HOW-TO TUTORIAL

Altair Embed® 2025.2

Common Simulation and Code Generation Tasks

Intellectual Property Rights Notice:

This Intellectual Property Rights Notice is exemplary, and therefore not exhaustive, of the intellectual property rights held by Altair Engineering Inc. or its affiliates. Software, other products, and materials of Altair Engineering Inc. or its affiliates are protected under laws of the United States and laws of other jurisdictions.

In addition to intellectual property rights indicated herein, such software, other products, and materials of Altair Engineering Inc. or its affiliates may be further protected by patents, additional copyrights, additional trademarks, trade secrets, and additional other intellectual property rights. For avoidance of doubt, copyright notice does not imply publication. Copyrights in the below are held by Altair Engineering Inc. or its affiliates. Additionally, all non-Altair marks are the property of their respective owners. If you have any questions regarding trademarks or registrations, please contact marketing and legal.

This Intellectual Property Rights Notice does not give you any right to any product, such as software, or underlying intellectual property rights of Altair Engineering Inc. or its affiliates. Usage, for example, of software of Altair Engineering Inc. or its affiliates is governed by and dependent on a valid license agreement.

Altair® HyperWorks®, a Design & Simulation Platform
Altair® AcuSolve® ©1997-2025
Altair® Activate® ©1989-2025
Altair® Automated Reporting Director™ ©2008-2022
Altair® Battery Damage Identifier™ ©2019-2025
Altair® CFD™ ©1990-2025
Altair Compose® ©2007-2025
Altair® ConnectMe™ ©2014-2025
Altair® DesignAI™ ©2022-2025
Altair® DSim® ©2024-2025
Altair® DSim® Cloud ©2024-2025
Altair® DSim® Cloud CLI ©2024-2025
Altair® DSim® Studio ©2024-2025
Altair® EDEM™ ©2005-2025
Altair® EEvision™ ©2018-2025
Altair® ElectroFlo™ ©1992-2025
Altair Embed® ©1989-2025
Altair Embed® SE ©1989-2025
Altair Embed®/Digital Power Designer ©2012-2025
Altair Embed®/eDrives ©2012-2025
Altair Embed® Viewer ©1996-2025
Altair® e-Motor Director™ ©2019-2025
Altair® ESAComp® ©1992-2025
Altair® expertAI™ ©2020-2025
Altair® Feko® ©1999-2025
Altair® FlightStream® ©2017-2025
Altair® Flow Simulator™ ©2016-2025
Altair® Flux® ©1983-2025
Altair® FluxMotor® ©2017-2025
Altair® GateVision PRO™ ©2002-2025
Altair® Geomechanics Director™ ©2011-2022
Altair® HyperCrash® ©2001-2023
Altair® HyperGraph® ©1995-2025
Altair® HyperLife® ©1990-2025
Altair® HyperMesh® ©1990-2025
Altair® HyperMesh® CFD ©1990-2025
Altair® HyperMesh ® NVH ©1990-2025
Altair® HyperSpice™ ©2017-2025
Altair® HyperStudy® ©1999-2025
Altair® HyperView® ©1999-2025
Altair® HyperView Player® ©2022-2025
Altair® HyperWorks® ©1990-2025
Altair® HyperWorks® Design Explorer ©1990-2025
Altair® HyperXtrude® ©1999-2025
Altair® Impact Simulation Director™ ©2010-2022

Altair® HPCWorks®, a HPC & Cloud Platform

Altair® Allocator™ ©1995-2025
Altair® Access™ ©2008-2025
Altair® Accelerator™ ©1995-2025
Altair® Accelerator™ Plus ©1995-2025
Altair® Breeze™ ©2022-2025
Altair® Cassini™ ©2015-2025
Altair® Control™ ©2008-2025
Altair® Desktop Software Usage Analytics™ (DSUA) ©2022-2025
Altair® FlowTracer™ ©1995-2025
Altair® Grid Engine® ©2001, 2011-2025
Altair® InsightPro™ ©2023-2025
Altair® InsightPro™ for License Analytics ©2023-2025
Altair® Hero™ ©1995-2025
Altair® Liquid Scheduling™ ©2023-2025
Altair® Mistral™ ©2022-2025
Altair® Monitor™ ©1995-2025
Altair® NavOps® ©2022-2025
Altair® PBS Professional® ©1994-2025
Altair® PBS Works™ ©2022-2025
Altair® Simulation Cloud Suite (SCS) ©2024-2025
Altair® Software Asset Optimization (SAO) ©2007-2025
Altair® Unlimited™ ©2022-2025
Altair® Unlimited Data Analytics Appliance™ ©2022-2025
Altair® Unlimited Virtual Appliance™ ©2022-2025

Altair® RapidMiner®, a Data Analytics & AI Platform

Altair® AI Hub ©2023-2025
Altair® AI Edge™ ©2023-2025
Altair® AI Cloud ©2022-2025
Altair® AI Studio ©2023-2025
Altair® Analytics Workbench™ ©2002-2025
Altair® Graph Lakehouse™ ©2013-2025
Altair® Graph Studio™ ©2007-2025
Altair® Knowledge Hub™ ©2017-2025
Altair® Knowledge Studio® ©1994-2025
Altair® Knowledge Studio® for Apache Spark ©1994-2025
Altair® Knowledge Seeker™ ©1994-2025
Altair® IoT Studio™ ©2002-2025
Altair® Monarch® ©1996-2025
Altair® Monarch® Classic ©1996-2025
Altair® Monarch® Complete™ ©1996-2025
Altair® Monarch® Data Prep Studio ©2015-2025
Altair® Monarch Server™ ©1996-2025
Altair® Panopticon™ ©2004-2025
Altair® Panopticon™ BI ©2011-2025
Altair® SLC™ ©2002-2025
Altair® SLC Hub™ ©2002-2025
Altair® SmartWorks™ ©2002-2025
Altair® RapidMiner® ©2001-2025

Altair One® ©1994-2025
Altair® CoPilot™ ©2023-2025
Altair® Drive™ ©2023-2025
Altair® License Utility™ ©2010-2025
Altair® TheaRender® ©2010-2025
OpenMatrix™ ©2007-2025
OpenPBS® ©1994-2025
OpenRadioss™ ©1986-2025

Contents

Introduction

This tutorial provides step-by-step instructions for performing common simulation and code generation tasks that will improve your Embed skills. Regardless of whether you are using Embed Pro, Embed SE, or Embed Personal, this tutorial refers to the product as Embed.

Simulation tasks

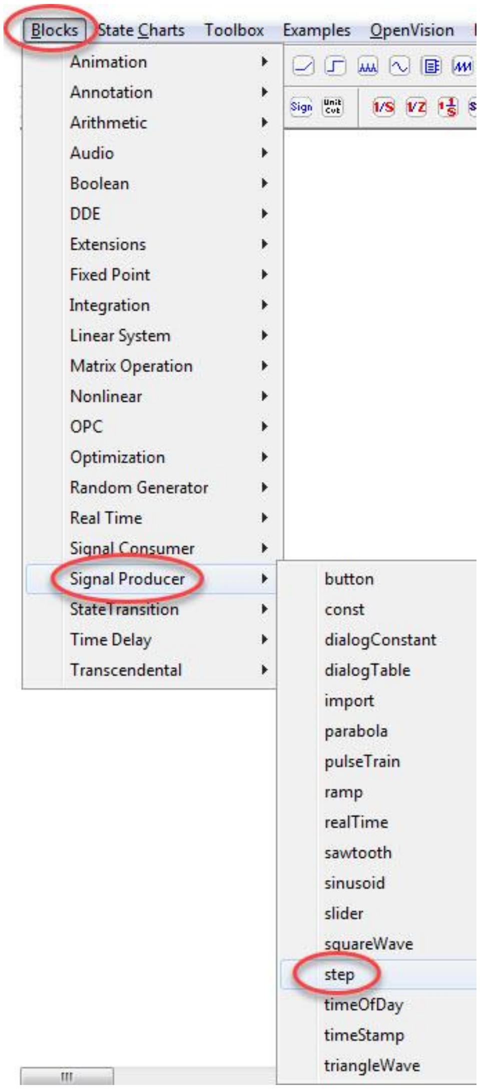
Building and simulating a second-order system

Inserting blocks

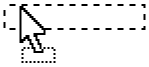
To construct a second-order system, you use a step block, two integrator blocks, and a plot block.

There are several ways to insert blocks into a diagram: from the Blocks menu, the Blocks and Diagram Browser, or the toolbar. This procedure shows how to insert blocks from the Blocks menu.

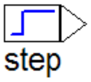
1. Choose **Blocks > Signal Producer** and click **step**.



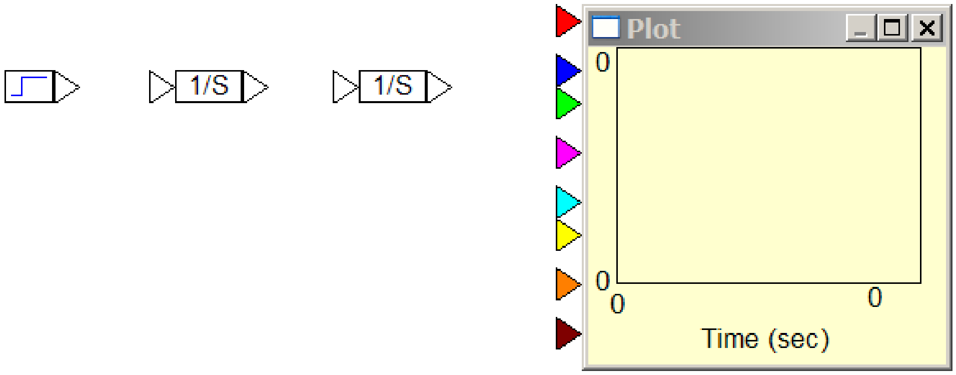
2. The **Blocks** menu disappears, and the pointer appears with a marquee attached to it.



3. Move the pointer to the work area and click to add the **step** block.

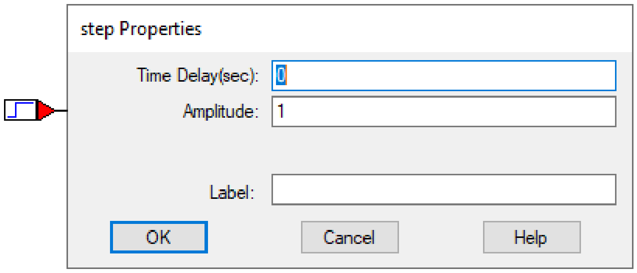


4. Repeat these steps to add two **integrator** blocks (**Blocks > Integration**) and a **plot** block (**Blocks > Signal Consumer**).



Setting block properties

Most blocks have properties that let you set attributes specific to the block. Right-click over the **step** block to display its Properties dialog.


step Properties

Time Delay(sec): 0

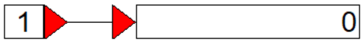
Amplitude: 1

Label:

For this example, no changes are required; however, it is worth noting that you can control the strength of the output signal and the time to delay before calculating the output signal. You can also create a block label that appears below the block when you activate **View > Block Labels**.

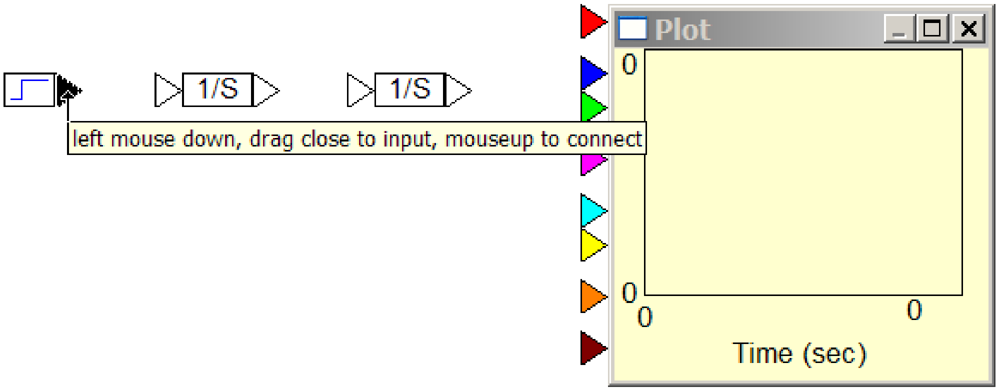
Connecting blocks

By connecting block, you can pass signal values, or data, from one block to another. You connect blocks by creating a wire between block input and output connectors or pins. The connectors have distinct colors to indicate the type of data being passed. Red connectors indicate the double data type.

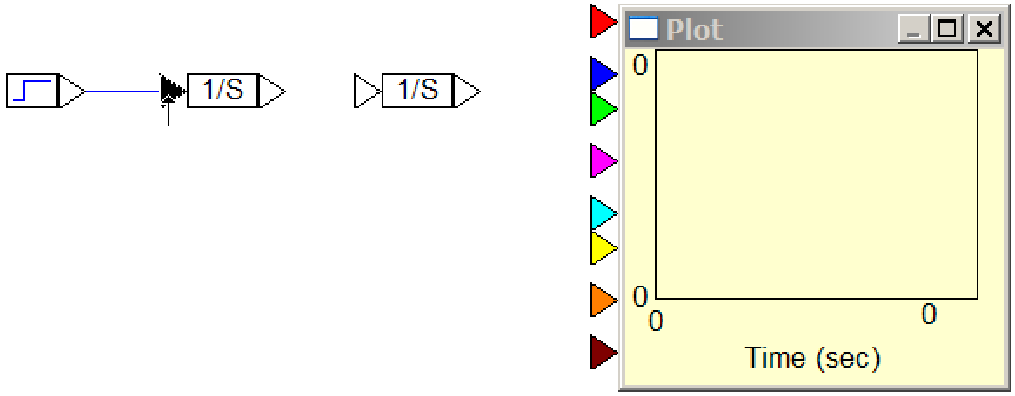


Note: The terms connectors and pins are used interchangeably and refer to the input and output ports on blocks.

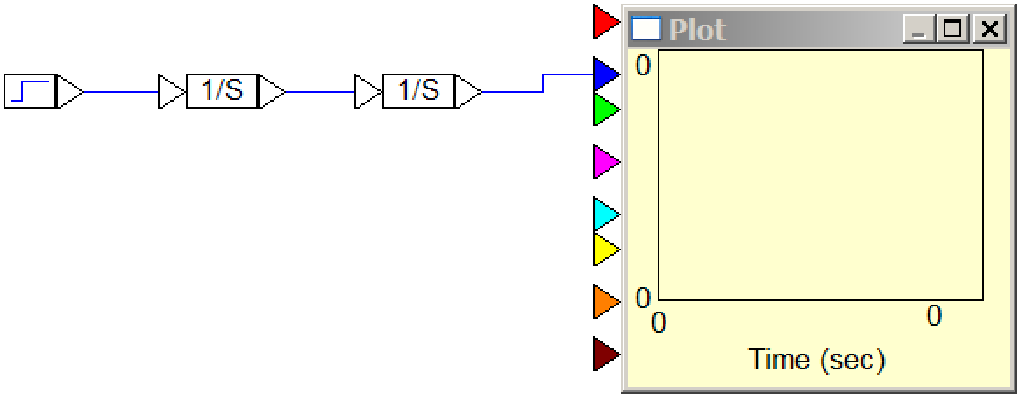
1. Point to the **output connector** on the **step** block. The pointer turns into an upward pointing arrow when it is over the connector.



2. Drag the pointer to the **input connector** on the **integrator** block. When you release the mouse button, the connection is completed.



3. When all the connections are complete, the diagram looks like this:

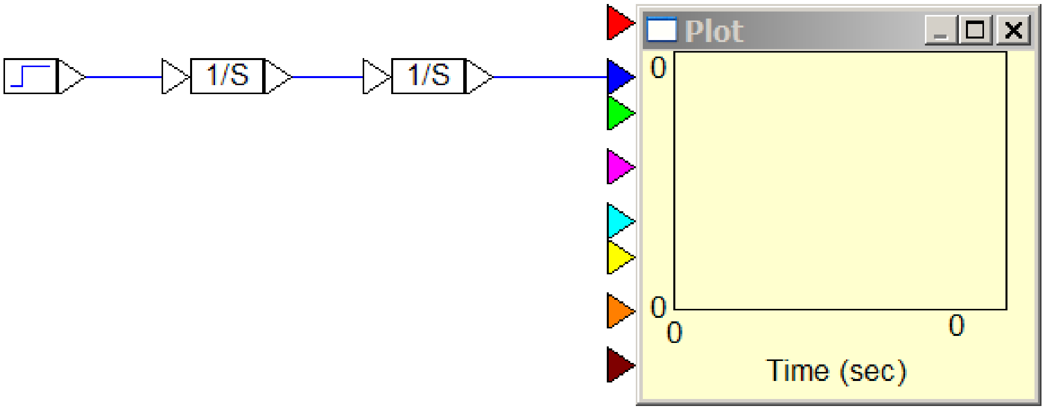


To undo a connection, point to the input connector on the block and drag the pointer away from the block. When you release the mouse button, the connection is removed.

Moving blocks

Moving blocks is one of the more common actions you perform on blocks. As you build your diagram, you will often have to move blocks around the work area. When blocks are connected, you can move them around the work area without breaking their connections.

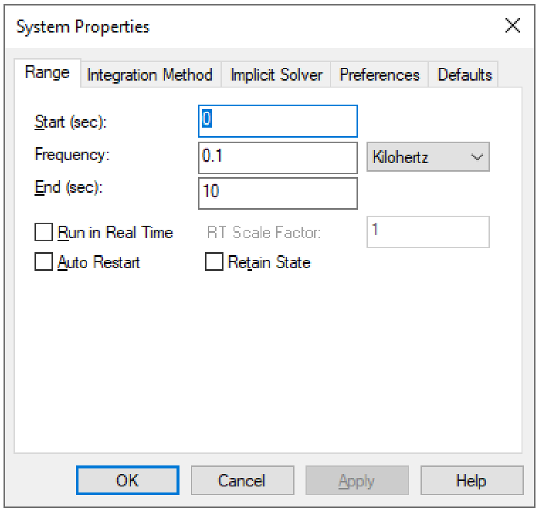
1. Position the pointer over the block and hold down the mouse button.
2. Drag the mouse to reposition the block.



Setting simulation properties

Before starting a simulation, you will need to set or review the simulation properties, which include such things as the step size, integration algorithm, and duration of the simulation. For this example, you need only set the simulation end time.

1. Choose **System > System Properties**.



System Properties

Start (sec): 0

Frequency: 0.1 Kilohertz

End (sec): 10

□ Run in Real Time RT Scale Factor: 1

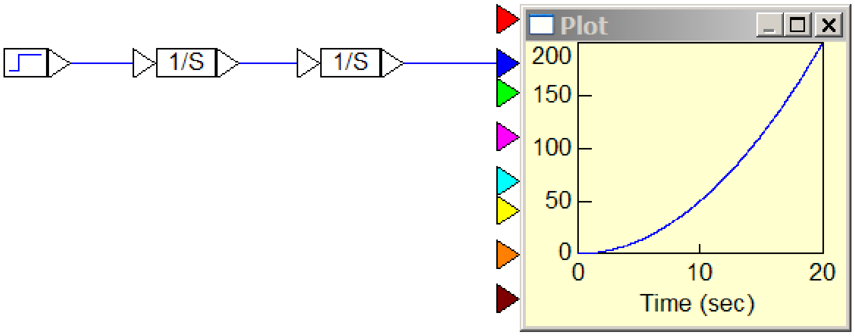
□ Auto Restart □ Retain State

2. In the **End** box, enter **20**, then click **OK**.

Embed sets the simulation end time to 20 sec and closes the dialog.

Running a simulation and viewing the results

The diagram is now ready to be simulated. To start the simulation, choose **System** > **Go**, or click ▶ in the toolbar.



The simulation runs until it reaches the specified end time. The plot block displays the simulation results for $x^2/2$ from 0 to 20 sec.

Saving your work

When you create a new diagram or edit an existing diagram, the work you do is saved in a temporary buffer. To make the changes permanent, use one of the **File > Save** commands or click the **Save** button in the toolbar. If your changes have not yet been saved, an asterisk appears after the diagram name in the title bar.

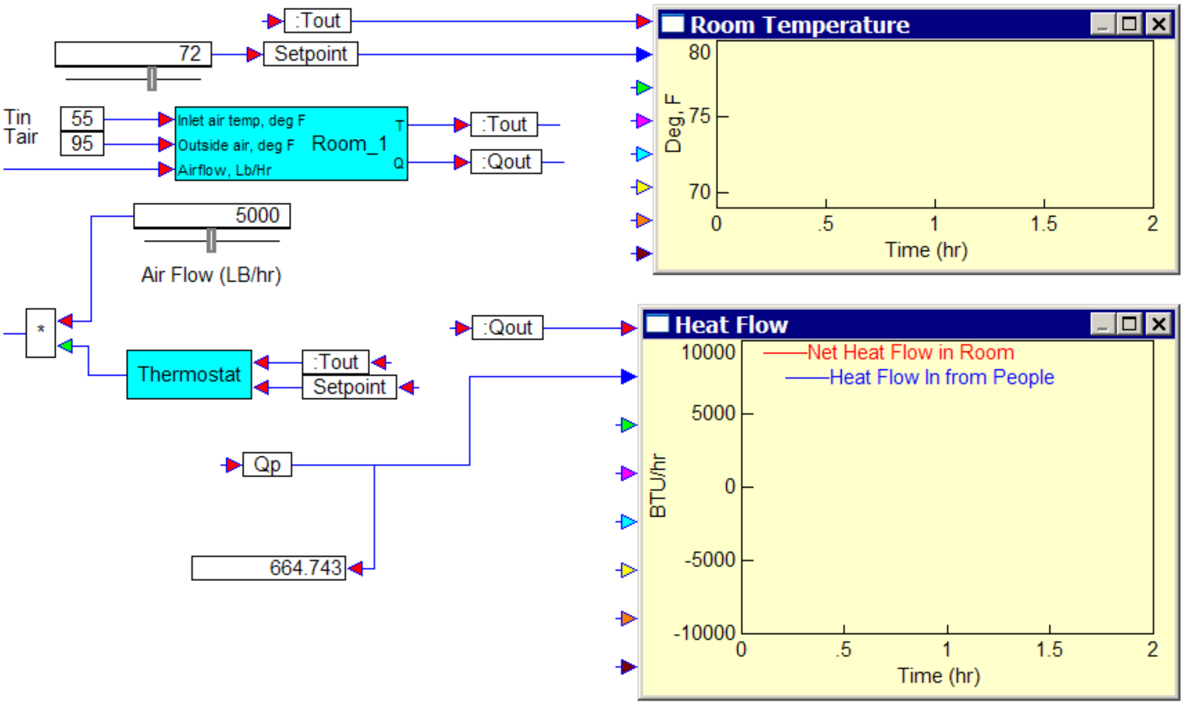
Simulating an HVAC diagram of single room cooling

The **RoomControl** diagram simulates an HVAC diagram of single room cooling with an ON/OFF thermostat. It has hysteresis in the controller and accounts for heat production from people in the room.

Opening and exploring RoomControl

1. Click **Examples** > **Applications** > **HVAC**.

2. Select **RoomControl**.



3. Start the simulation.

4. To stop the simulation, select **System** > **Stop**, or click ⏸.

Diagram properties

Parameters

Qp = rate of heat flow from people

Qin = rate of heat flow carried in by air entering room

Qe = rate of heat flow through room walls

Qnet = sum of all heat flow

Qp0 = heat given off by one person

Troom = room temperature

Tin = temperature of air entering room

Tout = temperature of air leaving room

Tair = temperature of air surrounding room

C = thermal capacitance of air in room

Cr = C + thermal capacitance of furniture and interior walls

R = thermal resistance of walls

w = air flow out of room

S = specific heat of air

A = wall area

P = number of people in room

Equations

Net heat flow (Qnet) is given by:

Qnet = Q p + Qin + Qe

where:

Qp = P * Pp0

Qin = w * S (Tin – Troom)

Qe = (Tair – Troom) * A/R

Substitution yields:

Qnet = P * Qp0 + w *S (Tin – Troom) + (Tair – Troom) * A/R

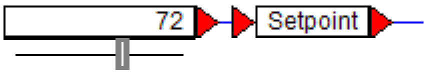
Room temperature = Qnet/Cr

Things to do with RoomControl

This diagram computes the temperature in a room into which cooled air is flowing. People in the room are used as heat input disturbances.

Setpoint

The setpoint is controlled by a dynamic slider block that specifies the desired temperature in the room.



You can adjust the temperature as the simulation progresses by sliding the gray rectangular box. As you change the setpoint, you can observe how quickly the diagram responds to the changes. The temperature is initially set to 72 °F. The allowable range is 50 °F – 85 °F.

Thermostat

The thermostat is a simple ON/OFF control with hysteresis. It allows fluctuation of 1 °F above the setpoint before turning on the Air Flow to blow cool air into the room. The Air Flow remains on until the temperature drops 1 °F below the setpoint. The temperature at which the thermostat turns ON and OFF around the setpoint is specified by the deadband setting in the thermostat subsystem. It is initially set to 2. You can change the setting to see how the diagram responds.

Air Flow

The Air Flow is controlled by a dynamic slider block that blows cool air into the room when the thermostat turns ON.

Room

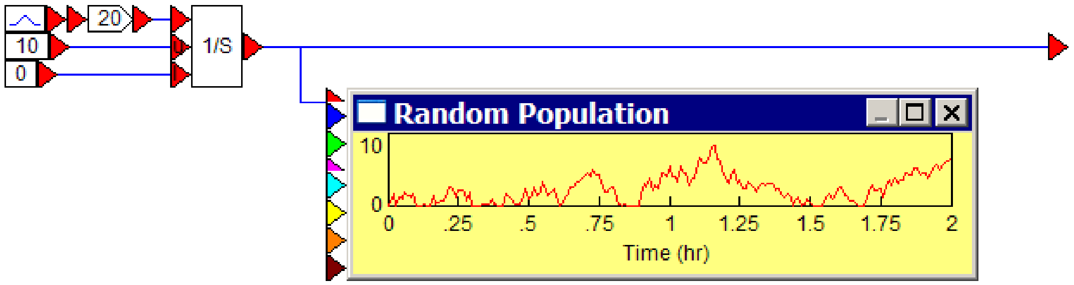
The room is modeled as a simple box with heat flowing in through the walls and heat mass in the room contents and interior walls. There is no heat storage in the room walls and the room air is completely mixed.

The following assumptions are made:

- A typical house (1500 sq ft) requires 3 tons of cooling
  - 1 ton of cooling = 12000 BTU/hr
  - Density of air = 0.076 Lb/f^3
  - 400 f^3/min = 1 ton of cooling
- For 1 ton of cooling, 60*0.076*400 = 1824 Lb/hr is required
- Specific heat of air = 0.24 BTU/(Lb - °F)
- All units in Lb/hr, °F
- Q = heat flow in units of BTU/hr = delta-T*S*w
- For cooling
  - Tin = 55 °F
  - Tair = 85 °F or higher
  - Troom = Tout = 68 °F – 75 °F

People

The number of people entering and exiting the room is a subsystem within the Room subsystem. The number of people is generated by integrating a Gaussian random number function.



Running the simulation

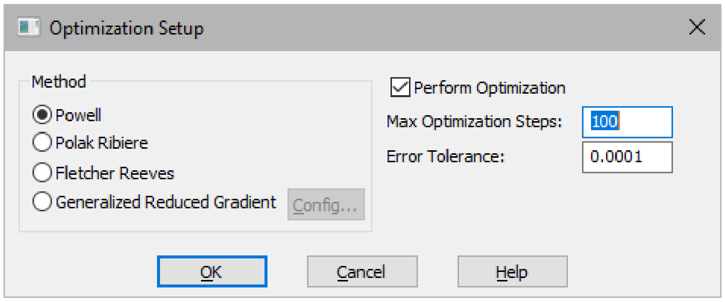
As you run the simulation, you can immediately see the how the temperature fluctuates as people enter and exit the room, as well as when and for how long cool air is blown into the room. By varying the setpoint and air flow, you can see how they affect the time it takes to cool down the room.

Optimizing functions

Optimizing a two-parameter function with no constraints

The **CURV2P** diagram is a simple two-parameter, curve-fitting application involving the approximation of the function Sin (πt) in the interval from 0 to 1. You will approximate this function by another function composed of two straight line segments. There are no constraints in this diagram.

1. Click **Examples** > **Optimize**.
2. Select **CURV2P**.
3. Choose **System** > **Optimization Properties**.



Optimization Setup

Method

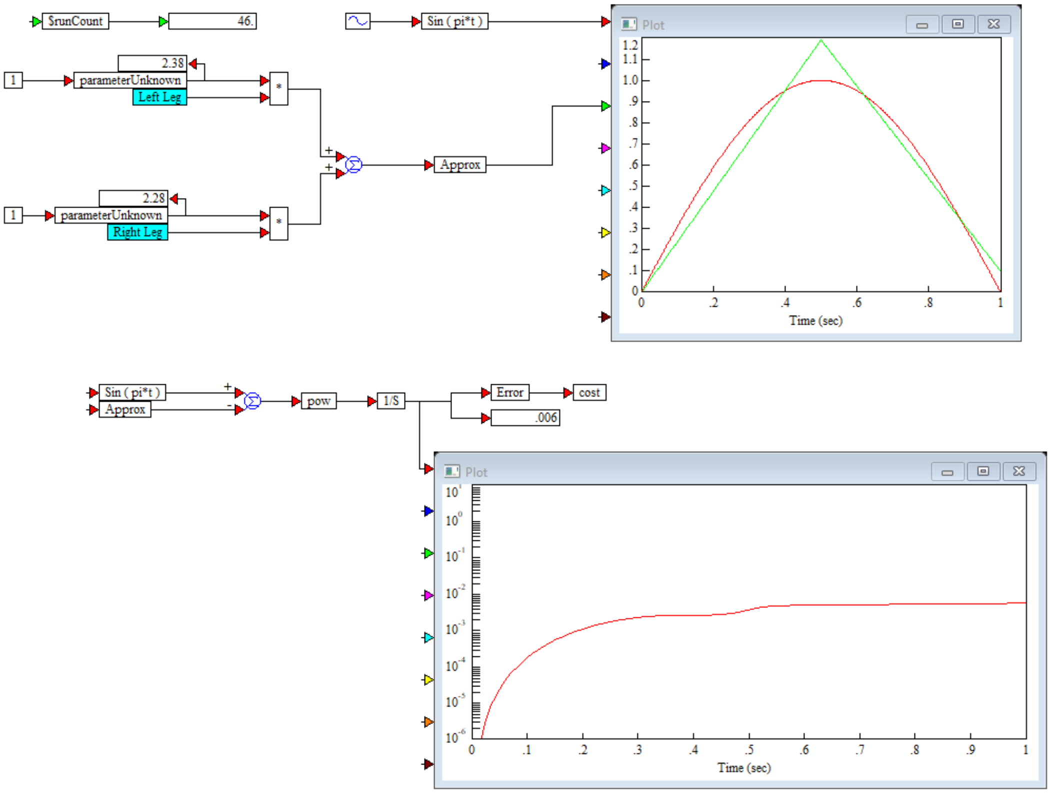
● Powell
○ Polak Ribiere
○ Fletcher Reeves
○ Generalized Reduced Gradient

☑ Perform Optimization

Max Optimization Steps: 100

Error Tolerance: 0.0001

4. Make the following selections:
   a. Under **Method**, activate **Powell.**
   b. Activate **Perform Optimization**.
   c. In **Max Optimization Steps**, enter 100 to set a limit on the number of optimization steps.
   d. In **Error Tolerance**, enter 0.0001 to define the relative accuracy of the simulation runs. In this case, three digits of accuracy are found in the solution.
5. Start the simulation.



The function Sin (πt) is produced by a **sinusoid** block with frequency 0.5 and amplitude 1. It is wired into the **Sin (pi*t)** variable. The approximating function **Approx** is the sum of **Left Leg** (a **step** block wired into an **integrator**block) with a **parameterUnknown** and **Right Leg (**a **step** block wired into an **integrator** block) with a **parameterUnknown**. Both curves are plotted.

To find the best multipliers for the approximating function to produce the smallest error, the multipliers are wired to **parameterUnknown** blocks — which alert Embed that optimization may be performed on these decision variables — and then into a **display** block so that the parameters can be monitored during the optimization run. Upper and lower bounds of 10 and -10 have been set in the **parameterUnknown** blocks. To view or change the bounds, right-click the **parameterUnknown** block.

The cost or objective function is computed by integrating the squared difference of the two curves, (Sin (pi*t) - Approx)^2 , from 0 to 1. The error is wired into a **cost** block to identify it as the objective function.

Each **parameterUnknown** has a **const**block with value 1 wired into it. This provides starting values for the **parameterUnknowns** (that is, the decision variables). A simulation run plots the two curves and computes the error for the starting values as 0.178.

After 46 simulation runs:

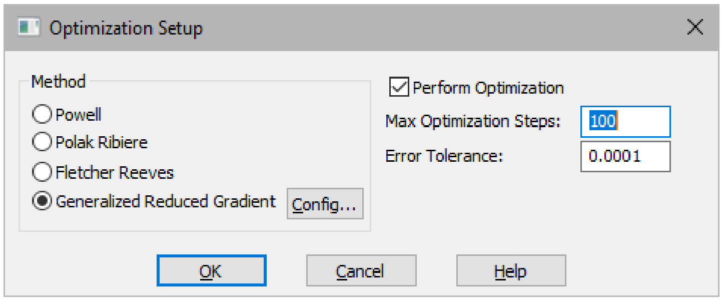
- The **cost** block has changed from 0.178 to 5.82e-3
- The **parameterUnknown** block (upper part of the diagram) has changed to 2.38
- The **parameterUnknown** block (lower part of the diagram) has changed to 2.28

In addition, a report is written to VSMGRG2.TXT that provides additional information on the optimization process.

Optimizing a two-parameter function with constraints

To solve a constrained optimization problem, you use **globalConstraint** blocks. These blocks identify constraints that depend on **parameterUnknowns** and are more complicated than the bound constraints. The **CURV2P1C** diagram illustrates the use of the **globalConstraint** block to constrain the area under the approximating function so that it cannot exceed 0.4.

1. Click **Examples** > **Optimize**.
2. Select **CURV2P1C**.
3. Choose **System** > **Optimization Properties**.



Optimization Setup

Method

○ Powell

○ Polak Ribiere

○ Fletcher Reeves

● Generalized Reduced Gradient Config...

☑ Perform Optimization

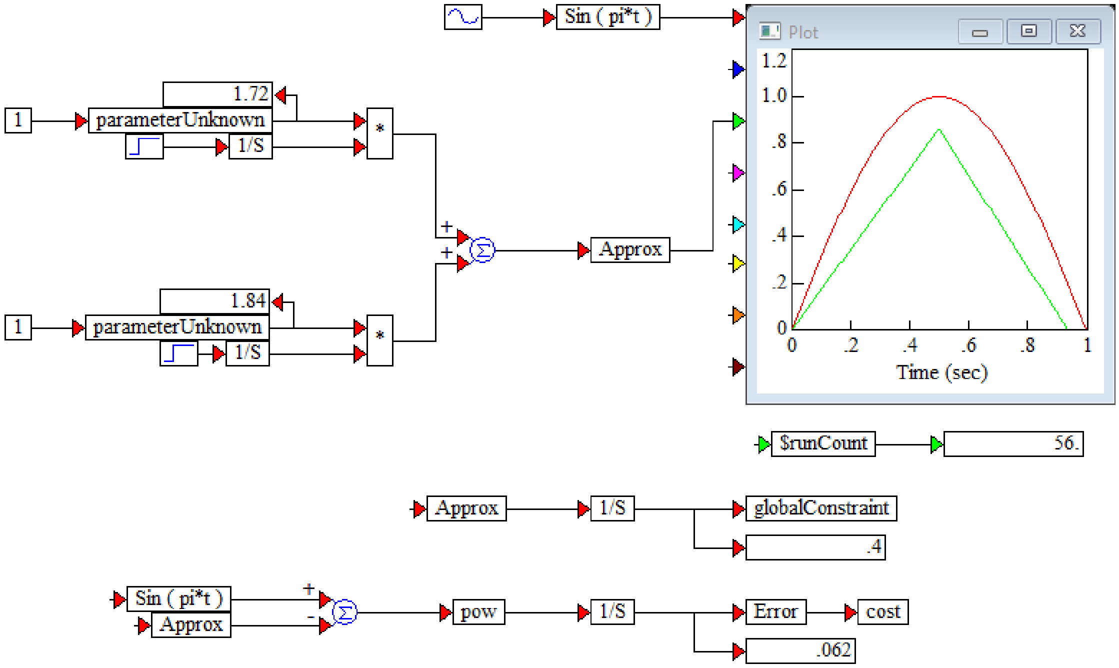
Max Optimization Steps: 100

Error Tolerance: 0.0001

OK Cancel Help

4. Make the following selections:
   a. Under **Method**, activate **Generalized Reduced Gradient.**
   b. Activate **Perform Optimization**.
   c. In **Max Optimization Steps**, enter 100 to set a limit on the number of optimization steps.
   d. In **Error Tolerance**, enter 0.0001 to define the relative accuracy of the simulation runs. In this case, three digits of accuracy are found in the solution.
   e. Click **OK**, or press **ENTER**.
5. Right-click the **globalConstraint** block and make the following changes:
   a. In **Upper Bound**, enter **0.4**.
   b. In **Lower Bound**, enter **0.0**.
   c. Click **OK**, or press **ENTER**

6. Start the simulation.



The constrained optimization run yields the **parameterUnknown** values of 1.72 and 1.84 and the cost value of 6.21e-2. The constraint is at its upper bound of 0.4, as should be expected.

Note: The exact answer to the analytic problem posed here may differ from the computed answer. This discrepancy shows up because the integration methods are not exact. You can verify this by decreasing the integration step size under System > System Properties and rerunning the simulation. The Embed solution to this problem differs (due to numerical truncation errors) from the analytic solution. Taking smaller step sizes makes this relationship clear.

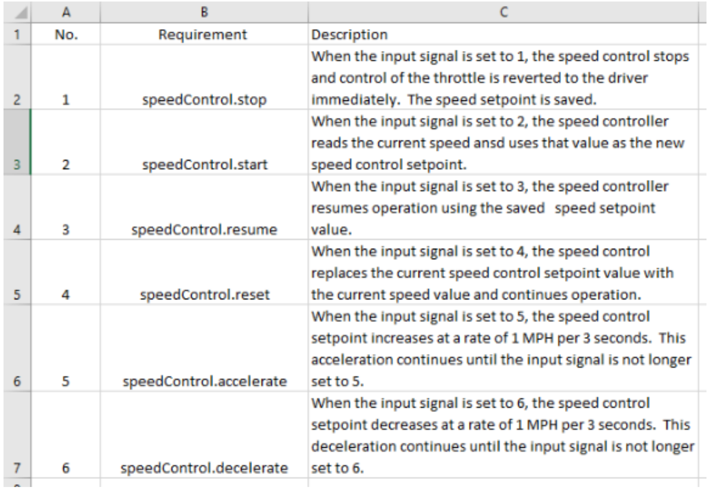
Optimizing a five-parameter function with constraints

The **CURV5P** diagram — under Examples > Optimize — approximates the function Sin (πt) on the interval 0 to 1. Five line segments are used in this diagram to get a better fit than what was gotten in **CURV2P.VSM** with only two parameters.

The objective function is the integral of the squared error between the two curves. Starting with all five **parameterUnknowns** set to 1, the starting value of the objective function is 0.14. Embed converges after 109 simulation runs with the minimized value of the objective function at 0.00027.

