0. Put the following in the initial value box of SineData: “zeros (1, 200). That is one, comma, 200, not twelve thousand.

Do the same for CosineData.

Your window should look like this

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.

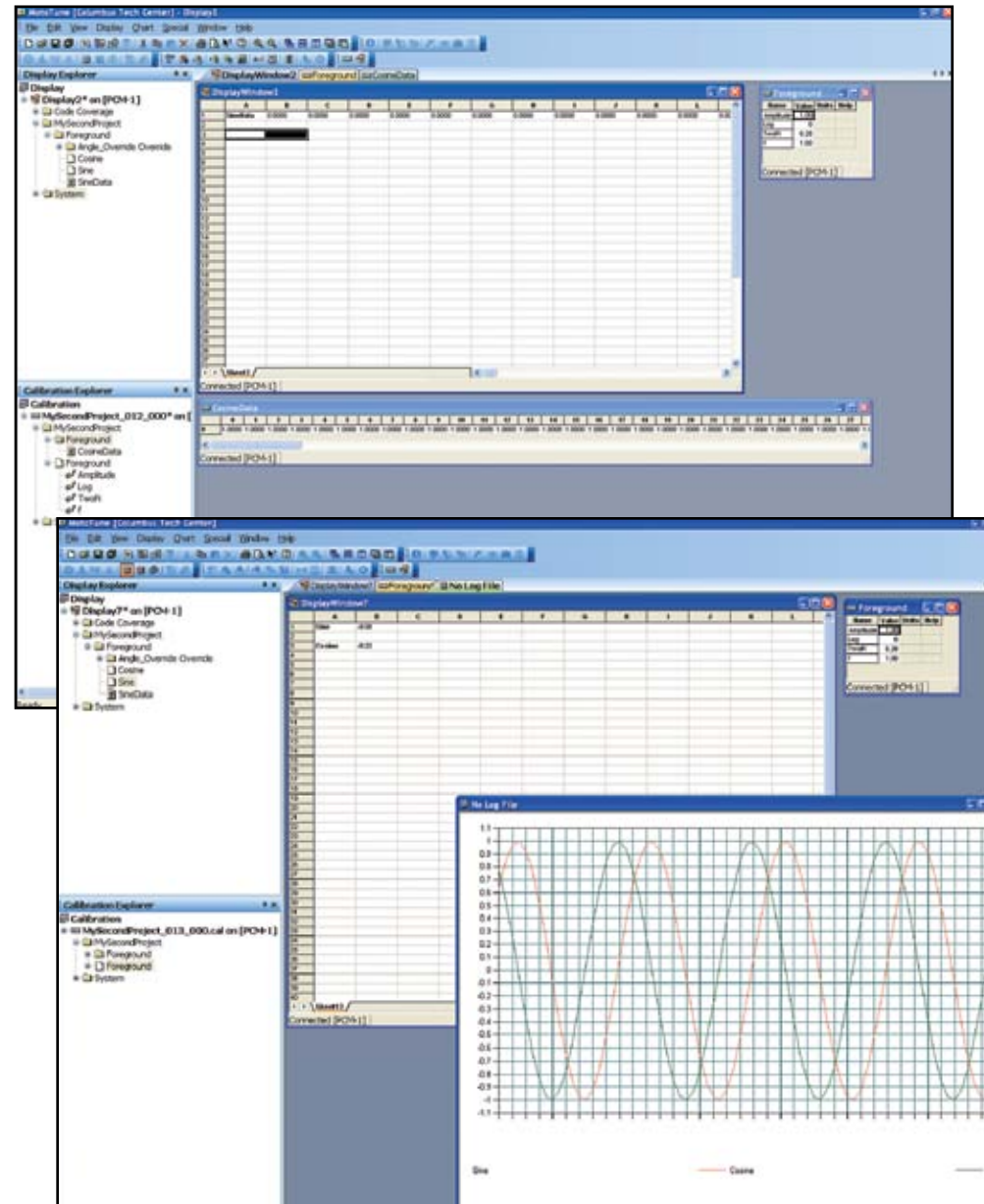
Select Calibration-Refresh Volatile Map (or press F5) and the CosineData array is updated.

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.

We are done with this example — you may close it.



## Throttle Control Challenge

The following example uses a slider potentiometer and an electronically controlled throttle assembly:

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

The Slider pot should be connected to XDRP, XDRG, and AN1M.

POT1 and POT2 should be connected to AN2M and AN3M respectively.

Consult the datasheet for your module to determine the appropriate wire number for each of the signals.

At the Simulink command line, use the `motohawk_project` instruction to open a new project. Name it `ThrottleControl`.

Double-click on the Foreground block and delete the Controller and Plant blocks.

From the MotoHawk Analog I/O Blocks library, drag in a `motohawk_ain` block.

Select "Allow I/O pin to be calibrated from MotoTune," and name the block `ThrottlePedal`.

Select AN1M from the pull down and click on "Apply" then "OK."

Drag in a Gain block and a `motohawk_probe` block.

Wire the `ThrottlePedal` block to the Gain block and the Gain block to the `motohawk_probe` block.

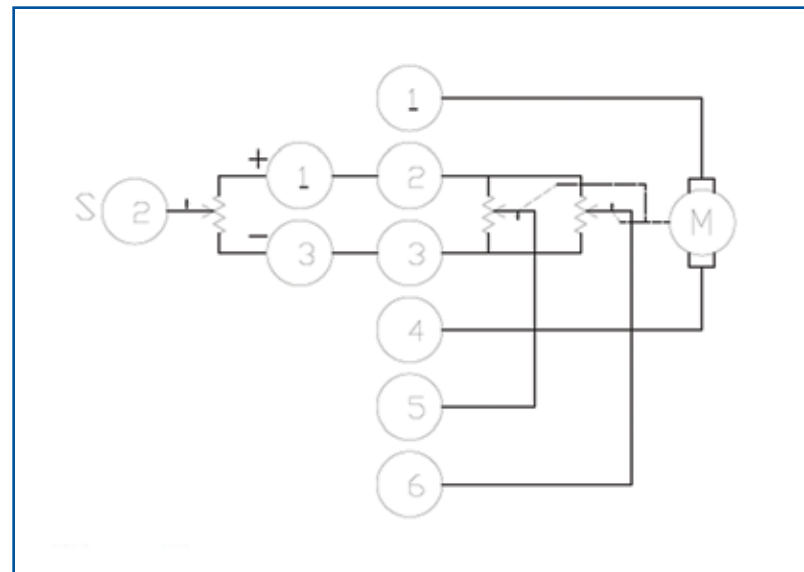
Set the Gain block Gain to  $100/1023$ .

Name your probe `SetPoint`.

Press CTRL - D.

Pin Number	Signal Name
1	Motor-
2	XDRG
3	XDRP
4	Motor+
5	POT2
6	POT1

Table 1: Electronic Throttle Connector Pinout



Electronic Throttle/Slider Potentiometer Schematic

In MATLAB 7.0, the following error message appears

MATLAB 7.0 supports a fixed point data type called `ufix16_eng19` which requires a separate license. Other versions of MATLAB will issue a data type mis-match error. This is because MATLAB uses a default data type of `double`, while the data type for a particular resource is dependant on the hardware.

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`.

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.

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.

Press CTRL - D again.

There should be no errors reported.

From the Format menu select Port/Signal Displays and check Port Data Types.

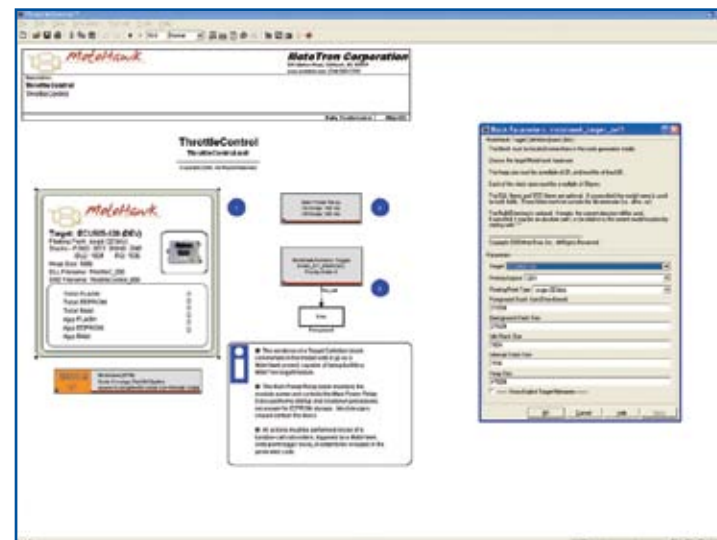
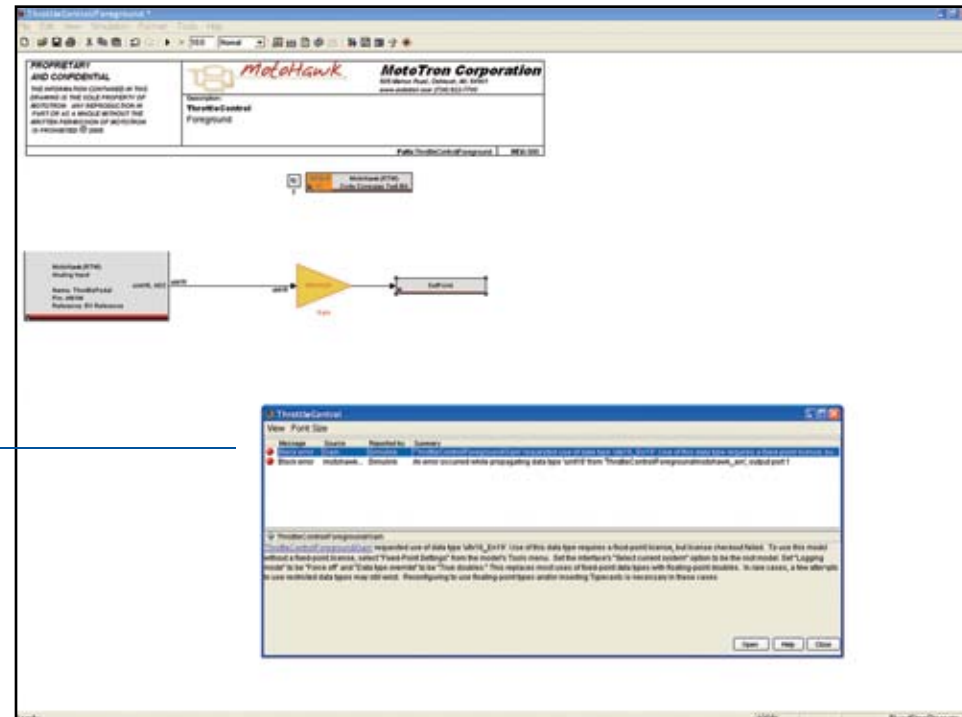
The data type appears adjacent to each wire. This is a convenient way to verify that your data types are consistent in your model.

Make copies of the analog input, data type conversion, gain, and probe blocks.

Highlight them and select Format-Flip Block (or CTRL - I).

Select AN2M for the analog input, name the probe Feedback.

Drag in a `motohawk_pwm` block from the Analog I/O Blocks library and select H1 as the resource.



Drag in a motohawk\_calibration block. Name it ETC\_Frequency and set the Default Value to 5000.

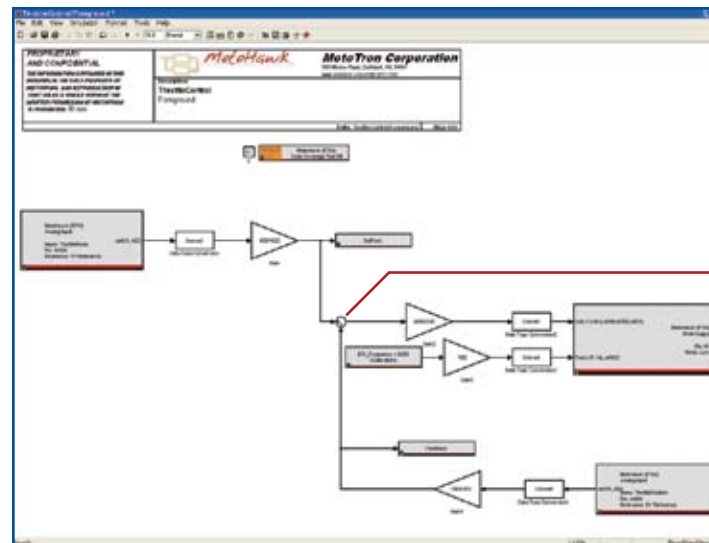
To make a proportional control like the one shown, double-click on the summing node and change the list of signs to  $|+|-$ .

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.

Press CTRL - B to build your model and use MotoTune to download it to the module.

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 calibratable Proportional and Integral gains.

Modify your drawing to look like the one shown.

Press CTRL - D to check your model.

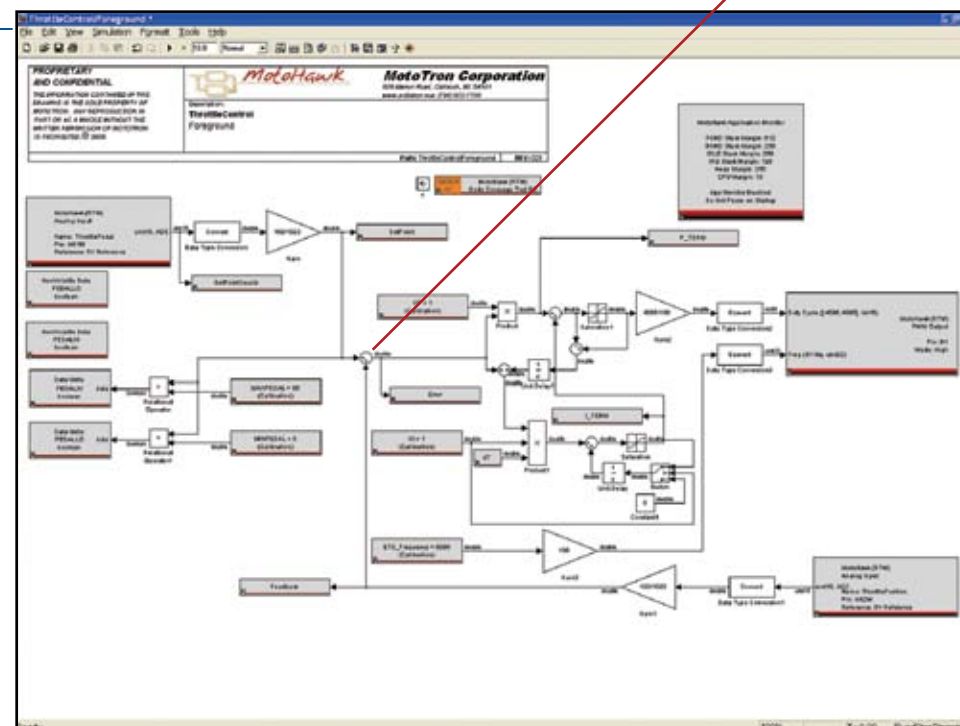
Then build it using CTRL-B.

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.

Set the Integral Gain to zero and increase the proportional gain until the throttle plate exhibits ringing when operated.

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

About Faults	1
MotoHawk Fault Theory of Operation	2
Fault Blocks	3
Fault Manager	3
Fault Definition	4
Set Fault & Clear Fault	4
Fault Status	5
Clear All Faults	5
Fault Action	6
Example	7-8

# FAULTS

system failure  
system failure  
system failure  
system failure

This section covers the basics of faults within MotoHawk.

## ... About Faults

**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 three 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 the containment or identification problems. That is the job of the application designer. MotoHawk will however, allow you to record the faults, help announce 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.**

**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. He 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.

**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** — meaning it will not signal a fault even if the X out of Y condition is satisfied.

**Sticky** — meaning that 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.

# MotoHawk

## Fault Management Blocks

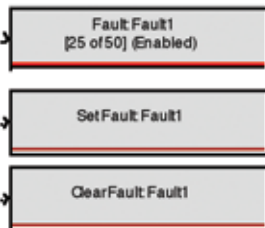
Copyright 2005 MotoTron, Inc. All Rights Reserved.

Provides functionality to manage a bit-packed fault data structure, to view and clear the status of faults from MotoTune, and to configure actions to be associated with any fault.

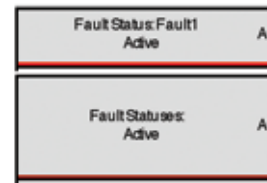
### Global Manager Definition



### Definition



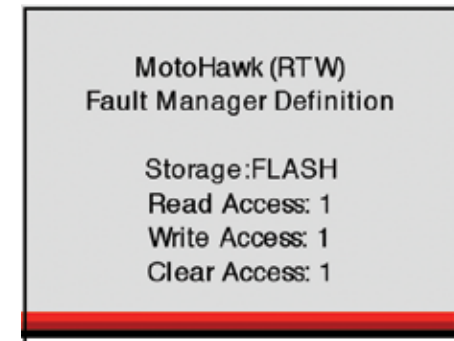
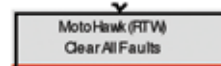
### Status



### Action



### Clear Faults



Block Parameters: motohawk\_fault\_manager

MotoHawk Fault Manager (mask) [link]

This block defines the Fault Manager, and must exist once in each model that contains Fault blocks.

Code is generated to cache the faults in a memory-efficient manner, and An interface is generated in MotoTune to display and configure the faults.

It is still legal to use Fault blocks without the Fault Manager, but without the definition block, it has the effect of removing all Fault functionality.

Copyright 2005 MotoTron, Inc. All Rights Reserved.

Parameters:

Storage:

Read Access Level:

Write Access Level:

Clear Access Level:

MotoTune Group String

OK Cancel Help Apply

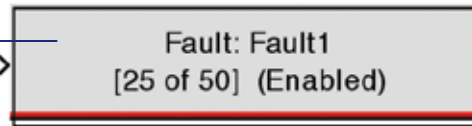
## Fault Definition

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.



**Block Parameters: motohawk\_fault\_def**

MotoHawk Fault Definition (mask) (link)

This block defines a Fault, which can be viewed from MotoTune, and may have several calibratable Fault Actions associated with it.

Faults have three boolean states associated with it: Suspected, Active, and Occured. If Suspected, the input to the block is true. If Active, the current fault has exceeded the logic associated with its behavior. A fault has Occured if it has ever transitioned from Active to Inactive, since processor reset.

If the input signal is true, the fault becomes Suspected until the count reaches X out of Y samples, when it becomes Active. The fault becomes inactive again after Y samples of false input.

A Sticky fault remains Active until explicitly cleared from MotoTune, or until shutdown. A Non-Sticky fault only stays Active while the count logic is true.

Each fault may declare up to 4 Actions to be taken, according to one of the Action Conditions.

Copyright 2005 MotoTron, Inc. All Rights Reserved.

---

Parameters

Fault Name

Mode

Faulty Samples (X)

Total Samples (Y)

Action 1

Action 1 Condition

Action 2

Action 2 Condition

Action 3

Action 3 Condition

Action 4

Action 4 Condition

Use uploaded Mode / X / Y values from MotoTune

Use uploaded Fault Actions from MotoTune

OK Cancel Help Apply



**Block Parameters: motohawk\_fault\_set**

MotoHawk Fault Set (mask) (link)

This block sets a predefined Fault, which can be viewed from MotoTune, and may have several calibratable Fault Actions associated with it.

If the input signal is true, the fault becomes Suspected until the count reaches X out of Y samples, when it becomes Active. The fault becomes inactive again after Y samples of false input.

Copyright 2005 MotoTron, Inc. All Rights Reserved.

---

Parameters

Fault Name

OK Cancel Help Apply

**Block Parameters: motohawk\_single\_fault\_clear**

MotoHawk Fault Set (mask) (link)

This block clears a predefined Fault, which can be viewed from MotoTune, and may have several calibratable Fault Actions associated with it.

If the input signal is true, the fault will be cleared.

Copyright 2005 MotoTron, Inc. All Rights Reserved.

---

Parameters

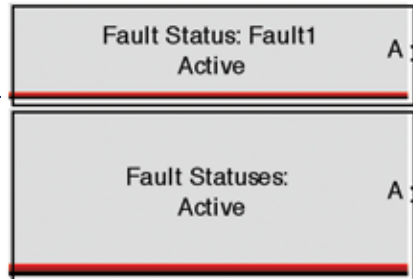
Fault Name

OK Cancel Help Apply

## Fault Status

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

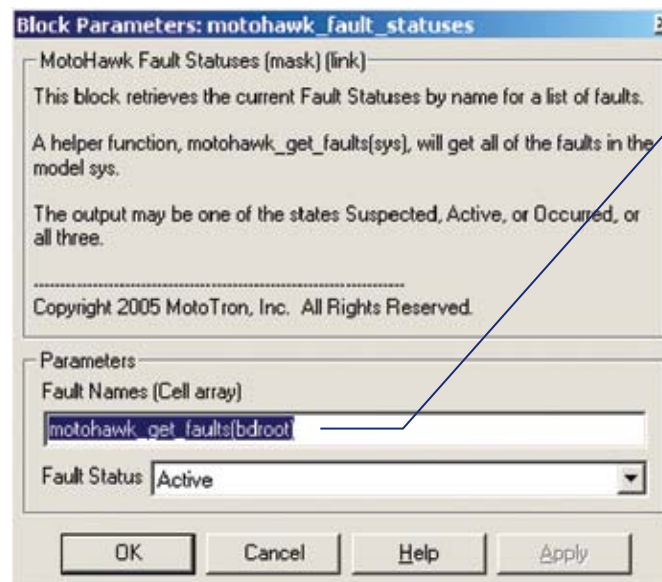
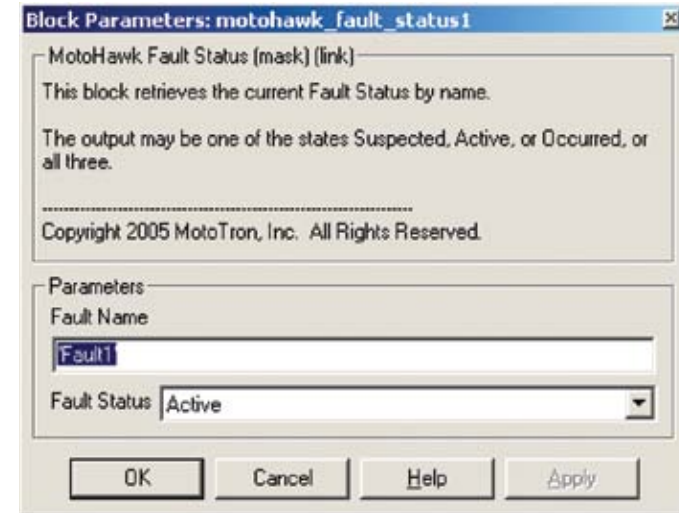
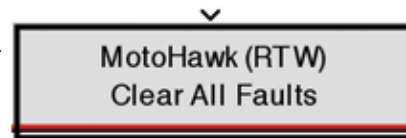
When reading multiple statuses, the output will be a vector of boolean values corresponding to the fault list.



## 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.

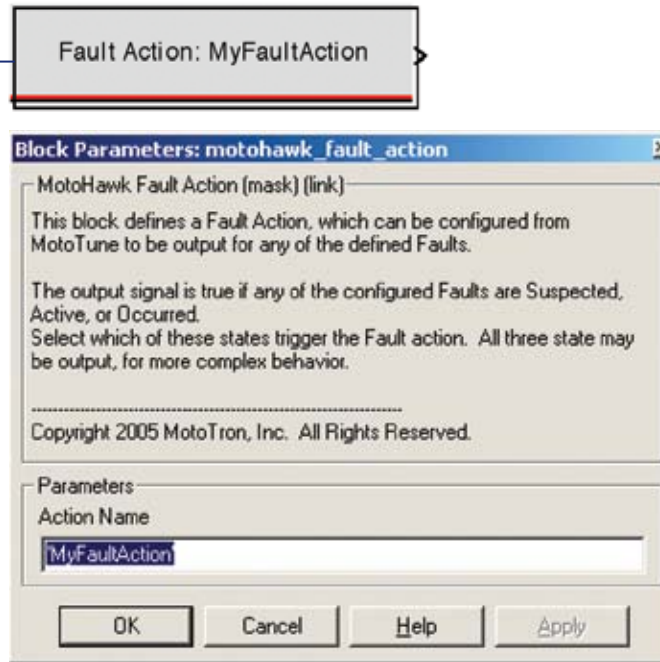


**Motohawk\_get\_faults(system:)**  
 This is a utility function that will retrieve all of the faults located in the system and its children. Use bdroot to find all faults within the model. The fault list returned by this function will be alphabetized.

## 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.



## Example

This example will demonstrate the Fault Blocks.

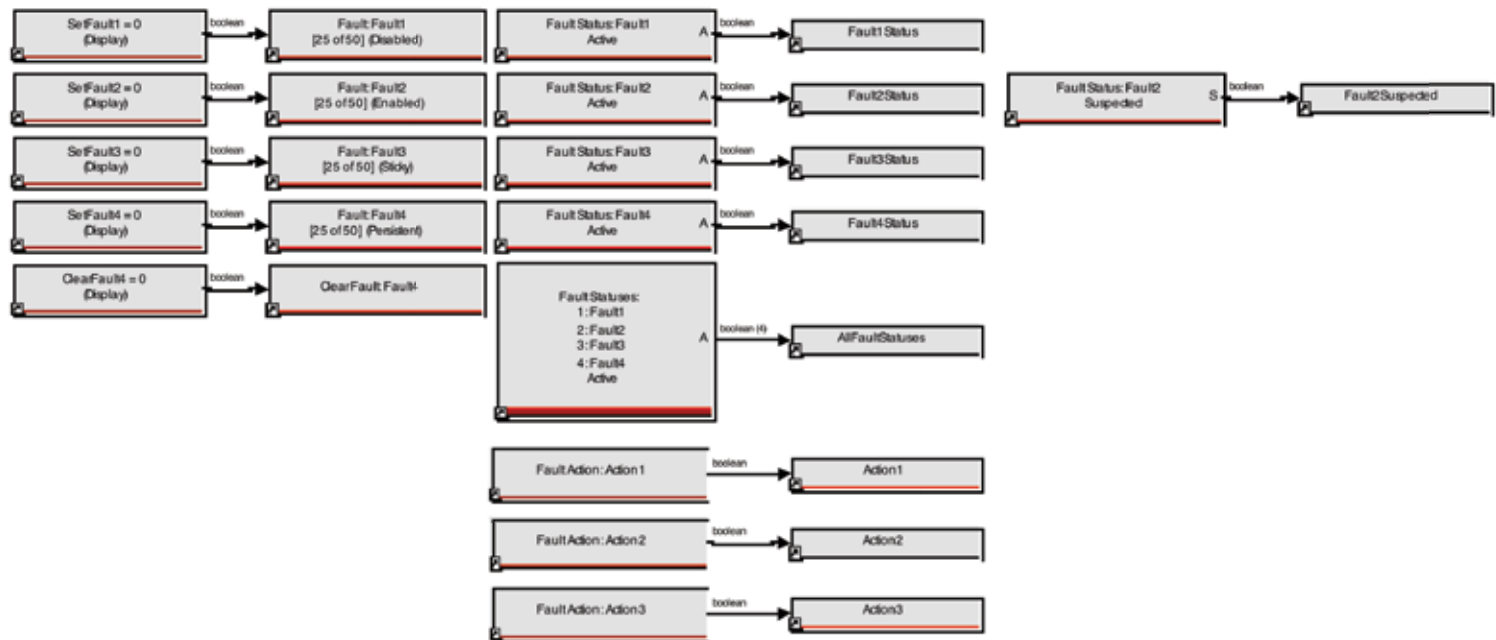
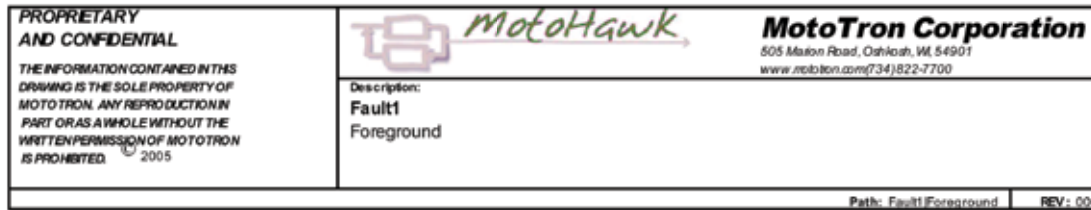
Start with `motohawk_project('Fault1.')`

Remove the existing contents of the Foreground subsystem

Create the model as shown. Build the model.

Run MotoTune, program the module, and open a display or open the `FaultExample.dis` file.

\*For a larger view of drawing, open `MotoHawk_Resource_Guide_11x17_drawings.pdf` on the included training cd.



**(example continued)**

Notice in MotoTune display explorer, there is a category for Faults that contains the display variables for:

- \_\_\_\_\_ Active Faults
- Occurred Faults
- Suspected Faults
- command that will 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.

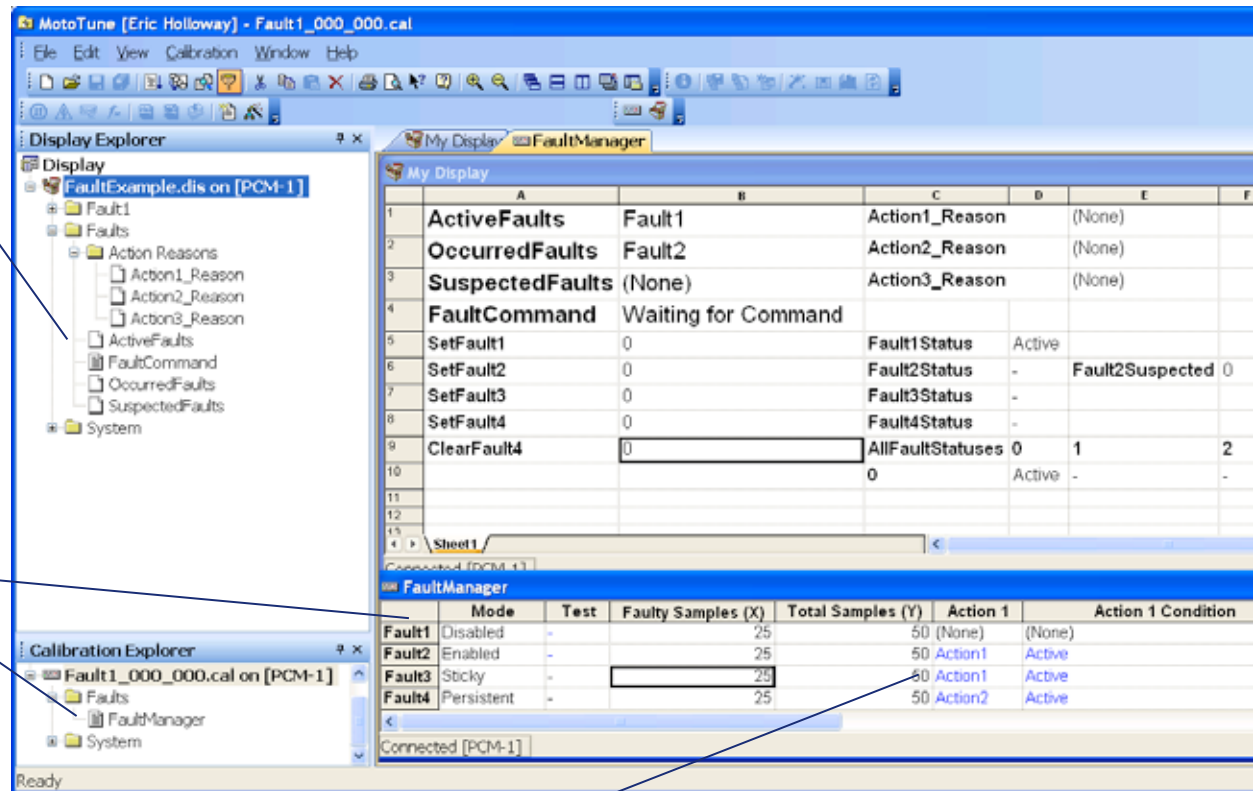
Open a calibration and notice the Faults category in the Calibration Explorer. The Fault Manager is located here. Open it up.

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.

\*For a larger view of drawing, open MotoHawk\_Resource\_Guide\_11x17\_drawings.pdf on the included training cd.



The screenshot shows the MotoTune interface with three main windows:

- Display Explorer:** Shows a tree view for 'FaultExample.dis on [PCM-1]' with folders for 'Fault1', 'Faults', 'Action Reasons', 'ActiveFaults', 'FaultCommand', 'OccurredFaults', 'SuspectedFaults', and 'System'.
- Calibration Explorer:** Shows 'Fault1\_000\_000.cal on [PCM-1]' with a 'FaultManager' folder.
- Fault Manager:** A table with columns: Mode, Test, Faulty Samples (X), Total Samples (Y), Action 1, and Action 1 Condition.
 

	Mode	Test	Faulty Samples (X)	Total Samples (Y)	Action 1	Action 1 Condition
Fault1	Disabled	-	25	50	(None)	(None)
Fault2	Enabled	-	25	50	Action1	Active
Fault3	Sticky	-	25	50	Action1	Active
Fault4	Persistent	-	25	50	Action2	Active

The 'My Display' window shows a grid of variables:

	A	B	C	D	E	F	
1	ActiveFaults	Fault1	Action1_Reason	(None)			
2	OccurredFaults	Fault2	Action2_Reason	(None)			
3	SuspectedFaults	(None)	Action3_Reason	(None)			
4	FaultCommand	Waiting for Command					
5	SetFault1	0	Fault1Status	Active			
6	SetFault2	0	Fault2Status	-	Fault2Suspected	0	
7	SetFault3	0	Fault3Status	-			
8	SetFault4	0	Fault4Status	-			
9	ClearFault4	0	AllFaultStatuses	0	1	2	
10			0	Active	-	-	





## CHAPTER 3 : CAN

About CAN	1
Introduction	2
CAN Bus Basics	2
Payloads	3
Protocols	3
What should a protocol specify?	4
Examples of Protocols	4
MotoHawk CAN Theory of Operation	4
Using CANKing to Observe the Bus	6
Basic CAN Blocks	8
CAN Channel Definition	8
CAN Transmit Raw	9
CAN Receive Raw	9
Slot Properties	10
Slot Receive Trigger	10
Example	11
Advanced CAN Blocks	13
Payload Bit Numbering	13
Standard ID Bit Numbering	13
Extended ID Bit Numbering	13
Message Definition Structure	14
Advanced Example	16
Recommended Usage of CAN Message Receive Blocks	18

**WARNING**  
**TO THE READER:**  
**CAN IS NOT DIFFICULT!**

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 via RS232. However, there are a couple of issues that can make it seem daunting – especially when talking about CAN protocols like J1939 or SmartCraft.