Linking an Excel spreadsheet to a block diagram

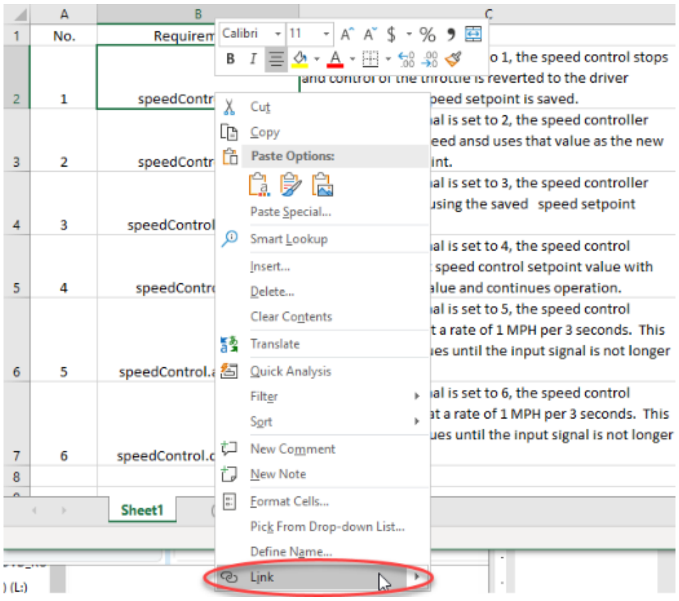
The following 3 x 7 Excel spreadsheet lists sample requirements for an automotive speed control system: column B lists the requirements and column C describes the requirements. In this example, **cell B2** is linked to the name that appears on the **label** block in the Embed diagram and the contents of **cell C2** is highlighted when you click the **label** block.



	A	B	C
1	No.	Requirement	Description
2	1	speedControl.stop	When the input signal is set to 1, the speed control stops and control of the throttle is reverted to the driver immediately. The speed setpoint is saved.
3	2	speedControl.start	When the input signal is set to 2, the speed controller reads the current speed ansd uses that value as the new speed control setpoint.
4	3	speedControl.resume	When the input signal is set to 3, the speed controller resumes operation using the saved speed setpoint value.
5	4	speedControl.reset	When the input signal is set to 4, the speed control replaces the current speed control setpoint value with the current speed value and continues operation.
6	5	speedControl.accelerate	When the input signal is set to 5, the speed control setpoint increases at a rate of 1 MPH per 3 seconds. This acceleration continues until the input signal is not longer set to 5.
7	6	speedControl.decelerate	When the input signal is set to 6, the speed control setpoint decreases at a rate of 1 MPH per 3 seconds. This deceleration continues until the input signal is not longer set to 6.

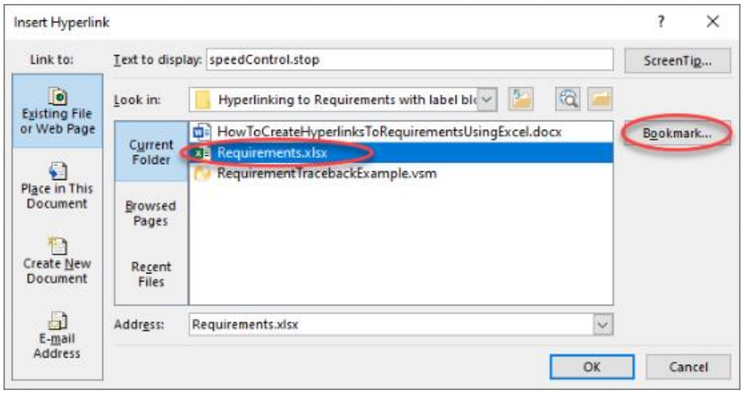
1. In your Embed diagram, insert a **label** block. If this is a new diagram, save it.

2. Go to your requirements spreadsheet, right-click **cell B2,** and select **Link** from the pop-up menu.



3. In the Insert Hyperlink dialog, select the **requirements spreadsheet** and click **Bookmark**.

   Note: The contents of **cell B2** (speedControl.stop) is displayed in **Text to display**.



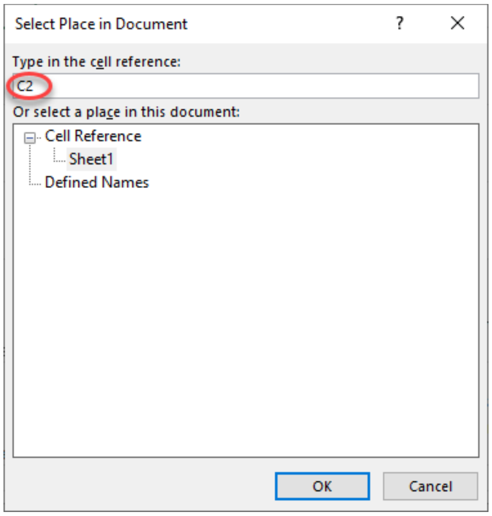
Text to display: speedControl.stop

Look in: Hyperlinking to Requirements with label blo

HowToCreateHyperlinksToRequirementsUsingExcel.docx
Requirements.xlsx
RequirementTracebackExample.vsm

Address: Requirements.xlsx

4. In the Select Place in Document dialog, enter **C2** (rather than B2) to link to the description of **speedControl.stop** and click **OK**.

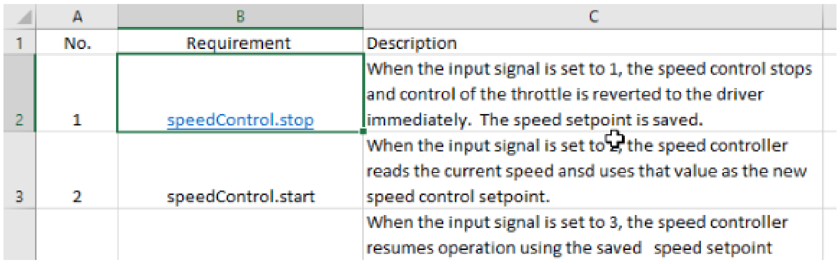


Type in the cell reference: C2

Or select a place in this document:

- Cell Reference
  - Sheet1
- Defined Names

5. In the requirements spreadsheet, **cell B2** is highlighted.

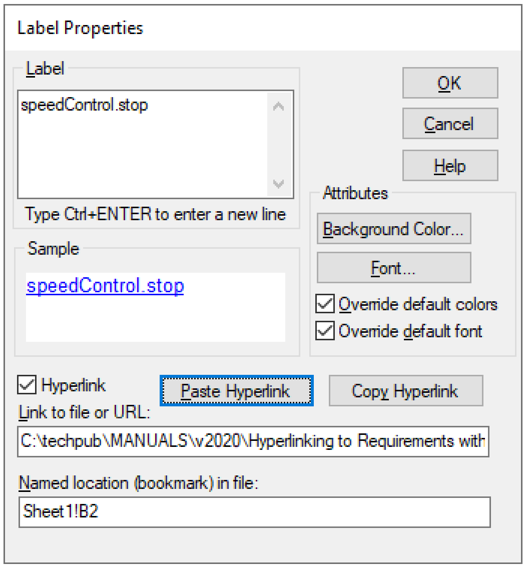


	A	B	C
1	No.	Requirement	Description
2	1	speedControl.stop	When the input signal is set to 1, the speed control stops and control of the throttle is reverted to the driver immediately. The speed setpoint is saved.
3	2	speedControl.start	When the input signal is set to 2, the speed controller reads the current speed ansd uses that value as the new speed control setpoint.
			When the input signal is set to 3, the speed controller resumes operation using the saved speed setpoint

6. Right-click **cell B2** and select **Copy** from the pop-up menu.

7. Return to the Embed diagram and right-click the **label** block.

8. In the Label Properties dialog, activate **Hyperlink** and click **Paste Hyperlink** to update it with the hyperlink information.



Label: speedControl.stop

Type Ctrl+ENTER to enter a new line

☑ Override default colors

☑ Override default font

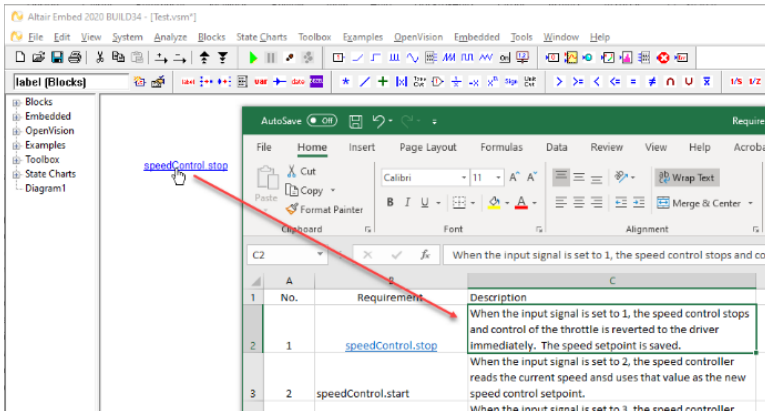
☑ Hyperlink

Link to file or URL: C:\techpub\MANUALS\v2020\Hyperlinking to Requirements with

Named location (bookmark) in file: Sheet1!B2

9. To display the description of **speedControl.stop**, revise the bookmark location in **Named location (bookmark) in file** to **Sheet 1!C2** and click **OK**.

10. In the Embed diagram, the **label** block appears as a hyperlink named **speedControl.stop**. When you click **speedControl.stop**, the requirements spreadsheet opens with a border around **cell C2.**


Altair Embed 2020 BUILD34 - [Test.vsm*]
File Edit View System Analyze Blocks State Charts Toolbox Examples OpenVision Embedded Tools Window Help
label (Blocks)
Blocks
Embedded
OpenVision
Examples
Toolbox
State Charts
Diagram1
speedControl.stop
AutoSave Off
Require
File Home Insert Page Layout Formulas Data Review View Help Acroba
Cut
Copy
Format Painter
Paste
Clipboard
Calibri
11
Font
Wrap Text
Merge & Center
Alignment
C2

When the input signal is set to 1, the speed control stops and co

	A	B	C
1	No.	Requirement	Description
2	1	speedControl.stop	When the input signal is set to 1, the speed control stops and control of the throttle is reverted to the driver immediately. The speed setpoint is saved.
3	2	speedControl.start	When the input signal is set to 2, the speed controller reads the current speed ansd uses that value as the new speed control setpoint.
			When the input signal is set to 3, the speed controller

Analyzing Van Der Pol's nonlinear system

Van Der Pol's nonlinear dynamic system is represented as the following differential equation:

$$\frac{d^2x}{dt^2} - (1{-}\propto x^2)\frac{dx}{dt} + x = 0$$

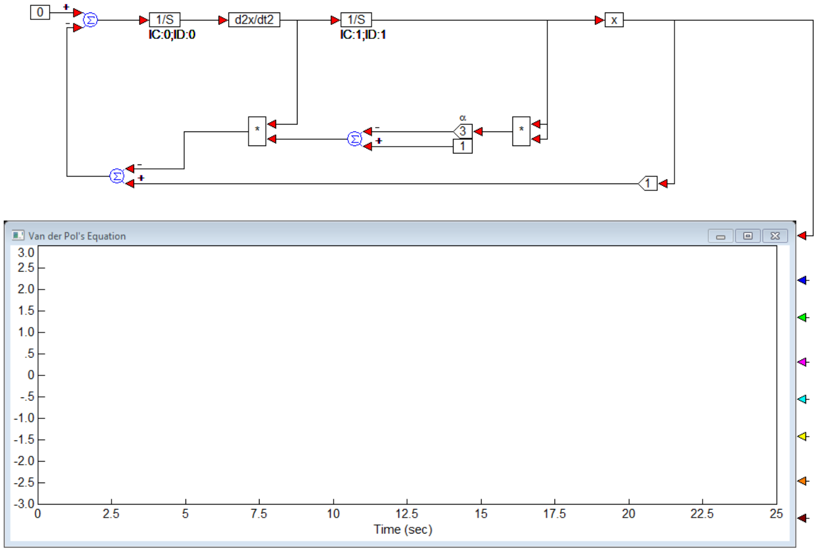
where at least one of the following initial conditions is met:

$$\frac{dx}{dt}(t_0) \neq 0$$

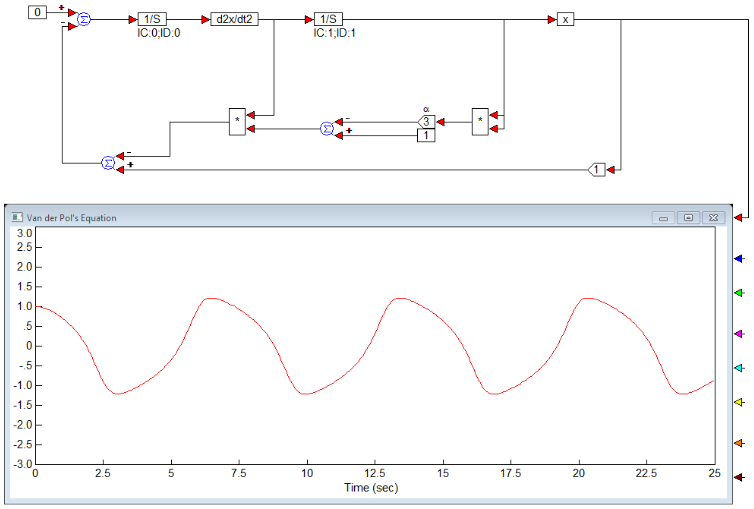
$$x(t_0) \neq 0$$

This section steps you through the process of building Van Der Pol's system in block diagram form, and generating ABCD state-space matrices, transfer function information, and Bode and root locus plots.

1. Convert Van Der Pol's system into block diagram form, or open **Examples > Applications > ControlDesign > VanDerPolSystem**.



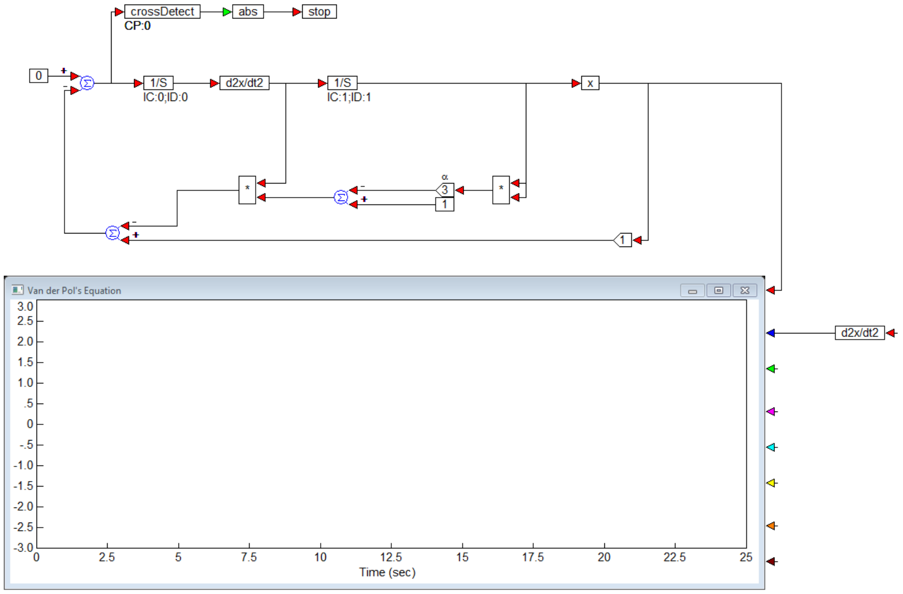
2. To satisfy the conditions of the equation, make the following block parameter assignments:
   - Set the gain (α) to **3**.
   - Set the initial condition of the first integrator (the block from which dx/dt is generated) to **0**.
   - Set the initial condition for the second integrator to **1**.
3. Choose **System > System Properties** and make the following selections.
   a. Under the **Range** tab, make the following changes:
      I. **Range Start**, enter 0.
      II. **Range End**, enter 25.
      III. **Step Size**, enter 0.05.
   b. Under the **Integration Method** tab, select **Euler** and click **OK**, or press **ENTER**.
4. Start the simulation.



Linearizing Van Der Pol’s nonlinear system

An interesting operating point about which to linearize the system is when the d2x/dt2 signal is equal to 0. At this point, the linearization results in stable poles.

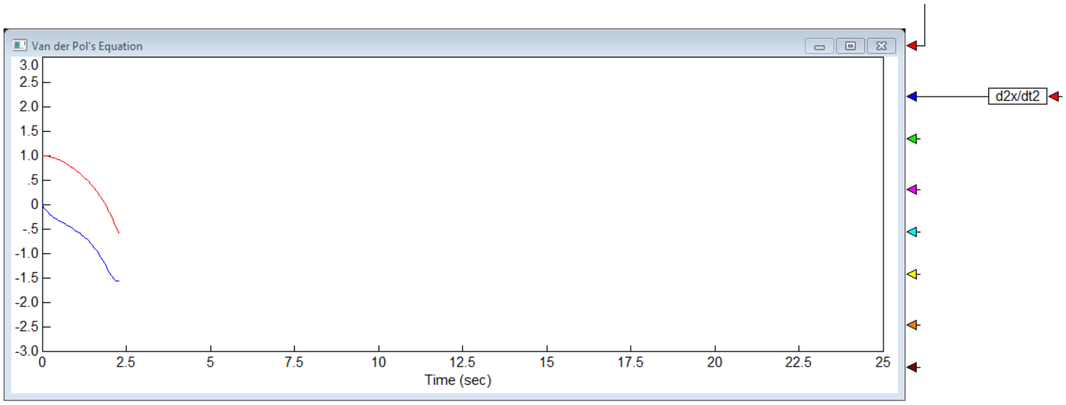
1. Wire a **crossDetect** block to the **d2x/dt2** signal.
2. Feed the output into an **abs** block, which is wired to a **stop** block.
3. Wire the **d2x/dt2** signal into the **plot** block.



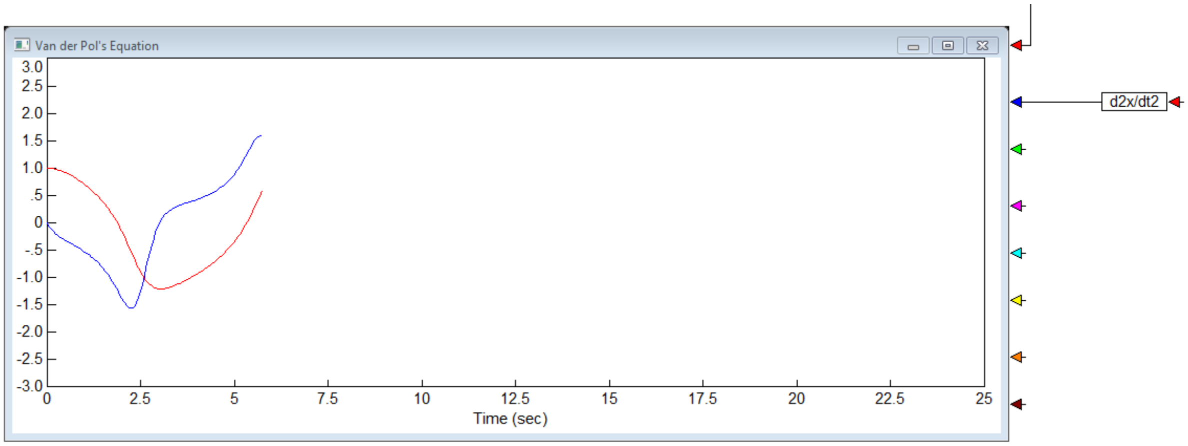
In this configuration, the simulation is automatically stopped when the d2x/dt2 signal is exactly 0. The crossDetect block outputs 1 or -1 depending on whether the crossing occurs with a positive or negative slope. Since the `stop` block stops the simulation only when the input is greater than or equal to 1, an abs block is introduced between them to ensure that the stop block receives only positive inputs.

4. Start the simulation.

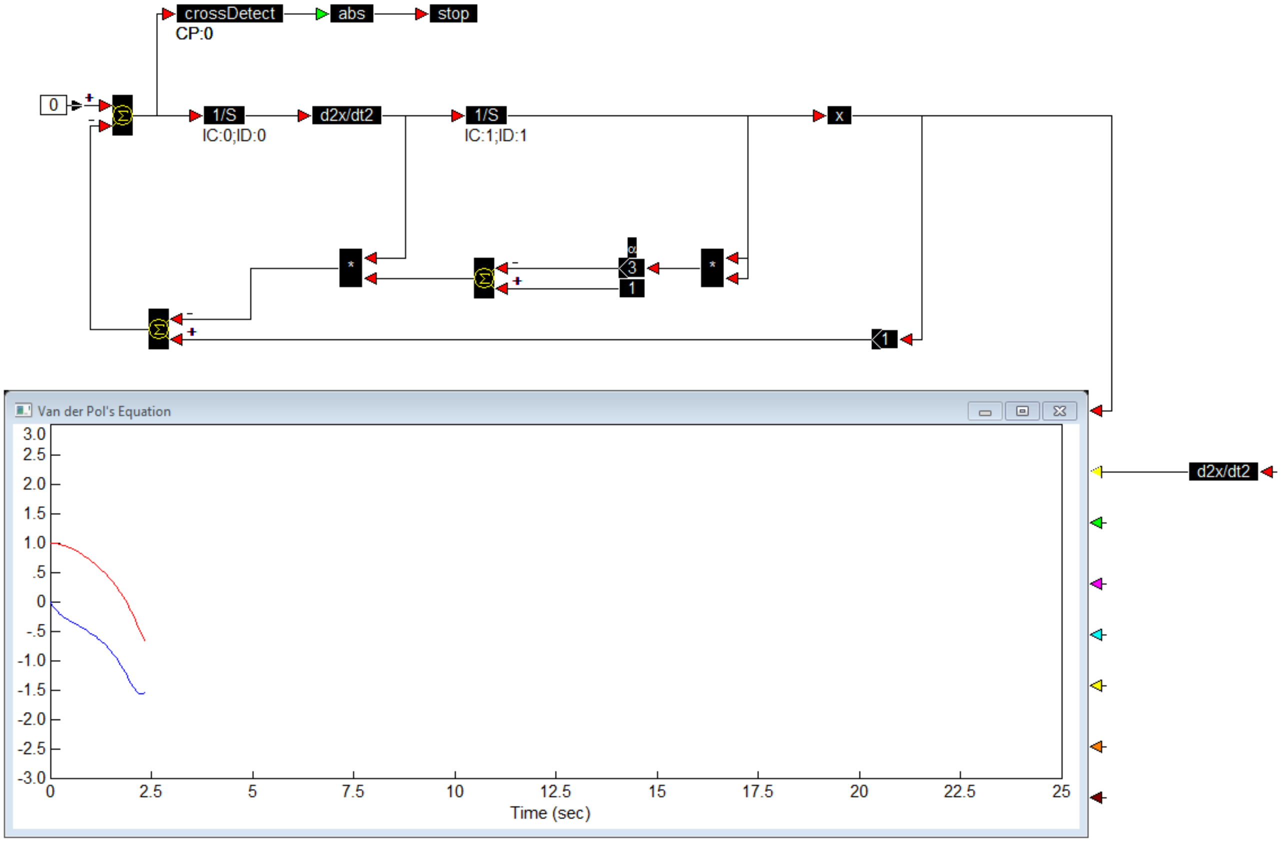
The simulation runs to the first occurrence of signal d2x/dt2 = 0.



5. Choose **System > Continue**, or click in the toolbar to continue the simulation to the next occurrence of d2x/dt2 = 0.



6. Select the **block set** to be analyzed. The block set includes all but the 0 input const block and the plot block.



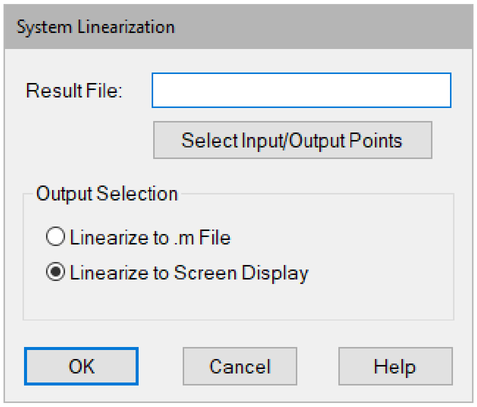
7. Choose **Analyze > Select Input/Output Points** to specify the reference points for the linearization.

8. Point to the output connector on the 0 input **const** block and click.

9. Point to the input connector on the **plot** block to which the **d2x/dt2** signal is wired and click.

10. Point to empty screen and click.

11. Choose **Analyze > Linearize**.



Result File:

Output Selection

○ Linearize to .m File

● Linearize to Screen Display

12. Do one of the following:

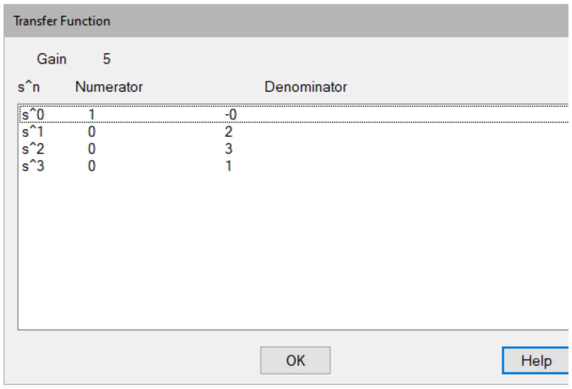
- To display the ABCD matrices in four separate, successive dialogs, select **Linearize To Screen Display**.
- To write the matrix information to an M file, select the **Linearize To .M File** and enter a file name in the **Result File** box. For this example, the contents of the file will be:

```
function [a,b,c,d] = vabcd
a = [-.384714 -7.26932;
    1 0 ];
b = [1 ;
    0 ];
c = [-.384714 -7.26932];
d = [1 ];
```

Generating transfer function information, root locus plots, and Bode plots

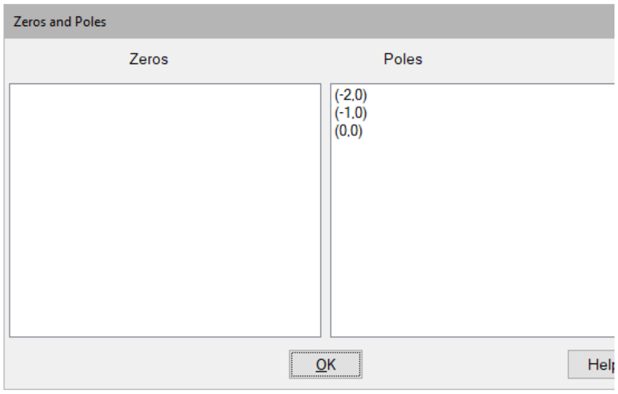
1. Choose **Analyze** > **Transfer Function Info**.

The numerator and denominator coefficients, and the zeros and poles are displayed in successive dialogs.



Gain 5

s^n	Numerator	Denominator
s^0	1	-0
s^1	0	2
s^2	0	3
s^3	0	1



Zeros	Poles
	(-2,0) (-1,0) (0,0)

The input to the system is the 0 constant (denoted as *R*), and the output is the dx/dt signal (as previously defined). The first dialog presents the transfer function as numerator and denominator polynomials in power of *s*. Denoting the output as:

$$y \equiv \frac{dx}{dt}$$

the transfer function is:

$$\frac{Y(s)}{R(s)} = \frac{s^2}{s^2 + 0.384714s + 7.26932}$$

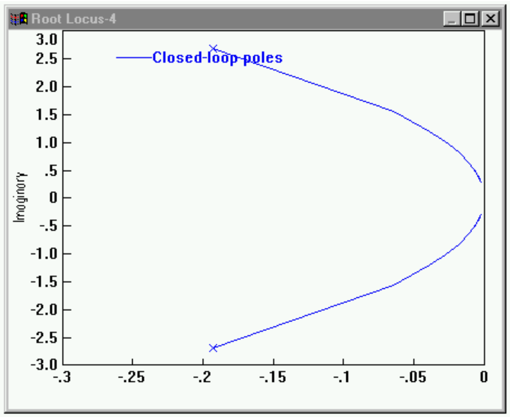
The gain (*s* = 0 gain) is 0 by inspection.

The second dialog presents the factors of both polynomials. The zeros are the roots of the numerator, and the poles are the roots of the denominator. At this operating point, the system has two real zeros and a complex conjugate pair of poles. The factored transfer function can be expressed as:

$$\frac{Y(s)}{R(s)}=\frac{(s+0)(s+0)}{(s-(-0.192-j2.68))(s-(-0.192+j2.68))}$$

2. Choose **Analyze** > **Root Locus**.

3. Resize and move the plot for easier viewing.

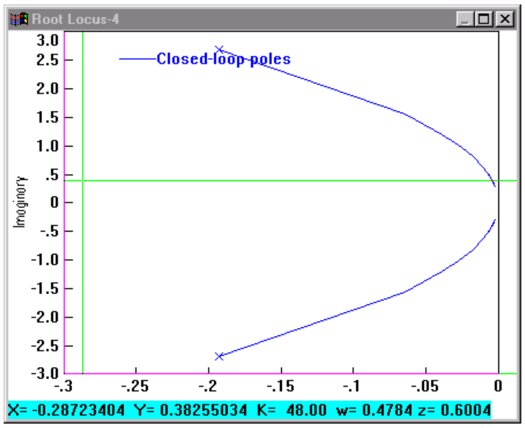


4. Choose **Edit** > **Block Properties**.

5. Click over the root locus plot.

6. Click **Read Coordinates**.

   The root locus plot reappears with crosshairs and status bar.



```
X= -0.28723404  Y= 0.38255034  K=  48.00  w= 0.4784 z= 0.6004
```

At the selected point, a gain of 48 in feedback around the transfer function results in a well-damped $(z \cong 0.6004)$ rapid responding system with a time constraint of:

$$\tau \approx \frac{1}{zw}=\frac{1}{(0.6004)(0.4784)}=\frac{1}{0.2872}=3.48s$$

A zero steady-state step error due to the integration at the origin.

7. Choose **Analyze** > **Frequency Range** to view the Bode plots used to determine the performance characteristics of a closed-loop system in frequency domain.

8. In the Bode Frequency Range dialog, do the following:

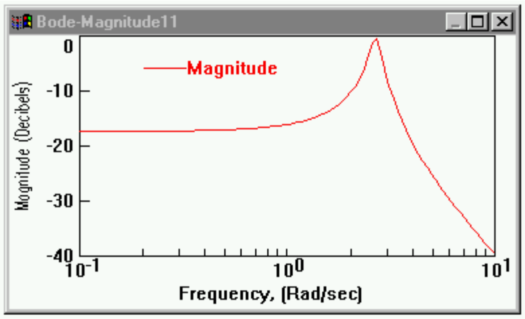
   a. In the Start box, enter **0.1**.

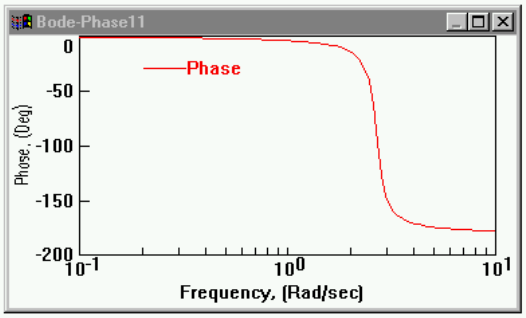
   b. In the End box, enter **10**.

c. In the Step Count box, enter **100**.

9. Choose **Analyze** > **Frequency Response**.

10. Resize and move the plots for easier viewing.



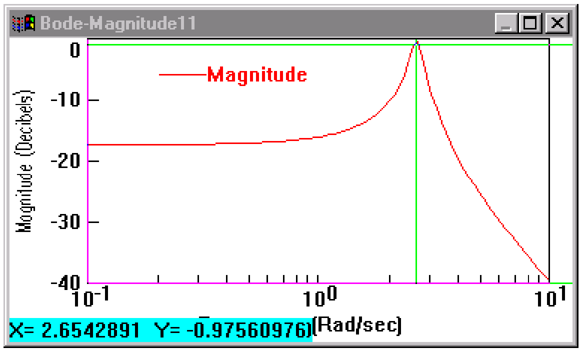


11. To determine the resonant frequency of the magnitude plot, invoke the plot crosshairs:

a. Choose **Edit** > **Block Properties** and click **Bode magnitude plot**.

b. Select **Options** and choose **Read Coordinates**.

The Bode magnitude plot reappears with crosshairs and a status bar.



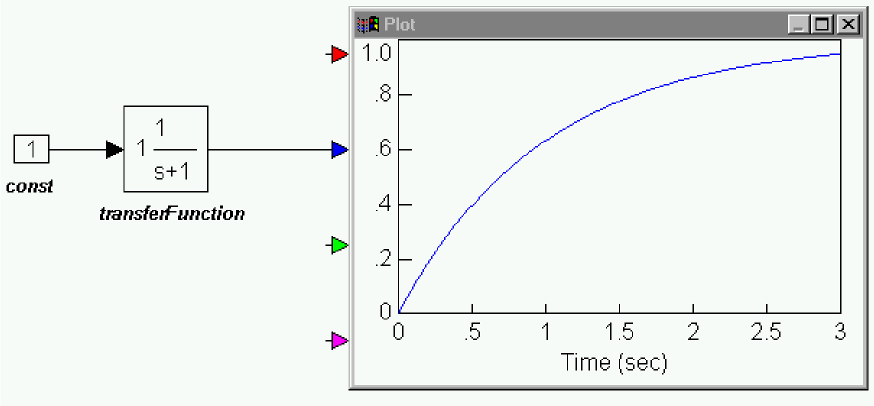
X= 2.6542891 Y= -0.97560976

The digital display of the magnitude plot reveals the *x* coordinate is 2.654 rad/sec, the resonant frequency of the system.

Note: This value agrees, within the granularity of the digital read-out, with the factored transfer function value of 2.68 rad/sec (as solved earlier).

Analyzing Nyquist stability of a type 0 system

To perform a Nyquist stability analysis, consider a simple type 0 system with the open-loop transfer function $GH(s) = \frac{1}{(s+1)}$ as shown in the diagram below:



To generate the Nyquist plot

1. Create the above diagram using a **const**, **transferFunction**, and **plot** block.
2. Enter the following polynomial coefficients to the **transferFunction** block:

   Numerator: 1

   Denominator: 1 1

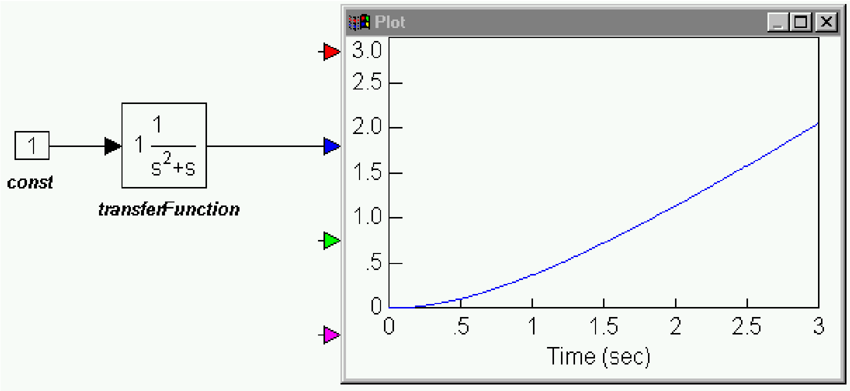
   Note: Always leave spaces between coefficient values.
3. Start the simulation.
4. Select the **transferFunction** block.
5. Choose **Analyze > Nyquist Response**.

   The Nyquist plot is displayed.
6. Drag on its borders to adjust its size.

The Nyquist plot for this system is a circle, with the real part of *GH*(*s*) on the horizontal axis and the imaginary part of *GH*(*s*) on the vertical axis. On this plot, the origin represents *GH*(*j*∞) and the point of intersection with the horizontal axis (*Re*(*GH*) = 1) represents *GH*(*j*0).

Analyzing Nyquist stability of a stable type 1 system

Consider a type 1 system with the open-loop transfer function $GH(s) = \frac{1}{s(s+1)}$ as shown below:



To generate the Nyquist plot

1. Create the above diagram using a **const**, **transferFunction**, and **plot** block.
2. Enter the following polynomial coefficients to the **transferFunction** block:

   Numerator: 1

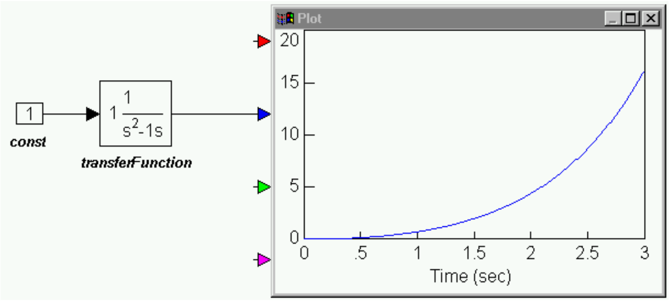
   Denominator: 1 1 0

   Note: Always leave spaces between coefficient values.
3. Start the simulation.
4. Select the **transferFunction** block.
5. Choose **Analyze > Nyquist Response**.
6. You are reminded that the system has poles on the imaginary axis, which will result in Nyquist circles at infinity. Click **OK**, or press **ENTER**.
7. In the Nyquist dialog, you have the option to change the maximum frequency range. The default is 10. Leave it unchanged and click **OK**, or press **ENTER**.

The point (-1,0) is not enclosed by the Nyquist contour. Consequently $N \leq 0$. The poles of $GH(s)$ at $s = 0$ and $s = -1$, neither of which are in the right-half plane, which means that $P = 0$. Therefore, $N = -P = 0$ and the system is absolutely stable.

Analyzing Nyquist stability of an unstable type 1 system

Another example of a type 1 system is the open-loop transfer function $GH(s) = \frac{1}{s(s-1)}$ as shown below:



To generate the Nyquist plot

1. Create the above diagram using a **const**, **transferFunction**, and **plot** block.
2. Enter the following polynomial coefficients to the **transferFunction** block:

   Numerator: 1

   Denominator: 1 -1 0

   Note: Always leave spaces between coefficient values.
3. Choose **System > Go**, or click ▶ in the toolbar to simulate the diagram.
4. Select the **transferFunction** block.
5. Choose **Analyze > Nyquist Response**.
6. You are reminded that the system has poles on the imaginary axis, which will result in Nyquist circles at infinity. Click **OK**, or press **ENTER**.
7. In the **Nyquist** dialog, you have the option to change the maximum frequency range. The default is 10. Leave it unchanged and click **OK**, or press **ENTER**.

The point (-1,0) is enclosed by the Nyquist contour. Consequently N > 0. Moreover, since the number of clockwise encirclements of the point (-1, 0) is one, N = 1. The poles of *GH*(*s*) are at *s* = 0 and *s* = +1, with the second pole appearing in the right-half plane. This implies that *P*, the number of poles in the right-half plane, equals 1.

In this case, N ≠ -*P*, which indicates that the system is unstable.

The number of 0s of 1 + *GH*(*s*) in the right-half plane is given by:

$$Z = N + P = 1 + 1 = 2$$

Creating a three-state pump with State Charts

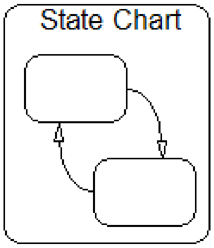
This example shows how to build a simple three-state pump. The pump operating states are defined as:

- Control OFF
- Control ON to pump water into the tank
- Control ON to pump water out of the tank

During simulation, the pump controls the water level in a tank by keeping the water within a specified minimum and maximum levels. An interactive ON/OFF button controls the pump. The tank drains completely if control is OFF, but it will never overflow.