### ... About CAN

The CAN standard was invented by Bosch in the early 1990's to facilitate the communication of data between devices in a vehicle. **CAN literally means Controller Area Network.**

All MotoTron Control Solutions modules are compatible with the current standard – CAN 2.0B. Most of our modules have at least one CAN port, while a few have as many as three.

All of our drive-by-wire marine applications require two busses for reliability and redundancy, so many modules are equipped with a pair of busses.

On to the basics of what makes a CAN bus.

## Introduction

This section covers the basics of the CAN databus and how to use the MotoHawk facilities to interface your module to a CAN bus.

## 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:

- \_\_\_\_\_ **Lower ID values have higher priority on the bus (and standard IDs are higher priority than extended IDs)**
- \_\_\_\_\_ **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.

### A frequent question is... How much data can a CAN bus transfer?

There is plenty of sophisticated math can run. Or, you can remember that 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 Woodward’s MotoTron Control Solutions hardware provide all of 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.

## What should a protocol specify?

### Most protocol specifications will define Message Definitions which include:

- ID (including whether it is extended or standard)
- A description of any of the meaning of any ID bits
- A 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
- The device responsible for transmitting the message
- The expected number of bytes in the payload
- The contents of the payload
- The size of each content item in bits
- The location of each content item in the payload
- The data type of each of the content items
- The byte packing order of each of the content items
- A 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.

### **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.

(MotoHawk CAN theory of operation continued)

### 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.

### ... 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 0x00 0x00 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.](http://www.kvaser.com/))

Kvaser publishes a free CAN tool, CANKing, which allows you to observe the bus and even send messages. You will need this tool routinely. Download CANKing from our website at [http://mcs.woodward.com/.](http://mcs.woodward.com/)

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

Choose “Template” to start a new project.

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. 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.

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

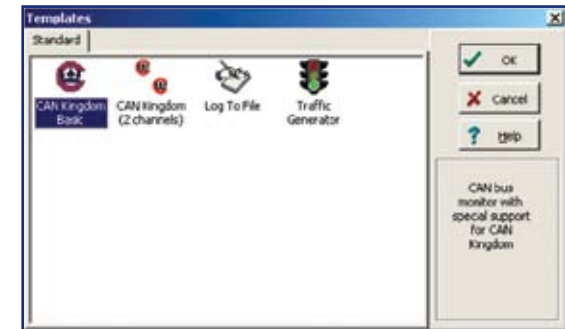
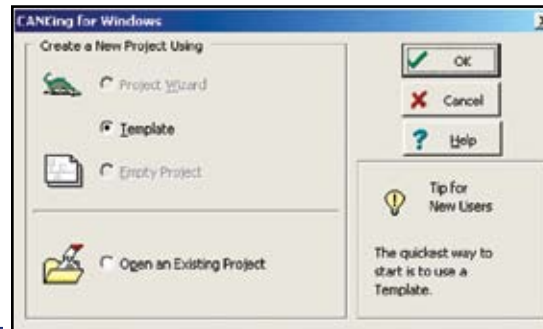
If there is traffic on the bus, the “Bus Load” bar will give you an idea of how much bandwidth is being consumed. If the “Error Passive” indicator illuminates, there are 3 possible reasons:

\_\_\_\_\_ No bus terminator

\_\_\_\_\_ Incorrect BAUD rate

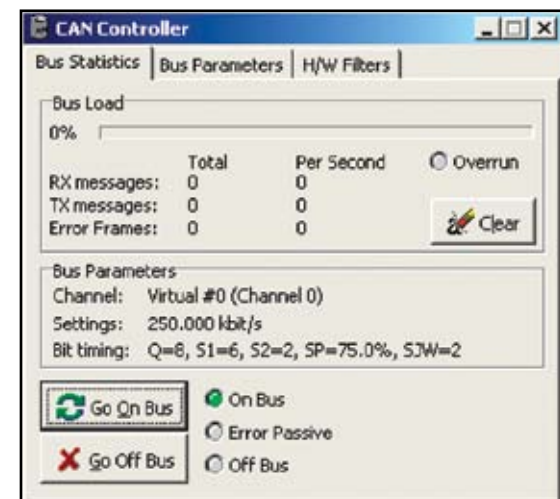
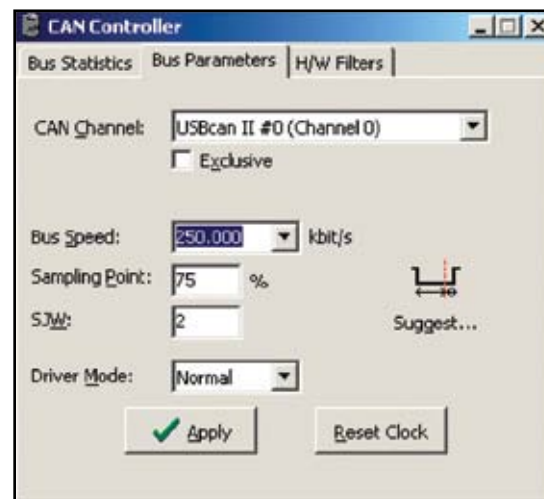
\_\_\_\_\_ No other modules on the bus (because the modules are not operating or there is a wiring problem.)

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.



**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.



Open the Messages menu and select the Universal page to get a window that will allow you to test transmission of messages.

Transmit anything and you will either see the state “Error Passive” or the message will appear in the “Output Window.”

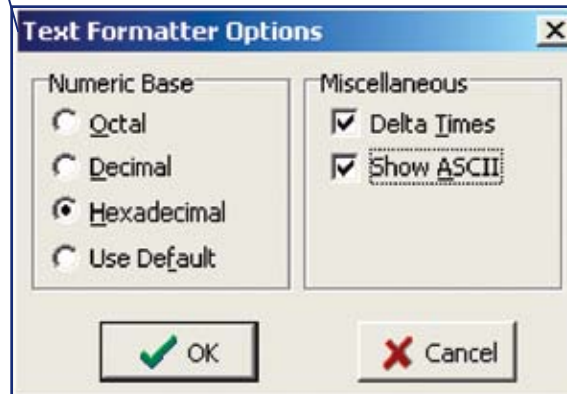
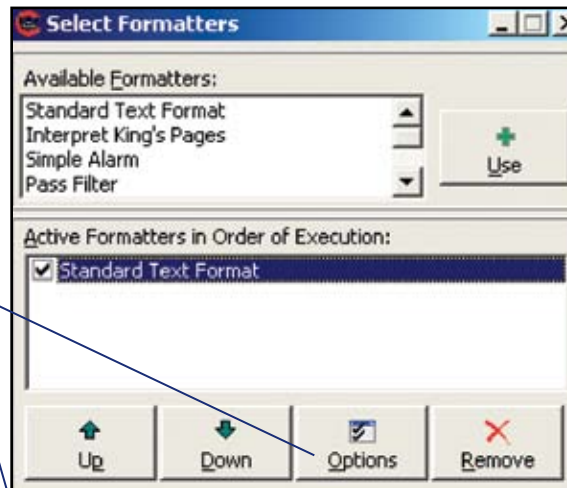
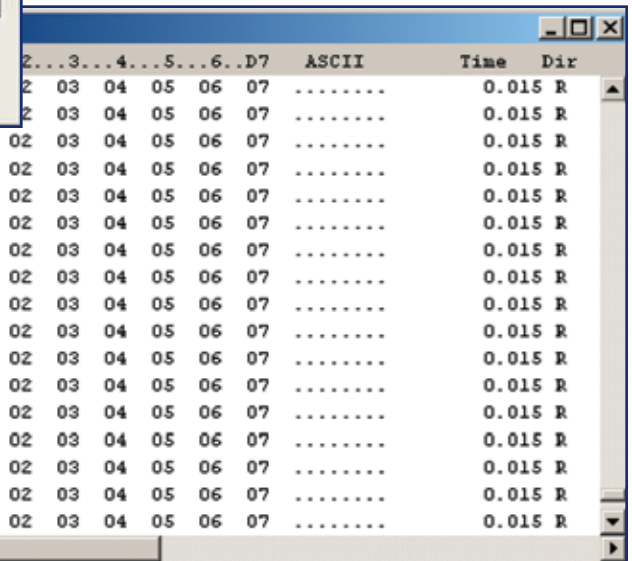
To display the messages in a useful form, find the “Select Formatters” window, select the “Standard Text Format” in the “Active Formatters in Order of Execution” list, and press the “Options” button.

The window “Text Formatter Options” will appear

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.

2	3	4	5	6	D7	ASCII	Time	Dir				
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R
07FF	8	00	01	02	03	04	05	06	07	.....	0.015	R

## 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.**

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

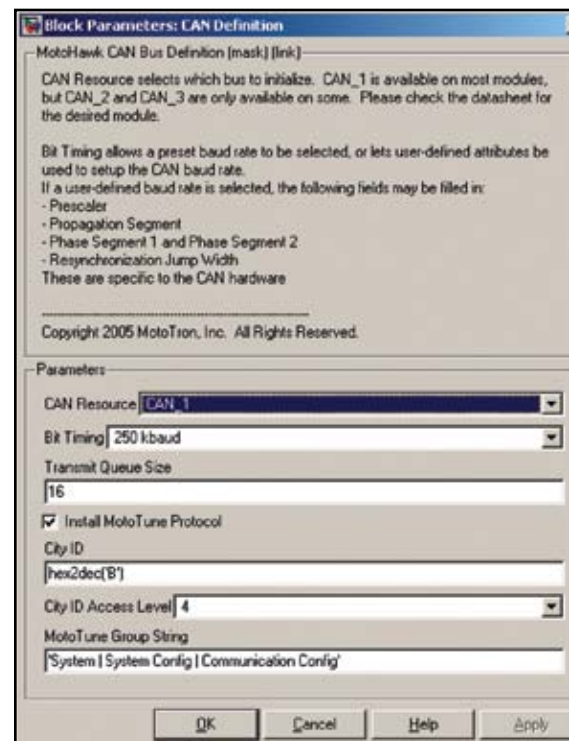
**City IDs :** The City ID is a MotoTune protocol value that essentially identifies the device. City ID 11 (0x0b) is the default for all of our modules. City ID 2 (0x02) is the ID for MotoTune. If you monitor the can bus while MotoTune is active, you will see extended message IDss like 0x00000b02 and 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 MotoTune to several modules. Each module must have a different City ID.

```

MotoHawk CAN Definition

Bus: CAN_1
Bit Timing: 250 kbaud
TX Queue: 16 messages

MotoTune Protocol Enabled
City ID: 0x0B (PCM-1)
    
```





## CAN Transmit Raw

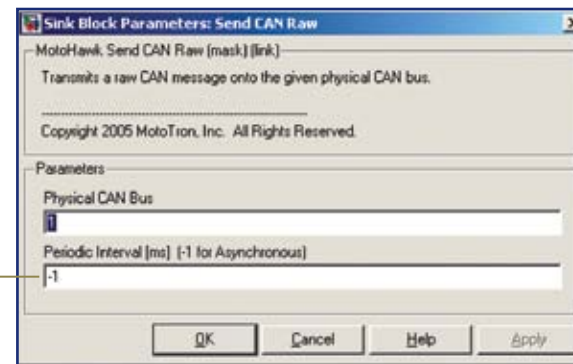
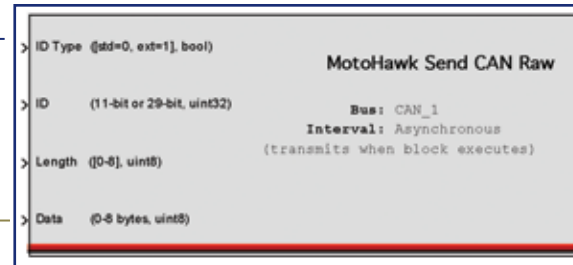
This block can have multiple instances within your model.

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.

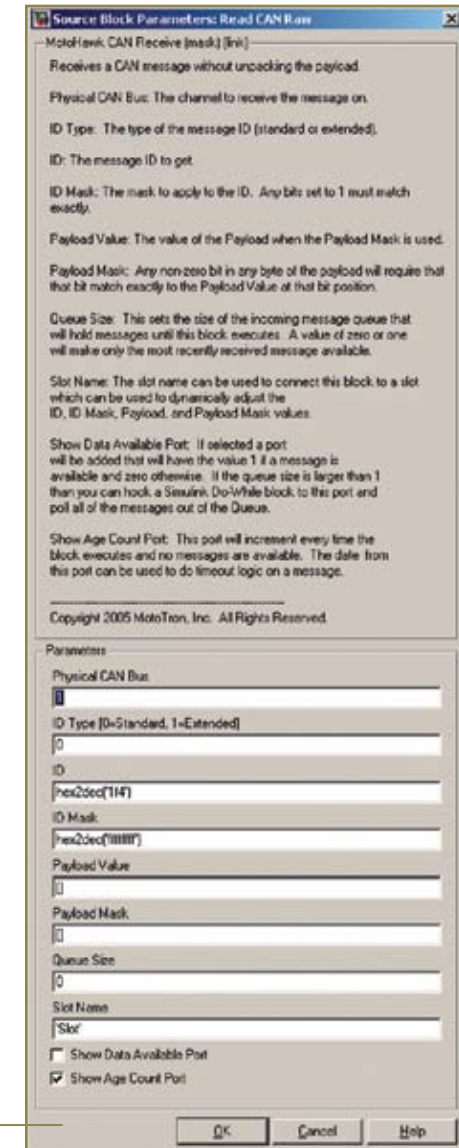
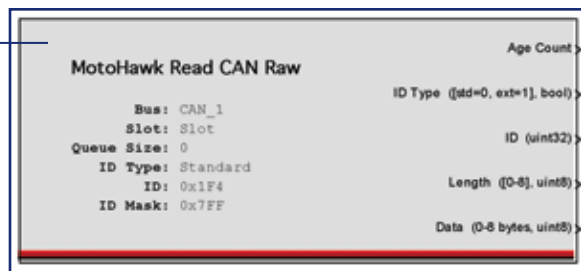


## CAN Receive Raw

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.)

**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.



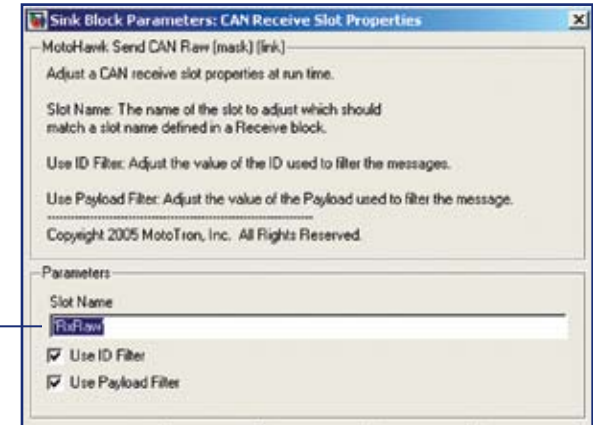
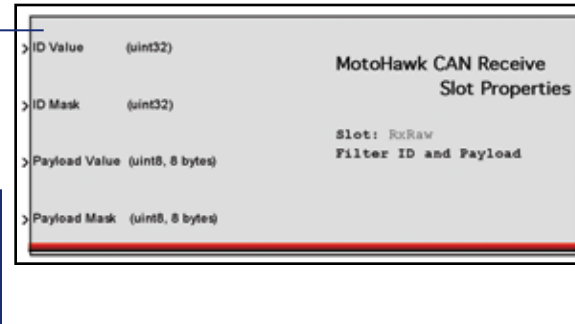
## Slot Properties

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 either 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.