A state chart block is a container block inside which you define the operating modes of the pump.

1. Open a new diagram.
2. Choose **State Charts** > **state chart** to create a container for the state chart.
3. Click anywhere in the work area to insert the **state chart** block. You will see the following:



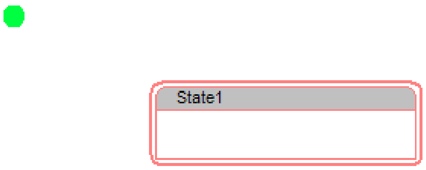
4. Right-click the **state chart** block to enter the state chart design environment.

Inserting states

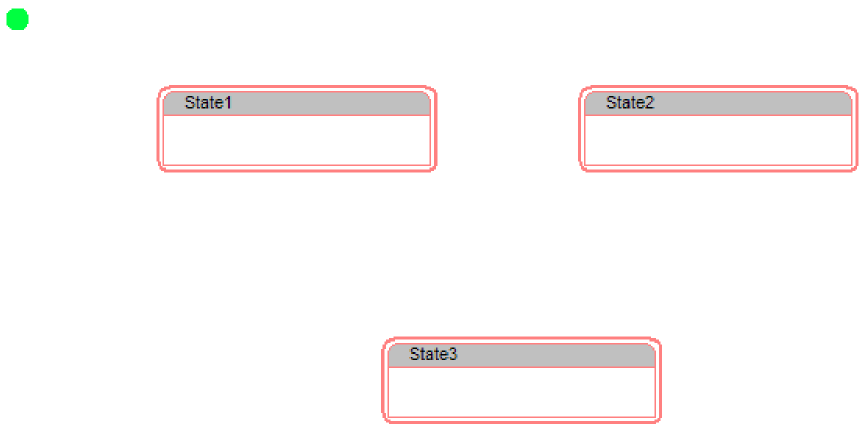
For this example, you will use an initial state indicator and three simple states to represent the three states of the pump.

1. Choose **Start Charts > initial state indicator**.
2. Click anywhere in the work area to insert the initial state indicator.

3. Choose **State Charts > state**.
4. Click anywhere in the work area to insert the state.



5. To create a three-state system, insert two more state blocks into the work area. Your diagram will look like this:



Creating transitions

A transition is a relationship between two states that indicates when an object can move the focus of control to another state once certain conditions are met.

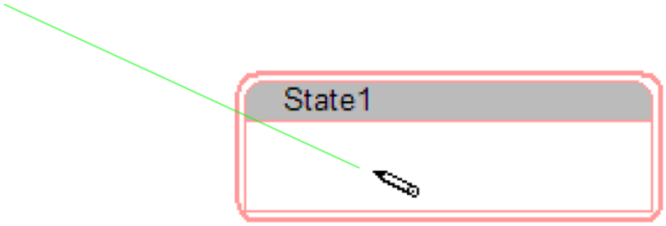
A transition is represented as line between two states. An arrowhead at one end of the transition indicates the direction of the transition.

When a state has multiple transitions exiting it, the transitions are numbered to indicate evaluation order.

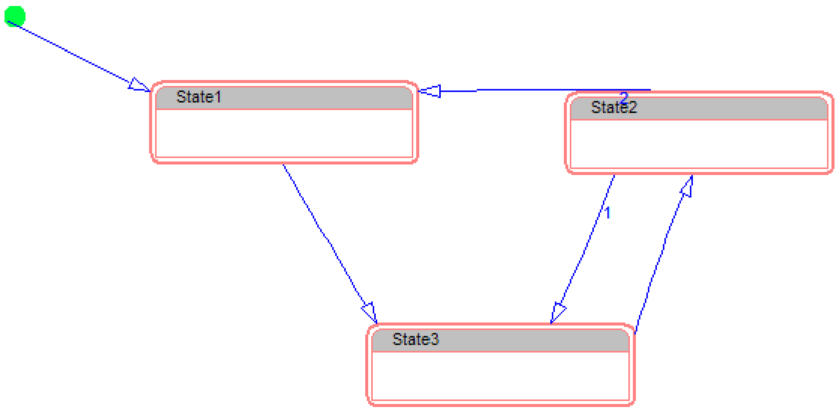
After you create a transition, you can define a transition specification for it.

By default, transitions are drawn as lines that can be bowed.

1. Point to the edge of a **source state**. The cursor changes to ✎ .
2. Drag into the **target state** and release the mouse button.



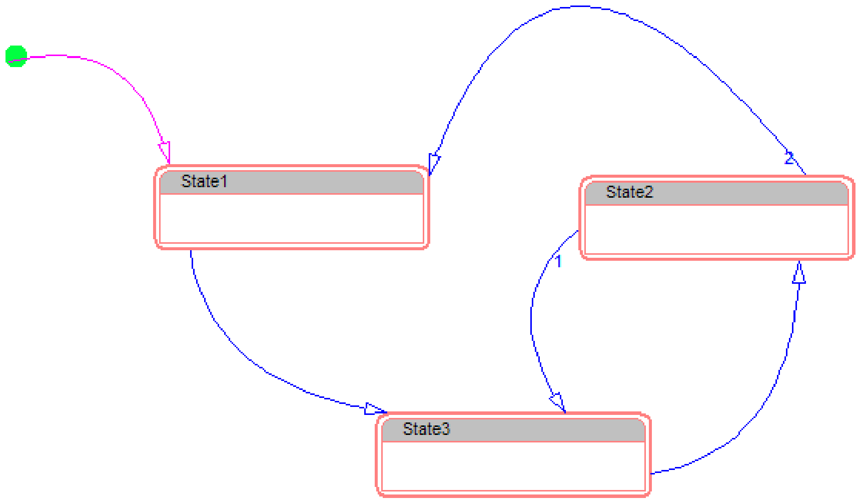
3. The transition appears as a line from the source state to the target state.
4. Repeat this exercise to create the following:



Notice that when multiple transitions are coming from a given state, Embed labels them according to evaluation order. Transitions with lower numbers have higher priorities.

Bending and moving transitions

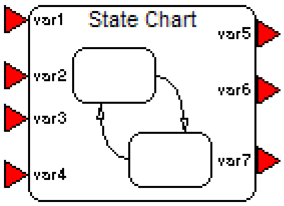
1. Click the transition that you want to bend or move. The transition turns purple.
   - To bend the transition, drag the transition.
   - To move where the transition connects to a state, drag the transition connector.
2. Repeat step 1 for each transition until you have the following:



Defining state chart variables

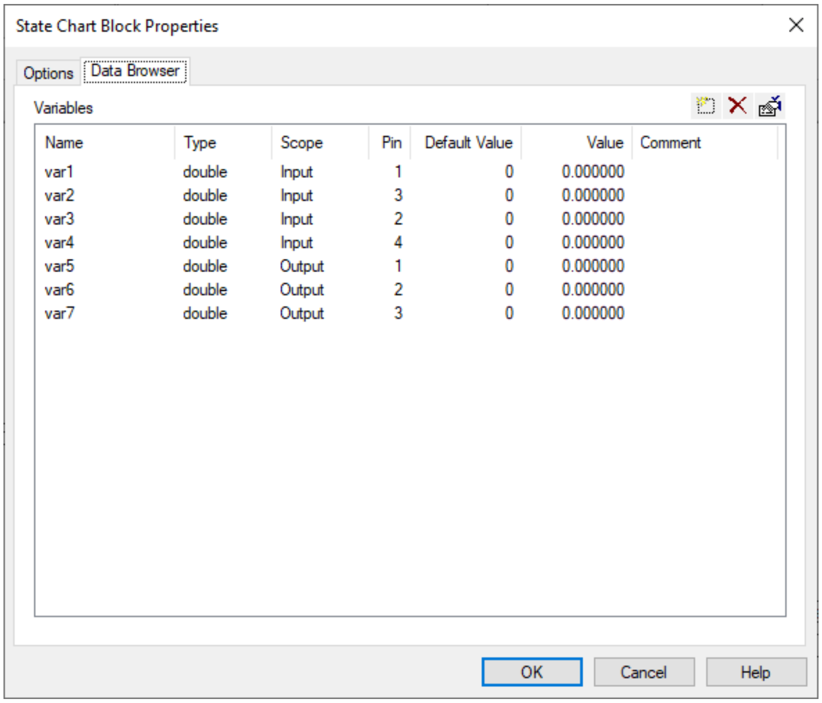
To exchange data between the Embed diagram and the state chart, you use variables. State chart variables are declared in the State Chart Block Properties dialog. In this example, you will declare four input variables and three output variables.

1. Add four input connectors and three output connectors to the state chart block using **Edit > Add Connector** (or ⇥ toolbar button). Your state chart block will look like this:



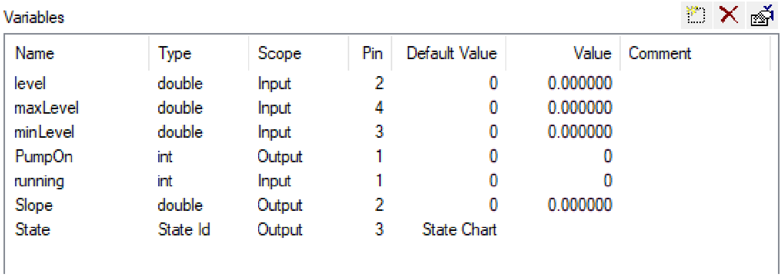
Note: Activate **View > Connector Labels** to display the connector labels.

2. To edit the attributes for each variable, CTRL+right-click the **state chart** block.
3. Click the **Activity Manager** tab.



Name	Type	Scope	Pin	Default Value	Value	Comment
var1	double	Input	1	0	0.000000	
var2	double	Input	3	0	0.000000	
var3	double	Input	2	0	0.000000	
var4	double	Input	4	0	0.000000	
var5	double	Output	1	0	0.000000	
var6	double	Output	2	0	0.000000	
var7	double	Output	3	0	0.000000	

4. To edit a variable, double-click each **attribute** (name, type, and scope) and make the changes shown below:



Name	Type	Scope	Pin	Default Value	Value	Comment
level	double	Input	2	0	0.000000	
maxLevel	double	Input	4	0	0.000000	
minLevel	double	Input	3	0	0.000000	
PumpOn	int	Output	1	0	0	
running	int	Input	1	0	0	
Slope	double	Output	2	0	0.000000	
State	State Id	Output	3	State Chart		

5. Click **OK**.

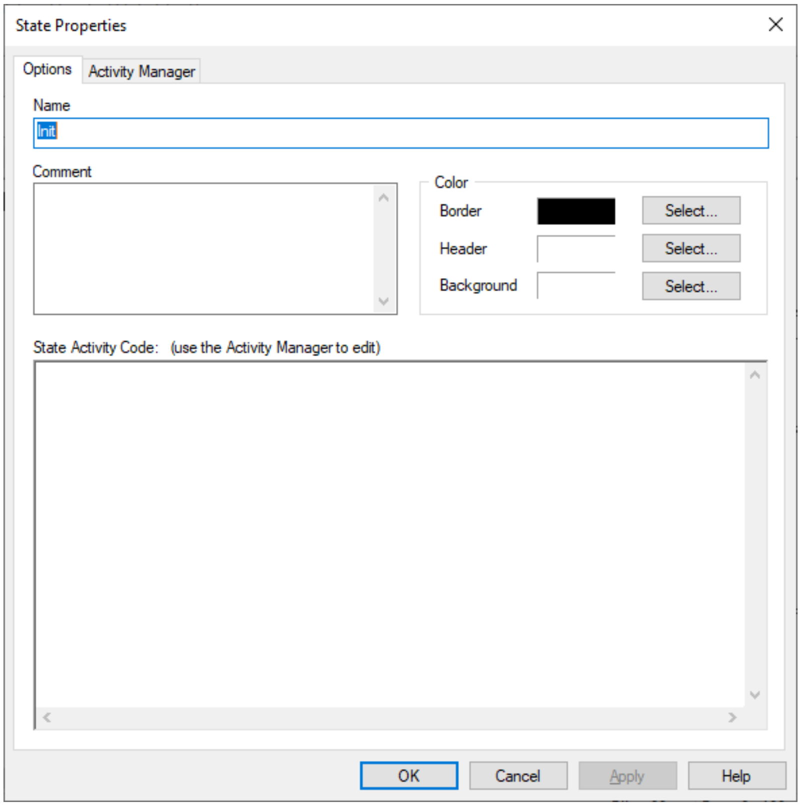
Configuring states

Configuring a state includes naming it and optionally assigning a behavior (C code) to selected actions. A state has three pre-defined actions (entry, exit, and do) and any number of inner transitions that are fired by triggers.

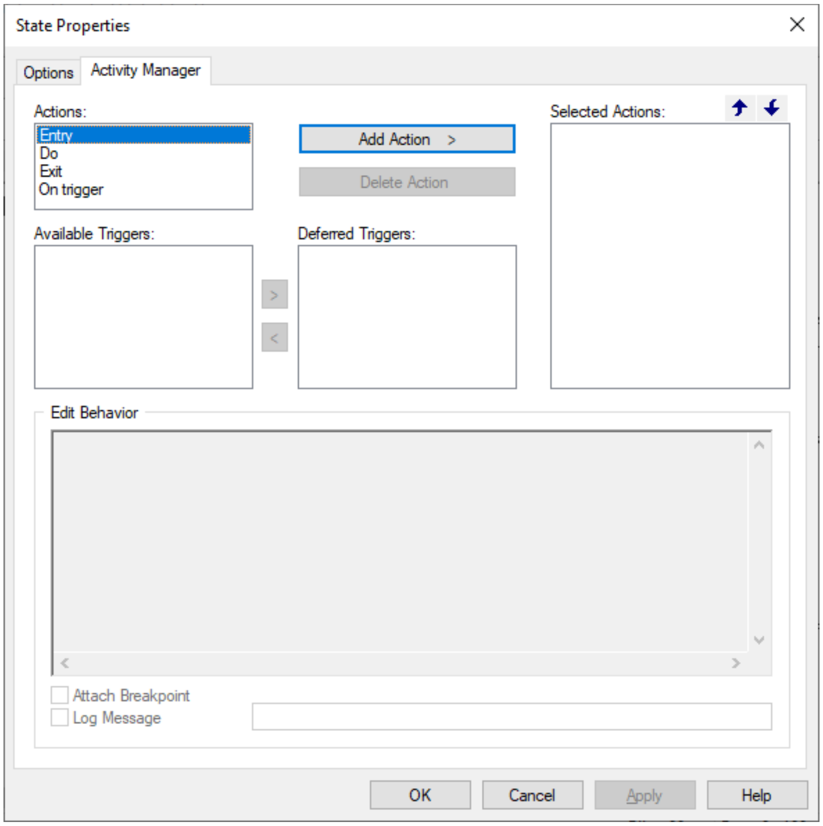
1. Right-click the **State1 title bar**.

2. In the State Properties dialog under Options, do the following:

- Under Name, enter **Init**.
- Under Color, select a **color** for the border and a **color** for the header.

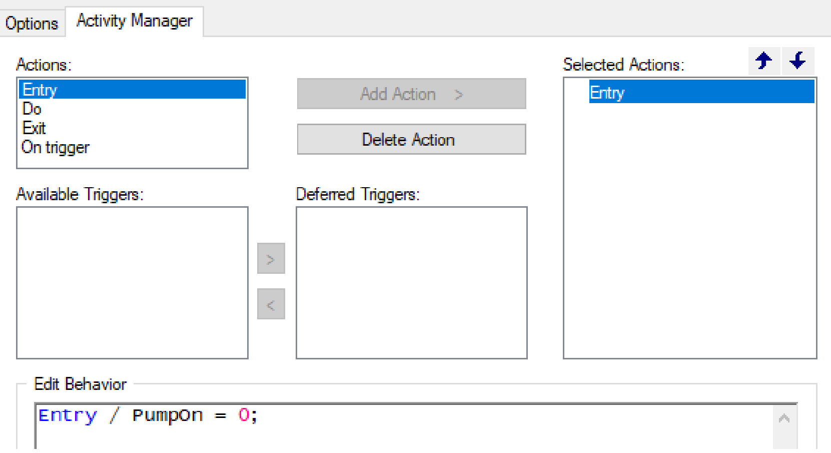


3. Click the Activity Manager tab to select an action and enter behaviors.



4. Under Actions, select **Entry** and click **Add Action**.

5. Under Edit Behavior, enter the **C code** to indicate the pump is OFF, as shown below:

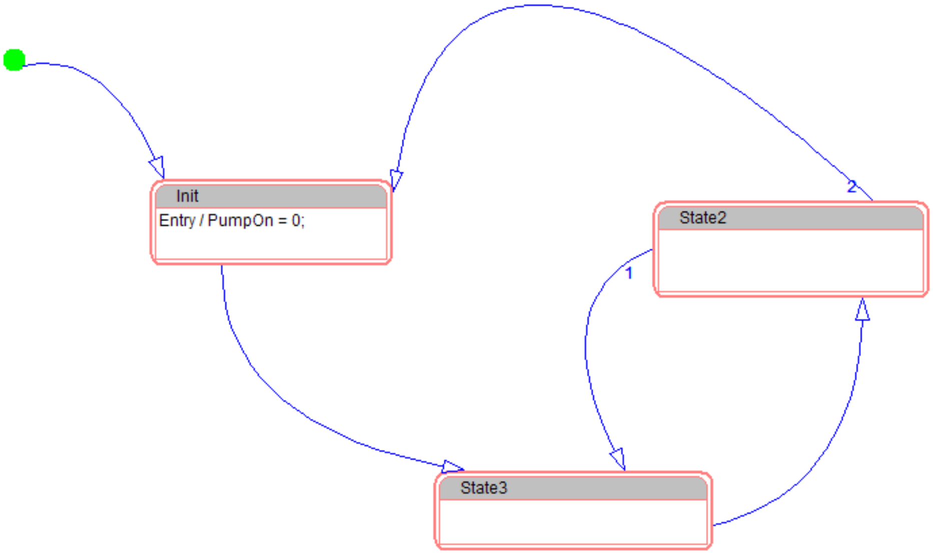


```
Entry / PumpOn = 0;
```

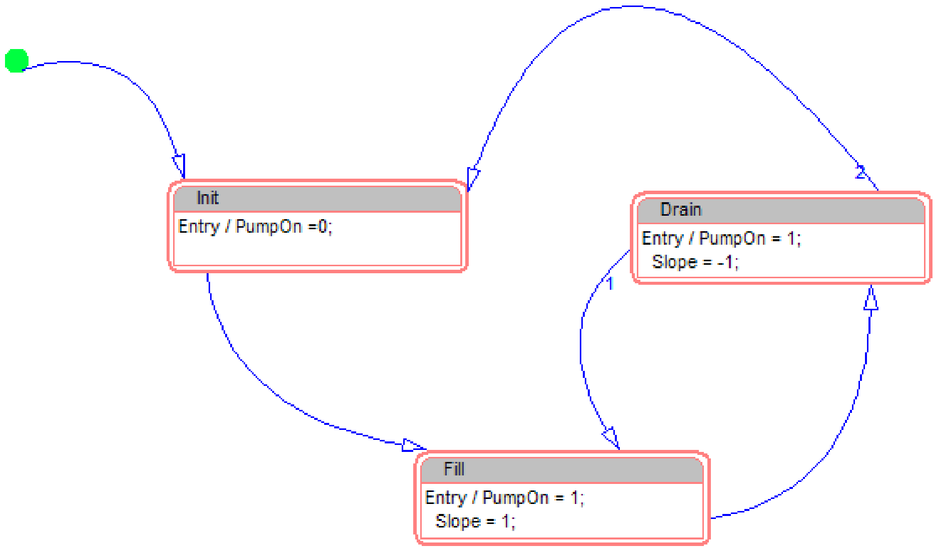
Note: If you are unfamiliar with the C language, refer to C: A Software Engineering Approach.

6. Click **OK**.

   Your state chart will look like this:



7. Repeat steps 1 – 6 for **State2** and **State3** so that your state chart looks like this:



Defining transition specifications

A transition has the following basic format:

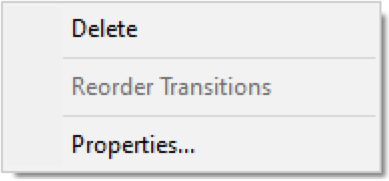
trigger(s) [guard] / behavior

Together, the triggers and guards represent a logical expression that evaluates to TRUE or FALSE. When the logical expression is TRUE, the transition is taken to the next state. When it is FALSE, another transition is tested; if there are no other transitions, the state of origin remains active.

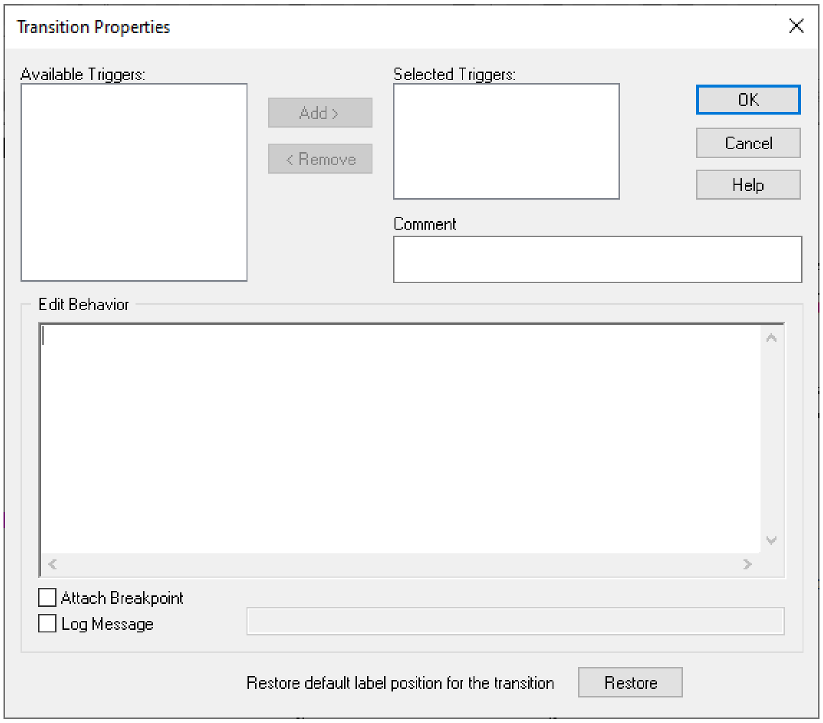
For this simple example, there are no triggers or behaviors in the transition specification, only guards.

1. Right-click the **transition** between **Init state** and **Fill state**.

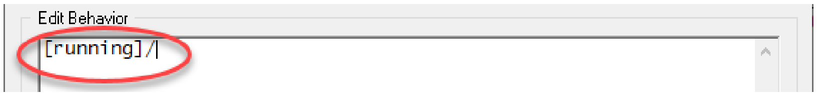
   The following pop-up menu appears:

2. Click **Properties**.



3. Under Edit Behavior, enter the **guard** using the C language. Enclose the code in square brackets and terminate with a forward slash.



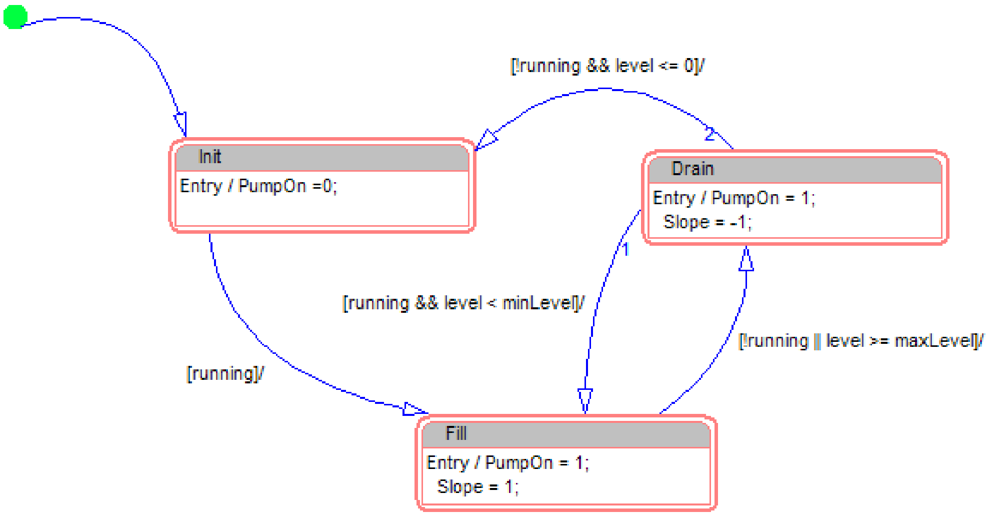
```
[running]/
```

4. Click **OK**.

5. Repeat these steps to add the following guards to the remaining transitions.

Transition	Guard
Fill to Drain	[!running || level > = maxLevel]/
Drain to Init	[!running && level <=0]/
Drain to Fill	[running && level < minLevel]/

Your state chart will look like this:

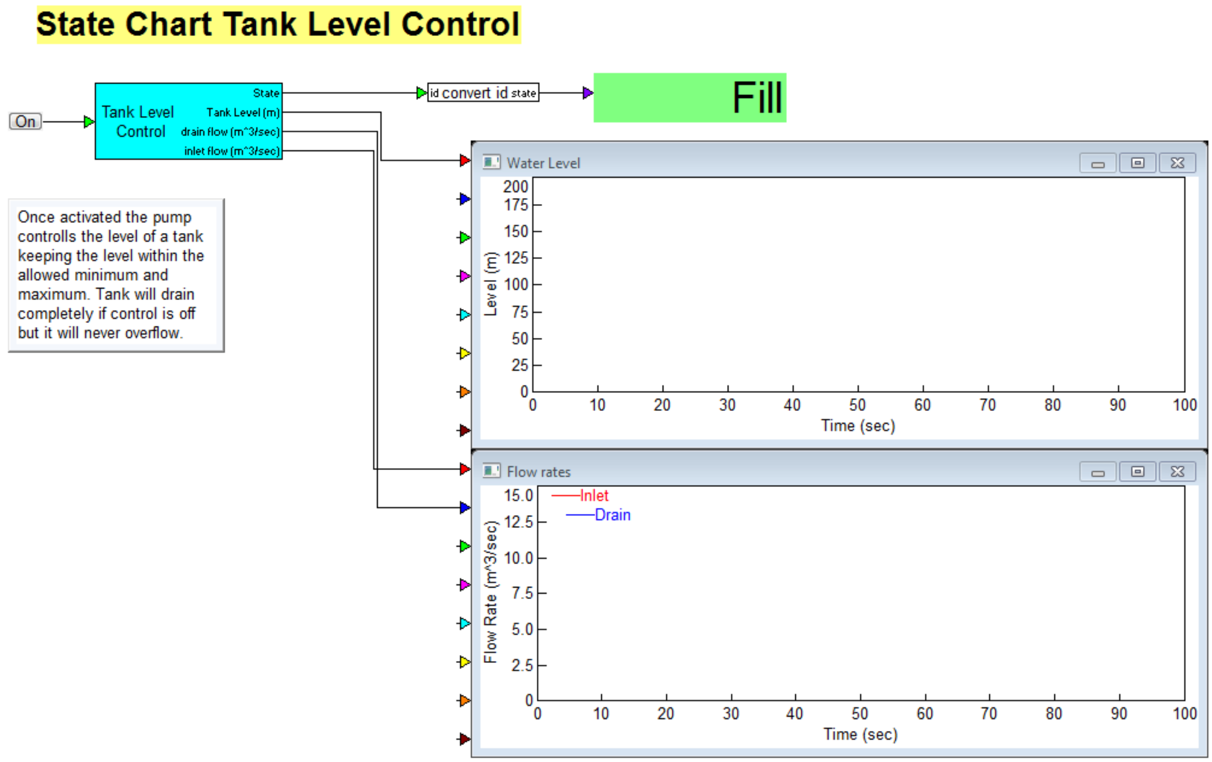


Annotating state charts

You use the comment and label blocks to annotate your state charts. These blocks are located under both the State Charts and Blocks > Annotation menus.

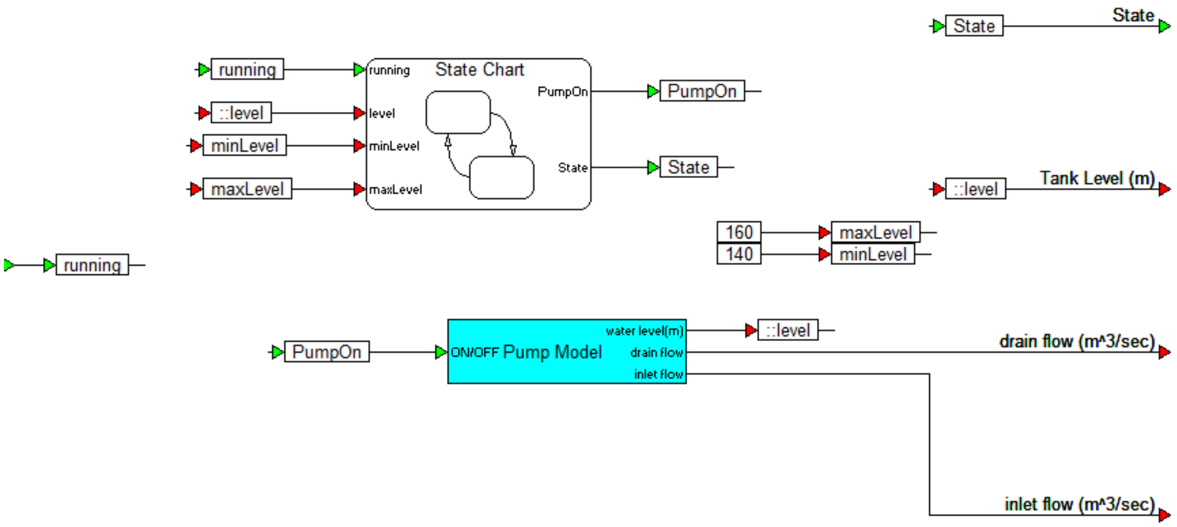
Setting up the block diagram to interact with the state chart

For the state chart to exchange data with the block diagram, you must create the pump dynamics, and link the dynamics via variables to the state chart. At this point, you can open an existing diagram (**Examples > Applications > State Charts > stateChartTank)** that contains a state chart like the one you just created.



Once activated the pump controlls the level of a tank keeping the level within the allowed minimum and maximum. Tank will drain completely if control is off but it will never overflow.

The state chart is inside **Tank Level Control**.



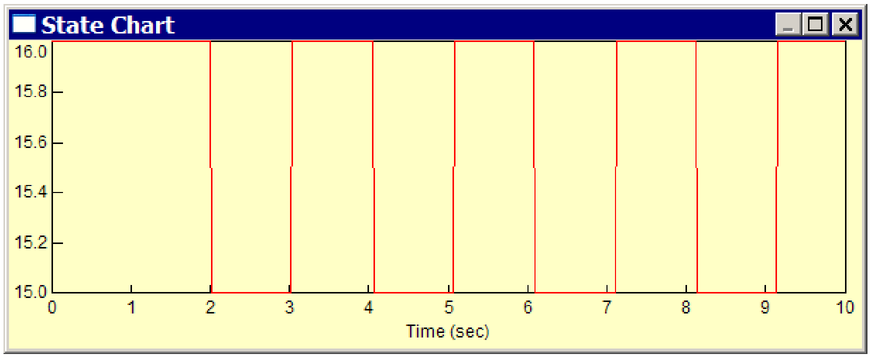
Pump Model contains an integrator block to define the pump logic.

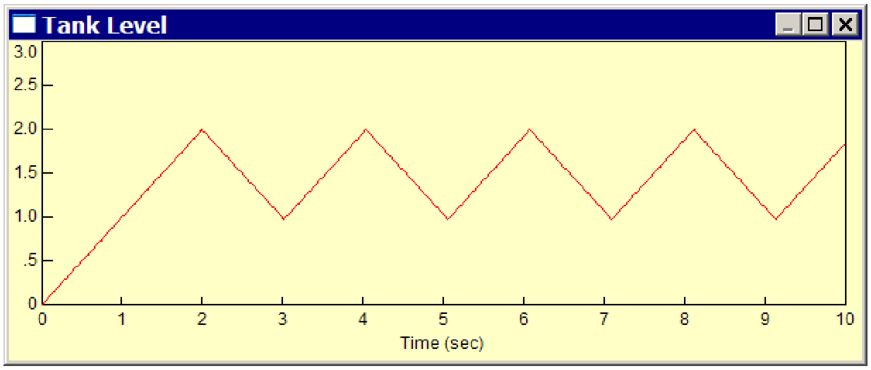
Simulating the state chart

Before simulating the state chart, you can examine and set your simulation parameters in the System Properties dialog.

- Choose **Simulate > Go**, or press ▶ in the toolbar.

In the state chart, the active state is highlighted to show it is executing. In the top level Embed diagram, the two plots State Chart and Tank Level monitor the pump and the tank level, respectively.





Note: At the top level of the diagram, the button block wired into **Tank Level Control** must be turned ON.

Importing blocks from PSIM

In PSIM, you cannot generate code for Arduino, Raspberry Pi, STMicroelectronics, and most Texas Instruments devices. You can, however, create DLLs from PSIM schematics and automatically import the DLLs as blocks into Embed. These imported blocks can be used to represent controllers and plants.

What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
PSIM	https://altair.com/psim ; you only need PSIM if you want to open the PSIM schematic, you don't need it to follow along with the example

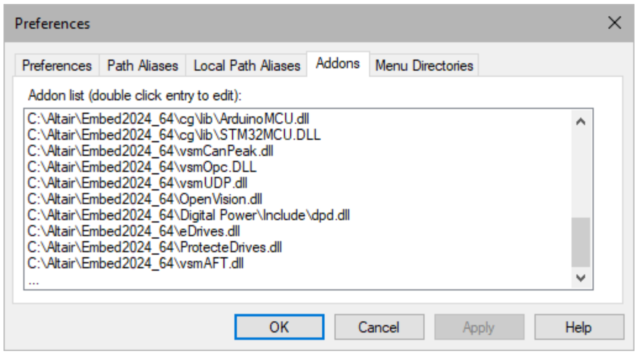
Simulating and validating data with an imported block from PSIM

This example uses an existing Embed diagram — **PSIM Sim Example** — to compare the simulated performance of a closed-loop control system. The diagram is divided into two sections: MODEL-1 and MODEL-2. In MODEL-1, the **Plant** compound block is designed entirely in Embed. In MODEL-2, **PSIM Codegen DLL Plant** is a compound block that contains a plant DLL generated in PSIM and exported to Embed. When simulated, MODEL-1 and MODEL-2 produce the same results.

Note: To execute the code on a Texas Instruments F28069M device, click here.

1. Add the **PSIM-generated block** to Embed's **Imported Blocks** menu.

a. Click **Edit > Preferences > Addons**.



Addon list (double click entry to edit):

```
C:\Altair\Embed2024_64\cg\lib\ArduinoMCU.dll
C:\Altair\Embed2024_64\cg\lib\STM32MCU.DLL
C:\Altair\Embed2024_64\vsmCanPeak.dll
C:\Altair\Embed2024_64\vsmOpc.DLL
C:\Altair\Embed2024_64\vsmUDP.dll
C:\Altair\Embed2024_64\OpenVision.dll
C:\Altair\Embed2024_64\Digital Power\Include\dpd.dll
C:\Altair\Embed2024_64\eDrives.dll
C:\Altair\Embed2024_64\ProtecteDrives.dll
C:\Altair\Embed2024_64\vsmAFT.dll
...
```

b. Scroll to the bottom of the **DLL list** and click **…**.

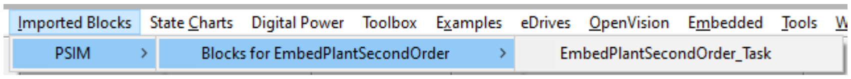
c. In the Open dialog, navigate to ***<Embed-installation-directory>*\Examples\Blocks\Extensions\ImportedBlocks\PSIM\EmbedPlantSecondOrder (C Code).**

d. Select **EmbedPlantSecondOrder.dll** and click **Open**.

This DLL was previously generated in PSIM. It is added to the bottom of the Addons list.

e. Click **OK**, or press **ENTER**.

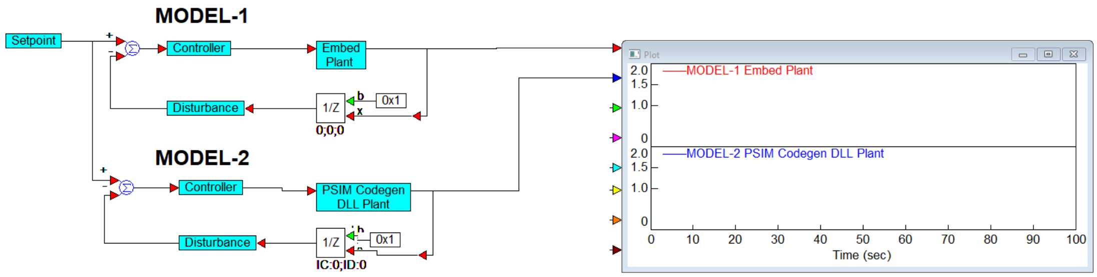
A block corresponding to the DLL is added to the **Imported Blocks** menu.



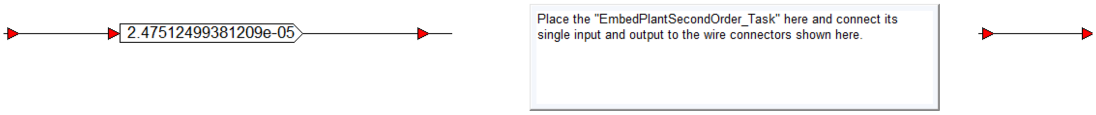
Note that **_Task*n*** is appended to the block name. If you generate additional DLLs from the schematic, _Task*n* differentiates them.

2. Add the **imported block** to the **PSIM Sim Example** diagram.

a. Click **Examples > Blocks > Extensions > Imported Blocks > PSIM > PSIM Sim Example**.

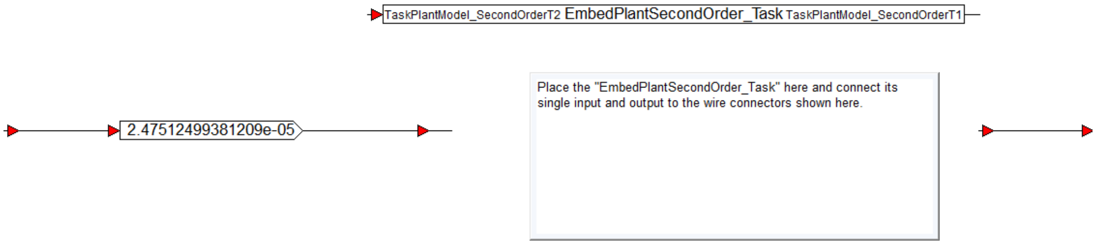


b. Under **MODEL 2**, right-click **PSIM Codegen DLL Plant** compound block to dive into the next lower level of the block.



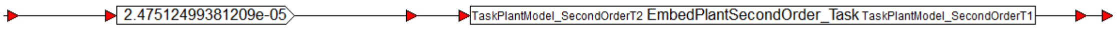
Place the "EmbedPlantSecondOrder_Task" here and connect its single input and output to the wire connectors shown here.

c. Click **Imported Blocks > PSIM > Blocks for EmbedPlantSecondOrder > EmbedPlantSecondOrder_Task** and insert the **imported PSIM block** into the diagram.


TaskPlantModel_SecondOrderT2
EmbedPlantSecondOrder_Task
TaskPlantModel_SecondOrderT1
2.47512499381209e-05

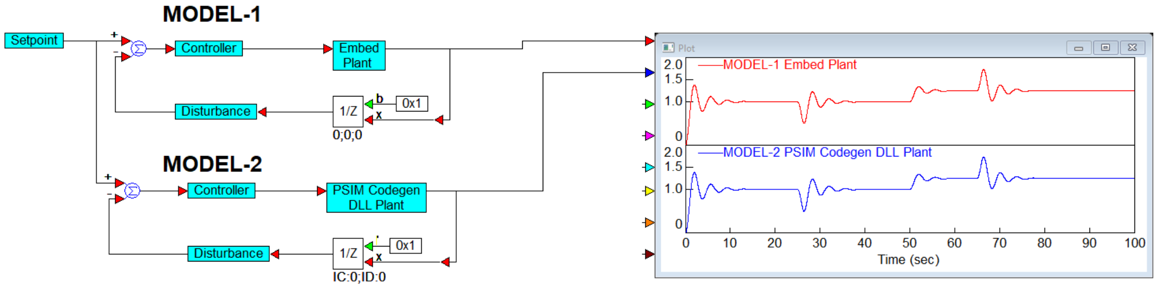
Place the "EmbedPlantSecondOrder_Task" here and connect its single input and output to the wire connectors shown here.

d. Replace the **comment** block with the **imported PSIM** block and wire it into the diagram.


2.47512499381209e-05
TaskPlantModel_SecondOrderT2 EmbedPlantSecondOrder_Task TaskPlantModel_SecondOrderT1

e. Right-click on empty screen space to return to the top level of the diagram.

3. Start the simulation and compare the simulation results.



Importing blocks from Twin Activate

In Twin Activate, you cannot generate code for Arduino, Raspberry Pi, STMicroelectronics, and most Texas Instruments devices. You can, however, create DLLs from Twin Activate diagrams and automatically import the DLLs as blocks into Embed. These imported blocks can be used to represent controllers and plants.

What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Twin Activate	https://altair.com/twin-activate ; you only need Twin Activate if you want to open the Twin Activate diagram, you don't need it to follow along with the example

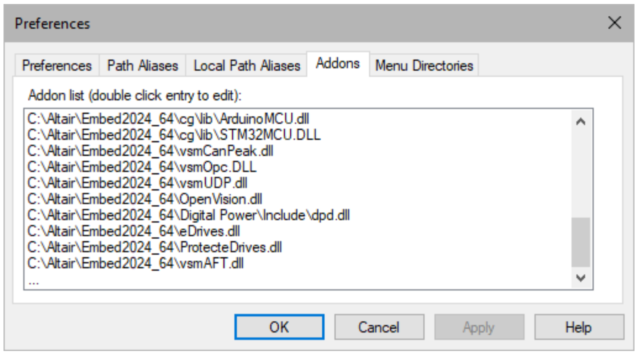
Simulating and validating data with an imported block from Twin Activate

This example uses an existing Embed diagram — **Twin Activate Sim Example** — to compare the simulated performance of a closed-loop control system. The diagram is divided into two sections: MODEL-1 and MODEL-2. In MODEL-1, the **Embed Controller** compound block is designed entirely in Embed. In MODEL-2, **Twin Activate Codegen DLL Plant** is a compound block that contains a controller DLL generated in Twin Activate. When simulated, MODEL-1 and MODEL-2 should produce the same results.

Note: To execute the code on a STMicroelectronics STM32410RB device, click here.

1. Add the **Twin Activate-generated block** to Embed's **Imported Blocks** menu.

   a. Click **Edit > Preferences > Addons**.



Preferences

Addon list (double click entry to edit):

```
C:\Altair\Embed2024_64\cg\lib\ArduinoMCU.dll
C:\Altair\Embed2024_64\cg\lib\STM32MCU.DLL
C:\Altair\Embed2024_64\vsmCanPeak.dll
C:\Altair\Embed2024_64\vsmOpc.DLL
C:\Altair\Embed2024_64\vsmUDP.dll
C:\Altair\Embed2024_64\OpenVision.dll
C:\Altair\Embed2024_64\Digital Power\Include\dpd.dll
C:\Altair\Embed2024_64\eDrives.dll
C:\Altair\Embed2024_64\ProtecteDrives.dll
C:\Altair\Embed2024_64\vsmAFT.dll
...
```

   b. Scroll to the bottom of the **DLL list** and click **….**

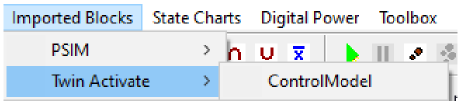
   c. In the Open dialog, navigate to **<*Embed-installation-directory*>\Examples\Blocks\Extensions\ImportedBlocks\Twin Activate\Code Generation.**

   d. Select **ControlModel.dll** and click **Open**.

   This DLL was previously generated in Twin Activate. It is added to the bottom of the Addons list.

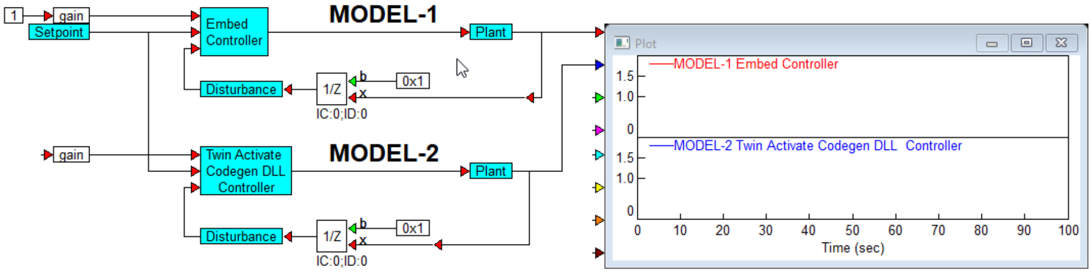
e. Click **OK**, or press **ENTER**.

A block corresponding to the DLL is added to the **Imported Blocks** menu.

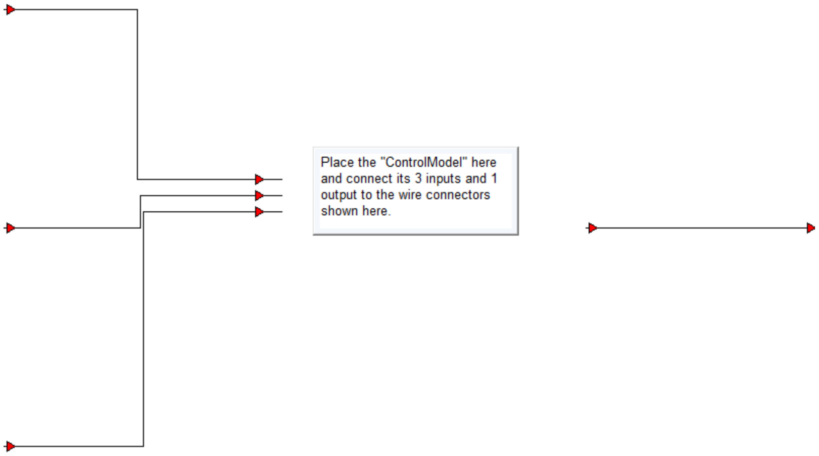


2. Add the **imported block** to the **Twin Activate Sim Example** diagram.

a. Click **Examples > Blocks > Extensions > Imported Blocks > Twin Activate > Twin Activate Sim Example**.

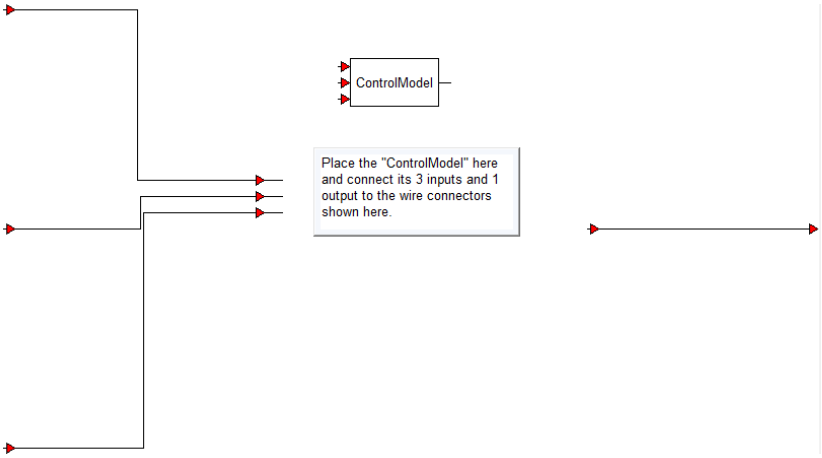


b. Under **MODEL-2**, right-click **Twin Activate Codegen DLL Controller** compound block to dive into the next lower level of the block.



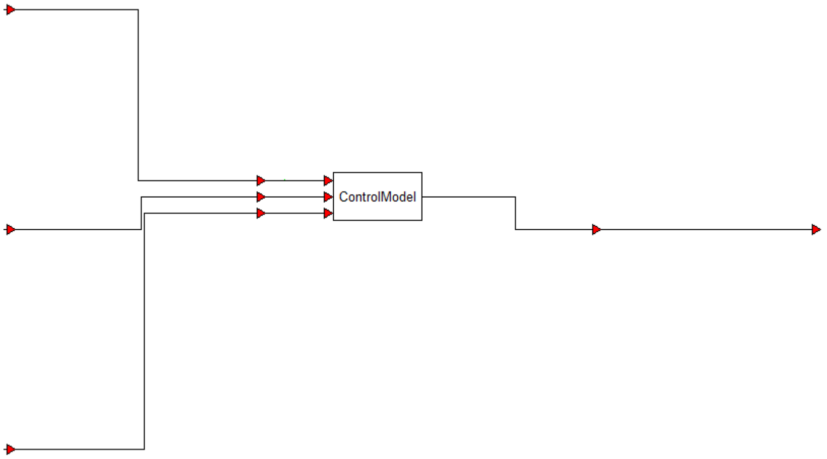
Place the "ControlModel" here and connect its 3 inputs and 1 output to the wire connectors shown here.

c. Click **Imported Blocks > Twin Activate > ControlModel** and insert the **block** into the diagram.



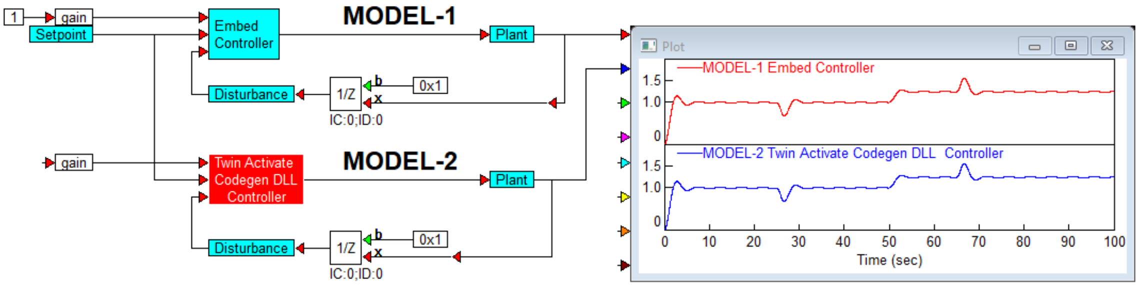
Place the "ControlModel" here and connect its 3 inputs and 1 output to the wire connectors shown here.

d. Replace the **comment** block with the **ControlModel** block and wire it into the diagram.



e. Right-click on empty screen space to return to the top level of the diagram.

3. Start the simulation and compare the simulation results of **MODEL-1 Embed Controller** and **MODEL-2 Twin Activate Codegen DLL Controller** in the plot.



Converting a floating-point elevator door system to fixed-point

This example describes how to implement an elevator door control system in floating point, then convert the controller to scaled, fixed point. The diagrams used are under Examples > Fixed Point:

Floating-Point Diagram: otis_elevator_regular

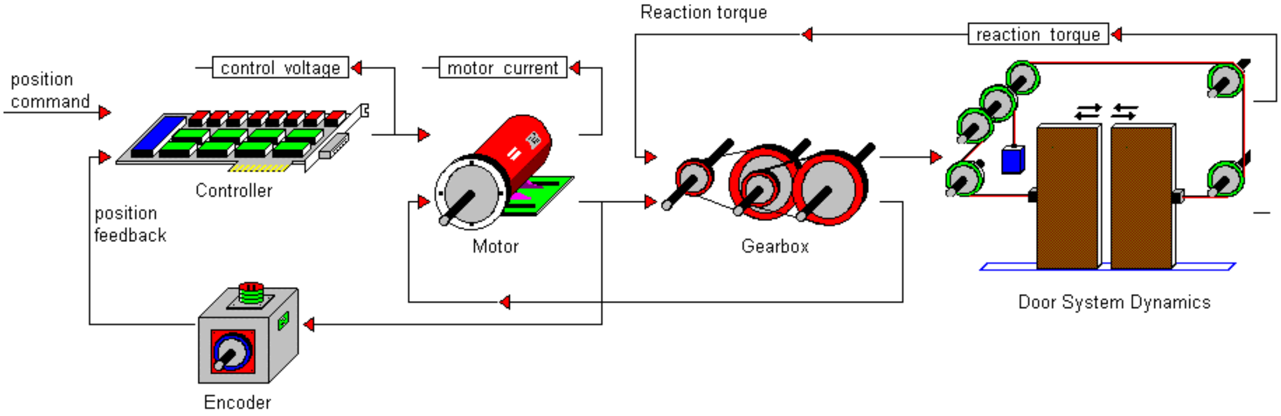
Fixed-Point Diagram: otis_elevator_fixed_point

Floating-point implementation

The **otis_elevator_regular** diagram is a simplified elevator door control system consists of a DC motor driving a gearbox that in turn manipulates the door position through a series of pulleys. The controller accepts open and close commands as inputs, and controls the magnitude and polarity of voltage that is applied to the DC motor. An encoder provides motor rotor shaft position feedback to the control system.

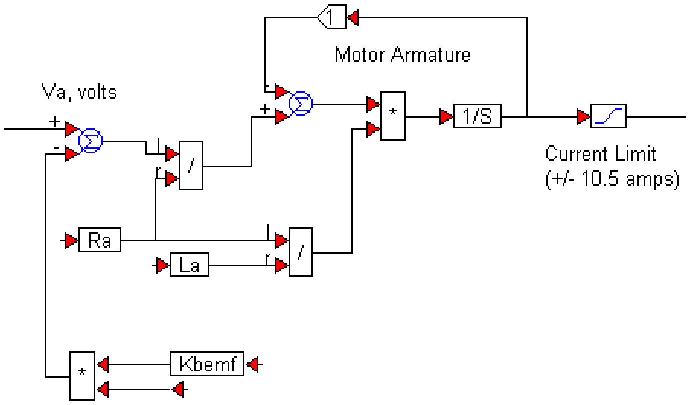
1. Click **Examples > Fixed Point**.

2. Select **otis_elevator_regular**.

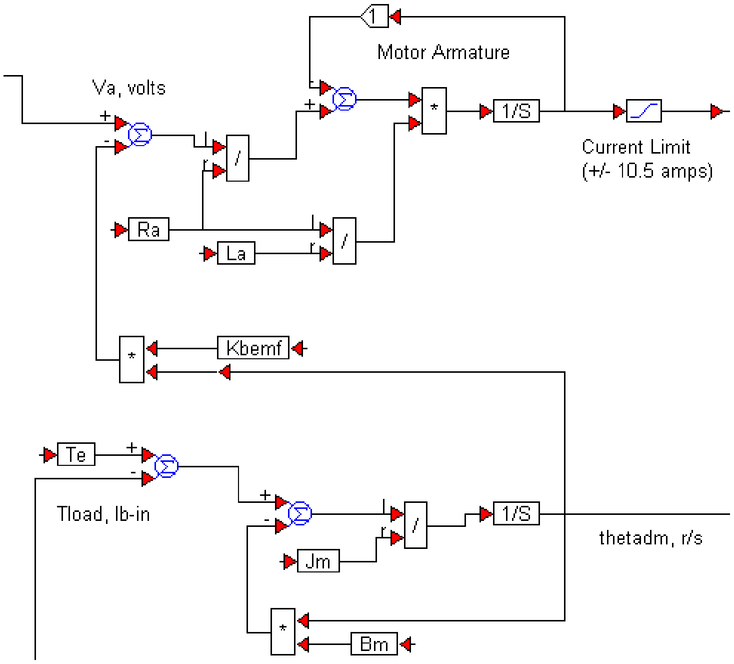


Constructing the motor

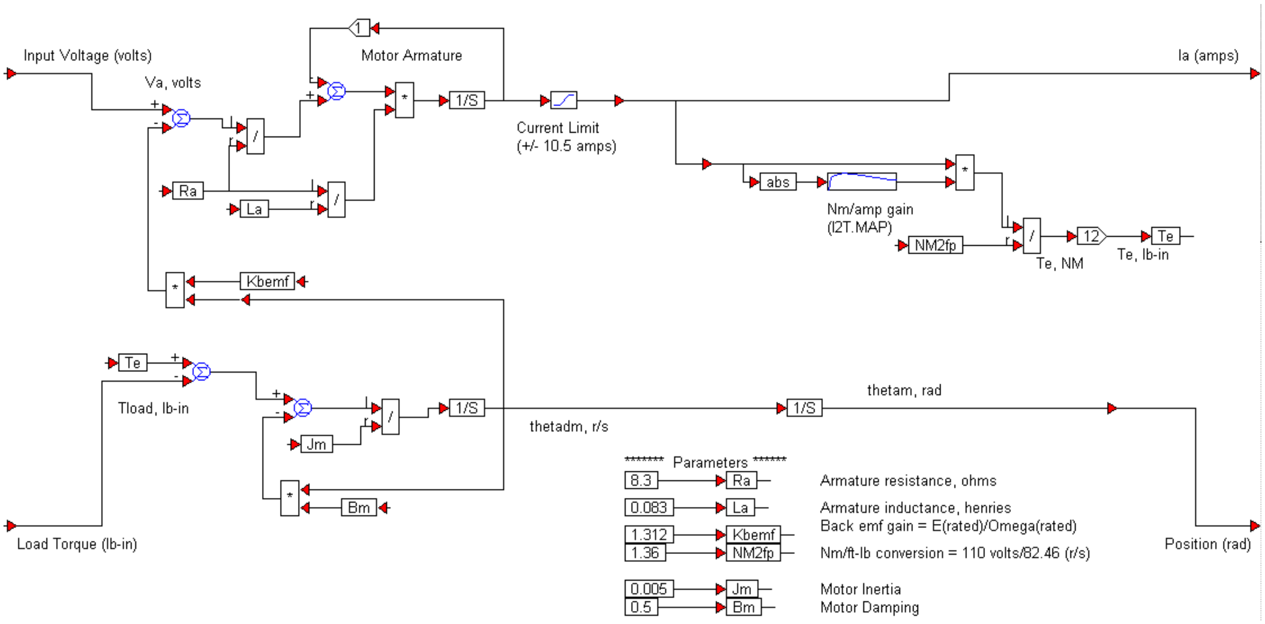
To model the DC motor, the effective voltage is the difference between the applied voltage and back-emf. The motor armature is modeled as a simple first-order system with resistance *Ra* and inductance *La*. The motor current is limited to +/- 10.5 A.



To compute the back-emf, the back-emf gain *Kbemf* is multiplied by the angular velocity. Angular velocity is computed using the electrical torque and load torque.



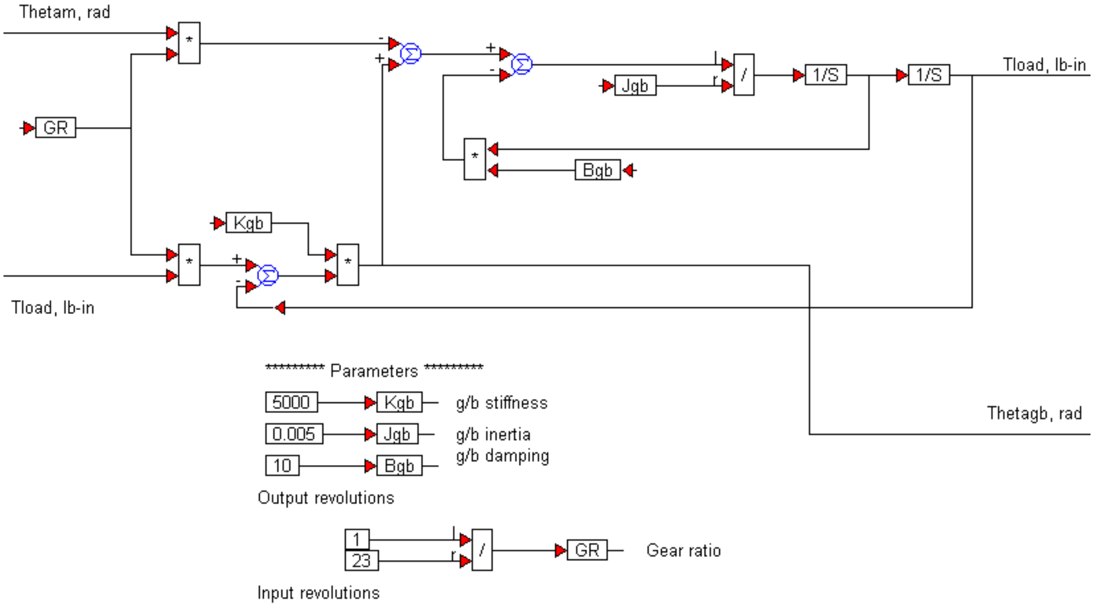
A set of const blocks defines the required motor parameters. The motor angular velocity *thetadm* is integrated to yield position. The electrical torque *Te* is computed using the motor armature current, and the I2T.MAP look-up table contains the motor's current-torque characteristics. This data is obtained from the motor's specification sheets or through the vendor. The complete motor diagram is shown below:



Note: The eDrives > eMotor (Legacy) toolbox includes a full set of pre-configured, pre-tested, and ready-to-use blocks, such as motors, amplifiers, loads, sensors, and controllers.

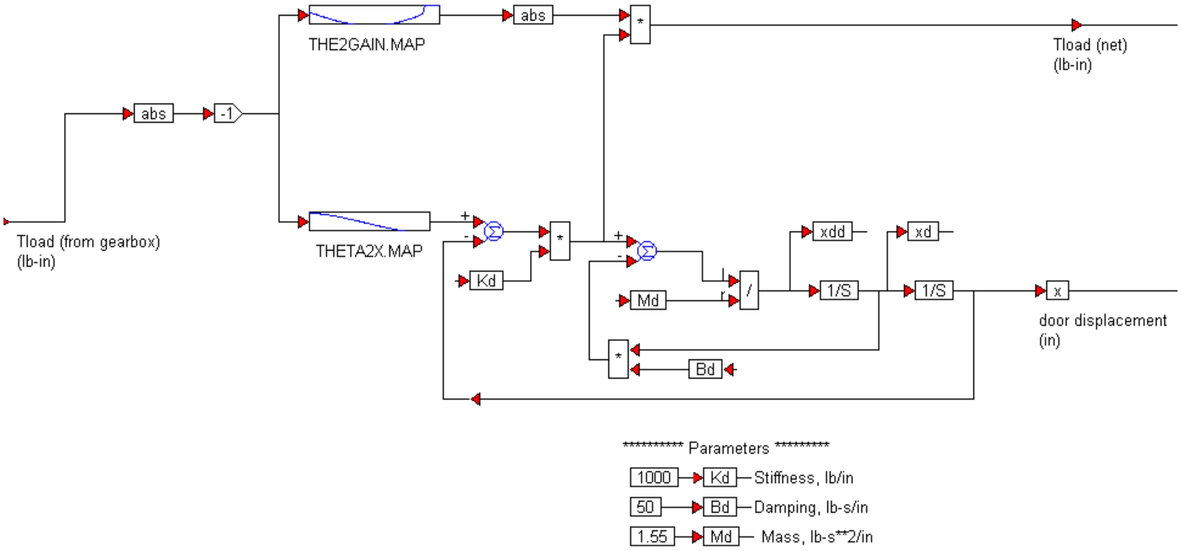
Constructing the gearbox

In addition to increasing or decreasing the number of output revolutions relative to the input revolutions, a typical gearbox also introduces additional inertia, stiffness, and damping effects into the system. A basic rotational load diagram is implemented below. Detailed rotational and translational diagrams are included in the eMotor toolbox. The completed gearbox diagram can be realized as shown below:



Constructing the door system

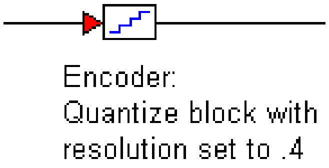
At a basic level, the door system can be thought of as an additional translational load that is connected to the motor through an intermediate rotational load (that is, the gearbox.) As such, the door-system imposes its own mechanical elements to the system: mass, stiffness, inertia, and damping. The input to the diagram is the rotation/position of the gearbox, and the outputs are the linear displacement of the door assembly in inches and the total system mechanical load torque in lb-in. The simple second order dynamical diagram and the related system parameters can be implemented as shown below:



Two look-up tables — THE2GAIN.MAP and THETA2X.MAP — are used for easy modeling of the dependency of the load torque aspects and the relationship between the angular gear-shaft motion and the linear motion of the door-assembly.

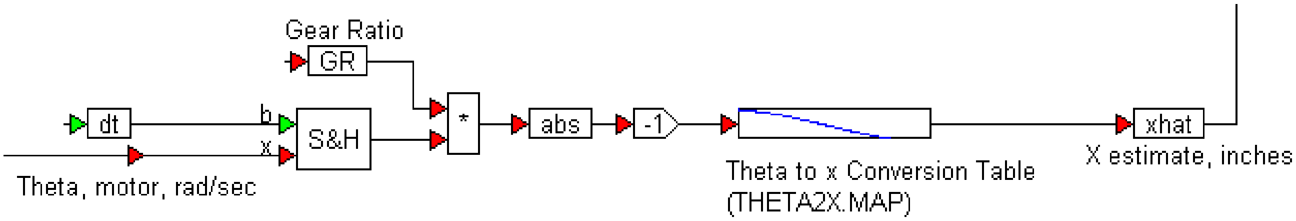
Constructing the encoder

A basic encoder can be modeled simply as a quantizer using the quantize block with the resolution set to 0.4.



Constructing the controller

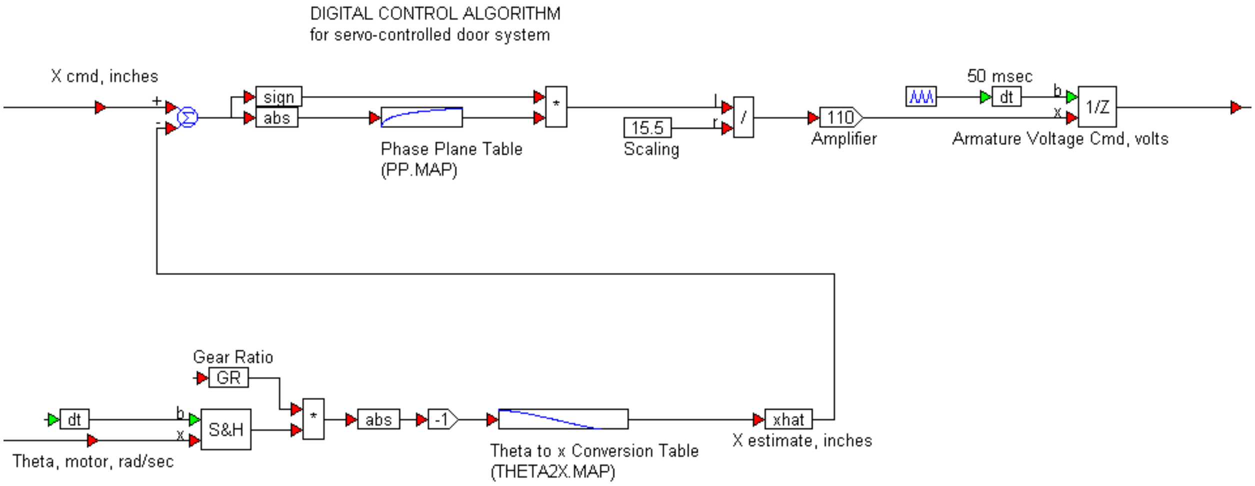
The controller takes two inputs: the open/close command and the actual position of the gear-shaft as estimated by the encoder. The encoder feedback is converted into inches, the same units as the command input using simple arithmetic, and the look-up table THETA2X.MAP gives the relationship between angular gear-shaft position and equivalent door-assembly linear displacement as previously seen. The conversion logic is shown below:



The error is calculated by subtracting the estimated actual door displacement from the commanded displacement. While the absolute value of the error determines the amplitude of the control voltage to be applied, the output of the sign block is used to determine the polarity of the voltage to be applied (that is, whether the door is to be opened or closed).

Another look-up table (PP.MAP) determines the recommended control voltage ratio. The recommended control voltage is converted into a ratio by scaling it with the maximum value from the table (15.5) and fed to a simple proportional control stage represented by a gain of 110. The output of the proportional stage is sampled at 50ms to represent the physical realities of implementing the control logic on a digital target such as a DSP or a microcontroller.

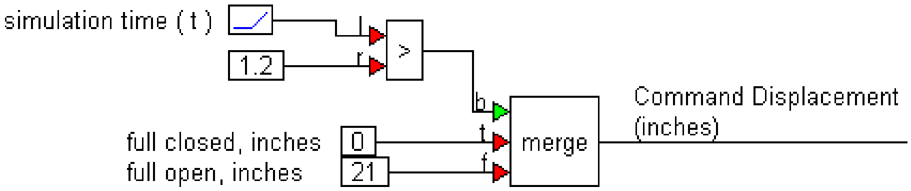
The complete controller structure is as shown below:



Constructing the open/close command

To test the system, an elevator door open-close cycle is constructed. A typical cycle is to open for 1.2s followed with a close command. An open command means the desired door displacement is 21 inches, while the close command implies that the desired door displacement is 0 inches.

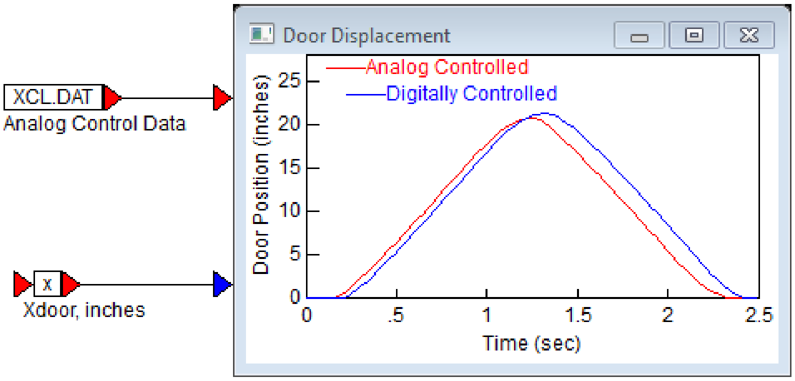
Using a ramp block to access the current value of simulated time *(t)* and using if-then-else logic with a merge block, you can implement the open/close cycle as shown below:



After connecting all the subsystems and assigning variables for monitoring (*control_voltage*, *motor_current* and *reaction_torque*), the system is complete.

Fixed-point implementation of the controller

When calibrating a new control design, a common way to validate a new design is to compare its performance with an existing floating-point implementation. In the **elevator_door_regular** diagram, system performance of the digital control system being designed is compared with the performance of an existing analog control system.



The door displacement profile of an existing system (XCL.DAT) is brought into the simulation using an import block and compared to the door displacement profile resulting from the current implementation.

By simulating the diagram, you can make refinements in the control strategy, as well as fine-tune the controller performance.

Once it is determined that the floating-point controller is performing adequately, the next step is to implement the controller using limited precision fixed-point blocks. This lets you simulate the behavior of the controller as it would behave as an embedded system in a fixed-point target, such as a DSP or a microcontroller.

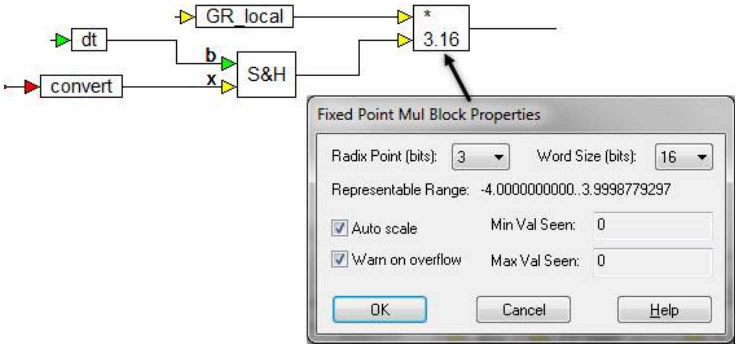
Converting to scaled fixed-point control

The underlying principle in converting an existing floating-point subsystem into an equivalent scaled fixed-point system is that every input, operation, and output be configured to reflect the realities of the target. Fundamental details — such as whether the target is an 8-, 12-, 16-, 24- or 32-bit processor — become important. As you traverse the controller implementation, from left to right, you encounter several operations, including summation, sample-and-hold, absolute value, sign, multiplication, constant, division, gain, and unit delay. Using Fixed Point blocks, each of these operations is replaced with the equivalent fixed-point operator, assuming a 16-bit target.

Blocks such as unitDelay (1/Z) and sampleHold are fixed-point-aware; that is, they automatically adjust to the incoming data type. Consequently, they can be used as is.

Constructing the encoder feedback

The output of variable *GR_local* and the output of the sampleHold block are wired into a fixed-point mul block with the following configuration:



Fixed Point Mul Block Properties

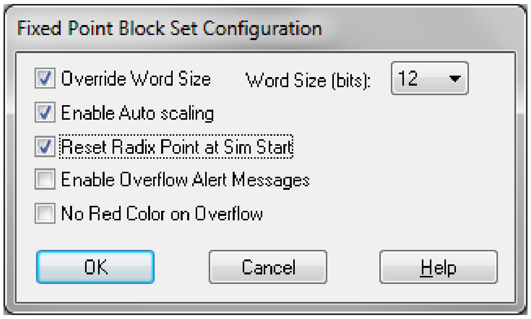
Radix Point (bits): 3   Word Size (bits): 16

Representable Range: -4.0000000000..3.9998779297

☑ Auto scale   Min Val Seen: 0

☑ Warn on overflow   Max Val Seen: 0

You can choose the radix point bits or select auto scaling. This option, together with the global settings in the Tools > Fixed Point Block Set Configure dialog, let you automatically monitor the maximum and minimum values seen by the block, and adjust the radix point bits to yield maximum precision while preventing numerical overflow.



Fixed Point Block Set Configuration

☑ Override Word Size   Word Size (bits): 12

☑ Enable Auto scaling

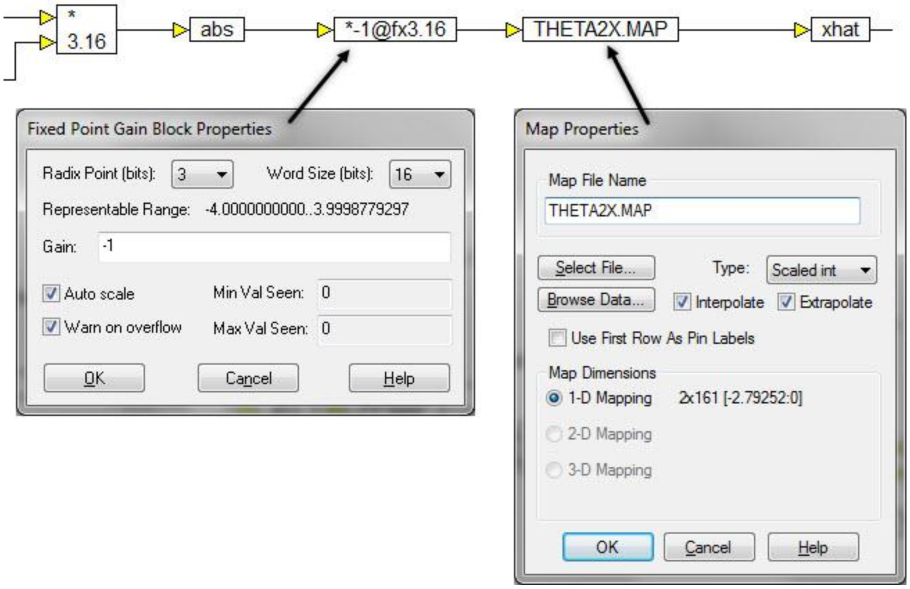
☑ Reset Radix Point at Sim Start

☐ Enable Overflow Alert Messages

☐ No Red Color on Overflow

Safely maximizing the dynamic range of each computation is by far the most time-consuming component in the rapid prototyping cycle. Fixed Point blocks reduce this tedious exercise to a few mouse clicks.

Next, the mul output is connected to an abs block to compute the absolute value, which in turn is fed into a fixed-point gain block. The gain output is wired into a map block, which points to the look-up table data file THETA2X.MAP and has a Scaled Int data type. THETA2X.MAP output is fed into the variable *xhat*.