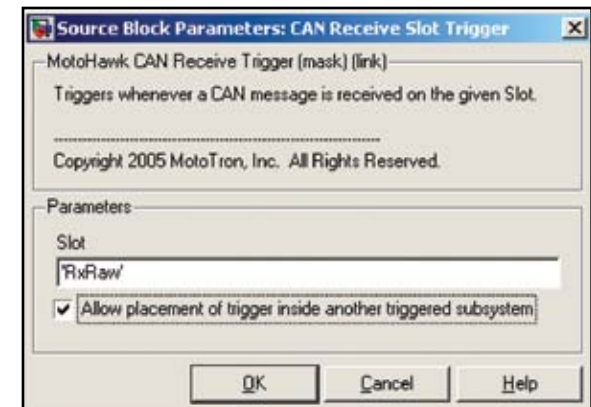
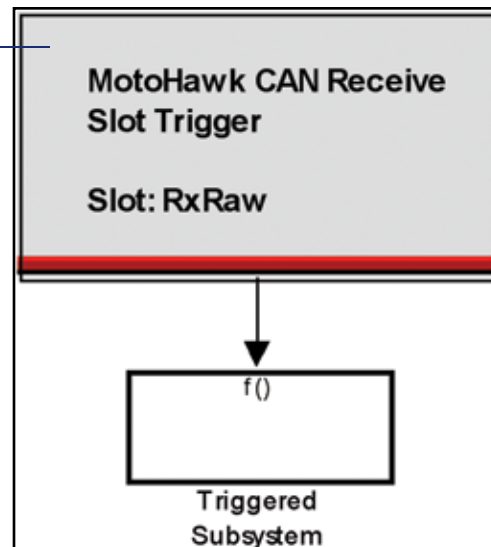
Usually this block is placed in a triggered subsystem, so that the slot properties are adjusted only on some conditions — such as at startup or on change of some state.



## 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

This example will demonstrate the basic CAN blocks.

Start with motohawk\_project Can1.

Remove the existing contents of the Foreground subsystem.

Create the model as shown – Build the model.

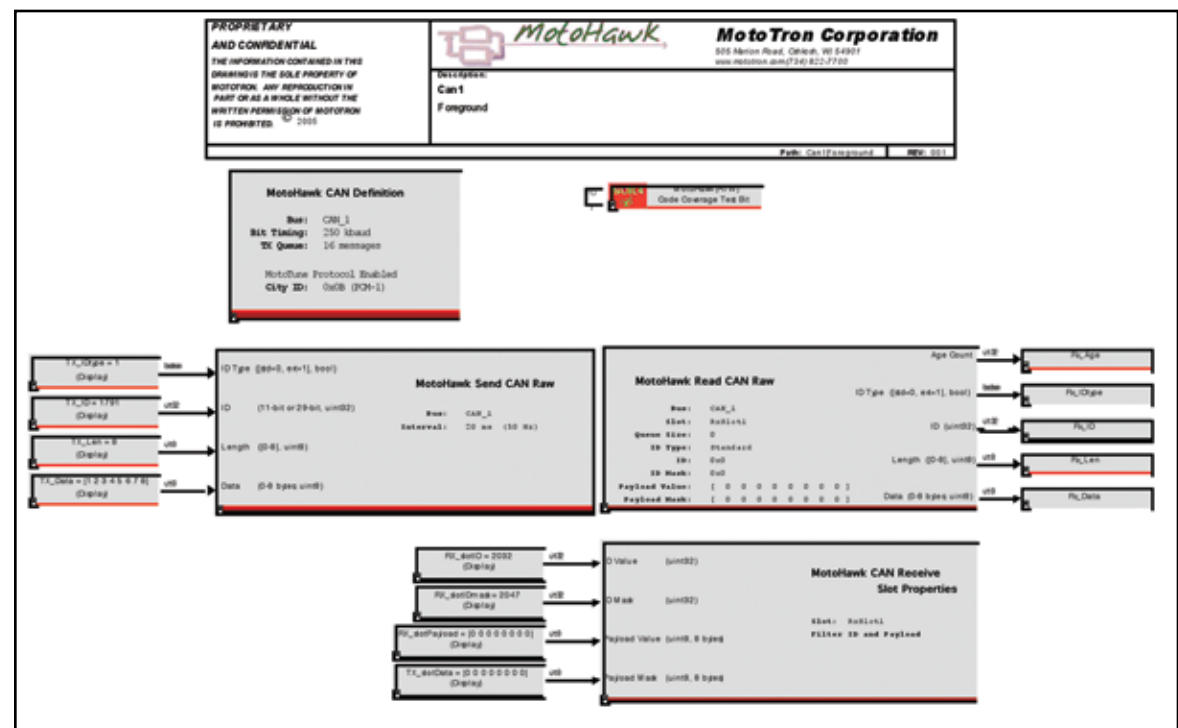
Run MotoTune, program the module, and open a display.

Run CANKing.

Right-click on the CANKing output window and select “Fixed Positions”.

In your MotoTune display – change the formatting of RX\_slotID, RX\_slotIDmask, RX\_ID, and TX\_ID to display hex.

\*For a larger view of drawing, open MotoHawk\_Resource\_Guide\_11x17\_drawings.pdf on the included training cd.



(example continued)

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

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.

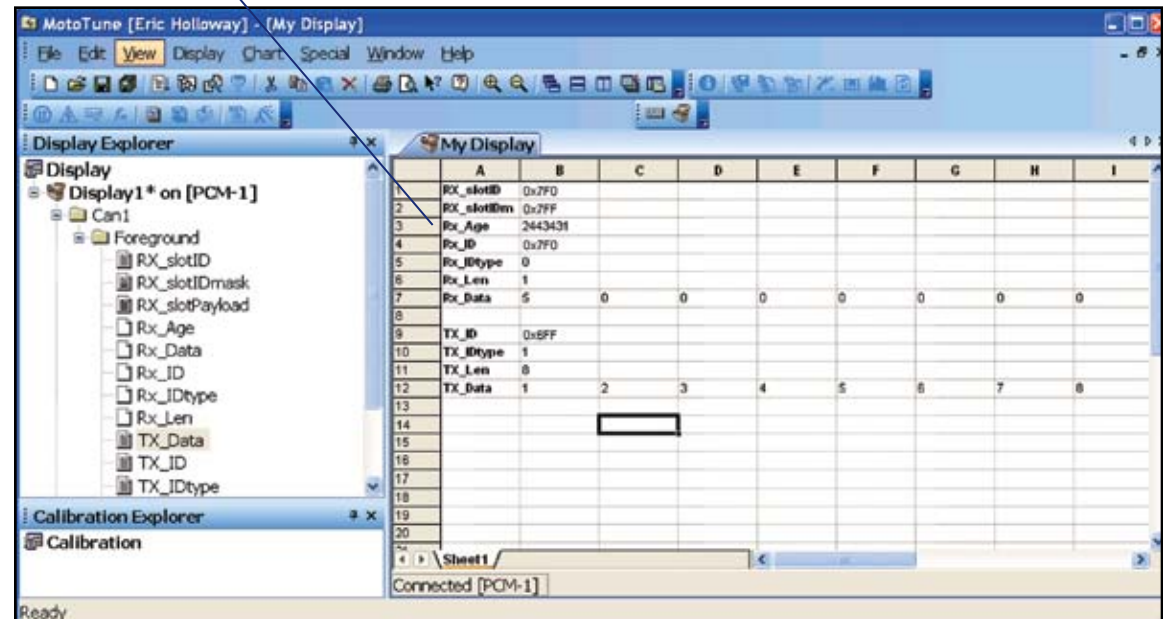
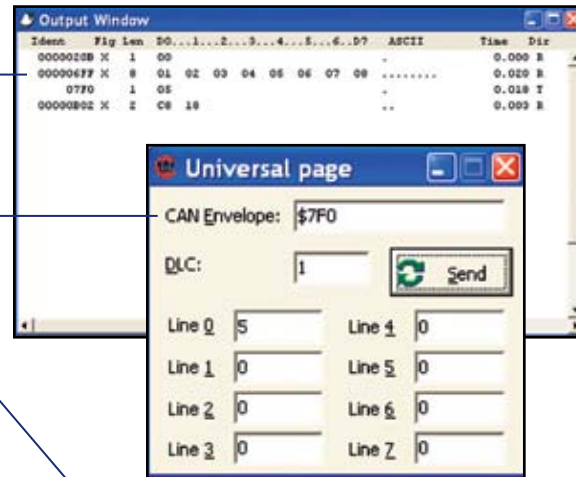
Adjust the slot ID and mask as well as the payload values to see how the messages are affected.

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.

The individual bytes of the payload may also be set using the \$ notation for hexadecimal.

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 to the right. 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.

### Extended ID Bit Numbering

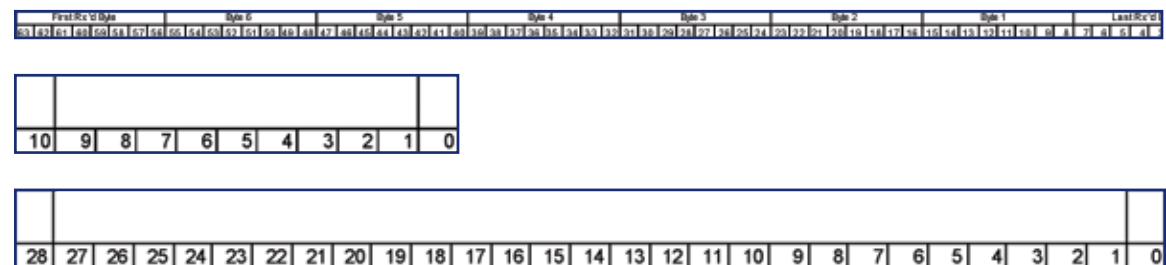
For extended IDs, the bit numbering is defined as shown.