Fixed Point Gain Block Properties

Radix Point (bits): 3 Word Size (bits): 16

Representable Range: -4.0000000000..3.9998779297

Gain: -1

☑ Auto scale Min Val Seen: 0

☑ Warn on overflow Max Val Seen: 0

Map Properties

Map File Name: THETA2X.MAP

Type: Scaled int

☑ Interpolate ☑ Extrapolate

☐ Use First Row As Pin Labels

Map Dimensions

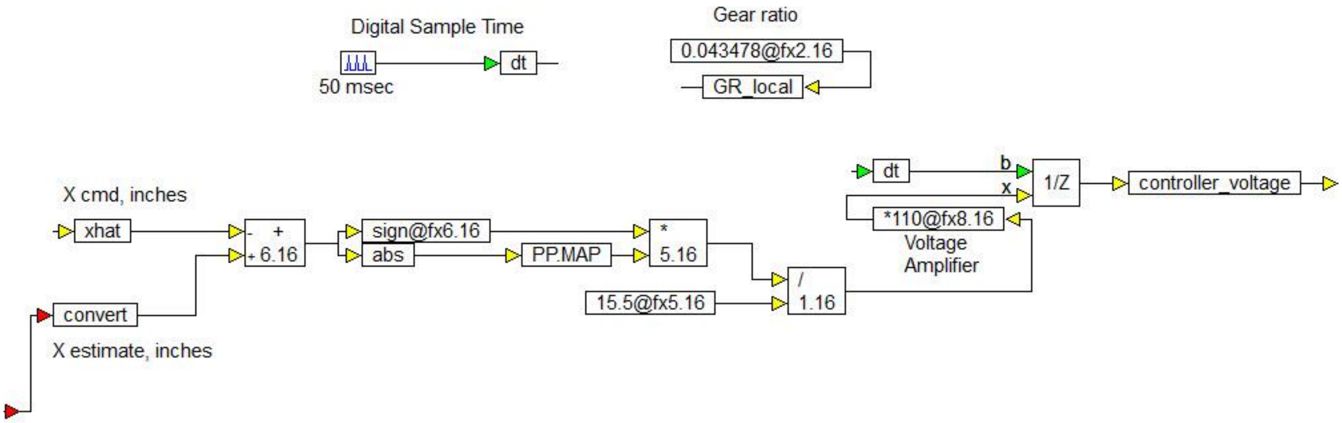
● 1-D Mapping 2x161 [-2.79252:0]

○ 2-D Mapping

○ 3-D Mapping

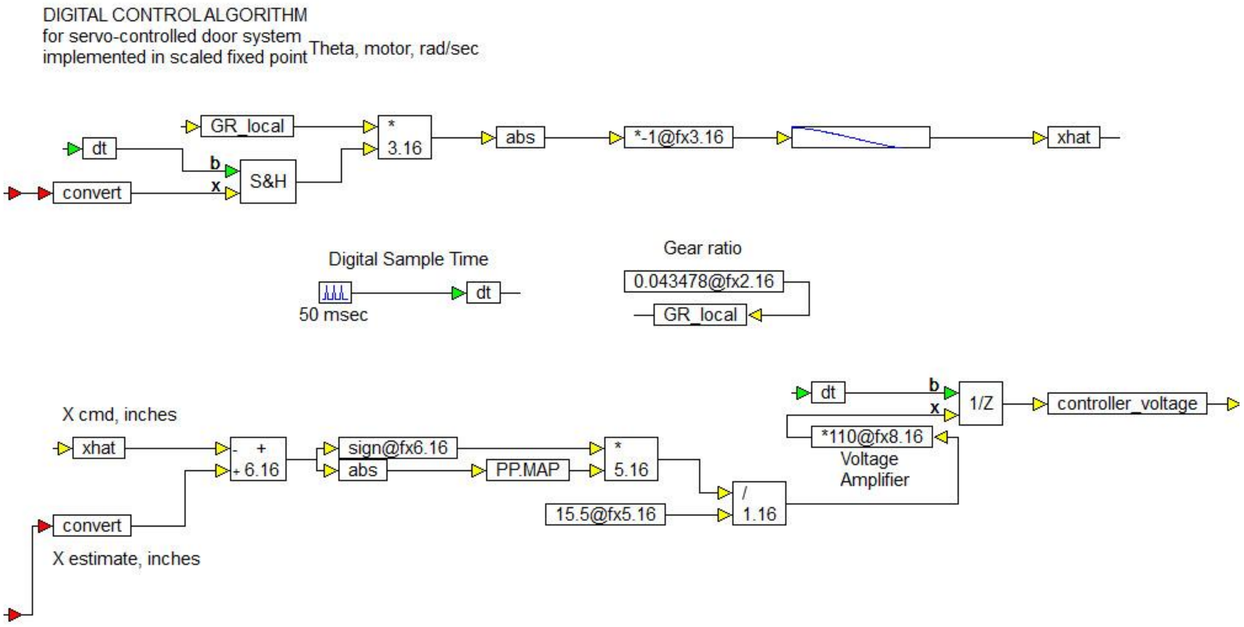
Constructing the control law

The control law is constructed using fixed-point sum, mul, div, const, and gain blocks. The gear ratio *GR_local* is defined as 0.043478 using a fixed-point const block. A 50 ms pulseTrain defines *dt*. The map block points to PP.MAP and sets the data type to scaled integer. The resulting control law implementation is:



Completing the controller implementation

To complete the controller implementation, the control law segment is connected to the Encoder feedback segment, and the output of the unitDelay block to the output:



The two inputs to the fixed-point Controller are scaled to the correct data types using convert blocks. The convert block connected to the encoder feedback needs a radix point precision of at least 8 bits while the convert block connected to the command input can be 6 bits.

When this simulation is executed, the controller is implemented in 16-bit scaled precision, while the rest of the simulation runs in double precision floating-point. This lets you simulate and validate the performance of the controller, as it would execute on the fixed-point target.

Prototyping the embedded control system

The fixed-point controller can easily be prototyped in a hardware-in-the-loop scenario or implemented on a target processor such as a DSP or a microcontroller. Furthermore, integrated solutions let you generate, compile, link, download, test, debug, and validate the entire application. This dramatically reduces development time and expenses while resulting in a high-quality product that is well tested and very dependable.

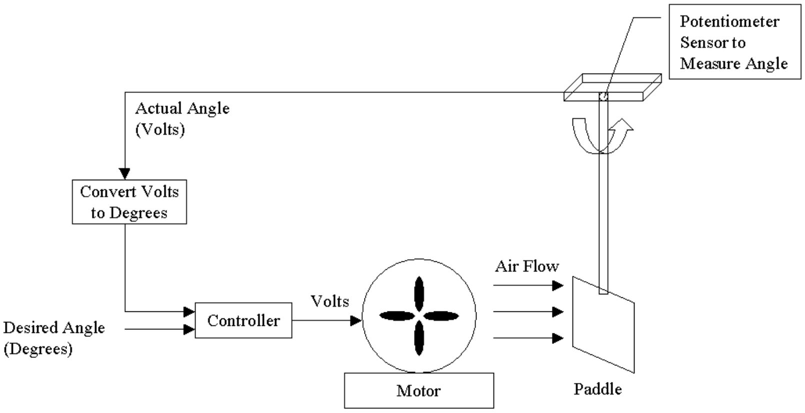
Implementing a PID position controller

This example describes how to implement a PID position controller in floating point. The diagram — **position_control_fixpoint** — is under Examples > Fixed Point.

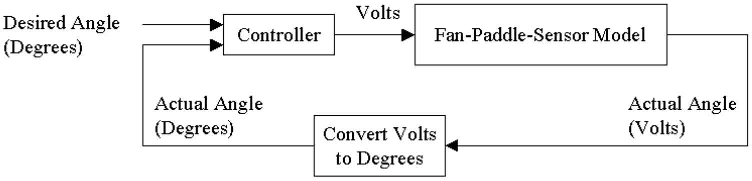
In most real-world cases, a scaled, fixed-point-based embedded controller controls a real system, such as automotive brake systems, machine tools, aerospace control surfaces, and other similar systems. In each case, the best way to prototype an embedded controller is to realize the controller in scaled fixed-point implementation that is native to the target platform. The rest of the simulation — such as sensors, plant diagram components, and actuators — are best simulated in double-precision floating-point to reflect the real-world application scenario most accurately.

The **position_control_fixpoint** diagram is an implementation of a PID controller for a position control application. The plant, controller, and other arithmetic operations are first implemented in double-precision floating point.

The system comprises an electrical motor connected to a small propeller that blows air on a paddle. The paddle is moved at an angle from the vertical. The control problem is to adjust the speed of the motor by varying its input voltage to maintain the paddle at a user-defined angle from the vertical. The system can be schematically represented as shown below:



For the prototyping process, the fan-paddle-sensor subsystem can be collapsed into a single diagram, as shown below:



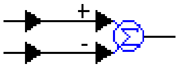
Constructing a floating-point PID position controller

The system represented above is built using standard blocks. Each of the three major components — Controller, Fan-Paddle-Sensor Model, and the Convert Volts to Degrees — are developed, linked, and simulated.

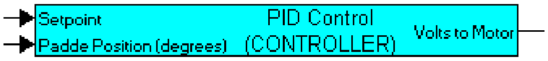
Constructing the controller

The controller has two inputs (desired and actual angles) and one output (voltage) to be applied to the motor.

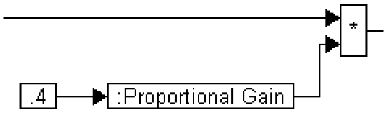
To begin modeling the controller, wire two wirePositioners to the inputs of a summingJunction block, and negate the second input.



For ease of implementation, these blocks are encapsulated in a compound block called **PID Control (CONTROLLER)** and the inputs and outputs are labeled appropriately.

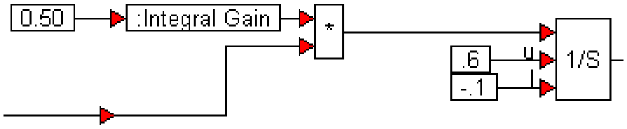


Inside **PID Control (CONTROLLER)**, the output of the summingJunction is passed through a gain of 0.001 and fed as the error input for computing *P* (proportional), *I* (integral), and *D* (derivative) components of the controller. The proportional term, encapsulated in a compound block named *proportional term* is implemented as:



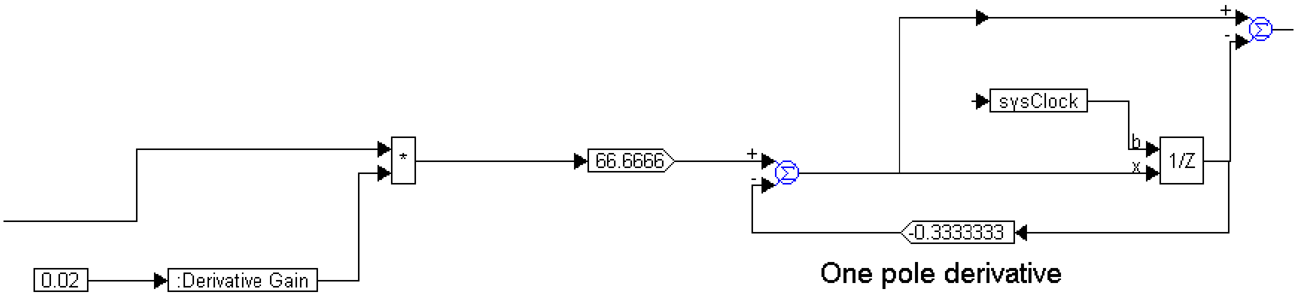
The proportional gain is set to 0.4.

The integral term, encapsulated in a compound block named **integral term** is implemented as:

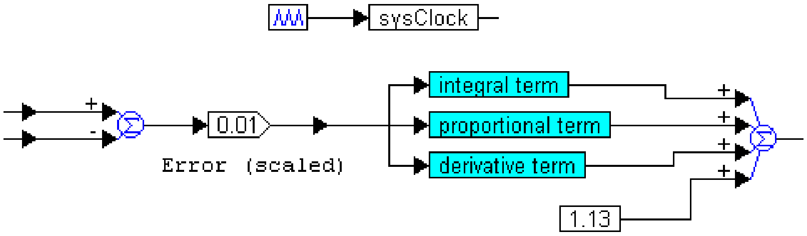


The integration is performed using a limitedIntegrator to prevent windup. The upper and lower limits are set to 0.6 and –0.1 respectively, and the integral gain is set to 0.50.

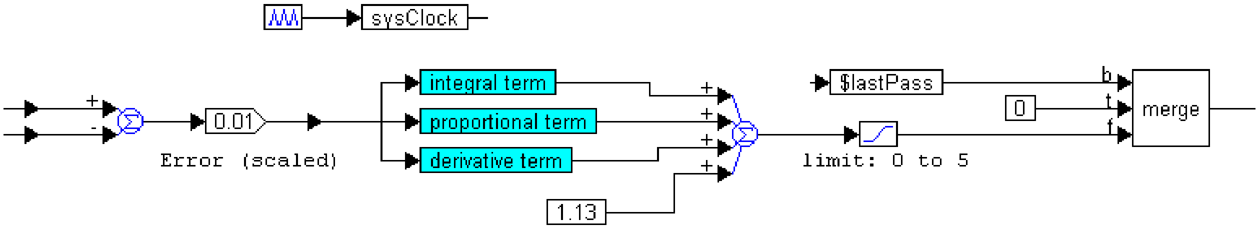
The derivative term, encapsulated as a compound block named **derivative term** is implemented as:



The computation of the derivative is implemented using a unitDelay block, two gain blocks, and two summingJunction blocks, as shown above. The *sysClock* clock input to the unitDelay is defined using a pulseTrain block, and the contributions of the *P*, *I*, and *D* terms are summed up, as shown below:



Because the motor needs a minimum of 1.13 V to turn, a constant bias of 1.13 V is added to the mix. To ensure that the voltage applied to the motor is within the rated voltage range, and to shut the motor down when the simulation run is complete, the limit and merge blocks are used, as shown below:



The output of the merge block is forced to 0 after the last step of the simulation, represented by the system variable *$lastPass*. For all other steps, the limit block restricts the output to the range 0 V to 5 V, as shown above.

Constructing the Volts to Degrees Converter

As is the case with many sensors, the potentiometer used in this application produces a voltage proportional to the actual quantity being measured: in this case the angle of the paddle from the vertical. Since the set point is in degrees, you must convert the volts corresponding to the actual angle to degrees of actual angle. The principle for modeling the conversion process is quite simple. You measure the voltage at 0° and 90° angles. Assuming a linear relationship between potentiometer volts and actual angle in degrees, the relationship can be written as:

actual angle = (*actual voltage* – *0deg voltage*) * *degrees_per_volt*

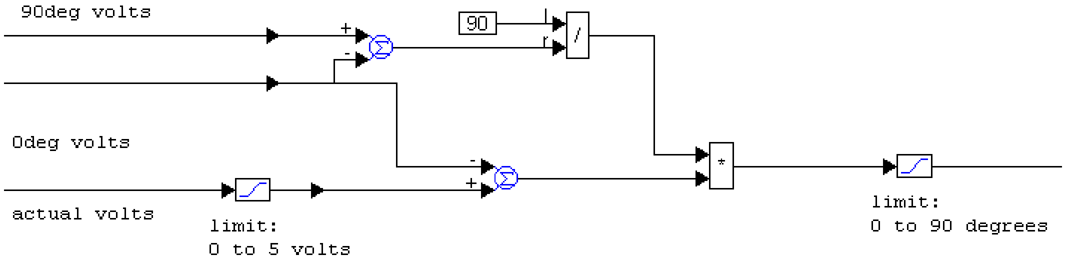
where degrees_per_volt is obtained from the two calibrating measurements as:

degrees_per_volt = (*90deg voltage* – *0deg voltage*) / 90

Combining the two relationships yields:

actual angle = (*actual voltage* – *0deg voltage*) * (*90deg voltage* – *0deg voltage*) / 90

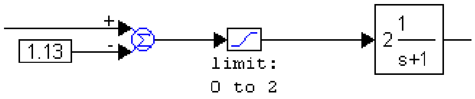
This relationship can be implemented using standard arithmetic blocks.



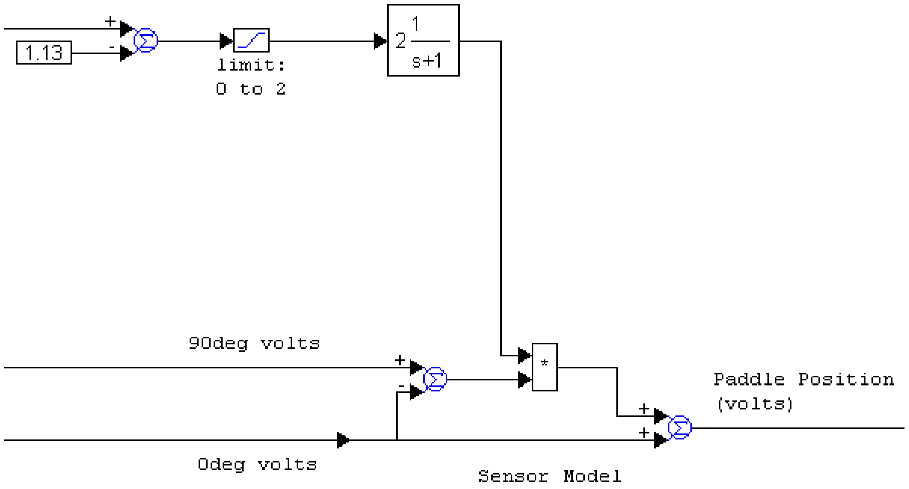
Two limit blocks are used to limit the actual volts to be within 0 V – 5 V and the output to be within 0 ºF – 90 ºF. This prevents the Volts to Degrees Converter subsystem from providing out-of-range values to the controller.

Constructing the fan-paddle-sensor

The key elements to capture in the fan-paddle diagram are the response profile and the lag between the input and output signals. In other words, when the input changes by a certain amount, how long does it take for the output to show the effects of the change in the input and how do the input and output amplitudes correlate. Based on this approach, subtract the 1.13 V that were added in the PID controller as the minimum bias voltage for the motor to run. The remainder is limited to be in the range 0 – 2. The time delay and response profiles can be modeled easily by a first order transfer function using the transferFunction block, as shown below:



Because the potentiometer converts angular motion into equivalent voltage and the calibrating voltage measurements for 0° and 90° are known, modeling the sensor is a simple arithmetic operation. The complete diagram for the fan-paddle-sensor subsystem is shown below:



This set of blocks is encapsulated in a compound block named *Fan-Paddle-Sensor Model (ACTUATOR+PLANT+SENSOR)*. Under System > System Properties > Range, set the simulation range to 0 – 100 with a step size of 0.01. A slider block with range set to 0 – 30 is used to specify the set-point angle, and a plot block is used to display the results. Two const blocks specify the 0° and 90° calibration voltages as 1.17 and 0.68, respectively.

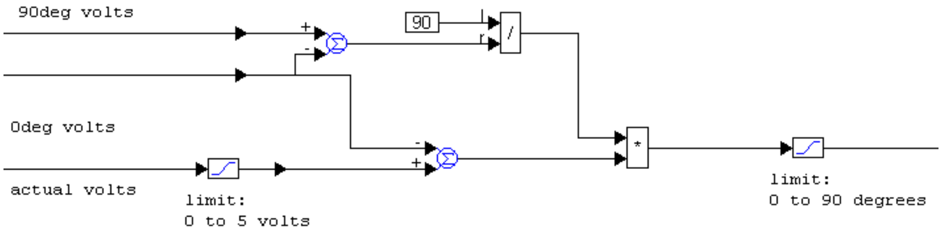
Fixed-point implementation of the PID position controller

Constructing the Fixed-Point Volts to Degrees Converter

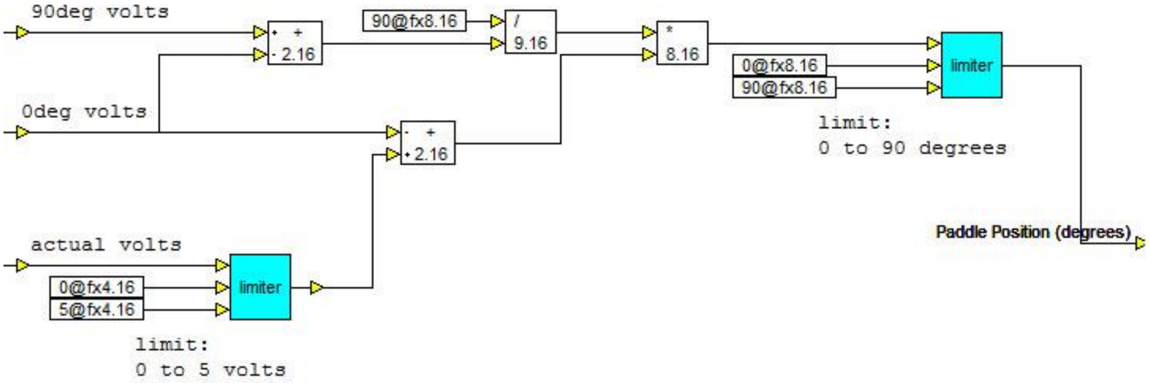
The floating-point implementation of the **Volts to Degrees Converter** was the arithmetic implementation of the equation:

actual angle = (*actual voltage* – *0deg voltage*) * (*90deg voltage* – *0deg voltage*) / 90

The actual implementation is:



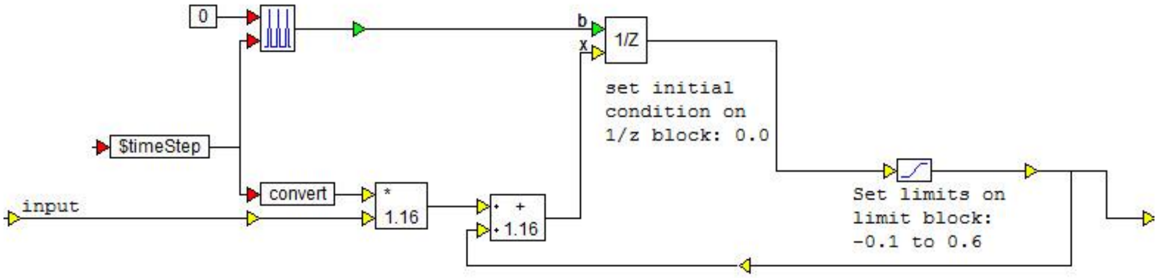
This relationship can be easily implemented using the fixed-point blocks sum, div, mul, limit, and const.



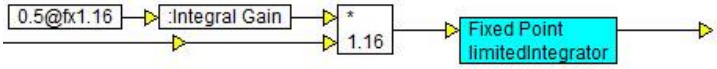
This set of blocks is encapsulated in the **Volts to Degrees (FIXED POINT)** compound block.

Constructing the controller

To implement the integral term, you use the fixed-point limitedIntegrator block, which expands to:

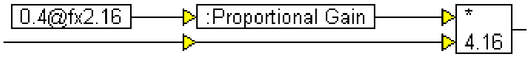


The integral term of the PID controller can be implemented as:

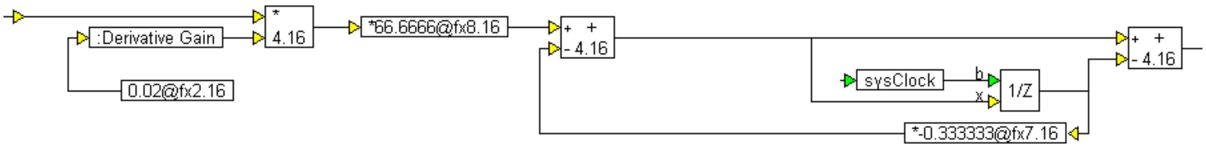


Compared to the floating-point implementation, the only differences are the fixed-point const and mul blocks used to define the integral gain and to perform multiplication, respectively. This set of blocks is encapsulated the **Integral Term** compound block.

The **Proportional Term** compound block contains the fixed-point equivalent.



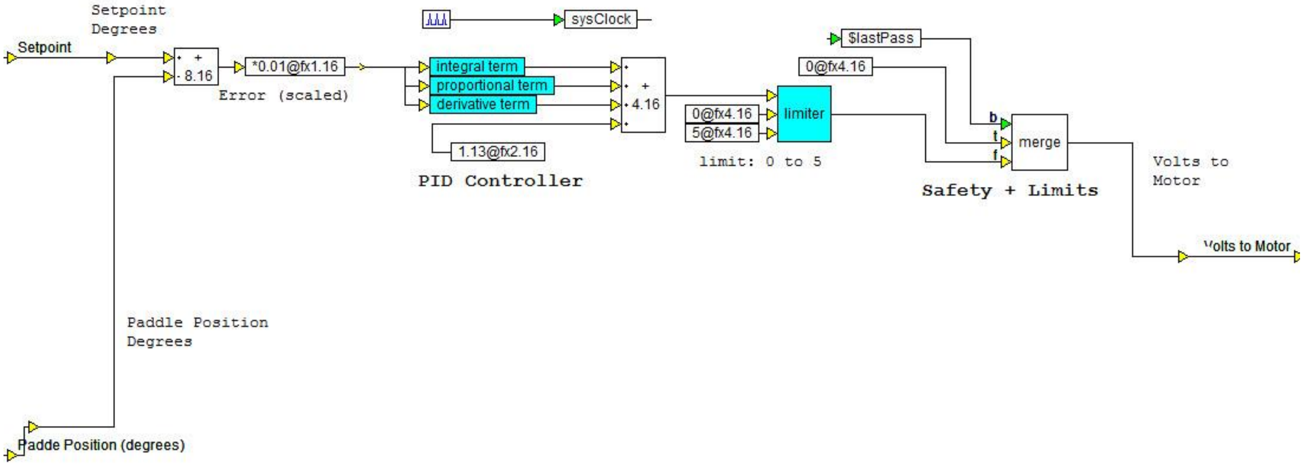
The arithmetic operations of the derivative term are replaced with fixed-point equivalents, const, mul, sum, and gain, as shown below:



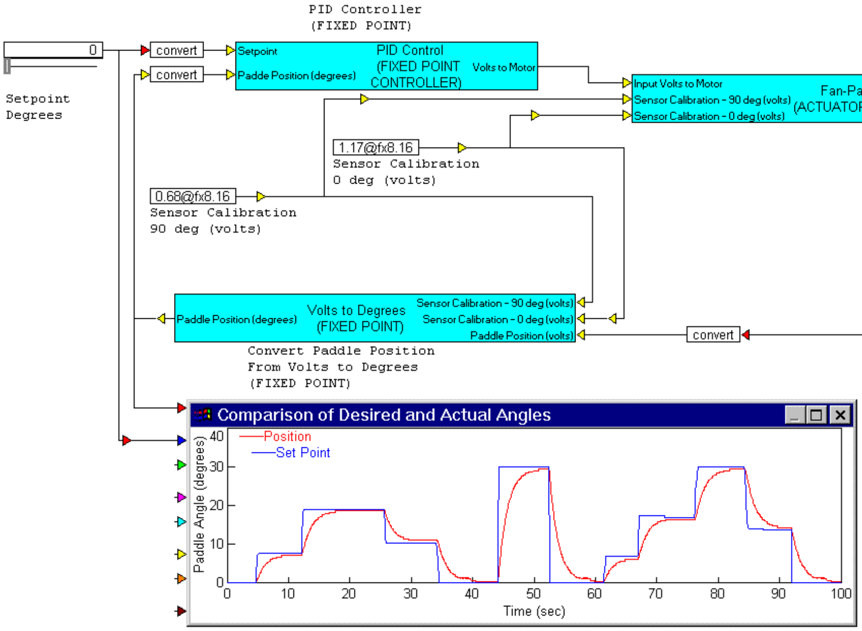
One pole derivative

These blocks are encapsulated in the **Derivative Term** compound block.

Combining the three control terms, the fixed-point PID control can be implemented as:



This set of blocks is encapsulated in the **PID Control (FIXED POINT CONTROLLER)** compound block. The complete system becomes:



Three convert blocks are used to ensure that the inputs to the **Controller** and the **Volts to Degrees** converter are the correct data type. Furthermore, the 0º and 90º calibration voltages are defined using fixed-point const blocks. The simulation parameters remain unchanged from the floating-point implementation.

It is important to note that in the simulation depicted above, the **PID Control** and **Volts to Degrees Converter** are simulated by Embed in scaled fixed-point while the **Fan-Paddle-Sensor** is simulated in floating-point. This means that you can simulate how a given control or logic prototype would execute on a fixed-point embedded system target, such as a DSP or a microcontroller. This lets you answer in a single design and simulation iteration, crucial questions, such as:

- Is it feasible?
- Will it work?
- Will it work on the embedded target that I have chosen or have in mind?
- Am I getting the most dynamic range for each of my variables?
- Can I guarantee that none of the variables will suffer numerical overflow for the entire range of inputs and outputs for which I am designing?
- Does my control system exceed or at least meet the design specifications?

The next step is to implement the fixed-point controllers and control logic on target hardware.

AC induction motor: speed control of a machine tool lathe

This example describes how to implement speed control of a machine tool lathe. The diagram — **Machine Tool** — is under Examples > eDrives > eMotors (Legacy) > AC Induction.

The typical machine tool lathe is operated from a single-speed motor drive, together with multiple gear selection to vary chuck speed. Here a simpler design is considered: one with a single 10:1 gear reducer and a variable speed control drive for a three-phase AC induction motor.

The lathe is required to operate with the following specifications:

- **Maximum work piece load:** 1 meter by 0.1 meter diameter aluminum bar stock
- **Chuck speed control range:** 30 – 400 RPM
- **Speed control accuracy:** ± 5 RPM from set point steady state
- **Maximum load torque**: not to exceed 0.3 N-m, introduced by the cutting tool

The motor specifications are given as:

Motor parameter	Value	Units
Stator resistance (per phase)	9.4	Ohms
Stator self-inductance (per phase)	0.402	Henries
Stator leakage inductance	0.032	Henries
Rotor resistance	7.1	Ohms
Rotor leakage inductance	0.032	Henries
Number of poles	2	
Rotor inertia	0.001	Kg-m2
Rotor viscous damping constant	0.0001	Kg-m2 - s

The moment of inertia of the chuck and moving drive assembly is given as 0.1 kg-m^2. The moment of inertia of the work piece is calculated as:

$$I = \frac{1}{2}Mr^2 = \left(\frac{1}{2}\right)(21.14kg)\left(\frac{0.1m}{2}\right)^2 = 0.026kg\,m^2$$

Since the axes of the chuck and work piece are coincident, they add to total 0.126 kg m^2.

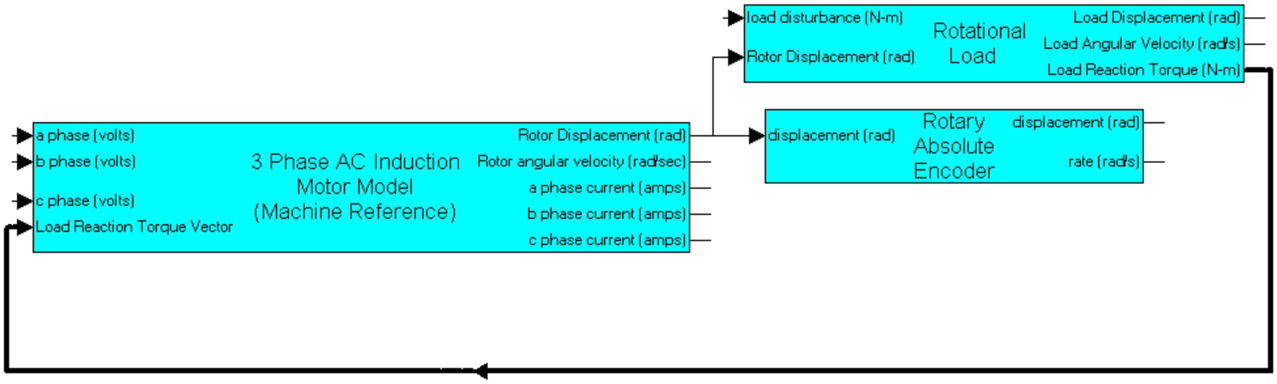
One effective way of controlling speed by an induction motor is to control the stator field frequency. Since stator flux is inversely proportional to frequency below the base frequency, it is necessary to adjust voltage proportional to frequency to maintain constant flux. For frequency above the base frequency (power supply limitation), the voltage is kept constant. This method is the basis of the design, with one minor improvement. The constant volts to frequency control mentioned above are used as a feed forward leg of a feed forward – proportional integral controller (PI). The PI component of the control is used to adjust any error that may occur due to motor slip and loading from the cutting tool. Motor speed is estimated from motor shaft position measured by an incremental encoder. To drive the motor, an inverter is used with six-step logic to switch polyphase-rectified voltage producing a balanced three-phase signal.

Setting up the motor, load, and encoder

The first step is to place the following eMotors blocks in your diagram:

- Rotational Load
- AC Induction Motor-Machine Reference
- Rotary Absolution Encoder

Wire the blocks together and use wirePostioner blocks to clearly represent the feedback of the load reaction torque to the motor diagram.



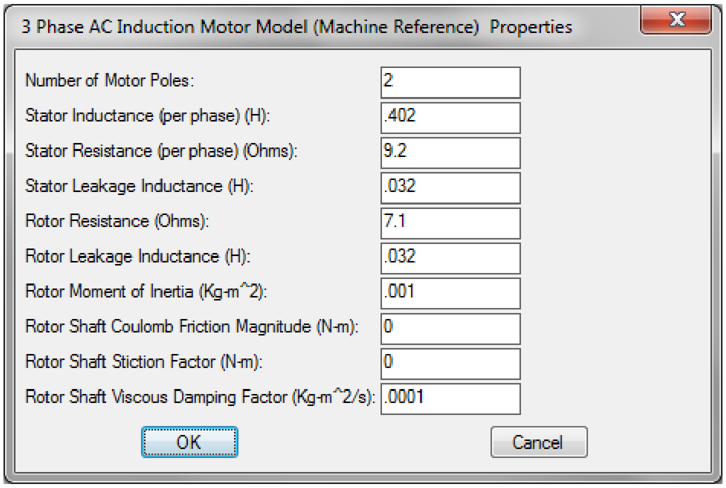
The rotational load diagram is used to simulate the lathe chuck and work piece. The rotary encoder diagram input is connected to the motor's rotor shaft displacement output connector. The motor displacement output is also connected to the rotational load diagram. To complete the dynamic interaction between the motor and load, the load reaction torque output connector must be connected to the load reaction torque vector input of the motor diagram.

Note: This wire is thicker than the other wired connections, indicating that it transmits a vector quantity. The vector contains load dynamic parameters that are reflected back to the motor dynamics through the coupling (linkage) mechanism. In this case, a 10:1 gear reduction.

Setting parameter values

The next step is to enter the parameters for the motor, load, and encoder. The parameter values can be changed later to see what effect they may have on the final control solution.

1. Set the **AC Induction Motor** block parameters as shown below. These parameter values are taken from the motor specifications table.

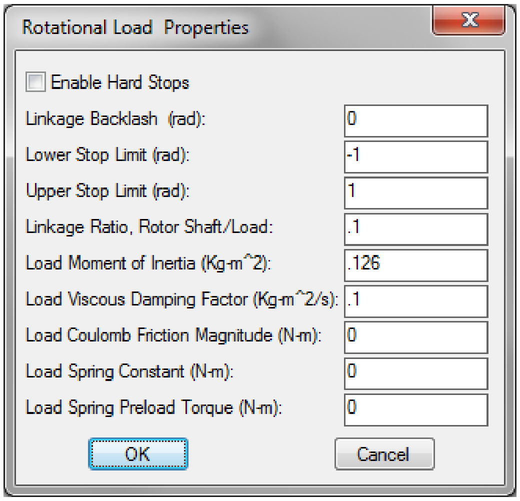


3 Phase AC Induction Motor Model (Machine Reference) Properties

Number of Motor Poles:	2
Stator Inductance (per phase) (H):	.402
Stator Resistance (per phase) (Ohms):	9.2
Stator Leakage Inductance (H):	.032
Rotor Resistance (Ohms):	7.1
Rotor Leakage Inductance (H):	.032
Rotor Moment of Inertia (Kg-m^2):	.001
Rotor Shaft Coulomb Friction Magnitude (N-m):	0
Rotor Shaft Stiction Factor (N-m):	0
Rotor Shaft Viscous Damping Factor (Kg-m^2/s):	.0001

2. Set the **Rotational Load** block parameters as shown below and note the following:

- The **Load Viscous Damping Factor** value is a rough guess.
- For the linkage ratio (gear ratio for this application), follow this rule: a factor less than 1.0 multiplies torque, and a factor greater than 1.0 multiplies speed; entering 1.0 produces a direct connection between motor and load.
- Default values are shown for the **Upper Stop Limit** and **Lower Stop Limit**, but since **Enable Hard Stops** is not activated, hard stop limits are not used in the diagram. Hard stops are useful in position control system applications.

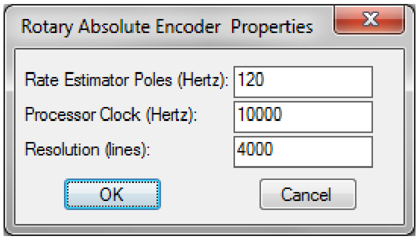


Rotational Load Properties

□ Enable Hard Stops

Linkage Backlash (rad):	0
Lower Stop Limit (rad):	-1
Upper Stop Limit (rad):	1
Linkage Ratio, Rotor Shaft/Load:	.1
Load Moment of Inertia (Kg-m^2):	.126
Load Viscous Damping Factor (Kg-m^2/s):	.1
Load Coulomb Friction Magnitude (N-m):	0
Load Spring Constant (N-m):	0
Load Spring Preload Torque (N-m):	0

3. Set the **Rotary Encoder** block parameters shown below:



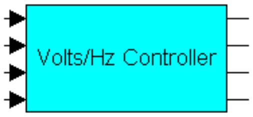
Rotary Absolute Encoder Properties

Rate Estimator Poles (Hertz):	120
Processor Clock (Hertz):	10000
Resolution (lines):	4000

Designing the volts/frequency controller for the motor

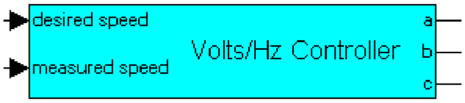
In this step, use a PID Controller-Digitalblock and a Square Wave Inverter-3 Phaseblock to design the volts/frequency controller for the motor.

After placing the blocks in your diagram, encapsulate them in a compound block using Edit > Create Compound Block. Name the compound block **Volts/Hz Controller**.



This block design requires only two inputs and three outputs. By default, when you create a compound block, Embed creates outputs for all the blocks contained in the compound, which may not be appropriate. In this case, you must remove two inputs and one output using Edit > Remove Connector

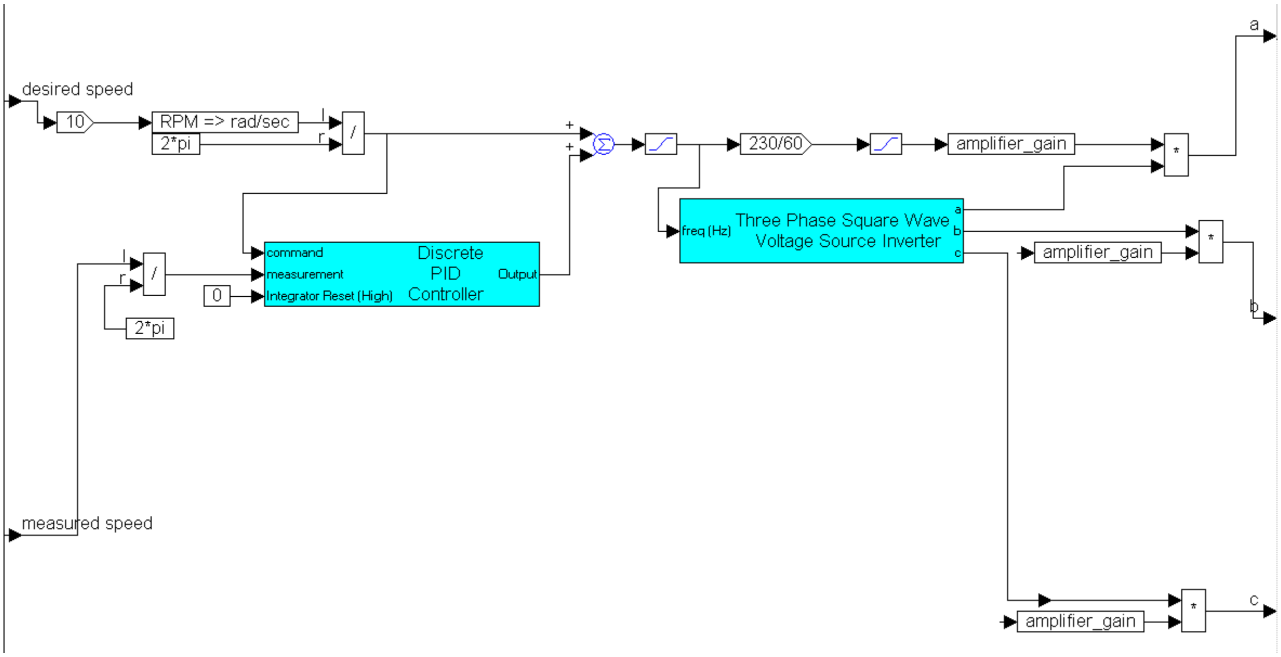
Label the input and output connectors.



Drill into Volts/Hz Controller and remove all unneeded wire connections within the compound by clicking, holding, and dragging the wires with the left mouse button to an open space and then releasing the button.

Customizing the Volts/Hz Controller block

Make the following modifications to **Volts/Hz Controller**:



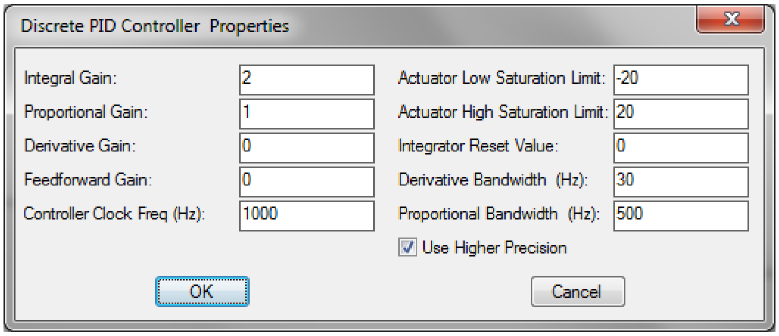
The input speed for this block is assumed to be the speed of the chuck; therefore, a gainblock is used to scale this speed up by a gear ratio of 10 since this controller affects the speed on the motor side. RPM is then converted to hertz by using a unitConversion block set to RPM⇒rad/sec and then dividing the output by 2π. The value 2π is produced by using a constblock set to 2*pi.

The measured speed comes from the Rotary Absolute Encoder and is in radians per second. This measurement is converted to hertz simply by dividing by 2*pi. The desired speed in hertz is fed into a summingJunctionblock, as well as the command input of the PID Controller-Digitalblock. The desired speed directly feeds the inverter/amplifier as the feed forward component of the control. PID Controller-Digitalblock output is used to correct for minor errors in the feed forward component. The sum of these two components is fed to the inverter/amplifier, the sum is limited to 70 Hz to prevent running the motor into its unstable region of control. The output of the limit block feeds the Square Wave Inverter-3 Phaseblock. The Square Wave Inverter-3 Phaseblock rail voltages must be set to 0 and 1 to provide logic control rather than bus level voltages:

The output of the control summingJunctionblock is scaled inversely proportional to frequency by using a gainblock with the factor 230/60. The output is then limited between 0 and 230 V, and defined as a variable with the user-defined name amplifier_gain.

Configuring the PID compensator

To configure the PID compensator, enter the following values into the PID Controller-Digitalblock:



Integral Gain:	2	Actuator Low Saturation Limit:	-20
Proportional Gain:	1	Actuator High Saturation Limit:	20
Derivative Gain:	0	Integrator Reset Value:	0
Feedforward Gain:	0	Derivative Bandwidth (Hz):	30
Controller Clock Freq (Hz):	1000	Proportional Bandwidth (Hz):	500

☑ Use Higher Precision

Since the feed forward and derivative gain are set to 0, the block is actually configured to operate as a PI controller. Saturation is set to limit the influence of the integral correction to ✆20 Hz. Proportional bandwidth is set at Nyquist frequency (½ the sampling frequency); derivative bandwidth does not matter in this controller. **Use Higher Precision** is activated to allow trapezoidal integration to be used.

Integral reset is not used on this controller, so a constblock with a value of 0 is fed into PID Controller-Digitalto prevent integral reset. The actual values for the proportional and integral gain were determined experimentally in the final configuration to obtain minor overshoot and settling in the control.

This completes the construction of the **Volts/Hz Controller** compound block.

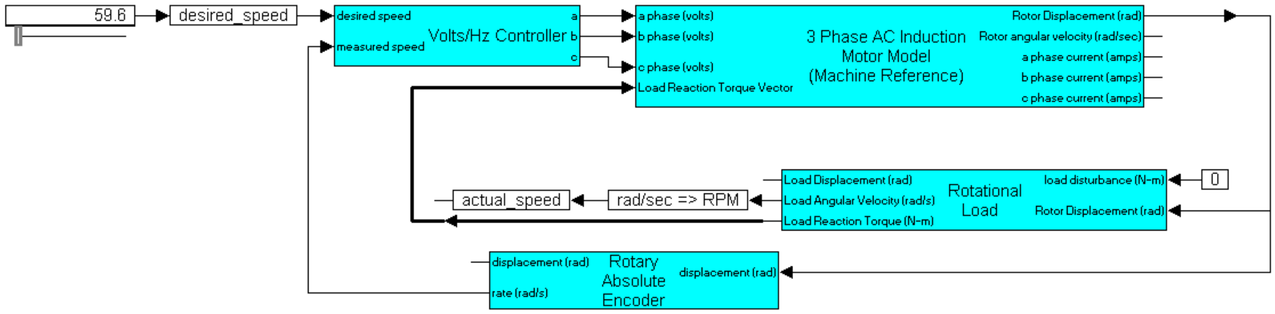
Wiring Volts/Hz Controller to the overall simulation

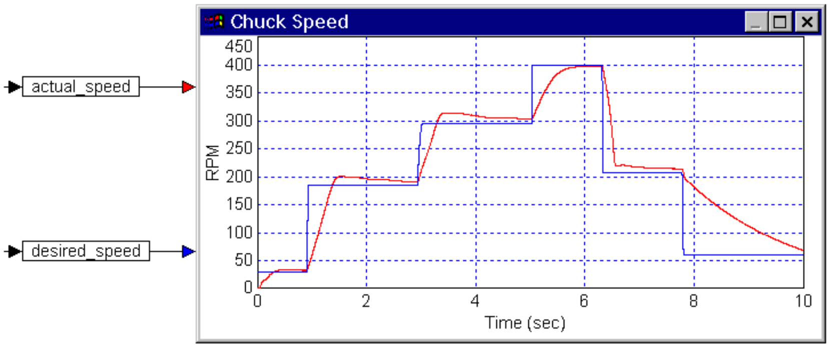
The three outputs for **Volts/Hz Controller** are connected to the corresponding inputs of the induction motor block. Measured speed from the Rotary Absolute Encoderblock is connected to the measured speed input of the **Volts/Hz Controller** block. A sliderblock, scaled between 30 and 400, is connected to the desired speed input of the **Volts/Hz Controller** block as RPM speed input. A plotblock is wired to compare the desired and actual speeds. The actual speed is determined by converting load angular velocity to RPM. A constblock set to 0 is connected to the load disturbance input of the rotational load diagram and variableblocks are used to make the diagram legible.

Before simulating the diagram, set the simulation range parameters

- Start Time = 0
- Step Size = 0.0001
- End Time = 10

Through minor exploration, the motor drive is found to have sufficient torque at all speeds to overcome maximum tool exertion.





Now with a working simulation, you have met the design requirements and can begin optimizing performance. For example, a fairly high-resolution encoder was used for estimating rate. How coarse can the resolution become before performance is degraded? Also, the motor may be oversized for the particular application. Surveys show that over 50% of the motors selected in the US are oversized for their application. Simulation provides a lower-cost alternative to performing extensive analysis or purchasing a variety of motors to empirically determine which is best suited for an application. This is true for any motion control application; not just limited to machine tools.

Brushless DC (BLDC/PMSM) motor: target tracking system

This diagram simulates a servo-controlled positioning system that maintains focal plane line of sight coincident with target angle. The permanent magnet synchronous motor diagram is selected as an actuator to provide fast response. The diagram — **Target Tracking** — is under Examples > eDrives > eMotors (Legacy) > BLDC.

Motor specifications

Automatically acquiring and maintaining the line of sight of a video camera or focal plane sensor is often required in various aerospace, defense, and security system applications. One way to mechanize such a system is to reflect the field of view through two independently-controlled mirrors that each rotate in axes orthogonal to one another. The object of the control system is to acquire the target, and by controlling rotation of each mirror, move the line of sight coincident with the target angle. This places the virtual image of the target in the center of the focal plane. Once the image of the target is acquired on the focal plane, an error in azimuth and elevation can be determined by a variety of image processing techniques, such as contrasting, differencing, and area parameter calculations.

For this simulation, such a mechanism is assumed, with a pipeline image processor providing direct angular azimuth and elevation measurements. The following design decisions are also assumed:

Motor type: Permanent magnet DC synchronous motor with Hall sensors for commutation sensing and control.

Motor parameter	Value	Units
Operating voltage	28	Volts
Magnetizing inductance	0.0009	Henries
Stator inductance (per phase)	0.001	Henries
Stator resistance (per phase)	0.5	Ohms
Torque constant	0.1035	Nm/A
Number of poles	2	
Rotor moment of inertia	8.5 E-06	kg-m2
Rotor shaft viscous damping factor	5.695 E-06	kg m2/s

For the simulation, a PWM Brushless Servo Amplifierblock is used with a base frequency of the PWM at 9000 Hz, along with a Hall Sensor block for commutation.

Precision current sense resistors produce voltage that is fed into a processor. An encoder provides motor shaft position and velocity. Encoder angle measurement and phase current measurements are used to obtain direct and quadrature current estimates through Clarke and Park transforms. Current and speed loops are used to set stiff inner loop performance.

Mechanical Load: Precision $\lambda/4$ flat oval mirrors mounted on a gear reducer shaft with rotation center coincident with reflecting surface represent the main load moment of inertia. A torsional spring with preload tension is used to help minimize backlash hysteresis. An optical encoder is provided with 16000 lines to measure mirror angle. PI compensation is used for controlling line of sight. Load parameters are:

Gear reduction	20:1
Backlash	0.0005 radians
Load moment of inertia	0.001 kg – m2
Load viscous damping	0.01 kg – m2/s
Load spring constant	0.01 N-m/rad
Load spring preload	0.1 N-m

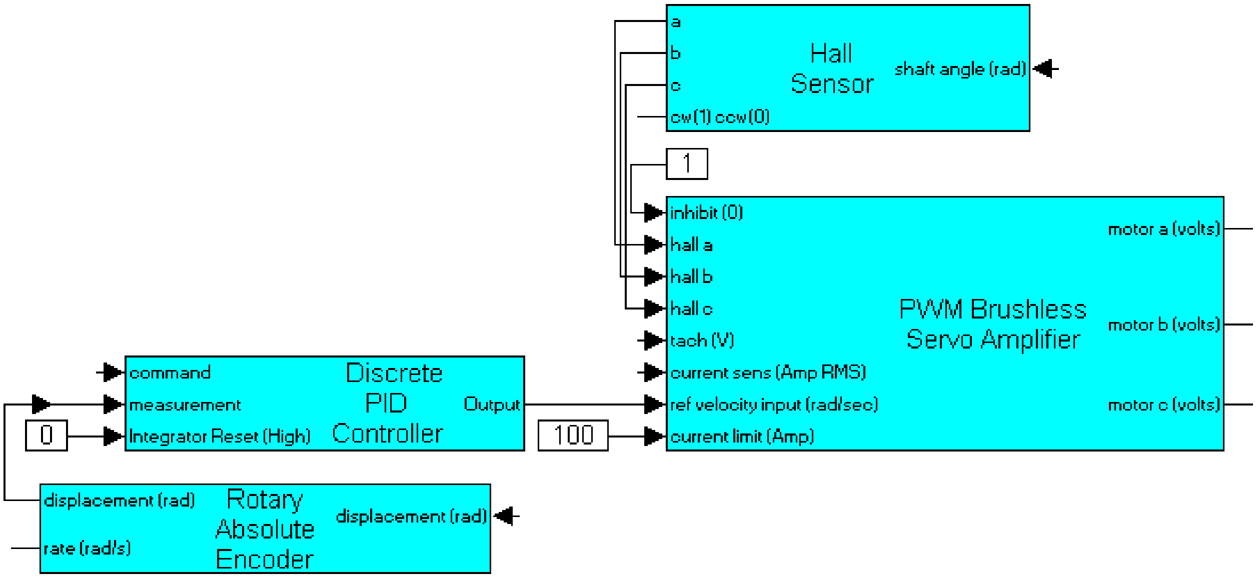
Pipeline Image Processor: Provides 60 Hz frame rate acquisition of target from focal plane array. Pixel resolution is sufficiently higher than expected control requirement of less than ± 3 degrees between target angle and line of sight in both axes. Hierarchical classification and size discrimination of blobs with subsequent calculation of the target centroid determine target position.

Simulation development

Place the following eMotors blocks in your diagram:

- Digital PID Controller
- Hall Sensor
- PWM Brushless Servo Amplifier
- Rotary Absolute Encoder

Flip the **Rotary Absolute Encoder** and **Hall Sensor** blocks using Edit > Flip Horizontal. Then arrange the blocks and wire them together, as shown below:



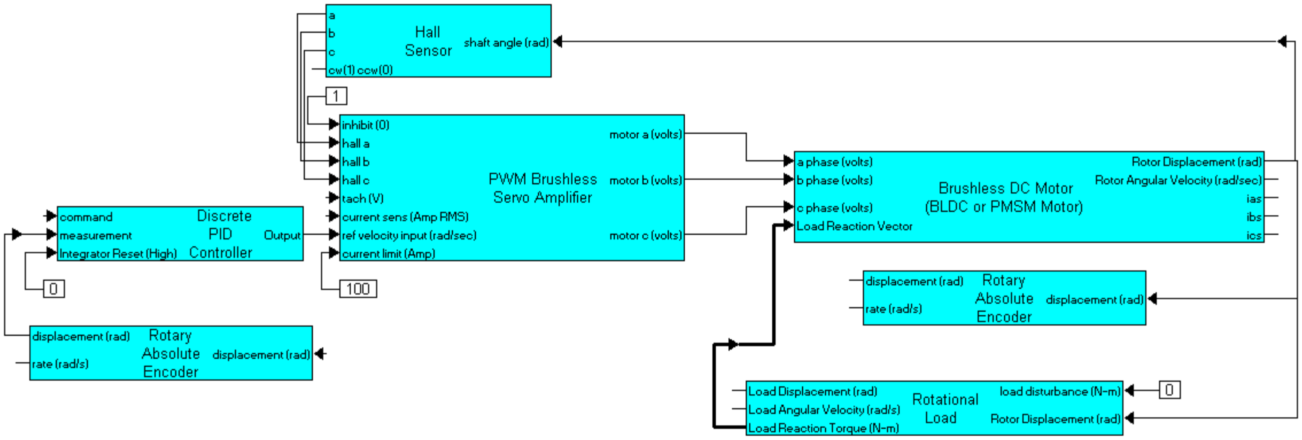
In this application, there is no reason to reset the integration of the **PID Controller-Digital**, so a 0 **const** is wired to **Integrator Reset (High)** to disable it. In other applications, repetitive control may be used, and **Integrator Reset (High)** may be required to re-initialize the control between repetitions.

A value of **100 A** is chosen for this example to make certain saturation does not occur. Later on, you might measure currents encountered in this simulation under highest load conditions and set a more appropriate current limit for the final design.

Next, place the following eMotor blocks in the diagram:

- Brushless DC Motor-Digital
- Rotary Absolute Encoder
- Rotational Load

Flip the **Rotary Encoder** and **Rotational Load** blocks and arrange the blocks as shown below:

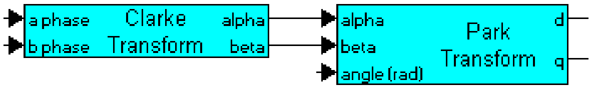


Connect the Rotor Displacement output on the **Brushless DC Motor-Digital** to these three blocks: the shaft angle input on the **Hall Sensor** block, the displacement input on the **Rotary Absolute Encoder**, and the Rotor Displacement input on the **Rotational Load**.

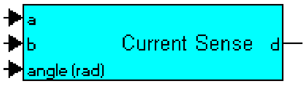
Connect the outputs on the **PWM Brushless Servo Amplifier** to the corresponding inputs on the **Brushless DC Motor-Digital**. Connect the Load Reaction Torque output on the **Rotational Load** to the Load Reaction Vector input on the **Brushless DC Motor-Digital**.

Lastly connect a **const** block with 0 set value to the load disturbance input on the **Rotational Load**. If there were other torques related to influences that could not be directly represented by the set parameters of the rotary load diagram, the load disturbance input provides a method for introducing such torques. For the target tracker, it might be conceivable to introduce torque noise induced by structural vibrations of the tracker mount. If the mount were part of a satellite payload, such vibrations could arise from solar array positioning systems. Noise profiles with specific power spectral densities can be generated in Embed using the Random Generator blocks and transferFunction block. Coefficients of the transfer function are determined by applying spectral factorization techniques to the known PSD.

Next, insert a **Park Transform** and a **Clark Transform** into the diagram and connect them as shown below:



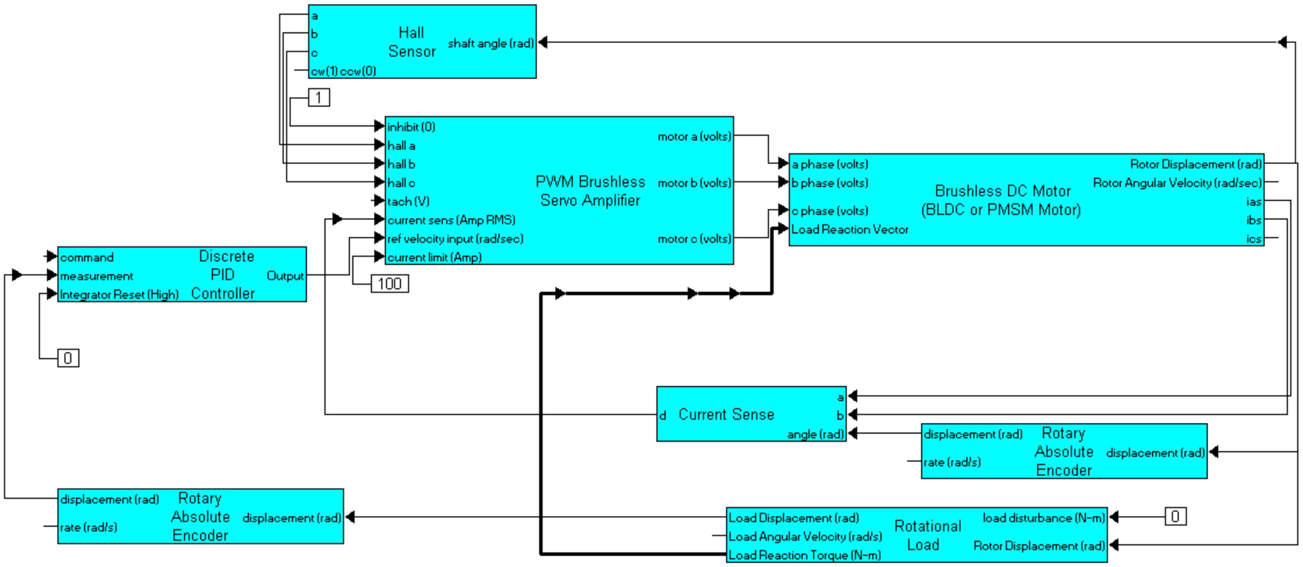
Encapsulate the blocks in **Current Sense**. Then label the input and output connectors as shown below:



Flip the block and connect the ias and ibs outputs on the **Brushless DC (BLDC/PMSM) Motor** to the corresponding a and b inputs on the **Current Sense**. Connect the displacement output of the **Rotary Encoder** to the angle input of the **Current Sense**.

Connect the Load Displacement output on the **Rotational Load** to the displacement input of the other **Rotary Absolute Encoder**.

Complete the wiring by connecting the output on **Current Sense** to the current sense input on the **PWM Brushless Servo Amplifier** and the rate output on the **Rotary Absolute Encoder** to the tach input on the **PWM Brushless Servo Amplifier**, as shown below:



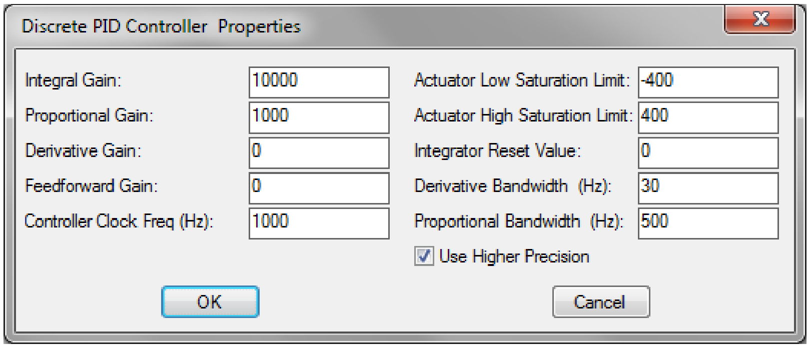
This diagram represents a cascade control loop. The inner loop senses and controls current; the middle loop senses and controls velocity; and the outermost loop senses and controls position.

Now the entire diagram must be collapsed into a single compound block named **X Axis Servo**. Reduce the number of inputs and outputs on **X Axis Servo** to one, and label the input connector commanded LOS and output connector actual LOS.

Then drill into **X Axis Servo** and make certain that the commanded LOS is connected to the command input on the **PID compensator** block and the displacement output of the **Rotational Load** is connected to the actual LOS output of the compound block.

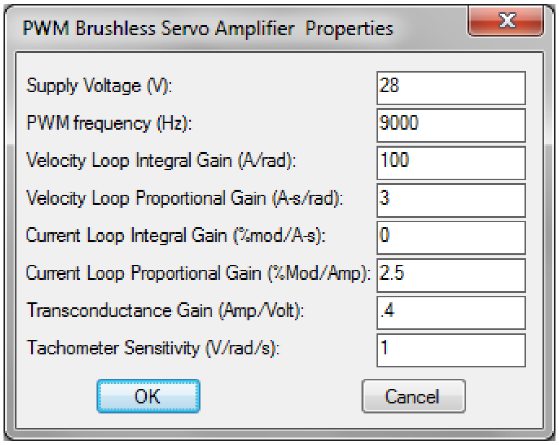
While still in the **X Axis Servo**, open the dialogs of each block and enter the following parameter values as specified by the design input.

PID Controller (Digital) block



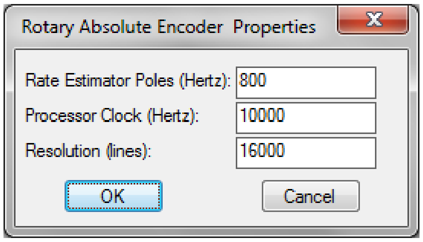
Integral Gain:	10000	Actuator Low Saturation Limit:	-400
Proportional Gain:	1000	Actuator High Saturation Limit:	400
Derivative Gain:	0	Integrator Reset Value:	0
Feedforward Gain:	0	Derivative Bandwidth (Hz):	30
Controller Clock Freq (Hz):	1000	Proportional Bandwidth (Hz):	500
		☑ Use Higher Precision	

PWM Brushless Servo Amplifier block



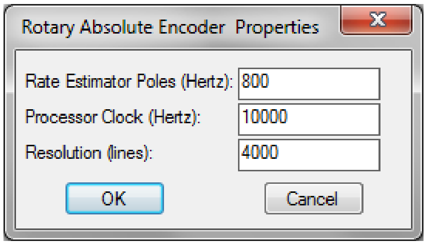
Supply Voltage (V):	28
PWM frequency (Hz):	9000
Velocity Loop Integral Gain (A/rad):	100
Velocity Loop Proportional Gain (A-s/rad):	3
Current Loop Integral Gain (%mod/A-s):	0
Current Loop Proportional Gain (%Mod/Amp):	2.5
Transconductance Gain (Amp/Volt):	.4
Tachometer Sensitivity (V/rad/s):	1

Rotary Encoder block that feeds back to PID Controller (Digital) block



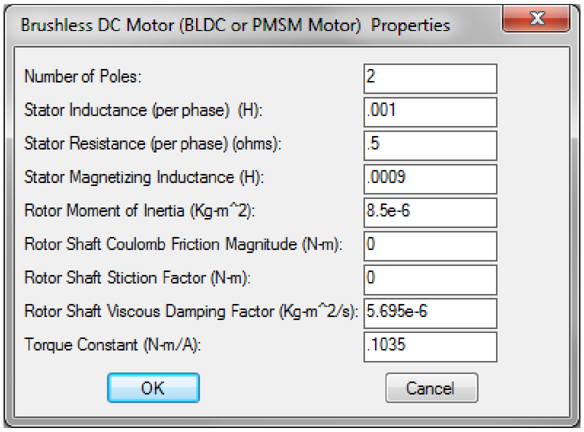
Rate Estimator Poles (Hertz):	800
Processor Clock (Hertz):	10000
Resolution (lines):	16000

Rotary Encoder block that feeds back into the PWM Brushless Servo Amplifier block



Rate Estimator Poles (Hertz):	800
Processor Clock (Hertz):	10000
Resolution (lines):	4000

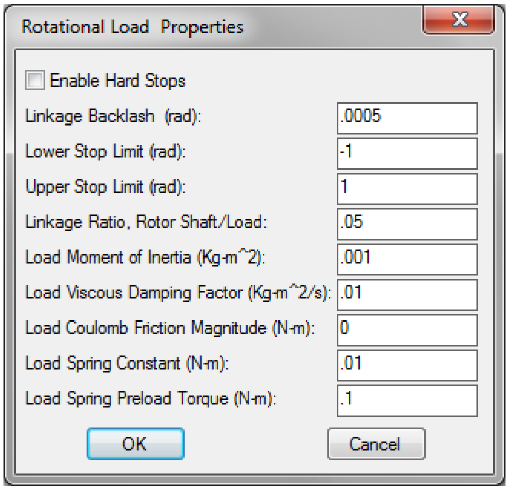
Brushless DC (BLDC/PMSM) Motor block



Brushless DC Motor (BLDC or PMSM Motor) Properties

Number of Poles:	2
Stator Inductance (per phase) (H):	.001
Stator Resistance (per phase) (ohms):	.5
Stator Magnetizing Inductance (H):	.0009
Rotor Moment of Inertia (Kg-m^2):	8.5e-6
Rotor Shaft Coulomb Friction Magnitude (N-m):	0
Rotor Shaft Stiction Factor (N-m):	0
Rotor Shaft Viscous Damping Factor (Kg-m^2/s):	5.695e-6
Torque Constant (N-m/A):	.1035

Rotational Load block



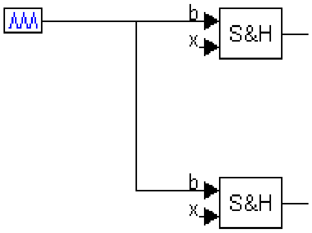
Rotational Load Properties

☐ Enable Hard Stops

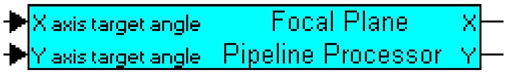
Linkage Backlash (rad):	.0005
Lower Stop Limit (rad):	-1
Upper Stop Limit (rad):	1
Linkage Ratio, Rotor Shaft/Load:	.05
Load Moment of Inertia (Kg-m^2):	.001
Load Viscous Damping Factor (Kg-m^2/s):	.01
Load Coulomb Friction Magnitude (N-m):	0
Load Spring Constant (N-m):	.01
Load Spring Preload Torque (N-m):	.1

This completes the *x*-axis of the servo controller. Completing the *y*-axis takes only a couple of keystrokes, as all dynamics for this axis mirror the *x*-axis. Make a copy of X Axis Servo using Edit > CopyIn the dialog for the newly-created X Axis Servo, change the block name to Y Axis Servo. At this point, there are two servo controllers in your diagram: an *x*-axis and a *y*-axis servo controller.

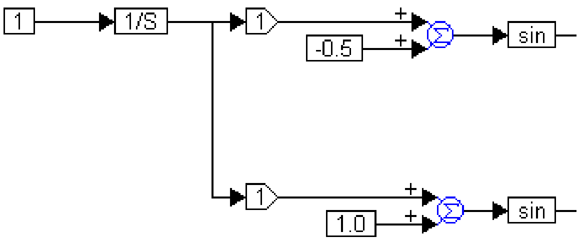
Next, create the pipeline image processor. For this processor, the dominant feature is the sample frame rate of 60 Hz. Place two sampleHoldblocks and a pulseTrainblock in your diagram, as shown below:



In the pulseTrain block, set the time between pulses to 1/60 (0.0167). Then encapsulate the three blocks in a compound block and name it Focal Plane Pipeline Processor.

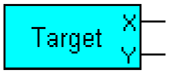


Create the following block configuration:

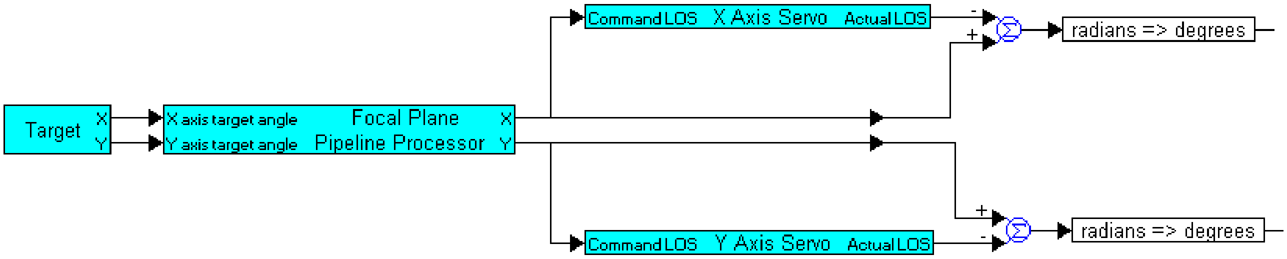


This creates an elliptical motion for the target in the X-Y plane. Frequency for each axis is the same (1 rad/sec); however, phase differs.

Collapse the blocks into a compound block and name it **Target.**



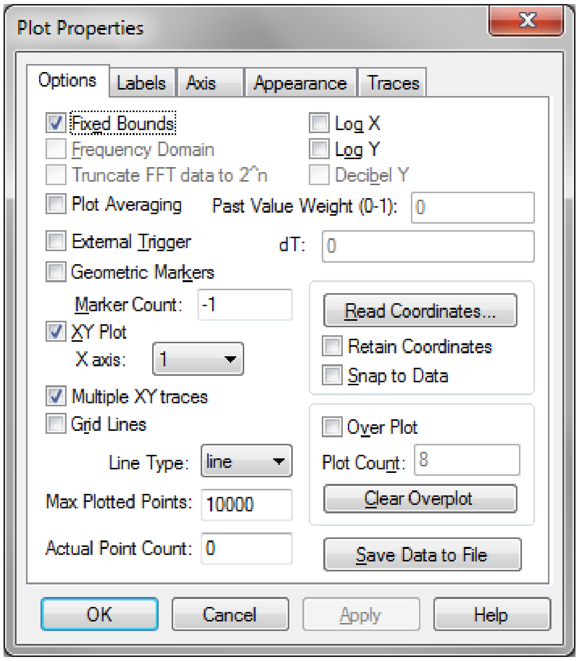
Connect the compound blocks as shown below:



The command line of sight (LOS) is set to the target angle that is determined by the pipeline processor. The difference between the target angle and actual line of sight is calculated using summingJunctionblocks that provide focal plane error. The error is converted into degrees using unitConversionblocks.

Setting up the plot blocks

Place a plotblock in the diagram and make the following selections in its dialog:



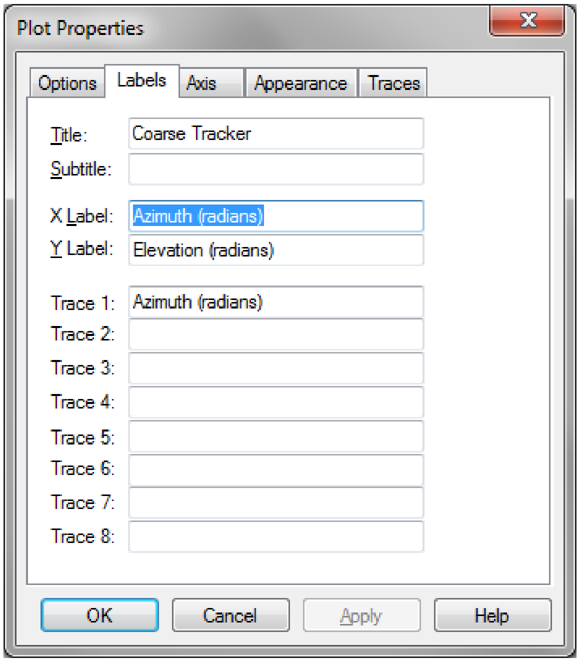
Plot Properties

Options | Labels | Axis | Appearance | Traces

☑ Fixed Bounds
☐ Frequency Domain
☐ Truncate FFT data to 2^n
☐ Log X
☐ Log Y
☐ Decibel Y
☐ Plot Averaging   Past Value Weight (0-1): 0
☐ External Trigger   dT: 0
☐ Geometric Markers
Marker Count: -1
☑ XY Plot
X axis: 1
☑ Multiple XY traces
☐ Grid Lines
Line Type: line
Max Plotted Points: 10000
Actual Point Count: 0
Read Coordinates...
☐ Retain Coordinates
☐ Snap to Data
☐ Over Plot
Plot Count: 8
Clear Overplot
Save Data to File

OK | Cancel | Apply | Help

The **Multiple XY Traces** parameter allows the display of the target motion independently from the servo line of sight.



Title: Coarse Tracker

Subtitle:

X Label: Azimuth (radians)

Y Label: Elevation (radians)

Trace 1: Azimuth (radians)

Trace 2:

Trace 3:

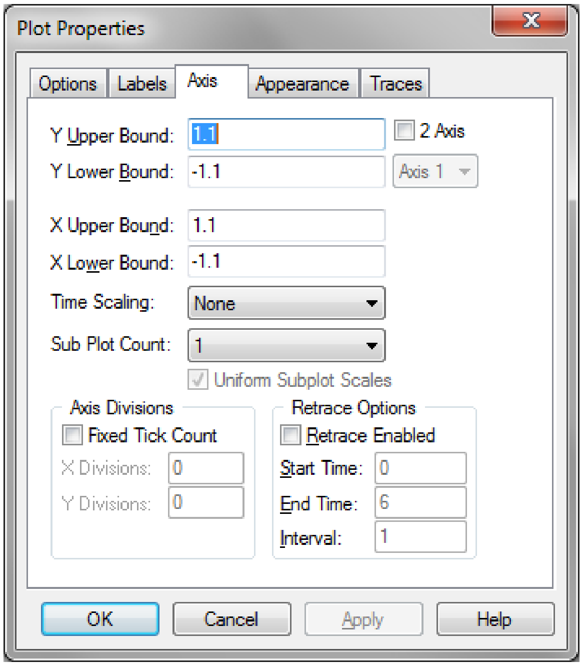
Trace 4:

Trace 5:

Trace 6:

Trace 7:

Trace 8:



Y Upper Bound: 1.1 ☐ 2 Axis

Y Lower Bound: -1.1 Axis 1

X Upper Bound: 1.1

X Lower Bound: -1.1

Time Scaling: None

Sub Plot Count: 1

☑ Uniform Subplot Scales

Axis Divisions

☐ Fixed Tick Count

X Divisions: 0

Y Divisions: 0

Retrace Options

☐ Retrace Enabled

Start Time: 0

End Time: 6

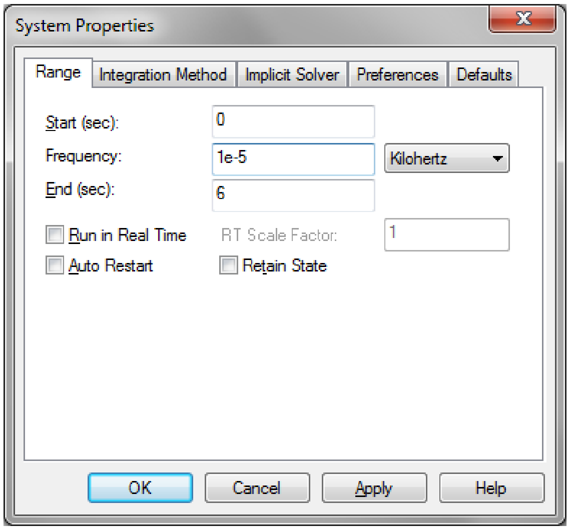
Interval: 1

Make a copy of the plot block and make these changes in the dialog:

1. Under **Labels**:
   - In the Title box, enter **Focal Plane**
   - Enter degrees as units instead of radians
2. Under **Options**:
   - Activate **Fixed Bounds**
   - De-activate **Multiple XY Traces**
3. Under **Axis**:
   - In **X Upper Bound** and **Y Upper Bound**, enter **5**
   - In **X Lower Bound** and **Y Lower Bound**, enter **–5**

Setting the simulation properties

Enter the following information in the System Properties dialog. For this simulation, a very small step size is necessary because pulse width modulation is being simulated at 9000 Hz.



System Properties

Range | Integration Method | Implicit Solver | Preferences | Defaults

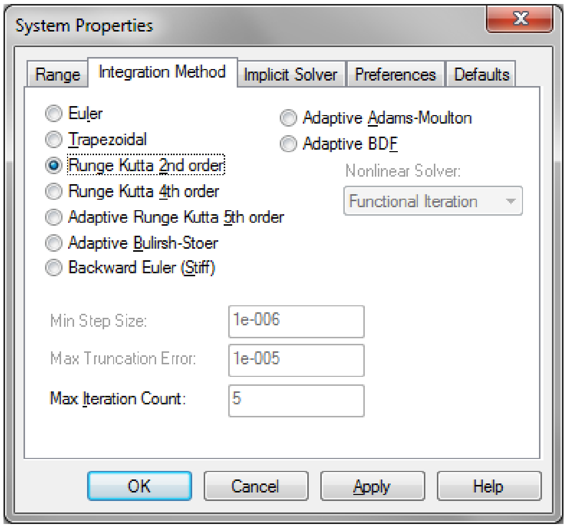
Start (sec): 0

Frequency: 1e-5 Kilohertz

End (sec): 6

☐ Run in Real Time RT Scale Factor: 1

☐ Auto Restart ☐ Retain State



System Properties

Range | Integration Method | Implicit Solver | Preferences | Defaults

○ Euler

○ Trapezoidal

● Runge Kutta 2nd order

○ Runge Kutta 4th order

○ Adaptive Runge Kutta 5th order

○ Adaptive Bulirsh-Stoer

○ Backward Euler (Stiff)

○ Adaptive Adams-Moulton

○ Adaptive BDF

Nonlinear Solver: Functional Iteration

Min Step Size: 1e-006

Max Truncation Error: 1e-005

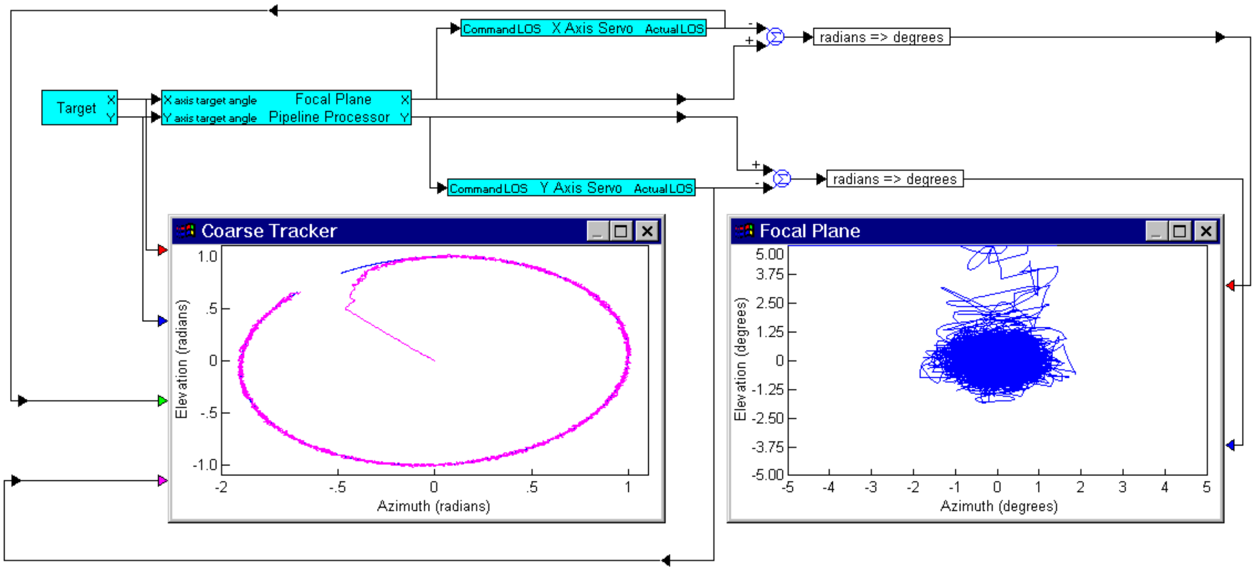
Max Iteration Count: 5

Final configuration requirements

Connect the X and Y outputs on **Target** to the first two inputs on the **Coarse Tracker plot** and the outputs on **X Axis Servo** and **Y Axis Servo** actual line of sights to the next two inputs on the same plot block.