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 the 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     - when set to 1, causes the message to use the ID of (default: 0)
                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.
```

(message definition structure continued)

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.

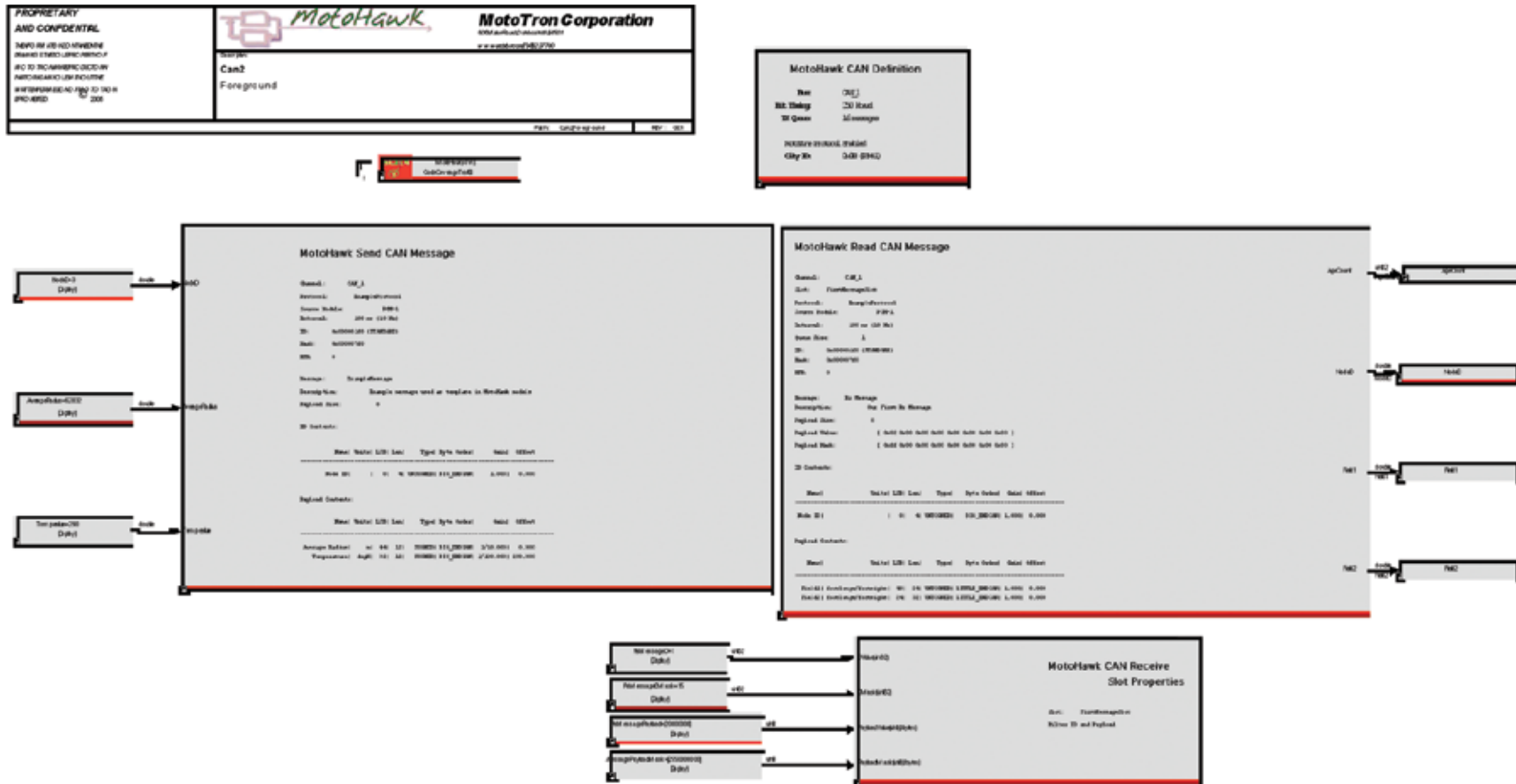
Structs in the `.idcontent{}` and `.fields{}` cell arrays may contain the following fields:

`.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

Create a new model using `motohawk_project('can2')`  
 Remove the existing contents of the Foreground subsystem  
 Create the model as shown. Build the model.  
 Run MotoTune, program the module, and open a display.  
 Run CANKing.

\*For a larger view of drawing, open `MotoHawk_Resource_Guide_11x17_drawings.pdf` on the included training cd.

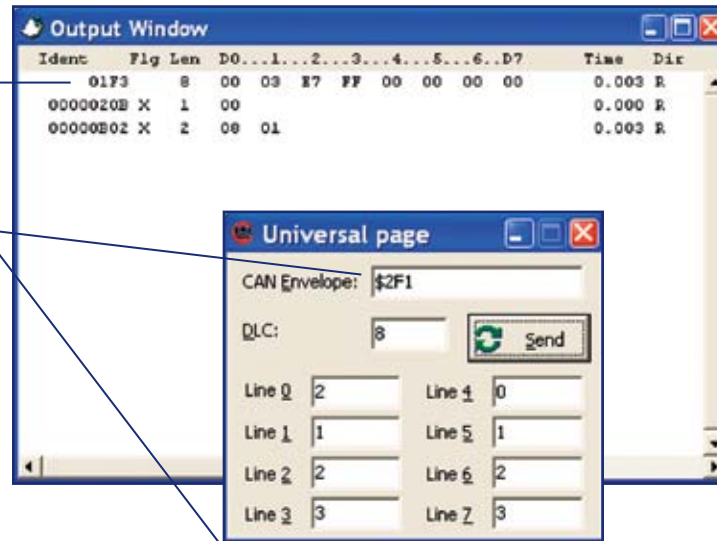




(example continued)

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

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

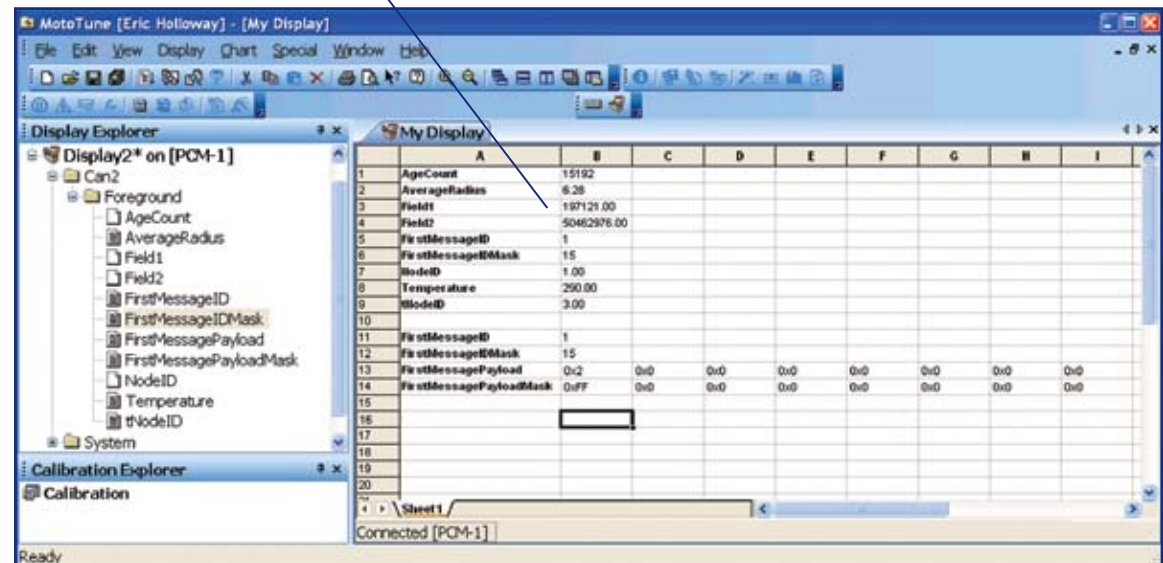


The image shows two overlapping windows from a diagnostic tool. The 'Output Window' displays a table of CAN messages:

Ident	Flg	Len	D0	1	2	3	4	5	6	D7	Time	Dir
01F3		8	00	03	E7	FF	00	00	00	00	0.003	R
0000020B	X	1	00								0.000	R
00000B02	X	2	08	01							0.003	R

The 'Universal page' window shows configuration for a CAN message:

- CAN Envelope: \$2F1
- DLC: 8
- Send button
- Line 0: 2, Line 1: 0, Line 2: 2, Line 3: 3, Line 4: 1, Line 5: 1, Line 6: 2, Line 7: 3



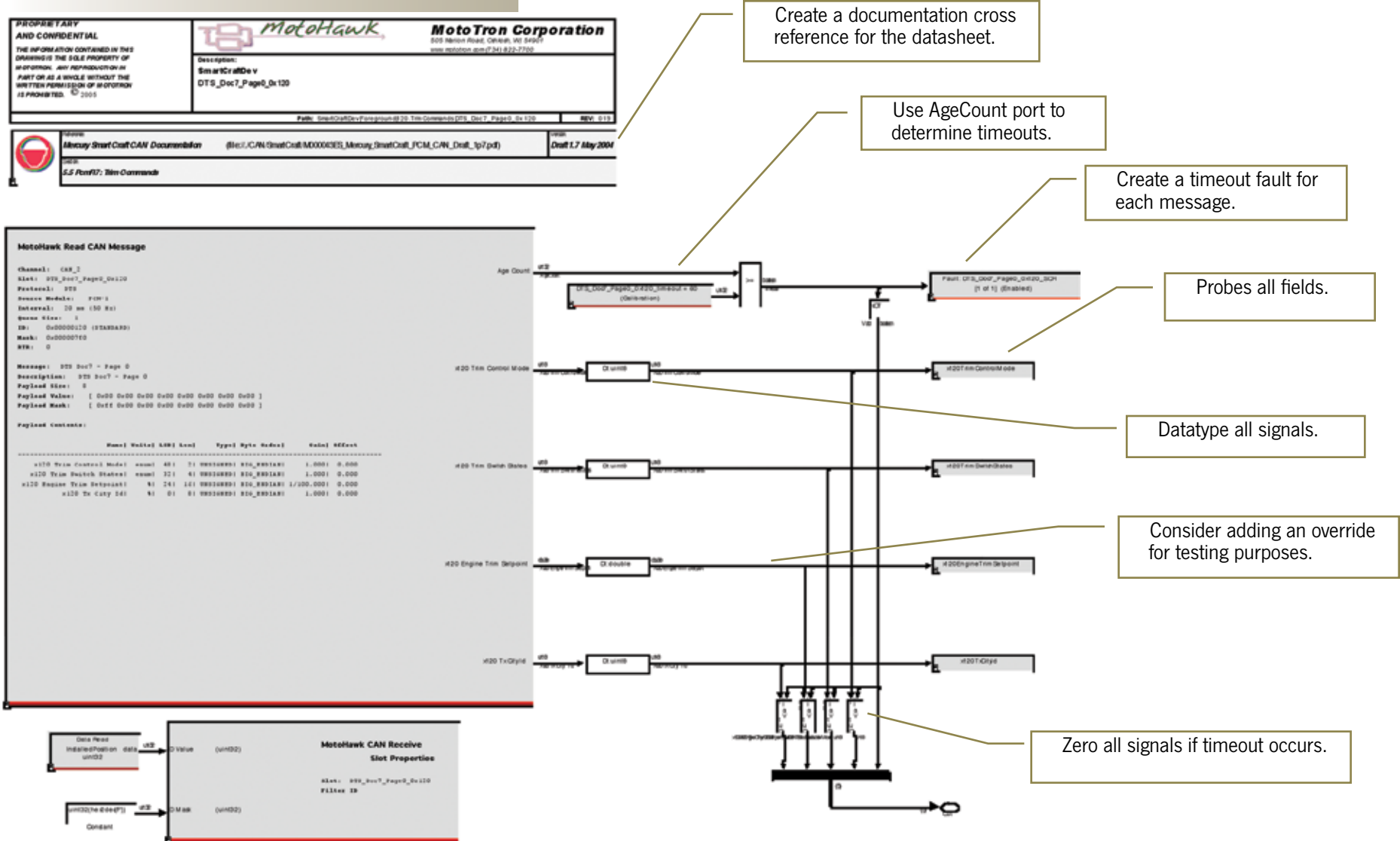
The image shows the MotoTune software interface. The 'Display Explorer' on the left shows a tree view for 'Can2' with various parameters. The main 'My Display' window shows a data table:

	A	B	C	D	E	F	G	H	I
1	AgeCount	15192							
2	AverageRadius	6.28							
3	Field1	197121.00							
4	Field2	50462976.00							
5	FirstMessageID	1							
6	FirstMessageIDMask	15							
7	NodeID	1.00							
8	Temperature	290.00							
9	NodeID	3.00							
10									
11	FirstMessageID	1							
12	FirstMessageIDMask	15							
13	FirstMessagePayload	0x2	0x0	0x0	0x0	0x0	0x0	0x0	0x0
14	FirstMessagePayloadMask	0xFF	0x0	0x0	0x0	0x0	0x0	0x0	0x0
15									
16									
17									
18									
19									
20									

The status bar at the bottom indicates 'Connected [PCM-1]'.

## Recommended Usage of CAN Message Receive Blocks

\*For a larger view of drawing, open [MotoHawk\\_Resource\\_Guide\\_11x17\\_drawings.pdf](#) on the included training cd.





## CHAPTER 4 : Memory Management

Variables types	1
Knowing your memory	2
Why so much different memory?	2
Knowing the hardware	2
Familiarize yourself with the interface	3
Block Parameters	3
Calibrations	6
Calibration : Default Value	7
Calibration : Output Data Type	7
Calibration : Access Levels	7
Calibration - Behavior	8
Calibration : View Value As	9
Calibration : MotoTune Help/Units	9
Calibration : MotoTune Min/Max	10
Calibration : MotoTune Precision - Gain/Offset/Exponent	11
Calibration : MotoTune Group	12
Probes	13
Probe : Name	13
Probe : Name Source	13
Probe : Read Access Level	14
Probe : View Value As	14
Probe – MotoTune Help / Units	14
Probe : MotoTune Precision Gain/Offset/Exponent	15
Probe – MotoTune Group	15
Overrides	16
Override : Name	16
Override : Name Source	16
Override : Override Access Level	17
Override : View Value As	17
Override : MotoTune Help/Units	18
Override : MotoTune Min/Max	18
Override : MotoTune Precision - Gain/Offset/Exponent	19
Override : MotoTune Group	20

# MEMORY MANAGEMENT

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 MotoTron Control Solutions 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 vardec's (Variable Declarations).

### ... What to remember and what not to remember .....

**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's MotoTron Control Solutions 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 can not 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 draw back 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's MotoTron Control Solutions modules come in two different versions.

**The development version has an added parallel EEPROM region where vardec's 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

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. 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.

## Block Parameters

MotoHawk has three basic blocks that allow viewable variables to appear in MotoTune: calibrations, displays, and probes.

Note that MotoTune makes probes a display variable, so a probe will appear in the display portion of MotoTune.

**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.

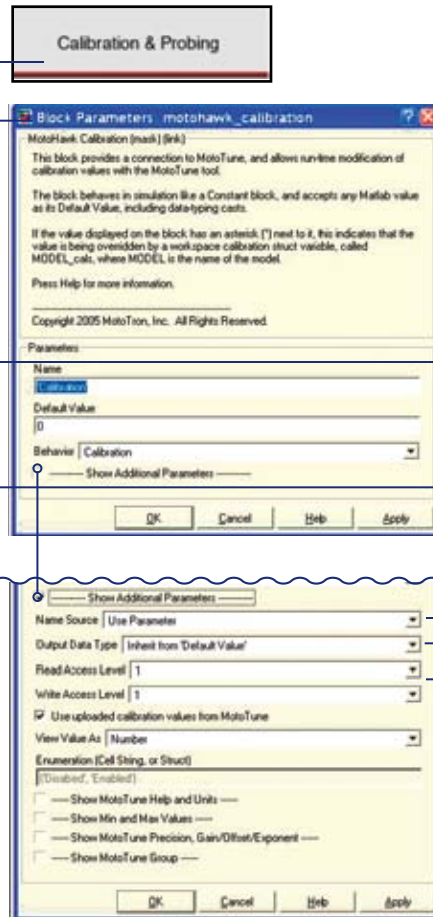
If a string is used, make sure to enclose the string in single quotes.

**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.

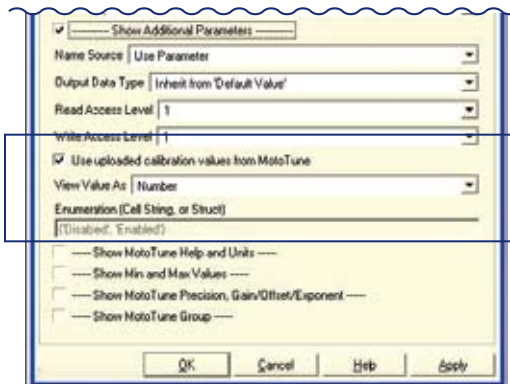
**Output Data Type :** By default, MATLAB makes all data types double. By not making a selection or specifying it in the “Default Value” field, then the output will default to double. Otherwise, you have two ways of specifying the data type. You can leave this field to “Inherit” from “Default Value” and enter the data type along with the “Default Value” field. (For instance, unit16(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.

## Block Parameters continued...



### 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.

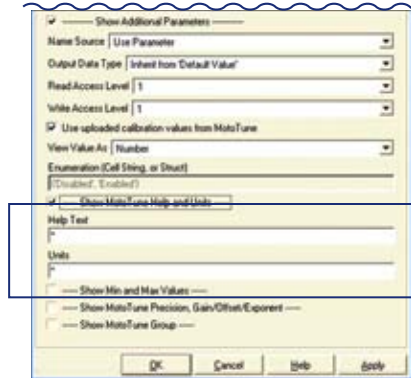
The MotoHawk upload calibration feature will make a MATLAB m-script for every defined vardec, but if a vardec is generated from a separate piece of software, you want your model to ignore the value located in the m-script file.

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.



### Help Text :

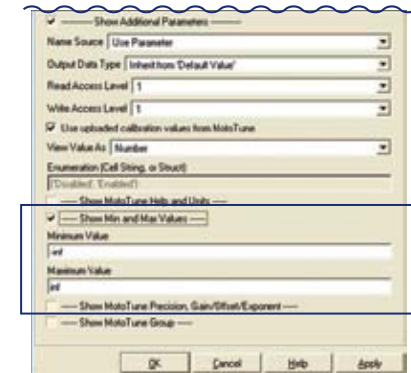
Text to aid the MotoTune users what this vardec does and what it might effect if changed. The text shows up automatically with calibrations. The help text and units automatically display with calibration values.

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

**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 :

Minimum Value for this vardec. This will clamp the signal in MotoTune.

If an attempt is made to go below this minimum value, then MotoTune will display a clamp value message and will force the value to this minimum value and no lower.

By default the minimum value is  $-\infty$  (-inf) to prevent MotoTune from clamping the value if it is changed.

### Maximum Value :

Maximum Value for this vardec. This will clamp the signal in MotoTune.

If an attempt is made to go above this maximum value, then MotoTune will display a clamp value message and will force the value to this maximum value and no higher.

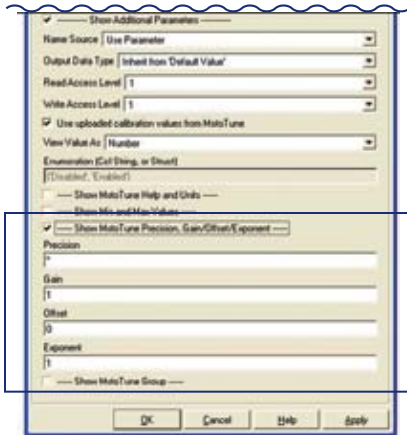
By default the maximum value is  $\infty$  (inf) to prevent MotoTune from clamping the value if it is changed.

### 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).



**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.

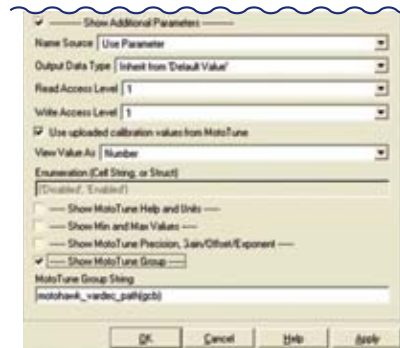
**Gain, Offset, Exponent** : These values only apply to how the variable will be displayed in MotoTune.

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 :

$$\text{MotoTuneValue} = (\text{value} * \text{gain})^{\text{exponent}} + \text{offset}$$

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.

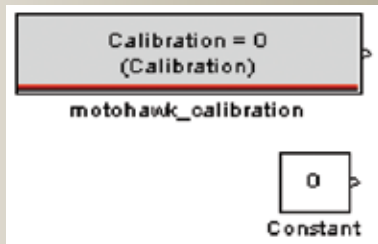
For instance, if you have a calibration in a model just under the foreground task in a model named example, then by default the calibration will be located under example/foreground/calibration in MotoTune.

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.

## 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.



## Calibration : Name

Because the name field is evaluated by MATLAB, the name field can accept any valid MATLAB expression that returns a string. Typically, that is useful when masking subsystems.

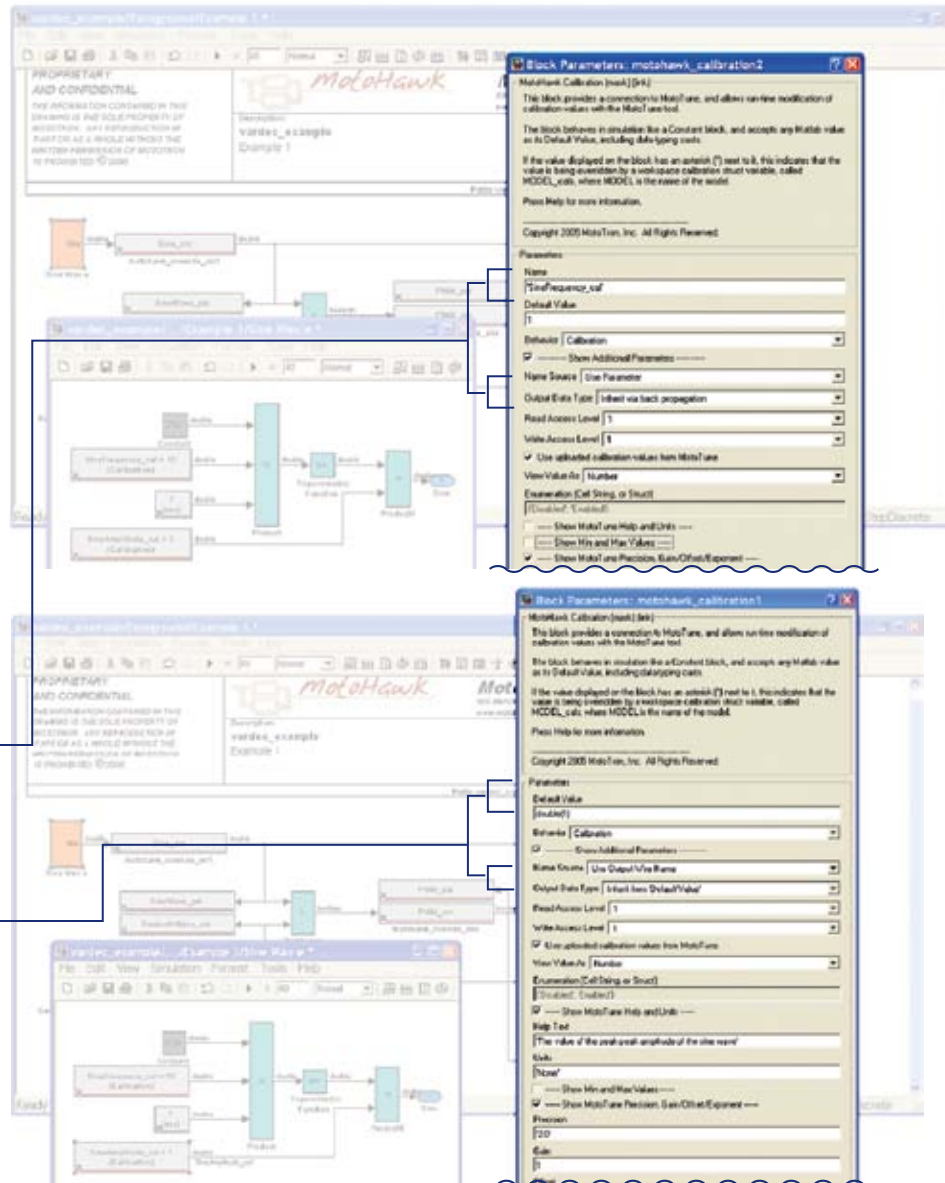
Most of the time, you will explicitly give the calibration a name, or use the output wire name as a reference for the name.

In example 1, the name for the sine wave frequency calibration is explicitly defined by selecting Name Source to be Use Parameter and providing a MATLAB string enclosed in single quotes in the name field.

## Calibration : Name Source

To make a calibration generic to the wire you attach it to, you can make this selection reference the output wire name as was done in the SineAmplitude\_cal block.

Notice that the name field no longer exists as it did in the SineFrequency\_cal block.





## Calibration : Default Value

Because the default value field is evaluated by MATLAB, the field can be any valid MATLAB expression that returns a number. This is the constant value this block will output unless calibrated by MotoTune to be something else. This can also be used to specify the data type.

## What are valid MATLAB expressions?

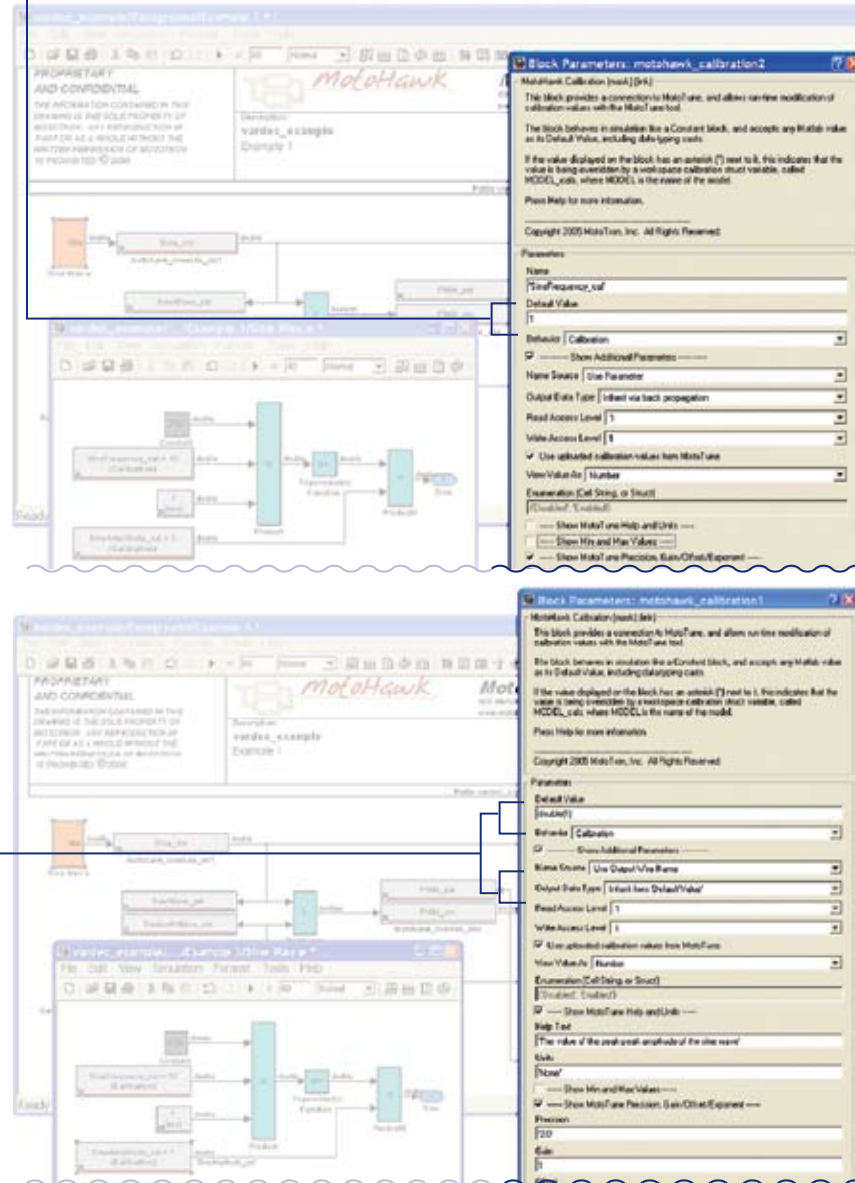
Matlab expressions can be workspace variables or MATLAB functions that return a number. Commonly, an m-script is made with calibration values stored by name. This m-script is called and all the calibrations are loaded in the workspace and the default values reference those values.

## Calibration : Output Data Type

Here is where to explicitly set the data type, or allow it to inherit via back propagation. Inherit via back propagation forces the data type decision based on how the calibration is used. By default, it inherits from the default value, which will be double if not specified.

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.

## Calibration : Access Levels



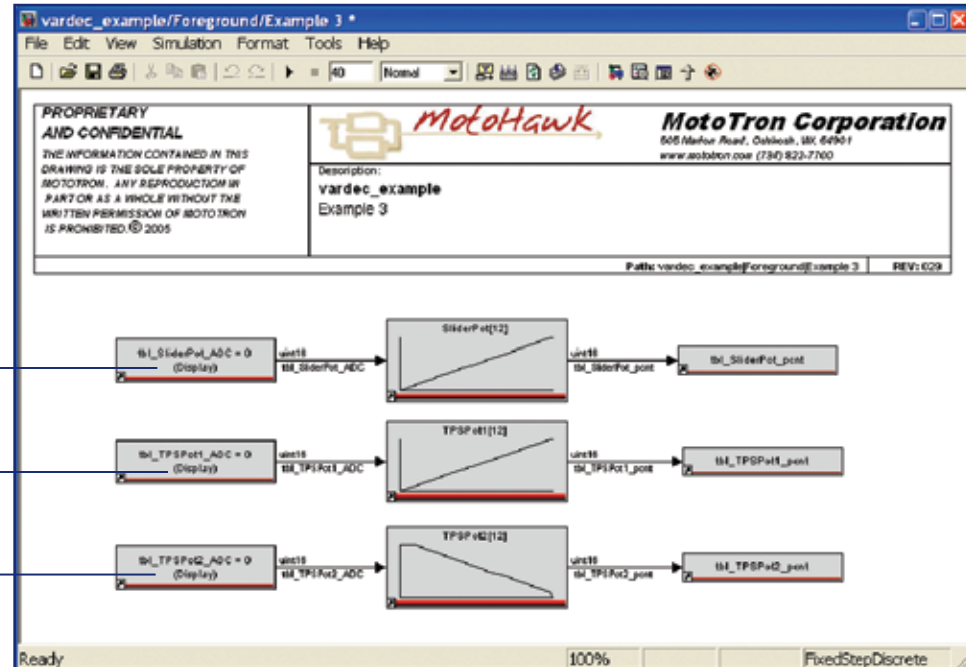
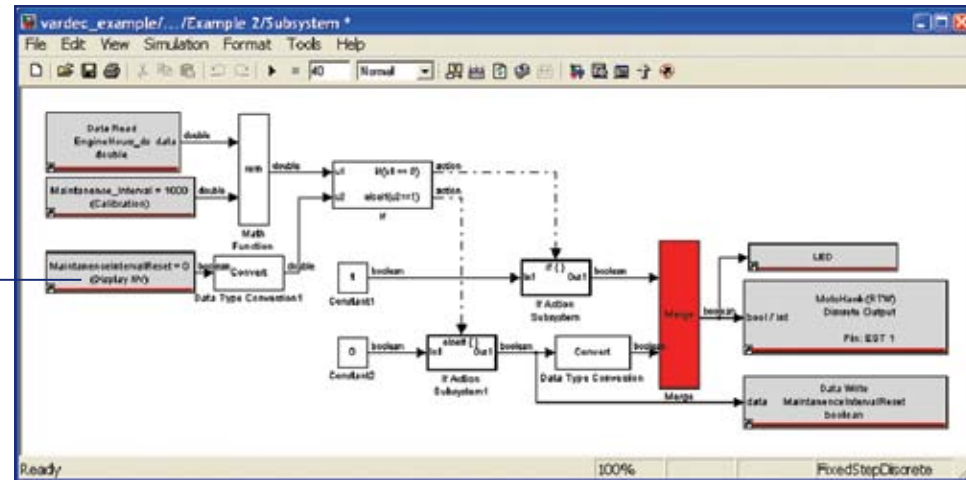
The image displays two screenshots of the Simulink Block Parameters dialog for 'MotoHawk Calibration' blocks. The top screenshot shows the 'Block Parameters: motohawk\_calibration2' dialog. The 'Default Value' field is set to '1'. The bottom screenshot shows the 'Block Parameters: motohawk\_calibration1' dialog. The 'Default Value' field is set to '(double)'. Both dialogs show various configuration options like 'Behaviors', 'Name Source', 'Output Data Type', and 'Read Access Level'.

## Calibration - Behavior

How will a user interact with this calibration? Will it be seen in the display or calibration window? Does it need to be accessible even on a production module?

In Example 2, \_\_\_\_\_  
 The Maintenance Interval Reset was specified as DisplayNV. This allows the value to be accessible, even on a production unit, so maintenance personnel can reset the alarm once maintenance has been performed.

In example3, \_\_\_\_\_  
 The calibrations are set to Display. The vardec's will still be stored in flash or parallel EEPROM, but will show up in the display pane of MotoTune and not the calibration pane.



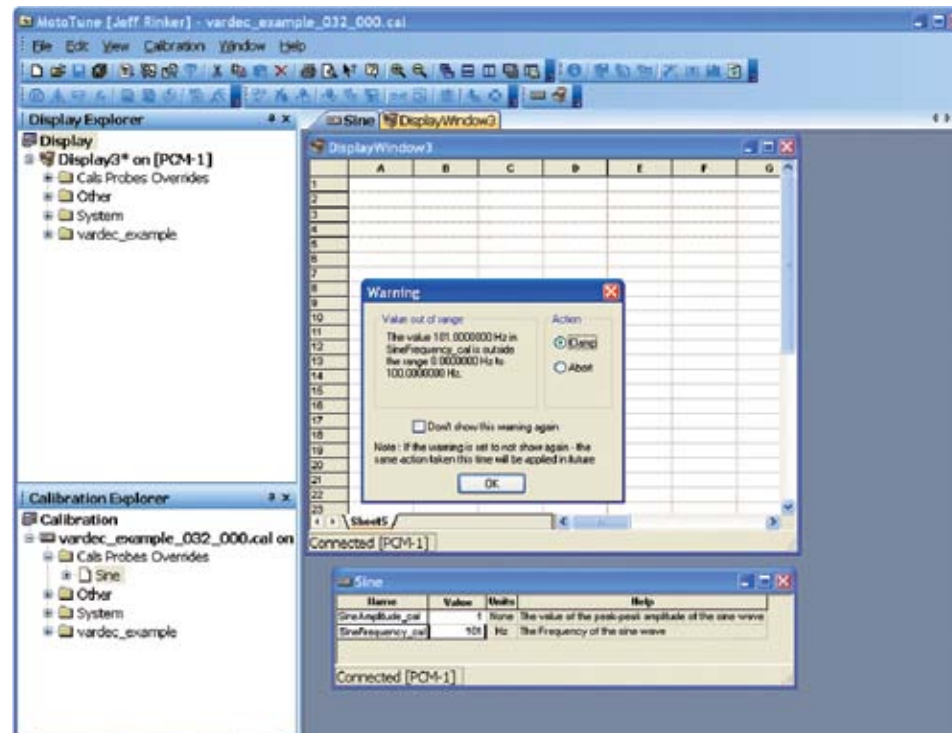
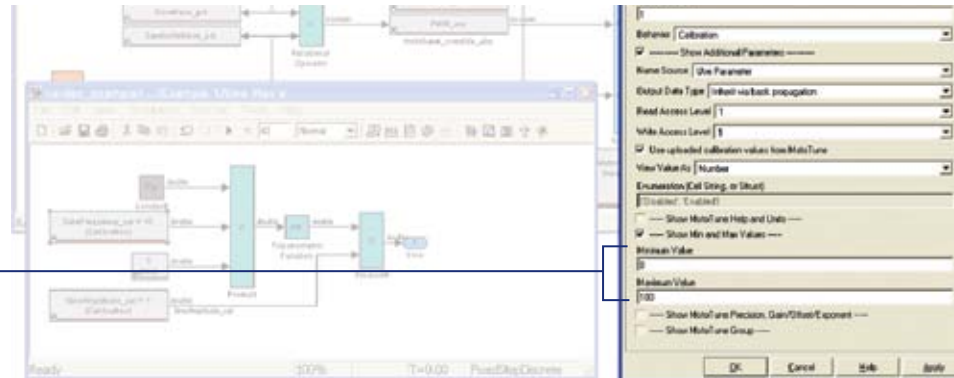


## Calibration : MotoTune Min/Max

In the SineFrequency\_cal a minimum and maximum was specified.

If you attempt to calibrate the value outside of the range, MotoTune will generate an error and inform the user it is outside the specified range.

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



## Calibration : MotoTune Precision Gain/Offset/Exponent

$$\text{MotoTuneValue} = (\text{value} * \text{gain})^{\text{exponent}} + \text{offset}$$

Precision is specified as a string or a valid MATLAB expression that returns a string in the format of width.decimal. The decimal will take precedence if the number becomes larger than the width specified on the left hand side of the decimal and will always maintain the specified decimal precision.

The width is specified to be 3 with 1 decimal precision. When the number is larger than 99.9, 1 decimal precision is maintained and the number increases to a width of 4 to display 100.0.

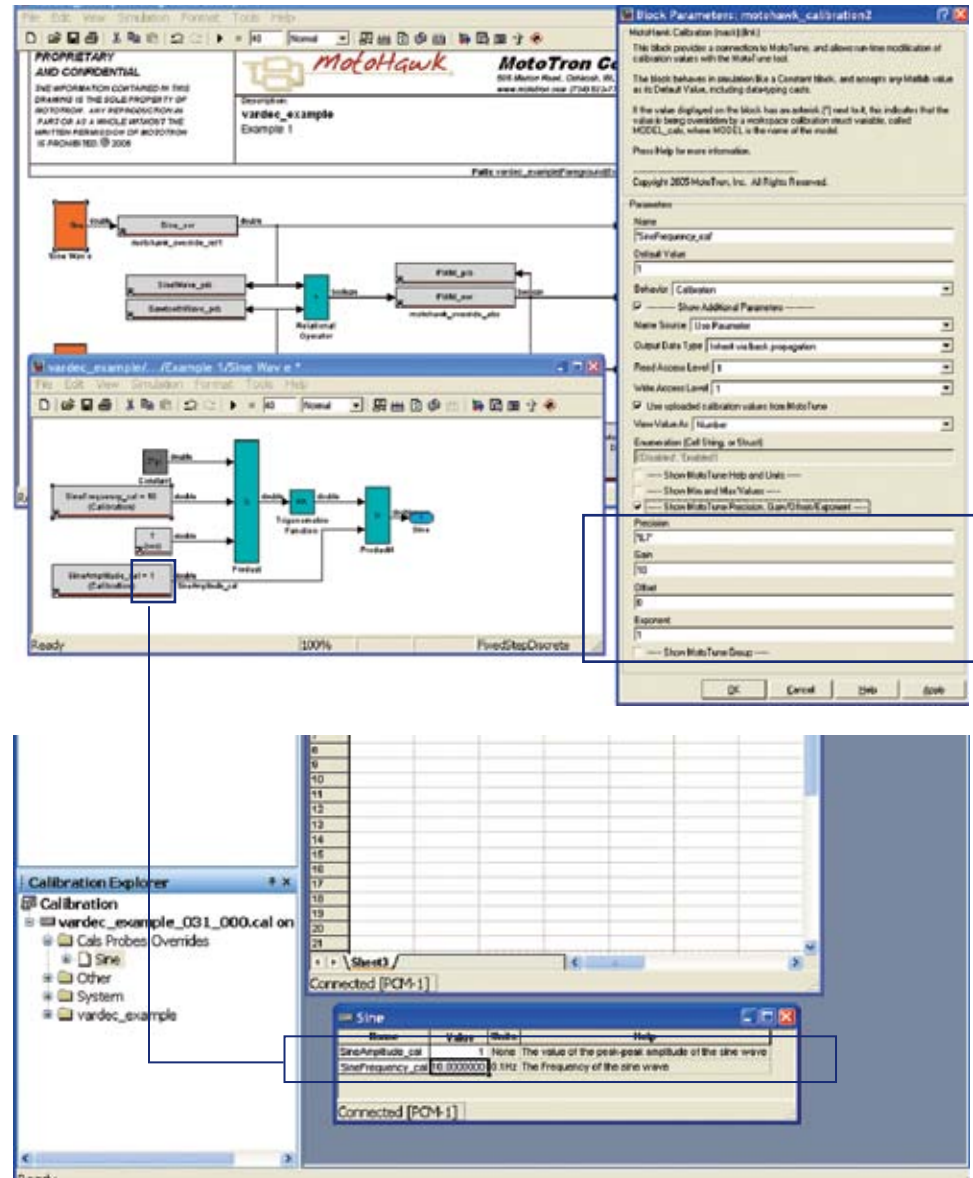
If the vardec is writable like a calibration, and the value is entered as 10.35, MotoTune will round the value up and display 10.4 to maintain the specified decimal precision. The rounding is merely for display purpose only and the value is actually 10.35. To see this you can change the precision in MotoTune to use 2 decimal precision and then the number would be displayed as 10.35. Changing variable precision from MotoTune is covered in the MotoTune chapter.

Gain/Offset/Exponent is used to display and calibrate the value differently than the actual use of the value down stream of the calibration block.

The default value is 1, but the value will be multiplied and scaled by the specified gain, offset, and exponent to be displayed in MotoTune as 10.

When changing the value in MotoTune — the above equation reverses.

In this case the value changed in MotoTune will be divided by 10.



The image displays several screenshots from the MotoTune software interface:

- Top Left:** A block diagram showing a signal flow from a sine wave input through a gain block, a summing junction with an offset, and an exponentiation block.
- Top Right:** A "Block Parameters" dialog box for a calibration block. It shows fields for Name, Gain, Offset, and Exponent. The Exponent field is highlighted with a blue box.
- Middle:** A "vardec\_example" block diagram showing a similar signal flow with a gain block, a summing junction, and an exponentiation block.
- Bottom Left:** A "Calibration Explorer" window showing a tree view of calibration files, including "vardec\_example\_031\_000.cal".