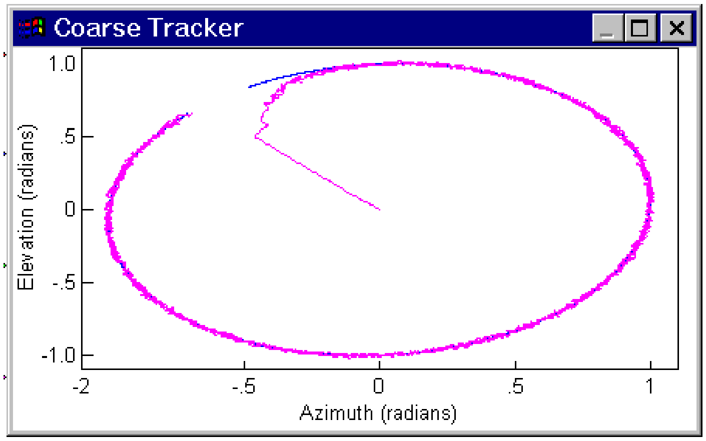
Connect the outputs on the two **unitConversion** blocks to the first two inputs on the **Focal Plane plot**.

Start the simulation.

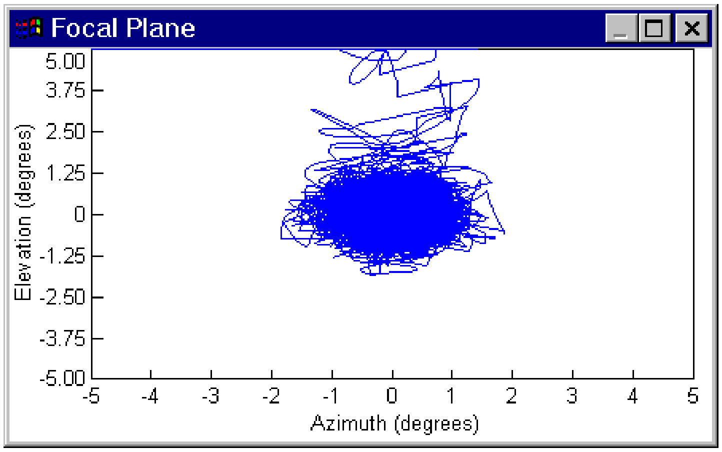


Simulation results

The Coarse Tracker plot shows the acquisition and tracking of the actual target's elliptical motion with the servo line of sight:



To better illustrate accuracy, the Focal Plane plot shows the focal plane error. The darkened circular area represents the time after the servo has acquired the target and begins tracking. These results show errors to be on the order of 1°, exceeding the requirement.



It should be noted that to get to this level of control required tuning of each of the control loops with multiple iterations before an acceptable control was achieved.

Exchanging data with Compose

Embed provides an Altair Compose OML interface that lets you add commands to Compose that:

- Invoke Embed
- Simulate VSM diagrams
- Change and save parameters within the VSM diagrams
- Simulate the VSM diagrams with the parameter changes

To communicate between Embed and Compose applications, a library named emRemote.dll is provided.

What you'll need

- Register Embed as a server.

  1. Launch a Command Prompt as administrator.
  2. Navigate to the Altair Embed directory and enter the following command:

     ```
     Vissim64 /Regserver
     ```

     Note: Other options are `/RegServer` or `/Register`. All options are case sensitive.

- Add the functions from the emRemote.dll to the Compose libraries list using the addLibrary function. In general, start the script with `addLibrary` and define the path to emRemote.dll as a parameter. For example:

  ```
  addTLibrary(‘C:\Altair\Embed2025\emRemote.dll');
  ```

To unregister Embed as a server, enter `Vissim64 /Unregserver`. Other options are `/UnregServer` or `/Unregister`. All are case sensitive.

Compose script structure

A typical Compose script used to set up and run an Embed diagram requires the following structure:

```
addLibrary('D:\Src\VissimDll\emRemote\x64\Debug\emRemote.dll');
emInitialize();
emLoadModel('D:\Garbage\RemoteTest\Diagram01.vsm');
%% Add model settings here
%%
%% Either run the model or compile it
emRunModel();
%% Get values from the model here
emSaveModel();
emDestroy();
```

Function	Purpose
addLibrary	Registers the functions exposed by the emRemote.dll with Compose.
emInitialize	Initializes the communication interface.
emLoadModel	Loads a specific Embed diagram.
%%	At this time, the diagram is ready to run or can be compiled. If you need to tune your diagram — for example, set simulation parameters, assign initial variable values, or change block parameters and properties — replace %% with the emSetSimParam, emSetValue, or emModifyBlock functions.
emRunModel	Runs the diagram.
emExecOperation	Compiles the diagram.
%%	When simulation ends, some values can be imported for postprocessing. Replace %% with the emGetValue function.
Script element	**Purpose**
emSaveModel	Saves the changed diagram.
emDestroy	Releases all objects and closes Embed.

You can also add the steps of setting up, running simulation, and postprocessing in a loop in the script.

Functions

addLibrary

Adds functions from emRemote.dll to the Compose libraries.

```
addLibrary(‘path-to-dll\emRemote.dll');
```

Example

```
addLibrary(‘C:\Altair\Embed2025\emRemote.dll');
```

emDestroy

Releases all objects and interface, and closes Embed. This function has no arguments.

```
emDestroy();
```

emExecOperation

Execute operations other than running diagram.

```
emExecOperation(quoted string);
```

The parameter string must have the following structure:

```
compile[=<compound name>]
```

At this time there is only one operation: compile.

Examples

```
emExecOperation(‘compile’);
emExecOperation(‘compile=Compound’);
```

emGetValue

Gets the value of the given variable from the Embed work space. The returned value can be assigned to a Compose variable.

```
emGetValue(quoted string);
```

The quoted string must be a valid variable name from the diagram.

Example

```
a=emGetValue(‘a’);
```

emInitialize

Initializes the interface to Embed. This function has no arguments.

```
emInitialize();
```

emLoadModel

Loads the specified Embed diagram into the Embed work space.

```
emLoadModel(quoted string);
```

The quoted string must be a valid path to the Embed diagram.

```
emLoadModel('D:\Garbage\RemoteTest\Diagram03.vsm');
```

emModifyBlock

Modifies block parameters and/or properties for a block with given ID.

```
emModifyBlock(quoted string: id, quoted string: parameters);
emModifyBlock(quoted string: id, quoted string: parameters, quoted string: properties);
```

Quoted string	Description
id	Block ID assigned to the block in Embed using **Edit > Assign Block id** command.
parameters	String of block parameters; if properties presented, it can be empty quoted string.
properties	String of block properties in VSM format

Examples

See Configuring Embed blocks.

emRunModel

Runs the diagram loaded in Embed. This function has no arguments.

```
emRunModel();
```

emSaveModel

Saves the previously loaded Embed diagram. Without the argument, this function saves the previously loaded diagram under the same file name. If there is an argument, it must contain a valid path and file name. The previously loaded Embed diagram will be saved under the given file name.

```
emSaveModel();
emSaveModel(quoted string);
```

Example

```
emSaveModel('D:\Garbage\RemoteTest\Diagram03_2.vsm');
```

emSetSimParam

Sets simulation parameters similar to command line parameters. The syntax for parameters to be set up must match Embed's command line syntax. See Embed help for details.

```
emSetSimParam(quoted string);
```

emSetValue

Assigns the given value to the given Embed variable.

```
emSetValue(quoted string: var, value);
```

Quoted string	Description
var	Valid Embed variable
value	Any Compose variable or constant with a type that can be evaluated as double, matrix, or string.

Example

```
a = 5;
```

```
emSetValue(‘a’, a);
```

Making an Embed block accessible from a Compose script

Before you can use the emModifyBlock function to access an Embed block from Compose, you must first assign a unique identifier to the block. To do so, follow these steps:

1. Start **Embed** and load the diagram to be launched later from **Compose**.
2. Navigate to the block.
3. Click **Edit > Assign Block id**.
4. Click the **block**.
5. In the dialog, enter a **unique identifier name**; then click **OK**.

Configuring Embed blocks

You use the emModifyBlock function to configure Embed block parameters and properties. There are two formats for the emModifyBlock function To determine which syntax to use, open the VSM file in a text editor and see how the block is listed:

- For blocks whose parameters are listed in a single line enclosed in parenthesis, use Method #1
- For complex blocks whose properties are listed over multiple lines, use Method #2

Method #1

In a VSM file opened in a text editor, the parameters for some blocks — such as the const and sinusoid block — are in a semi-colon-separated list enclosed in parenthesis. For example, the sinusoid block in VSM format looks like this:

```
N.4="sinusoid"(0,1,1)*18x24
```

```
i="sine"
```

In the first line, the numbers in parenthesis represent the time delay, frequency, and amplitude parameters for the sinusoid block. The second line is the unique identifier (set with the **Edit > Assign Block id** command in the corresponding diagram) for the sinusoid block. You use the emModifyBlock function in the following format to change a parameter value:

```
emModifyBlock(‘id’,’parameter-value;parameter-value;…’);
```

Thus, to change the amplitude to 2 for the sinusoid block shown above, enter the following emModifyBlock function:

```
emModifyBlock(‘sine’, ‘0;1;2’);
```

Method #2

In a VSM file opened in a text editor, the properties for more complex blocks have a multi-line format. For example, the transferFunction block in VSM format looks like this:

```
N.2="transferFunction"*52x17
n=""
Xi="0 "
Xg=1
```

```
Xn="1 "
Xd="1 1 "
XF=0,0,0,0,0,0,0,0,0,0,0,0,0,0
i=”tf”
```

Here, the initial value (Xi), gain (Xg), numerator (Xn), and denominator (Xd) are 0, 1, 1, and 1 1, respectively. The last line is the unique identifier (set with the **Edit > Assign Block id** command in the corresponding diagram) for the transferFunction block.

You use the emModifyBlock function in the following format to change a parameter value:

```
emModifyBlock(‘id’, ‘’, ‘properties in .vsm format’);
```

Thus, to change the denominator to 1 2 for the transferFunction shown above, use the third argument of emModifyBlock and keep the second argument empty.

```
emModifyBlock(‘tf’, ‘’, ‘Xi="0 "\nXg=1\nXn="1 "\nXd=\"1 2 "\ncEOF’);
```

Creating animation with White_Dune

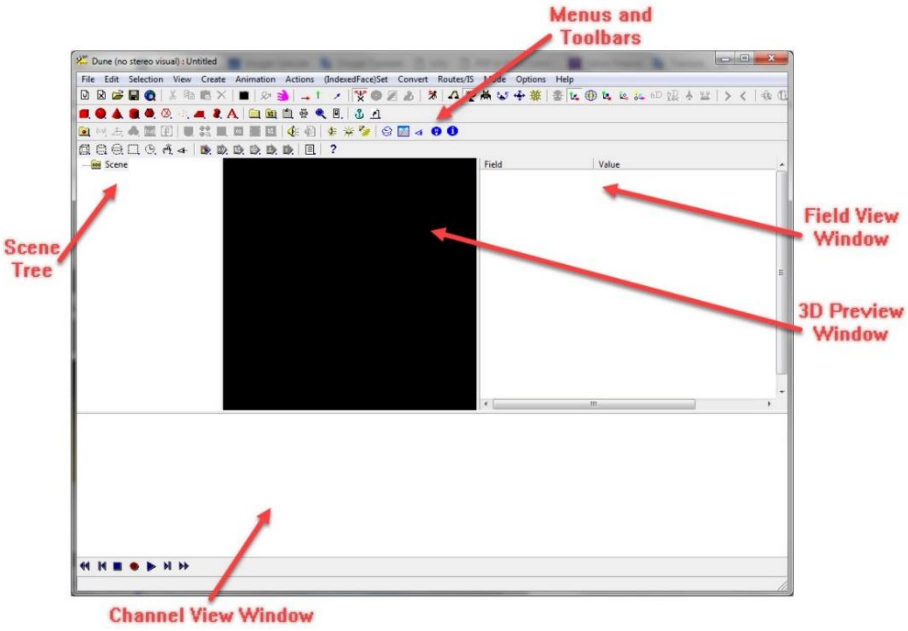
Embed 3D animation blocks let you connect to virtual reality models and manipulate the elements within the diagrams in real time and three-dimensional space. Virtual reality models can be created using a number of different 3D editing tools.

Here you will learn how to create a simple virtual reality rocket model using the White_Dune graphical editor, connect it to an Embed diagram, and add signals to the model to control and visualize the movement and appearance of the rocket elements as a simulation is running.

What you’ll need

Product	Where to get it
Embed Pro, Embed SE or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
White_Dune	White_Dune is a free, open source software package that lets you create and edit VRML97 files that can be read into Embed for simulation. To download White_Dune to your computer, go to http://wdune.ourproject.org/ and select either the Windows 10 or Windows 7 64-bit White_Dune executable.

When you start White_Dune for the first time, the following window appears:


Menus and Toolbars
Dune (no stereo visual) : Untitled
File Edit Selection View Create Animation Actions (IndexedFaceSet) Convert Routes/IS Mode Options Help
Scene
Field
Value
Field View Window
Scene Tree
3D Preview Window
Channel View Window

- **Menus and toolbars:** Contain commands and buttons for handling files; inserting and editing graphical objects, animation, and actions; changing window views; and working with nodes.
- **Field View window:** Contains the field values (numbers or character strings) of the currently selected node.
- **3D Preview window:** Shows the graphical output of the VRML file.
- **Channel View window:** Used for interpolator nodes. In this guide, the Channel View window is not used.
- **Scene Tree:** Shows the hierarchical structure of the VRML file.

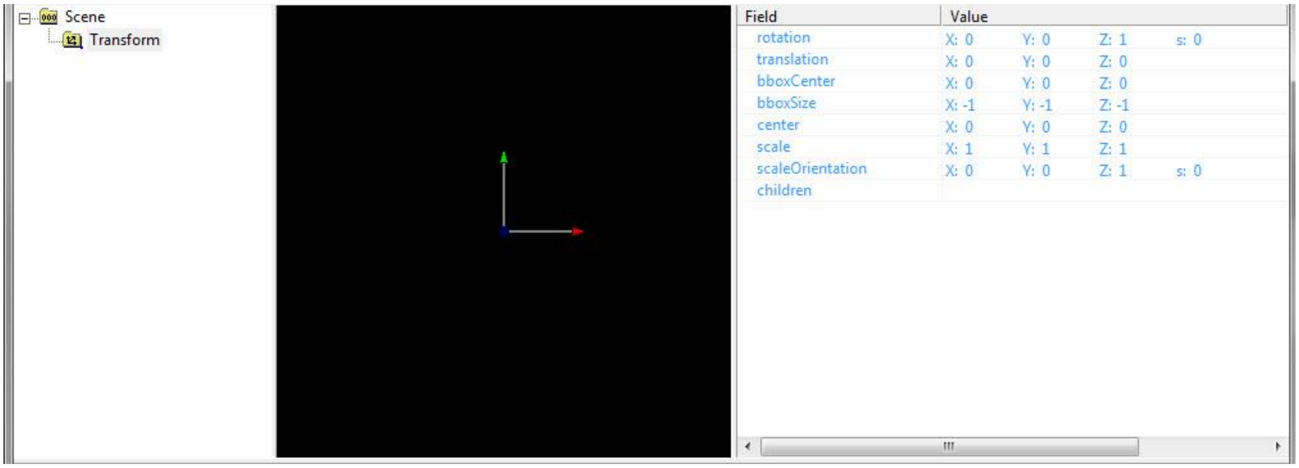
For detailed information on using White_Dune, see the online White_Dune documentation.

Building a virtual reality rocket model

In White_Dune, a virtual reality model is a hierarchy of nodes that define the elements of the model and the model structure.

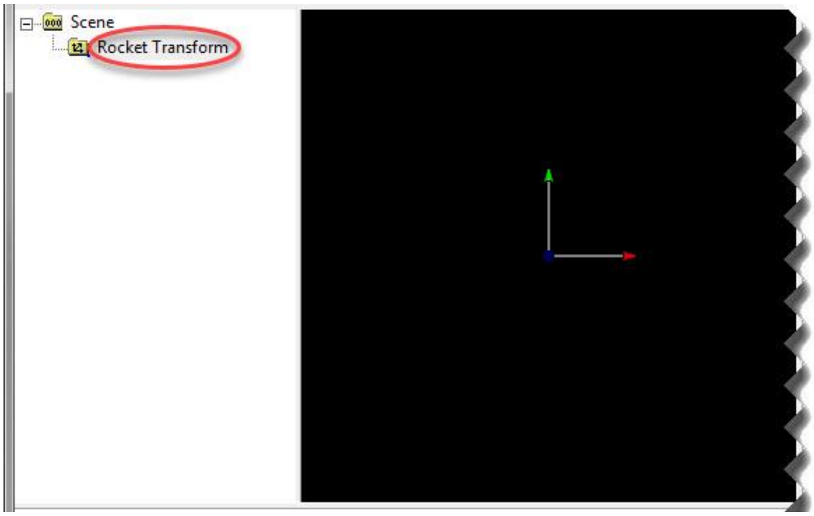
In this example, you will create a virtual reality rocket model that consists of three shape nodes: a cylinder base and two cones (a nose cone and an exhaust nozzle). You will learn how to color, move, and rotate the rocket, as well as how to resize the rocket in three-dimensional space. Later on, you will learn how to apply more advanced customizations to the rocket, include adding exhaust flames and background color.

1. Start **White_Dune**.
2. To create a single three-dimensional coordinate system that will control the rocket, select **Create > Grouping Node > Transform**.



Field	Value			
rotation	X: 0	Y: 0	Z: 1	s: 0
translation	X: 0	Y: 0	Z: 0	
bboxCenter	X: 0	Y: 0	Z: 0	
bboxSize	X: -1	Y: -1	Z: -1	
center	X: 0	Y: 0	Z: 0	
scale	X: 1	Y: 1	Z: 1	
scaleOrientation	X: 0	Y: 0	Z: 1	s: 0
children				

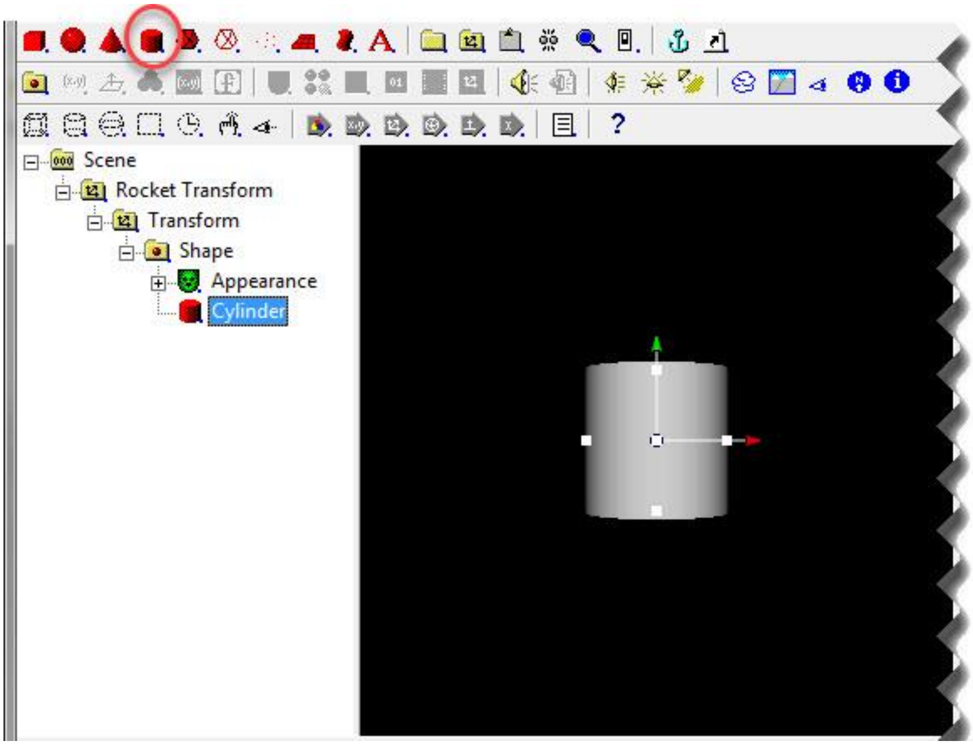
3. By default, when you create a node, it is unnamed. You must assign a unique name to each node that you want Embed to control. To do so, in the **Scene Tree**, select **Transform**, then click **Edit > DEF** and enter **Rocket.**



4. To create the cylinder base:

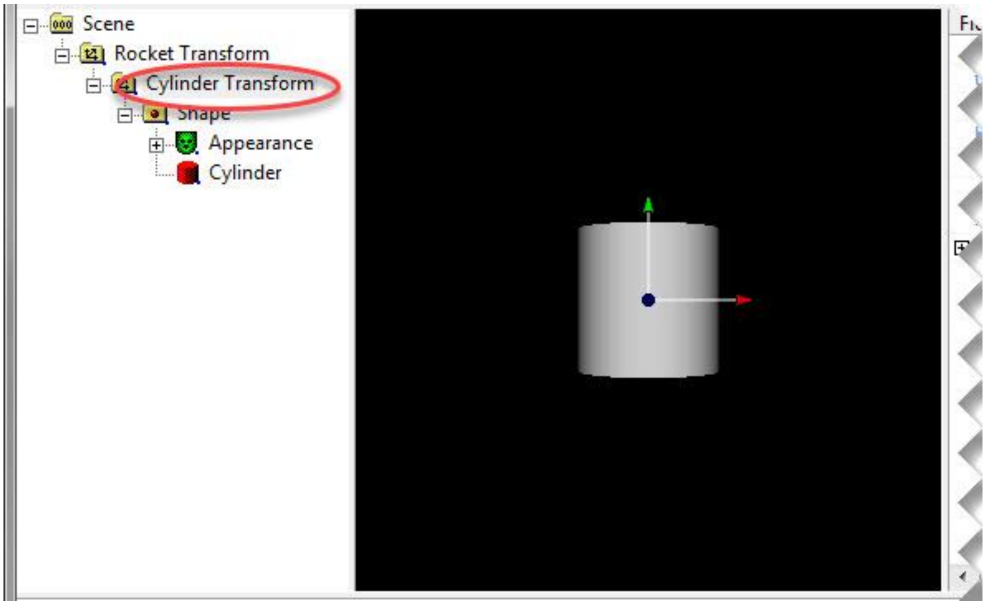
   a. In the **Scene Tree**, select **Rocket Transform**.

   b. In the **Toolbar**, click the **red cylinder**.



A cylindrical transform appears under the Rocket Transform and is displayed in the 3D Preview window.

   c. In the **Scene Tree**, select **Transform**, then click **Edit > DEF** and enter **Cylinder**.

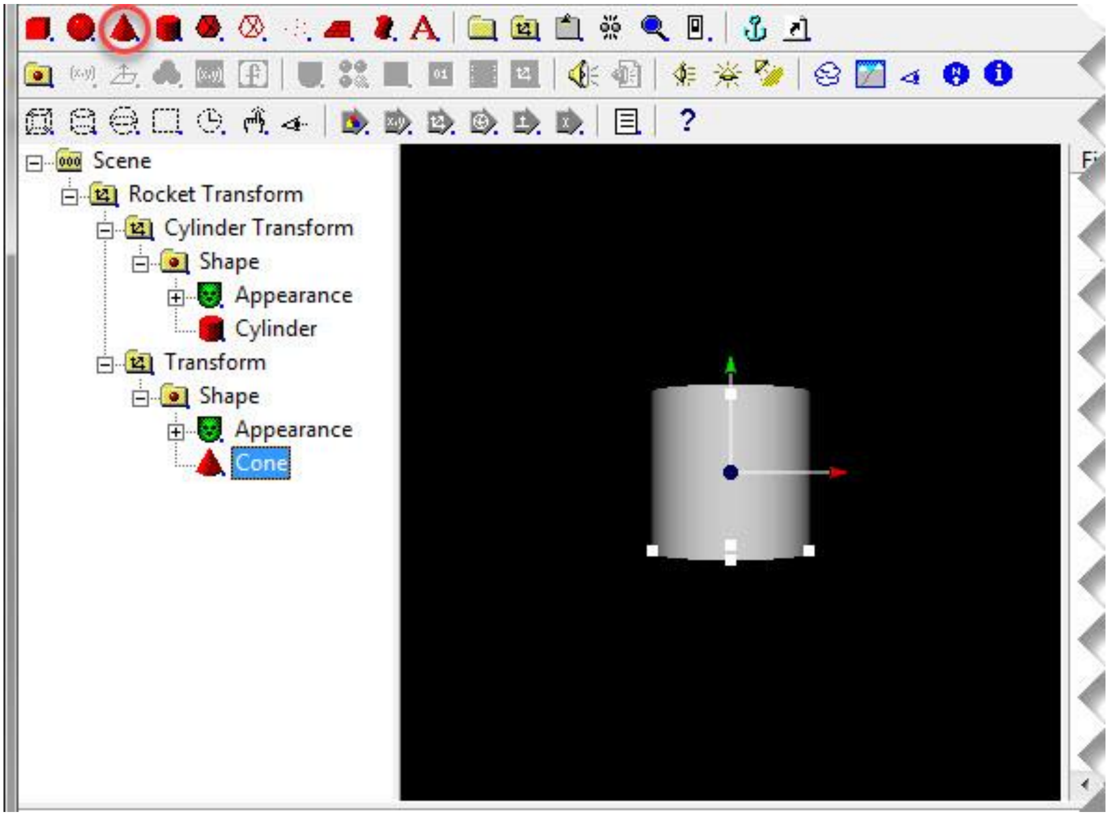


While this is not necessary for this model, by naming the Cylinder Transform, the cylinder can be controlled separately in Embed.

5. To create the nose cone:

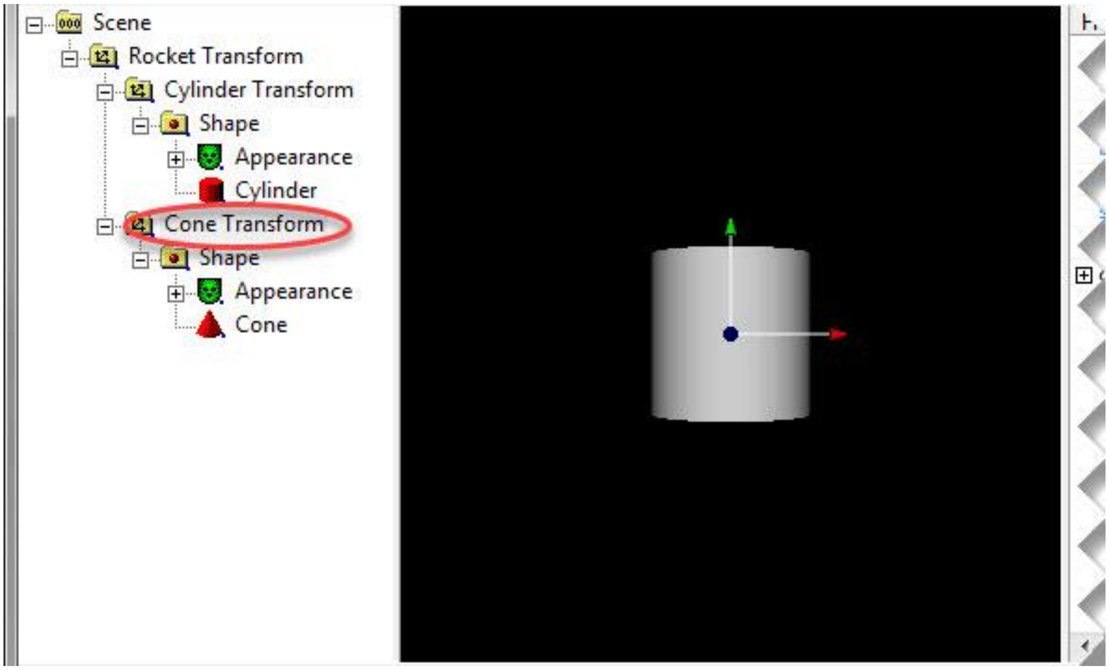
   a. In the **Scene Tree**, select **Rocket Transform**.

   b. In the **Toolbar**, click the **red cone**.



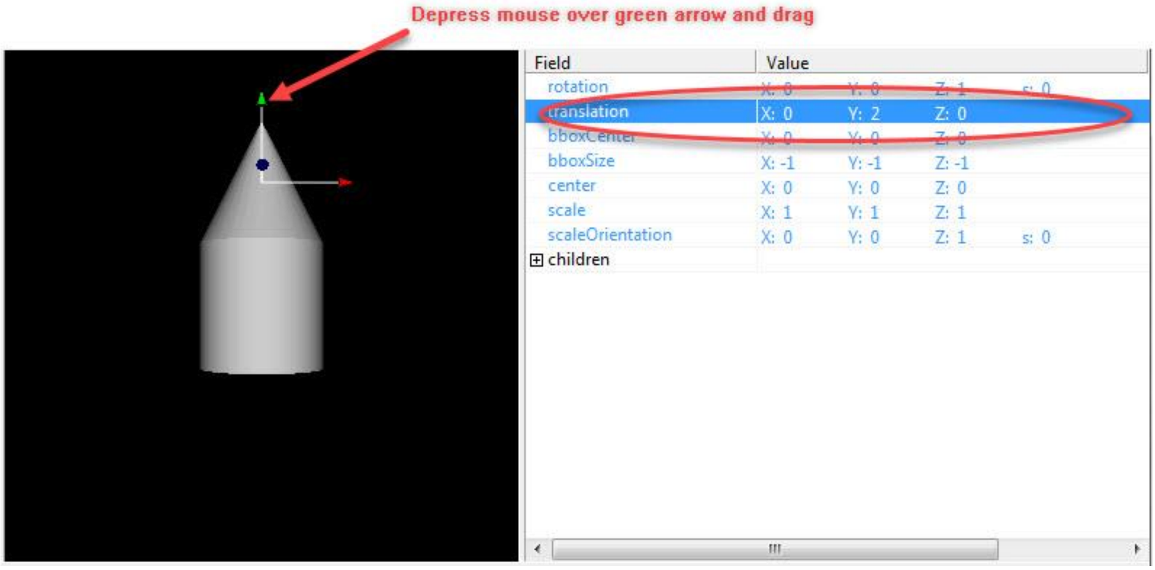
In the **3D Preview window**, the nose cone has been added; however, it is placed inside the cylinder. You position the nose cone correctly in Step 5d.

   c. In the **Scene Tree**, select **Transform**, then click **Edit > DEF** and enter **Cone**.



While this is not necessary for this model, by naming the Cone Transform, the nose cone can be controlled separately in Embed.

d. To move the cone to the correct position, you can either **drag the green arrow on the *y* axis upward** or enter the **Field translation values X:0; Y:2; Z:0** in the corresponding **Field View window**.



Depress mouse over green arrow and drag

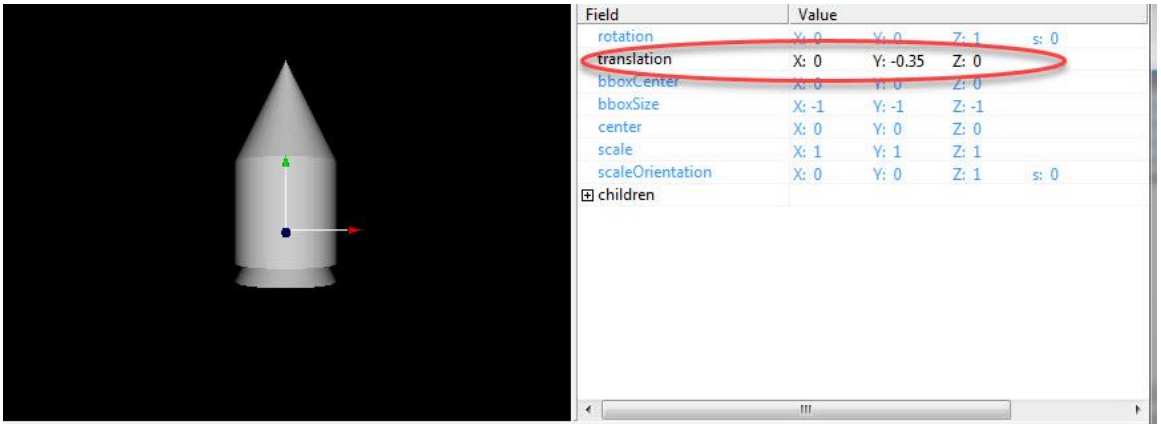
Field	Value			
rotation	X: 0	Y: 0	Z: 1	s: 0
translation	X: 0	Y: 2	Z: 0	
bboxCenter	X: 0	Y: 0	Z: 0	
bboxSize	X: -1	Y: -1	Z: -1	
center	X: 0	Y: 0	Z: 0	
scale	X: 1	Y: 1	Z: 1	
scaleOrientation	X: 0	Y: 0	Z: 1	s: 0
⊞ children				

When moving the nose cone with the mouse, you may not be able to position it precisely. In this case, simply edit the Field translation values.

6. To create the exhaust nozzle:

   a. Repeat steps 5a – 5c.

   b. To move the nozzle to the correct position, you can either **drag the green arrow on the *y* axis downward** or enter the appropriate **Field translation values** in the corresponding **Field View window**.



Field	Value			
rotation	X: 0	Y: 0	Z: 1	s: 0
translation	X: 0	Y: -0.35	Z: 0	
bboxCenter	X: 0	Y: 0	Z: 0	
bboxSize	X: -1	Y: -1	Z: -1	
center	X: 0	Y: 0	Z: 0	
scale	X: 1	Y: 1	Z: 1	
scaleOrientation	X: 0	Y: 0	Z: 1	s: 0
⊞ children				

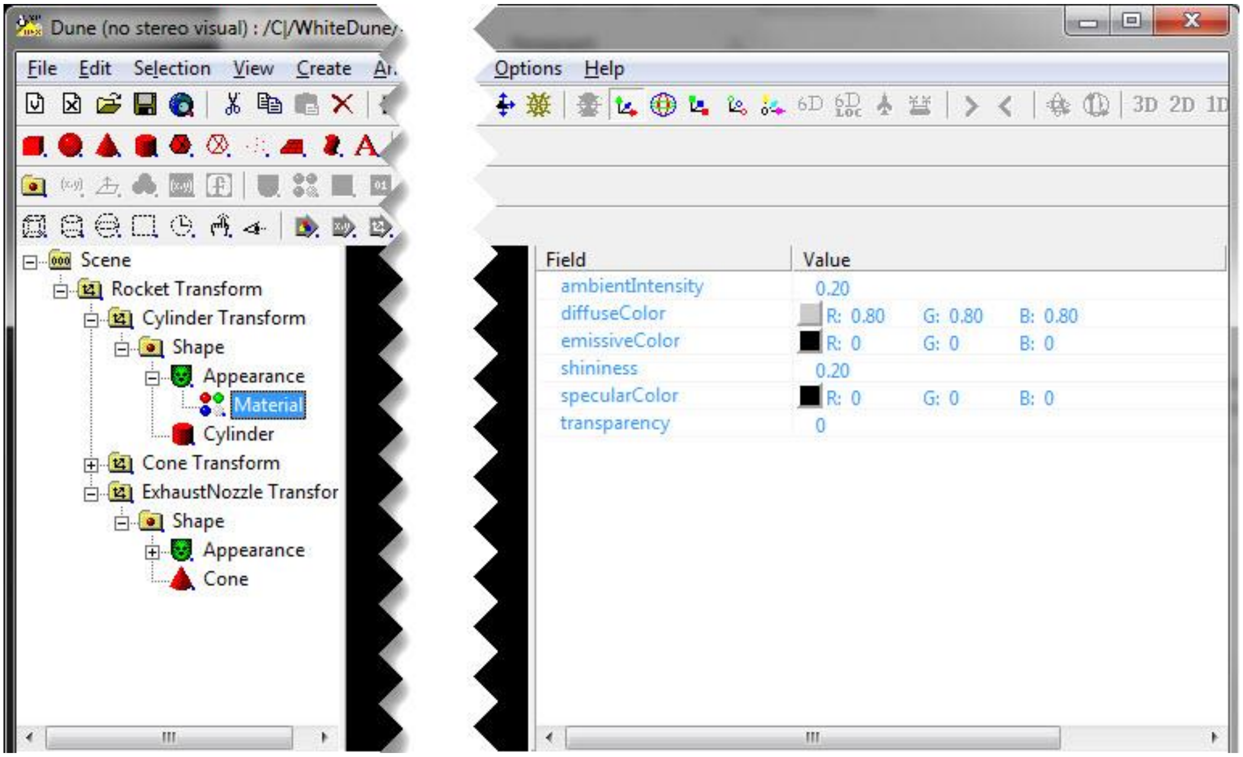
   c. Rename the transform to **ExhaustNozzle**.

   d. To resize the nozzle, follow the directions under Changing the dimensions of an element of the rocket.

7. Click **File > Save As** to save your newly-created virtual reality rocket model as a WRL file.

Changing the color of the rocket

After creating the rocket model, you may want to change its appearance before importing it into Embed. This is done by editing Field values for the Material nodes. You can alter node color, shininess, and transparency in values from 0 - 1.

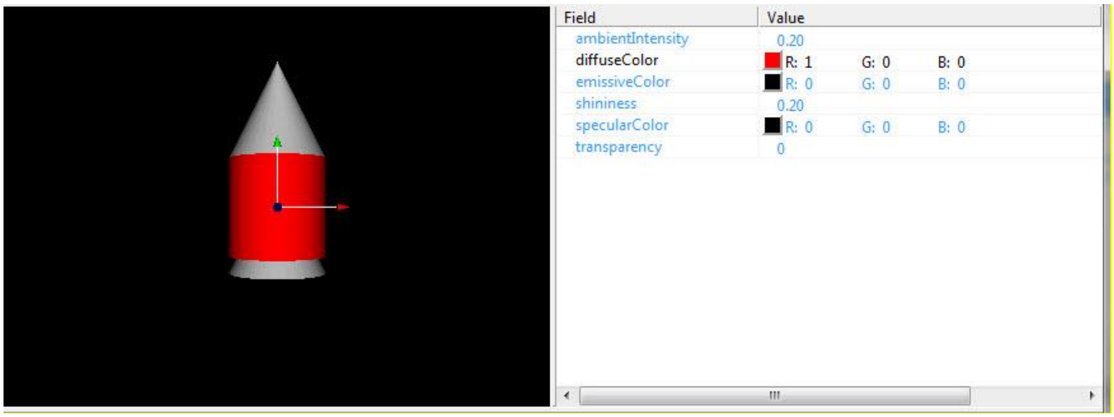


Field	Value		
ambientIntensity	0.20		
diffuseColor	R: 0.80	G: 0.80	B: 0.80
emissiveColor	R: 0	G: 0	B: 0
shininess	0.20		
specularColor	R: 0	G: 0	B: 0
transparency	0		

For the rocket model, the cylinder will be colored red, the nose cone will be colored white, and the exhaust nozzle will remain grey.