- Bottom Right:** A "Sine" block parameter dialog box showing a table of parameters:

Parameter	Value	Units	Help
SineAmplitude_cal	1	None	The value of the peak-great amplitude of the sine wave
SineFrequency_cal	0.000000	0.1Hz	The frequency of the sine wave

## Calibration : MotoTune Group

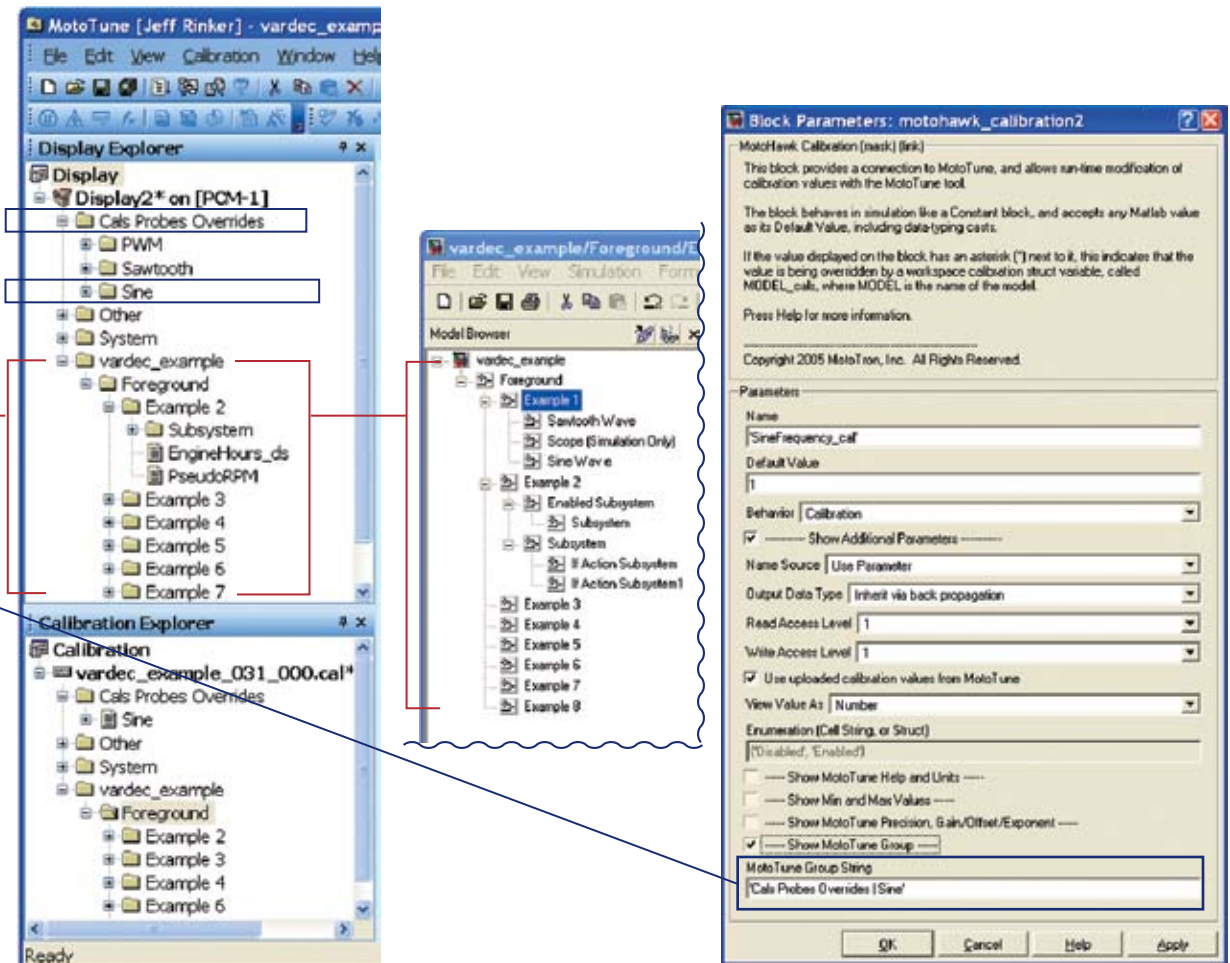
This is a valid Matlab expression that returns a string with folders separated by the vertical bar. It is used to organize system values into groupings that facilitate specific functions, such as in calibration or maintenance.

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.

Open Simulink's Model Browser (View/Model Browser Options/Model Browser)

Notice the similarity of the model browser's tree and the MotoTune tree. The vardec in this example have not yet been put into any specific MotoTune Groups.

You can also specify your own string with the MotoTune folder name separated by the vertical bar.



The screenshot displays three windows from the MotoTune tool:

- Display Explorer:** Shows a tree view of the model structure. The 'vardec\_example' folder is expanded, showing subfolders like 'Foreground' and 'Example 2'. Red boxes highlight the 'Cals Probes Overrides' and 'Sine' folders.
- Model Browser:** Shows a similar tree view for the 'vardec\_example' model, with 'Example 1' selected.
- Block Parameters: motohawk\_calibration2:** Shows the configuration for a calibration block. The 'MotoTune Group String' field is set to 'Cals Probes Overrides | Sine', which corresponds to the highlighted folders in the Display Explorer.

## 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. Meaning 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.

Remember – There are only 1,793 available vardec definitions. For 99% of the applications, this is more than enough room, but if your application uses many tables, the number of vardec's in your model can grow very rapidly.

### Probe : Name

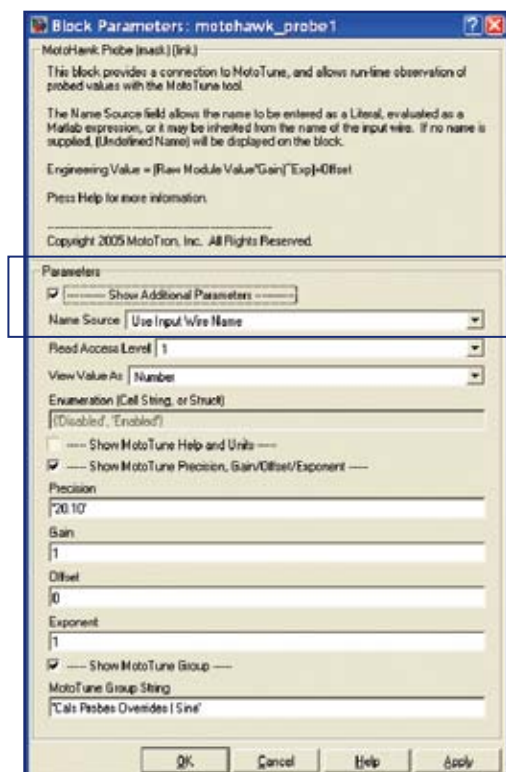
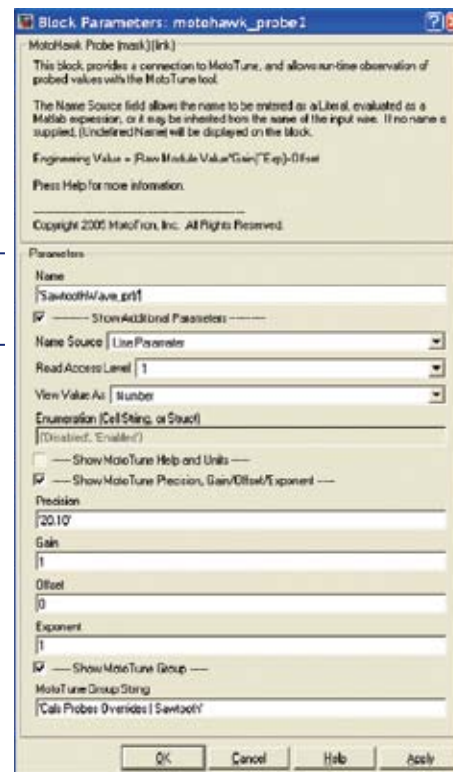
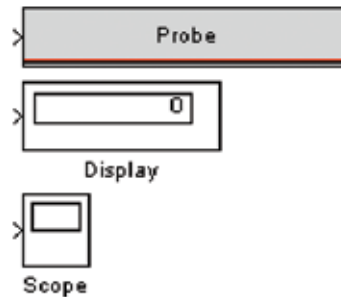
Because the name field is evaluated by Matlab, the name field can accept any valid Matlab expression that returns a string. Typically, this is valuable when masking subsystems. Most of the time you will explicitly give the probe a name or use the input wire name as a reference for the name.

In example1 of the vardec\_example.mdl, the sawtooth wave probe (SawtoothWave\_prb) has an explicitly defined name. The sine wave probe (SineWave\_prb) uses the input wire name as its name reference.

### Probe : Name Source

To make a probe generic to the wire you attach it to, you can make the block reference the input wire name as was done in the SineWave\_prb block.

Notice the name field no longer exists like in the SawtoothWave\_prb block.



### Probe : Read Access Level

Access level is a value from 1 through 4 (most open through most restricted.) There is a comparable access level written on each MotoTune dongle.

By default, MotoHawk dongles are given an access level of 4. This means that the dongle can access any vardec with access 4 or below. Since 4 is the highest, it can access everything in the model.

By default, access levels in the MotoHawk blocks are set to 1. This means that anyone with a dongle of access level 1 or above may access this block. Since 1 is the lowest, it can be accessed by everyone.

To restrict access with the MotoTune dongle, you will need to purchase dongles with access levels 1, 2, or 3 and change the appropriate access levels for each block with MotoTune interface.

A probe can only be read by a user, so there is only a read access level.

### Probe : View Value As

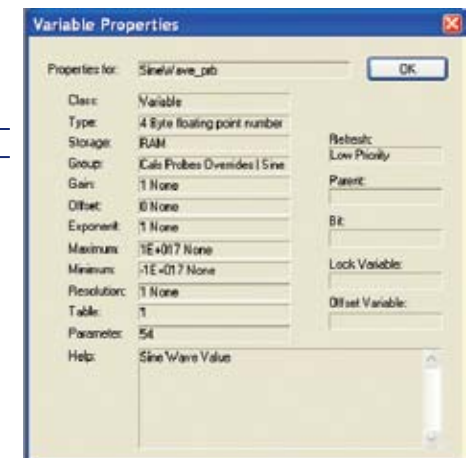
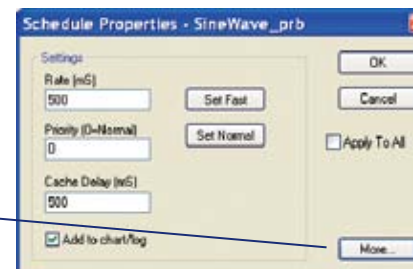
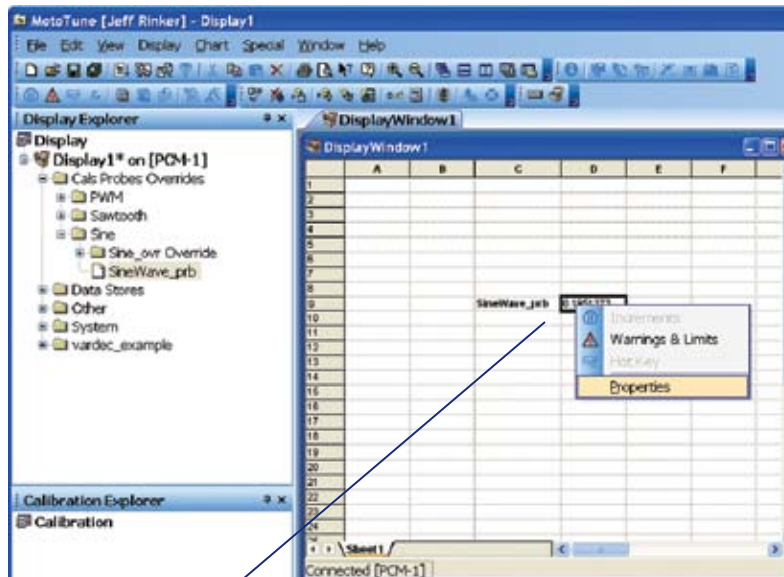
How will the MotoTune user view the data: text, number, or enumeration? Most probes will take on the form of a number, but enumerations can also be useful.

Only consider using an enumeration for a probe block when the wire you are probing has predictable values. If the input to the probe does not exist in the enumeration, the probe will display "undefined" in the MotoTune window.

### Probe – MotoTune Help / Units

Help and Unit information for MotoTune user.

For a probe, right click on the value of the probe and select properties. Another small window will appear, then hit "More" to display information about the probe.





## Probe : MotoTune Precision Gain/Offset/Exponent

$$\text{MotoTuneValue} = (\text{value} * \text{gain})^{\text{exponent}} + \text{offset}$$

Precision is specified as a string or a valid MATLAB expression that returns a string in the format of width.decimal. The decimal will take precedence if the number becomes larger than the width specified on the left hand side of the decimal and will always maintain the specified decimal precision.

The width is specified to be 3 with 1 decimal precision. When the number is larger than 99.9, 1 decimal precision is maintained and the number increases to a width of 4 to display 100.0.

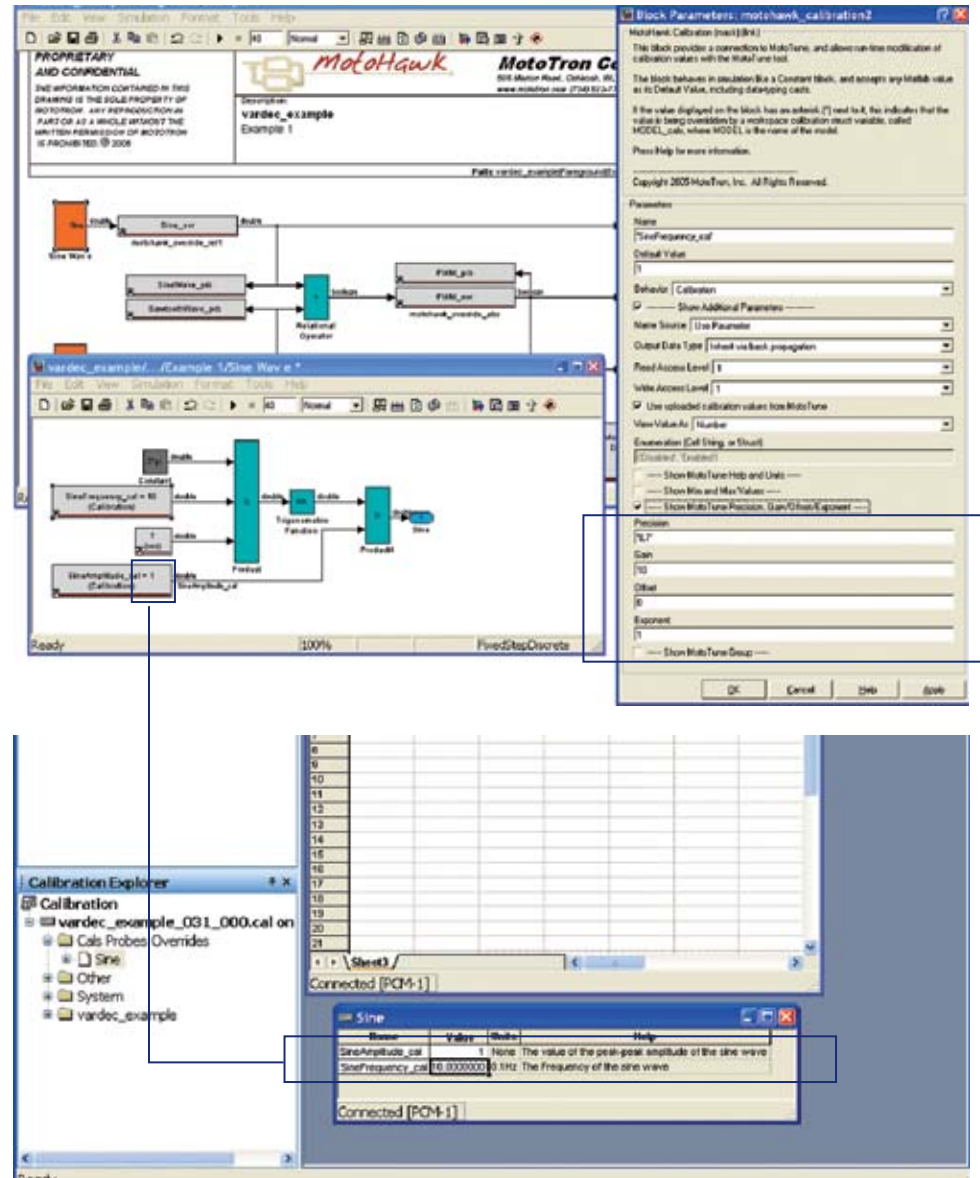
If the vardec is writable like a calibration, and the value is entered as 10.35, MotoTune would round the value up and display 10.4 to maintain the specified decimal precision. The rounding is merely for display purposes only and the value is actually 10.35. To see this you can change the precision in MotoTune to use 2 decimal precision and then the number would be displayed as 10.35. Changing MotoTune precision from MotuTune is covered in the MotoTune chapter.

Gain/Offset/Exponent is used to display and calibrate the value differently than the actual use of the value down stream of the calibration block. The default value is 1, but the value will be multiplied and scaled by the specified gain, offset, and exponent to be displayed in MotoTune as 10.

When changing the value in MotoTune — the above equation reverses. In this case the value changed in MotoTune will be divided by 10.

## Probe – MotoTune Group

This is a valid Matlab expression that returns a string with folders separated by the vertical bar. By default this value runs a function called motohawk\_vardec\_path(gcb). The location of each vardec will be the same as the model.



The image displays several screenshots from a Simulink environment and the MotoTune software interface. The top screenshot shows a Simulink model with a 'Motohawk' block and a 'vardec\_example' block. The middle screenshot shows the 'Block Parameters: motohawk\_calibration2' dialog box, which includes a 'Precision' field set to '10.35'. The bottom screenshot shows the 'Calibration Explorer' window, which lists various calibration files, including 'vardec\_example\_031\_000.cal'. The 'Sine' block parameters are also visible, showing 'SineAmplitude\_cal' set to 1 and 'SineFrequency\_cal' set to 10.000000.

If you turn on Simulink's Model Browser, you will notice the similarity of the model browser's tree and the MotoTune tree with the vardecs that have not yet been put into their respective MotoTune Groups. This is good if the Controls Engineer is also doing the testing, but for a calibrator, the levels may get too deep and you may want to specify a better, easier to use MotoTune layout. You can also specify your own string with the MotoTune folders separated by the horizontal bar.

## 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. Notice how under the PWM\_ovr Override\_Absolute block is two blocks with \_ovr and \_new appended to the base name given under the mask.

## Override : Name

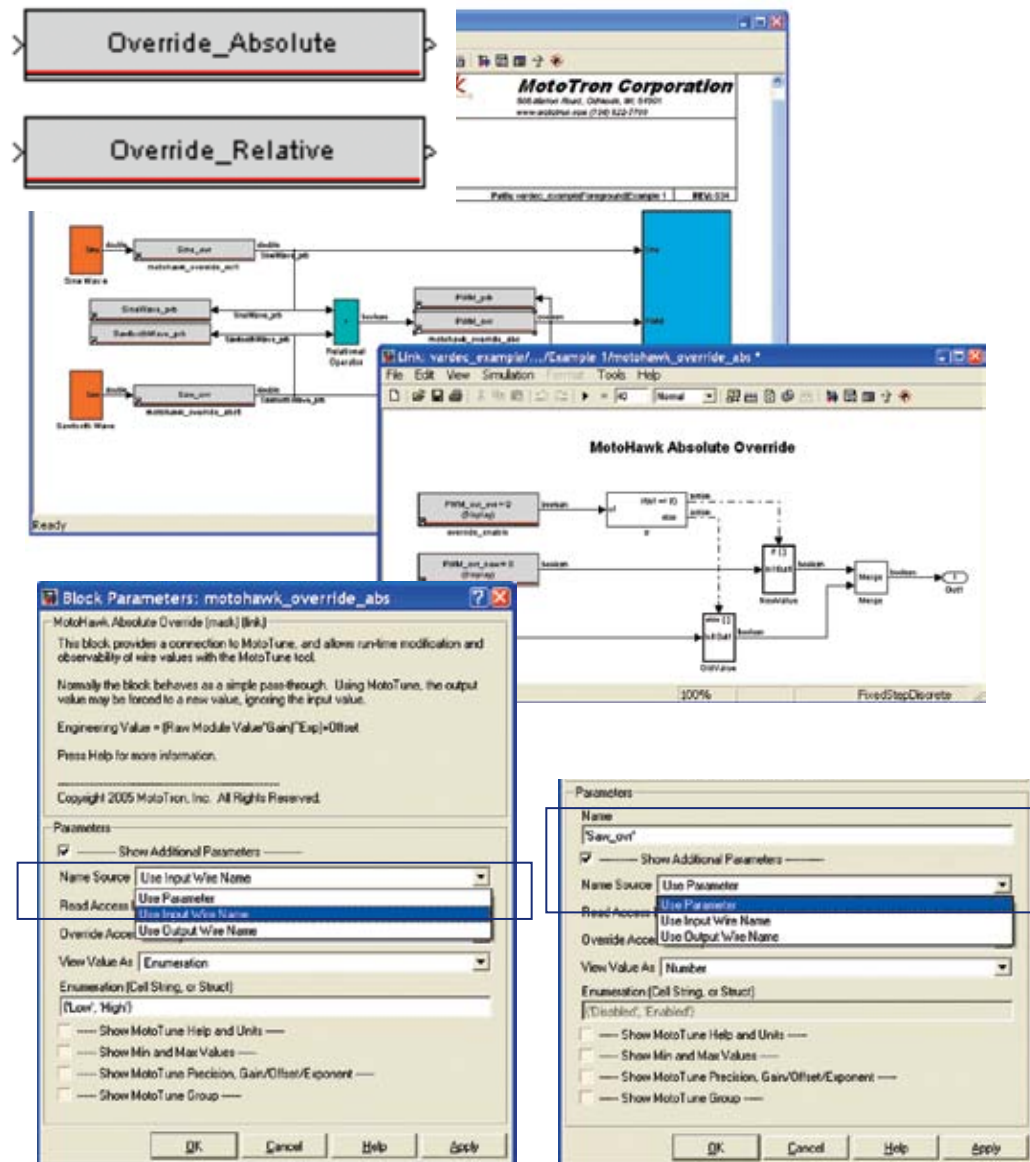
Because the name field is evaluated by MATLAB, the name field can accept any valid Matlab expression that returns a string. This is typically valuable when masking sub-systems.

## Override : Name Source

Similar to calibrations and probe blocks, the override blocks can inherit a name via the input or output wire name.

If you select either of these choices, the "Name" field disappears. Then, by specifying an appropriate name on the wire attached to the block, the name will change to this value.

When using the wire name to identify a vardec, avoid using spaces or special characters.





## Override : MotoTune Help/Units

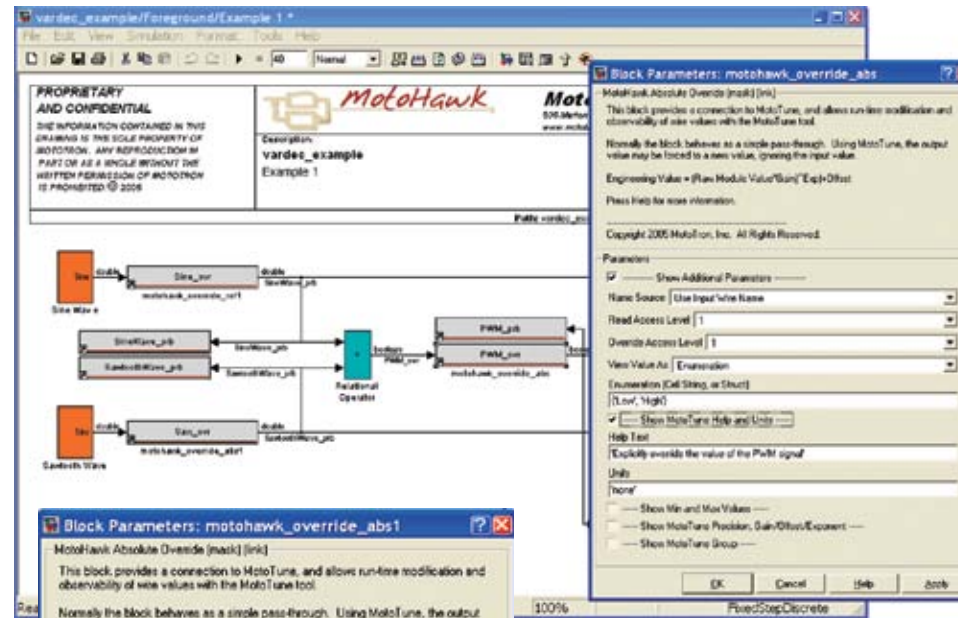
Help and Unit information for MotoTune user.

To display help and unit information for an override, right click on the value, select “Properties / More” to view the help and unit information.

## Override : MotoTune Min/Max

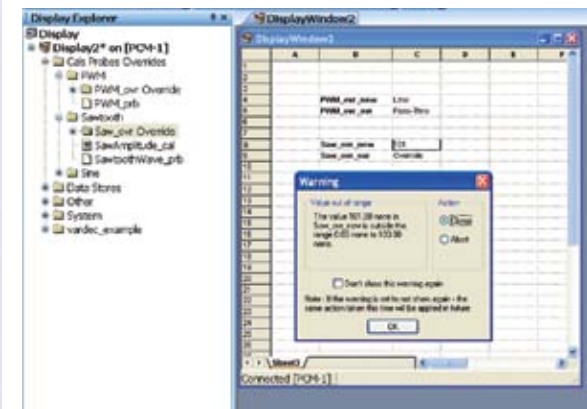
In the PWM\_ovr, a minimum and maximum was specified. In an attempt to calibrate the value outside of the range, MotoTune will generate an error and inform the user it is outside the specified range.

Useful to ensure a calibration is not accidentally changed outside of a range.



The screenshot shows the MotoHawk software interface. The main window displays a block diagram with several blocks connected by lines. A dialog box titled "Block Parameters: motohawk\_override\_abs1" is open, showing the following details:

- Block Name:** motohawk\_override\_abs1
- Description:** This block provides a connection to MotoTune, and allows run-time modification and observability of raw values with the MotoTune tool.
- Engineering Value:**  $(Raw\ Module\ Value * Gain) * ExpOffset$
- Parameters:**
  - Show Additional Parameters
  - Name Source: Use Input's Name
  - Read Access Level: 1
  - Override Access Level: 1
  - View Value As: Enumeration
  - Enumeration (Cell String, or Struct): [Low, High]
  - Show MotoTune Help and Units
- Help Text:** Explicitly override the value of the PWM signal.
- Units:** [None]
- Buttons:** OK, Cancel, Help, Auto



The screenshot shows the Display Explorer and DisplayWindow2. The Display Explorer shows a tree view of the system components. The DisplayWindow2 shows a table of data with columns A, B, C, D, E, F. A warning dialog box is displayed over the table, with the following text:

**Warning**

Value out of range  
The value 101.08 is in  
low, you should use the  
range 0.00 to 132.36  
max.

Buttons: OK, Abort

Checkbox:  Don't show this warning again

Note: If the warning is not set then again the same action over the time will be applied to data

## Override : MotoTune Precision Gain/Offset/Exponent

$$\text{MotoTuneValue} = (\text{value} * \text{gain})^{\text{exponent}} + \text{offset}$$

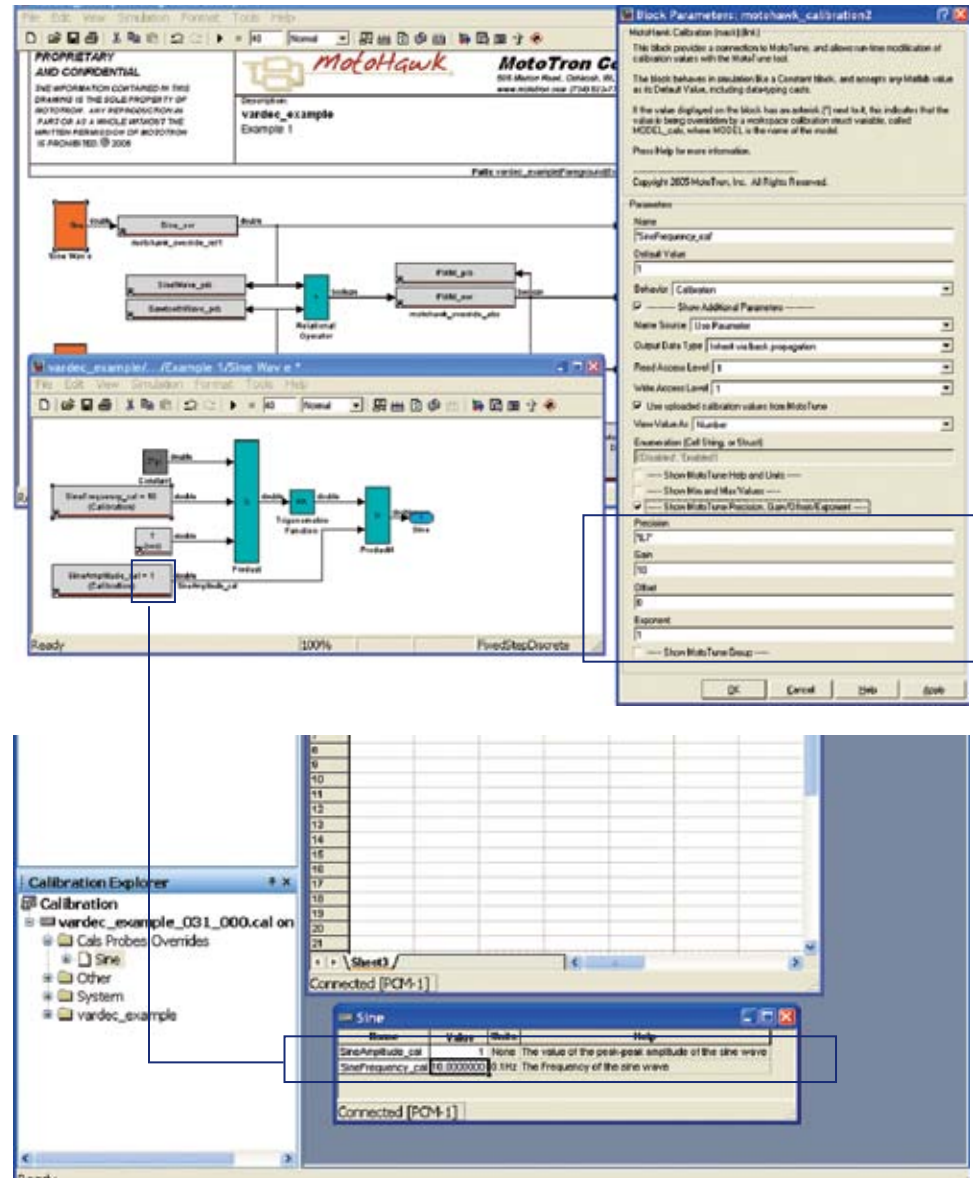
Precision is specified as a string or a valid MATLAB expression that returns a string in the format of width.decimal. The decimal will take precedence if the number becomes larger than the width specified on the left hand side of the decimal and will always maintain the specified decimal precision.

The width is specified to be 3 with 1 decimal precision. When the number is larger than 99.9, 1 decimal precision is maintained and the number increases to a width of 4 to display 100.0. (Fig12)

If the vardec is writable like a calibration, and the value is entered as 10.35, MotoTune would round the value up and display 10.4 to maintain the specified decimal precision. The rounding is merely for display purposes only and the value is actually 10.35. To see this you can change the precision in MotoTune to use 2 decimal precision and then the number would be displayed as 10.35. Changing MotoTune precision from MotoTune is covered in the MotoTune chapter.

Gain/Offset/Exponent is used to display and calibrate the value differently than the actual use of the value down stream of the calibration block. The default value is 1, but the value will be multiplied and scaled by the specified gain, offset, and exponent to be displayed in MotoTune as 10.

When changing the value in MotoTune — the above equation reverses. In this case the value changed in MotoTune will be divided by 10.



The screenshot displays the Simulink environment with a 'vardec\_example' block and its associated 'Block Parameters: motehawk\_calibration2' dialog. The dialog shows parameters for 'Precision', 'Gain', 'Offset', and 'Exponent'. A 'Sine' block is also visible, with its parameters set to 'SineAmplitude\_cal' = 1 and 'SineFrequency\_cal' = 0.000000. Below the dialog, a 'Calibration Explorer' window shows the 'vardec\_example\_031\_000.cal' file, and a 'Sine' block is shown with its parameters set to 'SineAmplitude\_cal' = 1 and 'SineFrequency\_cal' = 0.000000.

Time	Value	Units	Help
0	1	None	The value of the peak-peak amplitude of the sine wave
10	0.000000	Hz	The frequency of the sine wave

## Override : MotoTune Group

This is a valid Matlab expression that returns a string with folders separated by the vertical bar. By default this value runs a function called `motohawk_vardec_path(gcb)`. The location of each vardec will be the same as the model.

If you turn on Simulink's Model Browser, you will notice the similarity of the model browser's tree and the MotoTune tree with the vardec's that have not yet been put into their respective MotoTune Groups.

The default value for the MotoTune group is good for developers that are familiar with the code. To make it easier for calibrators, test engineers, or other individuals who will use MotoTune, but is unfamiliar with the actual code, you will want to specify a better, easier to use, layout.

You can also specify your own string with the MotoTune folders separated by the horizontal bar.