1. In the **Scene Tree**, under **Cylinder Transform > Shape > Appearance**, click on **Material** and select **diffuseColor in the Field Value** window.

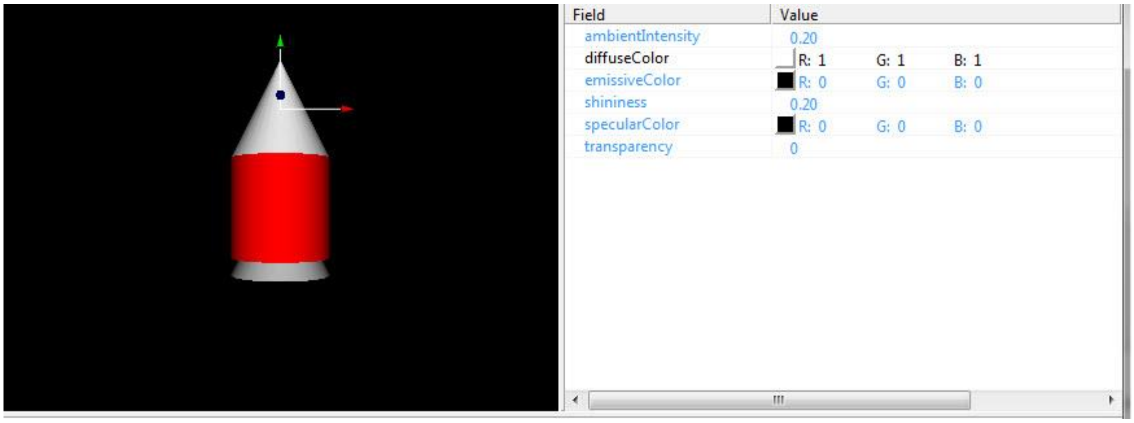
2. In the toolbar, click **Color Wheel** (🎨) to select a color or change the **diffuseColor** RGB values to 1 0 0.



Field	Value		
ambientIntensity	0.20		
diffuseColor	R: 1	G: 0	B: 0
emissiveColor	R: 0	G: 0	B: 0
shininess	0.20		
specularColor	R: 0	G: 0	B: 0
transparency	0		

3. In the **Scene Tree**, under **Cone Transform > Shape > Appearance**, click on **Material**.

4. Repeat Step 2 and color the nose cone white.



Field	Value			
ambientIntensity	0.20			
diffuseColor	□	R: 1	G: 1	B: 1
emissiveColor	■	R: 0	G: 0	B: 0
shininess	0.20			
specularColor	■	R: 0	G: 0	B: 0
transparency	0			

Feel free to experiment with the emissiveColor, specularColor, shininess, and transparency field values.

5. Click **File > Save** to save your work.

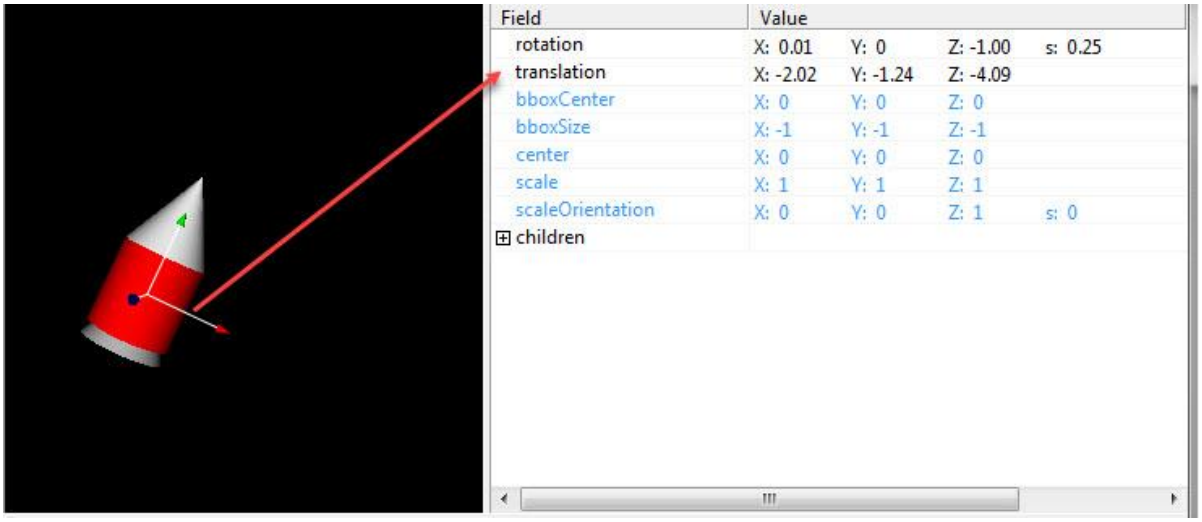
Moving the rocket

You move the rocket by moving its coordinate system relative to the world coordinate system.

1. In the **Scene Tree**, select **Rocket Transform**.

2. In the toolbar, click **Move** ( ).

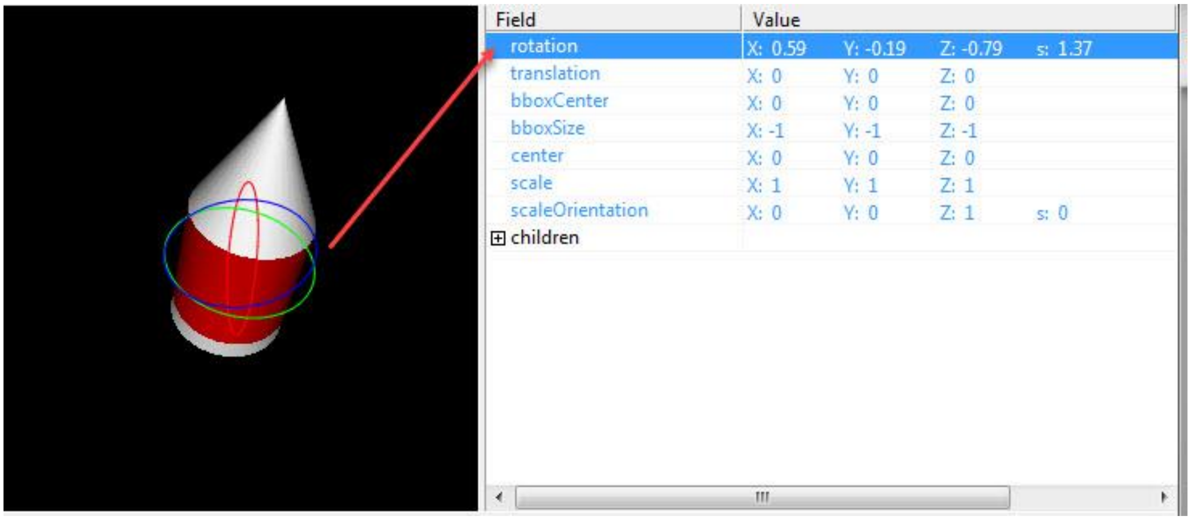
3. Move the rocket by dragging on the axes. The x-y-z translation values are updated accordingly.



Field	Value			
rotation	X: 0.01	Y: 0	Z: -1.00	s: 0.25
translation	X: -2.02	Y: -1.24	Z: -4.09	
bboxCenter	X: 0	Y: 0	Z: 0	
bboxSize	X: -1	Y: -1	Z: -1	
center	X: 0	Y: 0	Z: 0	
scale	X: 1	Y: 1	Z: 1	
scaleOrientation	X: 0	Y: 0	Z: 1	s: 0
⊞ children				

Rotating the rocket

1. In the **Scene Tree**, select **Rocket Transform**.
2. In the toolbar, click **Rotate** (⊕).
3. Rotate the rocket by dragging on the circles. The *x-y-z* rotation values are updated accordingly.



Field	Value			
rotation	X: 0.59	Y: -0.19	Z: -0.79	s: 1.37
translation	X: 0	Y: 0	Z: 0	
bboxCenter	X: 0	Y: 0	Z: 0	
bboxSize	X: -1	Y: -1	Z: -1	
center	X: 0	Y: 0	Z: 0	
scale	X: 1	Y: 1	Z: 1	
scaleOrientation	X: 0	Y: 0	Z: 1	s: 0
⊞ children				

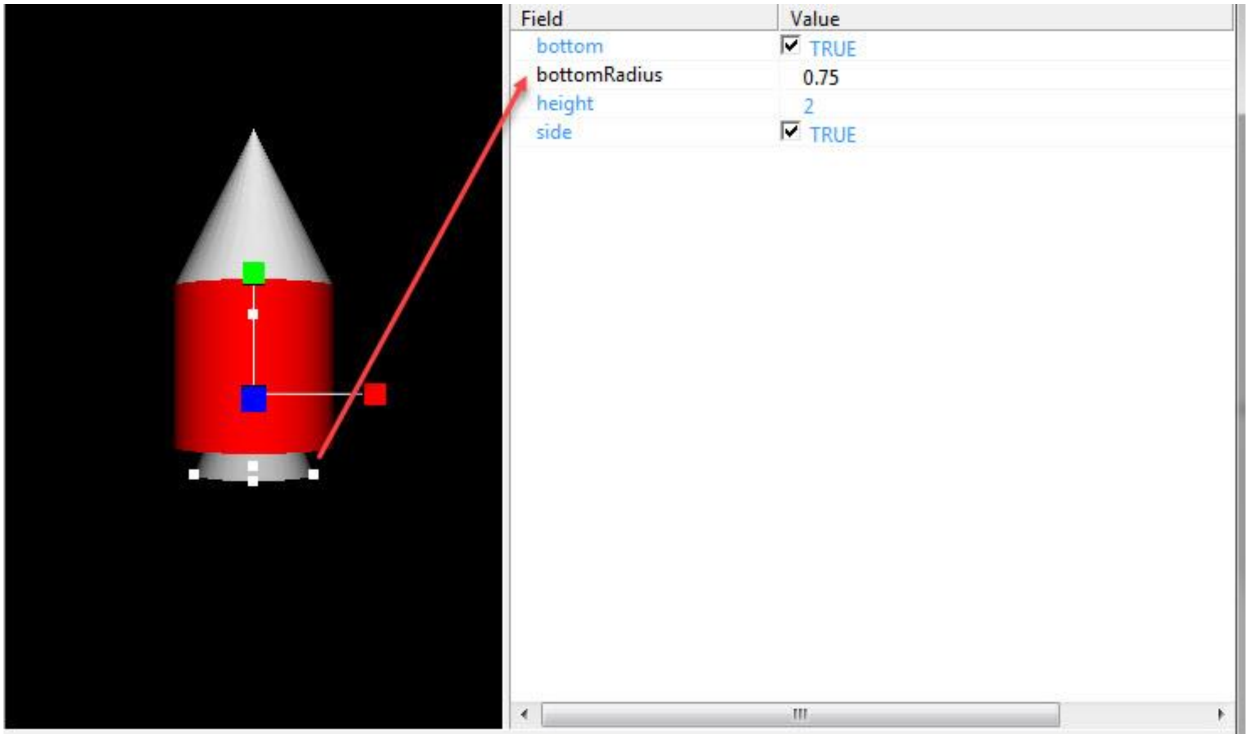
Moving and rotating an element of the rocket

You can move and rotate individual elements in the rocket by selecting the corresponding transform in the Scene Tree and repeating the steps under Moving the rocket and Rotating the rocket.

Changing the dimensions of an element of the rocket

You can make individual elements in a virtual world larger or smaller. In the rocket model, the exhaust nozzle is proportionally too large for the rocket and needs to be shrunk down.

1. In the **Scene Tree**, select ExhaustNozzle Transform > Shape > Cone.
2. In the **Field View** window, change **bottomRadius** to **0.75**. The exhaust nozzle shrinks accordingly.

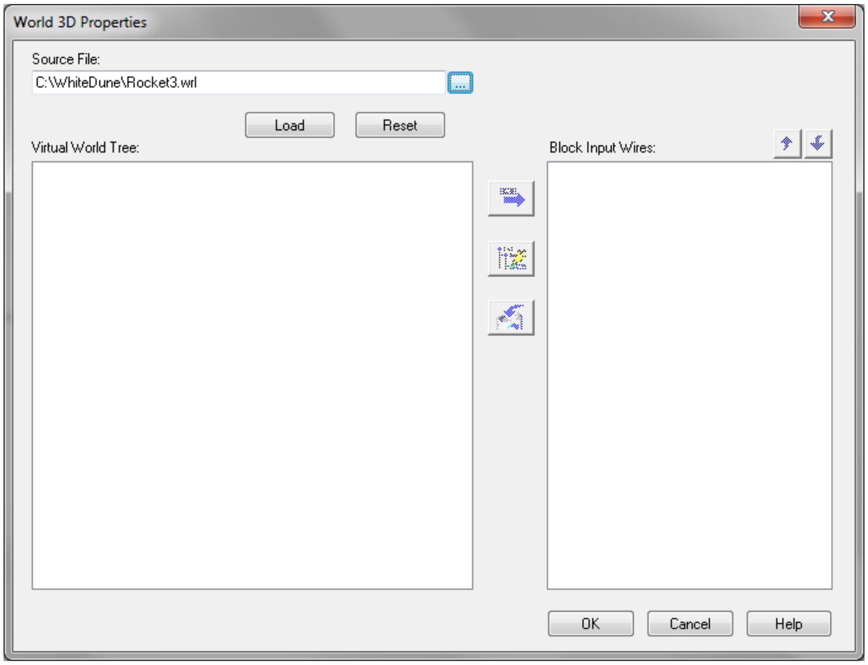


Field	Value
bottom	☑ TRUE
bottomRadius	0.75
height	2
side	☑ TRUE

Connecting the rocket model to an Embed diagram

After you create a virtual reality model, you connect the model to your Embed diagram so that it can interact with a dynamic system simulation. In White_Dune, virtual reality models are saved as WRL files. To load and view a WRL file in Embed, you use the world3D and animation 3D blocks.

1. Open a new diagram in Embed.
2. Under **Blocks > Animation**, insert a **world3D** and an **animation3D** block and wire the blocks together.
3. Right-click the **world3D** block.
4. In the **World 3D Properties** dialog, under **Source File**, click **…** to select the rocket WRL file you previously created.



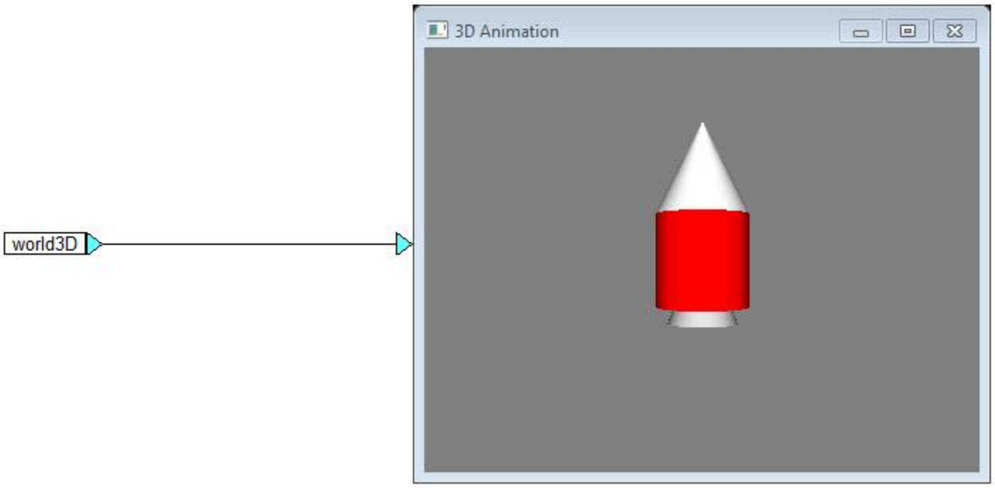
World 3D Properties

Source File: C:\WhiteDune\Rocket3.wrl

5. Click **Load**.

   The rocket WRL file is loaded into the world3D block.

6. Click **OK** and view the rocket in the **animation3D** block.



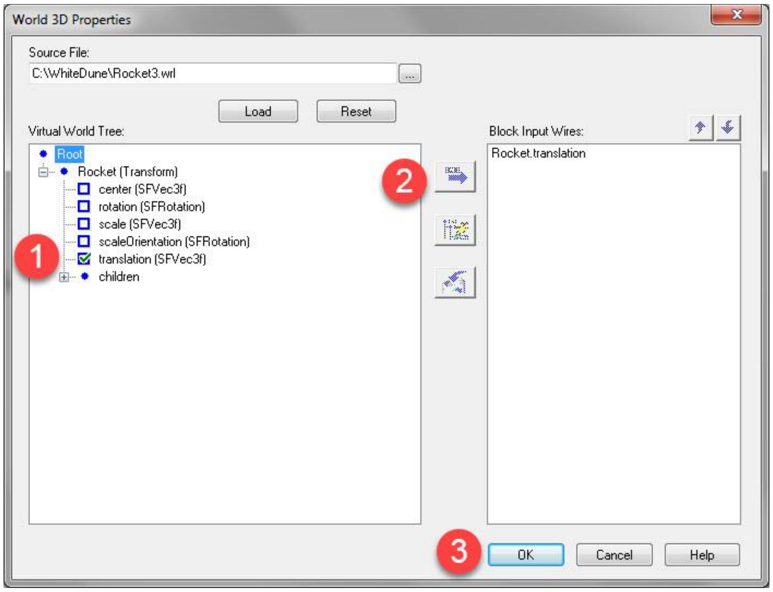
Visualizing rocket animation in Embed

Once a WRL file is loaded into the world3D block, you can control one or more VRML node fields — for example, center, rotation, scale, scaleOrientation, and translation — for the virtual reality model. For the rocket model, you can control these fields for the entire rocket or, because the two shape nodes for the rocket were named, you can also control them for the cylinder base or cone individually. In this section, you will control only rocket translation and rotation, but feel free to try out different control scenarios on your own.

Applying translation and rotation to the rocket

To access and control a VRML parameter in Embed, you assign the parameter to an input connector on the world3D block.

1. Right-click the **world3D** block.
2. In the **Virtual World Tree**, expand the **Rocket** hierarchy, if necessary.
3. Select **translation (SFVec3f)**, click , and then click **OK.**

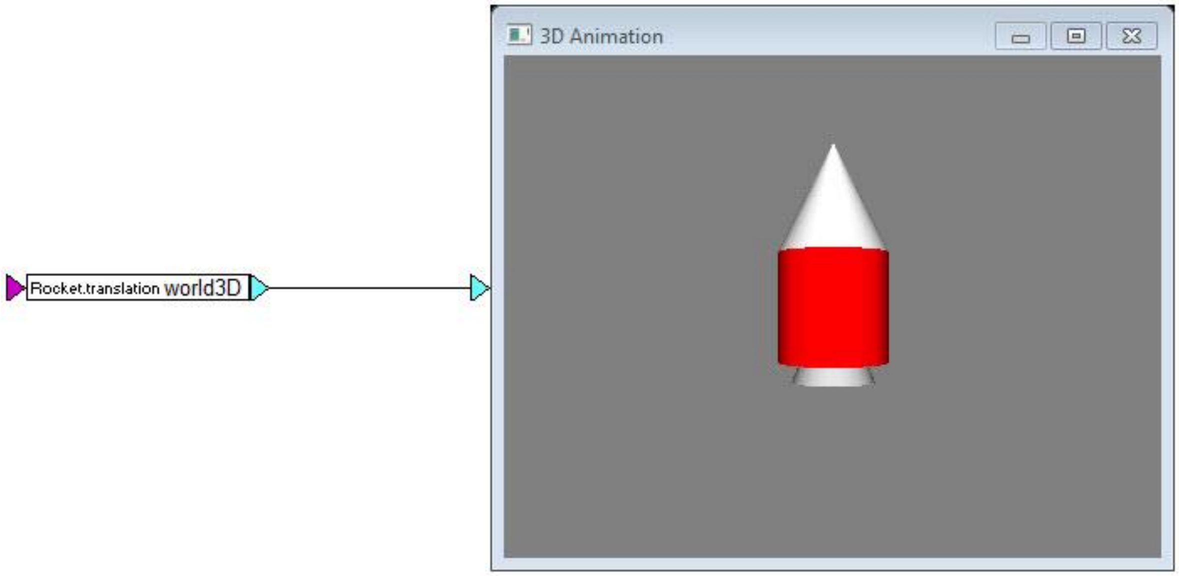


World 3D Properties

Source File: C:\WhiteDune\Rocket3.wrl

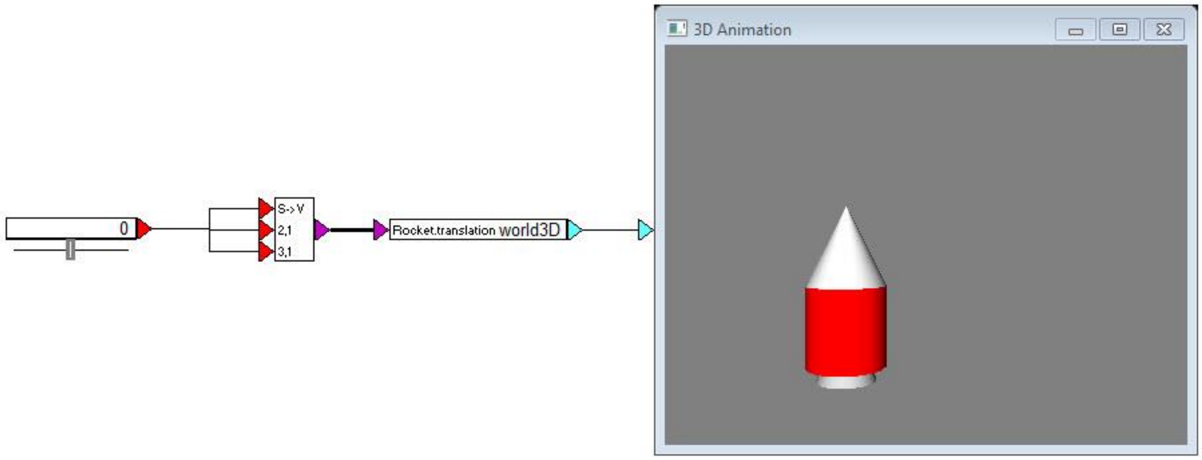
Block Input Wires: Rocket.translation

A Rocket.translation input connector is added to the **world3D** block. You can now apply signals to the **Rocket.translation** input to visualize rocket translation during simulation.

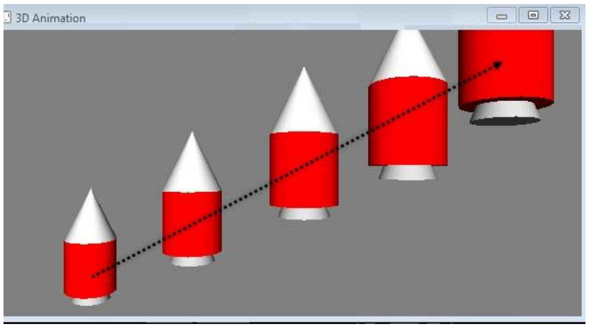


The **Rocket.translation** input accepts a three-element floating-point vector (*x, y, z*).

4. To send a vector signal to **Rocket.translation** input, wire a **scalarToVec block** to **Rocket.translation** and attach a **slider** to the **scalarToVec** inputs to control translation.



5. Click the ▶ button to begin the simulation and visualize the animation.

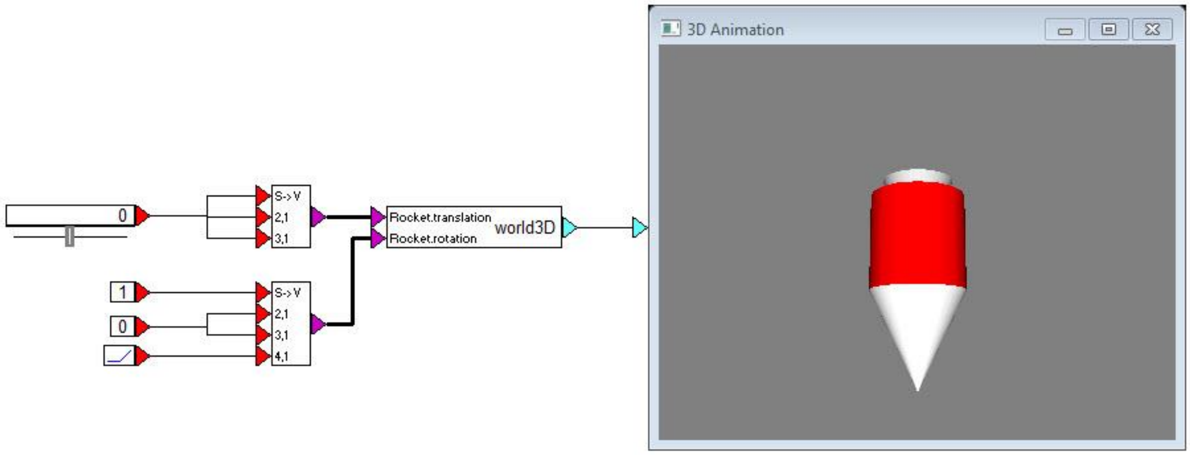


The dotted arrow shows the translation path as you slide the slider to higher values.

6. To add rotation to the rocket, repeat steps 1 – 3, but this time, select **rotation (SFRotation)**.

   Rocket.rotation accepts a four-element vector (*x, y, z,* and angle of rotation).

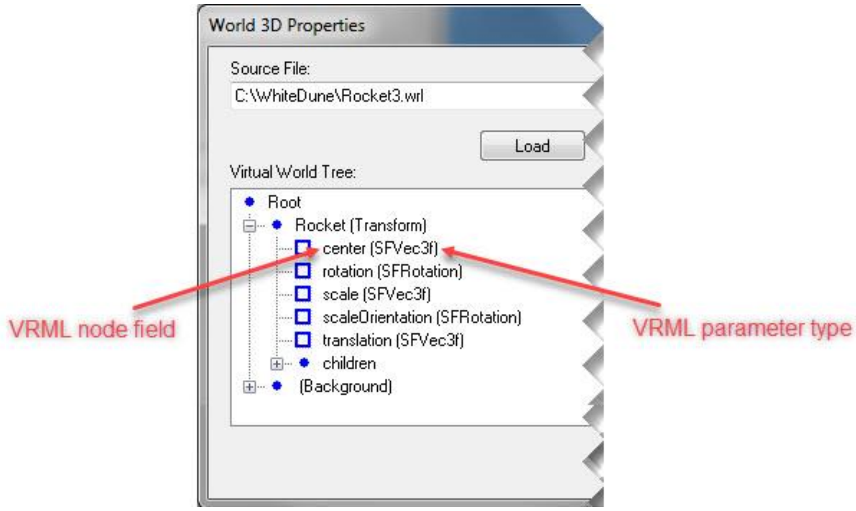
7. Wire a four-element **scalarToVec** block to **Rocket.rotation** and input signals to the **scalarToVec** block to control rotation.



In this configuration, as the simulation runs, the rocket has an axis of rotation of 1 0 0 and follows the translation path controlled by the slider.

VRML node fields and types for the world3D block

If a node has a given name, its VRML node field values can be controlled by Embed. In the world3D dialog, each VRML node field includes a corresponding VRML parameter type. The VRML parameter type defines the input signal type for the node field.



Source File: C:\WhiteDune\Rocket3.wrl

VRML node field

VRML parameter type

VRML Parameter Type	Description	Value
SFBool	Single Boolean	Any single value
SFFloat	Single float	Any single value
SFInt32	Single integer	Any single value
SFVec2f	Vector 2 floats	Two-element vector (*x, y*)
SFVec3f	Vector 3 floats	Three-element vector (*x, y, z*)
SFColor	Color	Three-element vector (R, G, B with values between 0.0 and 1.0)
SFTime	Time	Double
SFRotation	Rotation	Four-element vector (*x*, *y*, *z*, angle of rotation)
MFFloat	Multiple floats	Any vector
Mflut32	Multiple integers	Any vector
MFVec2f	Multiple vectors of 2	n x 2 matrix
MFVec3f	Multiple vectors of 3	n x 3 matrix
MFColor	Multiple colors	n x 3 matrix
MFTime	Multiple times	Any vector
MFRotation	Multiple rotations	n x 4 vector

Adding realism to your rocket model

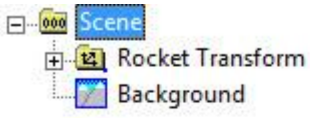
There are many ways to edit your virtual reality models to make them more realistic. This section describes how to add two visual effects: background colors and rocket exhaust flames.

Adding background

You use the Background node to specify the color of the sky and ground.

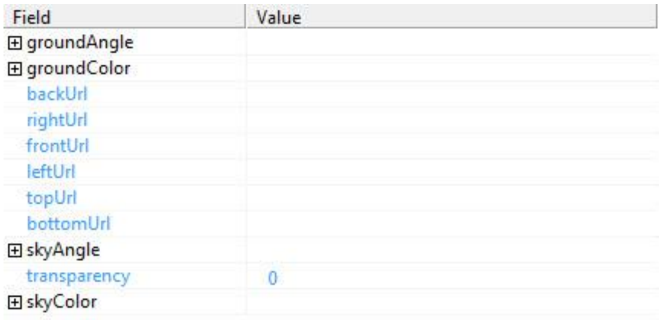
1. In the **Scene Tree**, select **Scene**.

2. In the toolbar, select **Background** ().



Scene
Rocket Transform
Background

3. When you want to work on the background, select **Background**. The Field View window displays the background fields and values for the sky and ground.

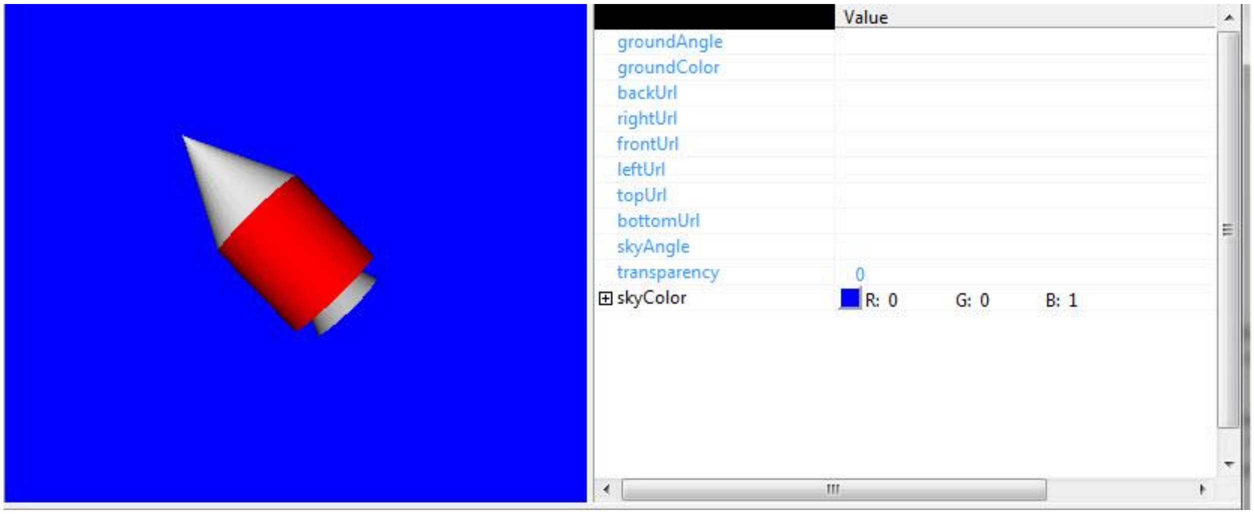


Field	Value
⊞ groundAngle	
⊞ groundColor	
backUrl	
rightUrl	
frontUrl	
leftUrl	
topUrl	
bottomUrl	
⊞ skyAngle	
transparency	0
⊞ skyColor	

To add background sky and ground color, you use the **skyColor**, **skyAngle**, **groundColor**, and **groundAngle** fields. The **Background** node also lets you define a background panorama layered between the sky-ground colors and the virtual reality model using the **backUrl**, **rightUrl**, **frontUrl**, **leftUrl**, **topUrl**, and **bottomUrl**. Detailed information on these fields, along with information on transparency and fog can be found in the online VRML documentation.

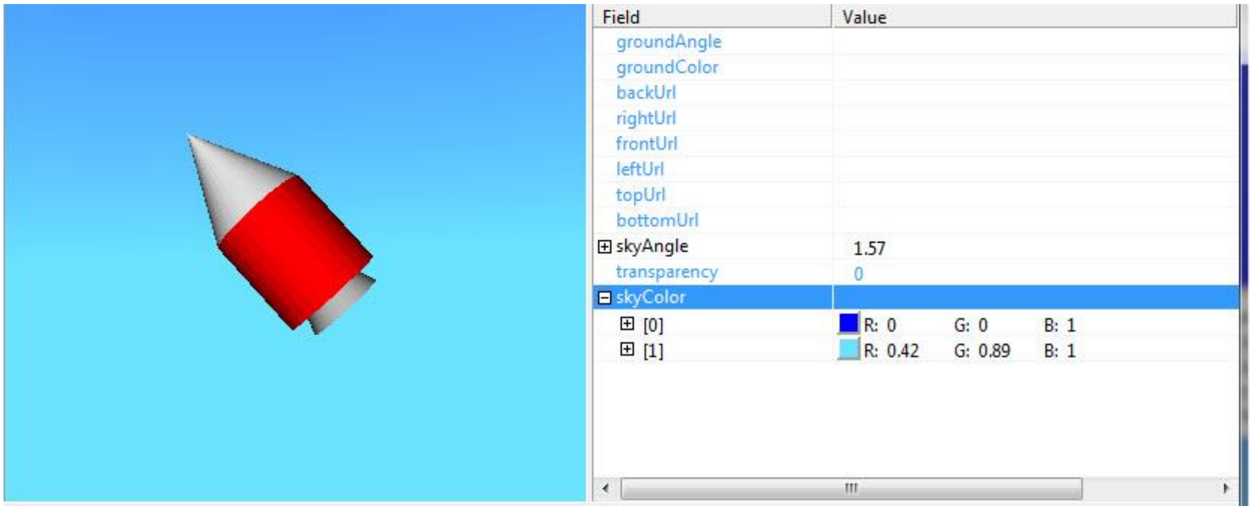
Defining the sky

In your 3D world, the sky is a limitless sphere that surrounds your virtual reality model. The sky can be a single color or consist of a blend of two or more colors that creates a gradient effect. The **skyColor** and **skyAngle** fields specify the sky color. To create a single-colored sky, specify the **skyColor** field as an **RGB** color (with values ranging from 0 – 1) and leave the **skyAngle** field empty. For example, below is a solid blue sky with an RGB value of 0 0 1.



	Value
groundAngle	
groundColor	
backUrl	
rightUrl	
frontUrl	
leftUrl	
topUrl	
bottomUrl	
skyAngle	
transparency	0
⊞ skyColor	R: 0 G: 0 B: 1

To create a gradient effect, you must specify at least two **skyColor** fields as **RGB** colors. The first value of **skyColor** is the color of the sky at 0.0 radians (that is, the zenith of the sphere). The **skyAngle** field specifies the angle of the gradient in radians. The angle ranges from 0.0 – pi in increasing values, where 0.0 radians is the zenith of the sphere; 1.57 radians (pi/2 radians or approximately 90°) is the natural horizon; and 3.14 radians (pi radians) is the nadir. You can specify as many colors and angles in a gradient sky as you want; however, because the first color is always the color at the zenith of the sphere, you must specify one less angle than color. Below is an example of a fading blue sky (RGB 0 0 1 to RGB 0.42 0.89 1). The angle of the gradient is 1.57 radians.



Field	Value
groundAngle	
groundColor	
backUrl	
rightUrl	
frontUrl	
leftUrl	
topUrl	
bottomUrl	
⊞ skyAngle	1.57
transparency	0
⊟ skyColor	
⊞ [0]	🟦 R: 0 G: 0 B: 1
⊞ [1]	🟦 R: 0.42 G: 0.89 B: 1

To specify an RGB color:

1. Expand the **skyColor field** by clicking the **+** sign, if it is not already expanded.
2. Under **Value**, click on the **color** you want to change.
3. Enter the **new value** or in the toolbar, click **Color Wheel** (🎨) to select a color.

To add gradients:

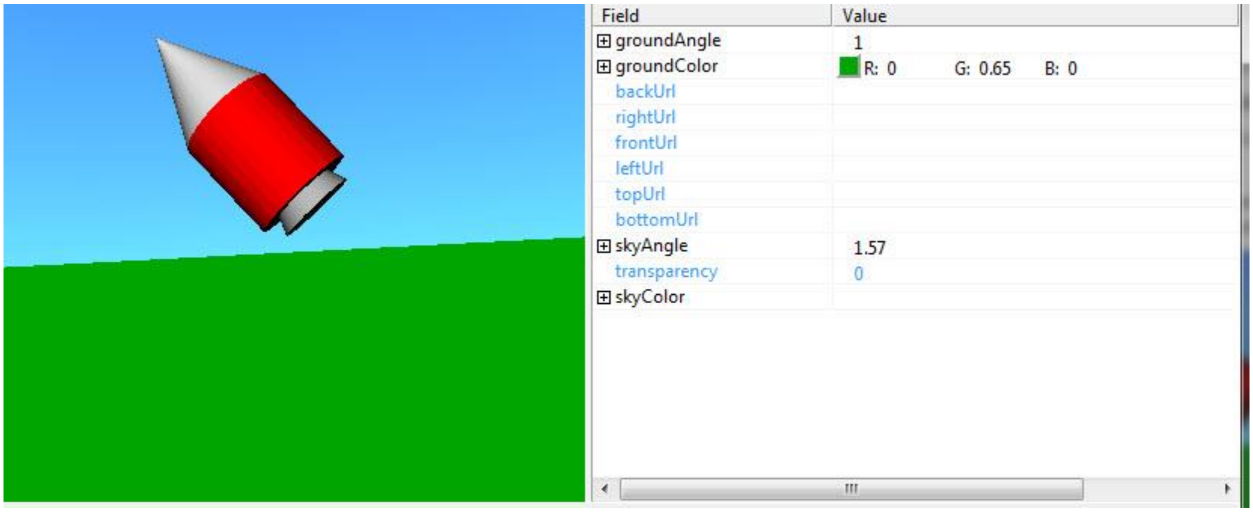
1. Expand the **skyColor field** by clicking the **+** sign, if it is not already expanded.
2. In the numbered list, click a **+** sign to add a gradient.

To add sky angles:

1. Expand the **skyAngle field** by clicking the **+** sign, if it is not already expanded.
2. In the numbered list, click the **+** sign to add an angle.
3. In the corresponding **Value** field, add a value in radians.

Defining the ground

The ground is a limitless sphere surrounding your virtual reality model. It can have solid or gradient color; however, because the ground sphere is inside the sky sphere, if you do not apply color to the ground sphere, you will see the sky color.



Field	Value
⊞ groundAngle	1
⊞ groundColor	🟩 R: 0 G: 0.65 B: 0
backUrl	
rightUrl	
frontUrl	
leftUrl	
topUrl	
bottomUrl	
⊞ skyAngle	1.57
transparency	0
⊞ skyColor	

Solid green ground (RGB 0 0.65 0) has been added to the background using the **groundAngle** and **groundColor** fields. The rules that apply to the skyAngle and skyColor fields apply to the **groundAngle** and **groundColor** fields.

To specify an RGB color:

1. Expand the **groundColor** field by clicking the **+** sign, if it is not already expanded.
2. Under **Value**, click on the **color** you want to change.
3. Enter the **new value** or in the toolbar, click **Color Wheel** (🎨) to select a color.

To add gradients:

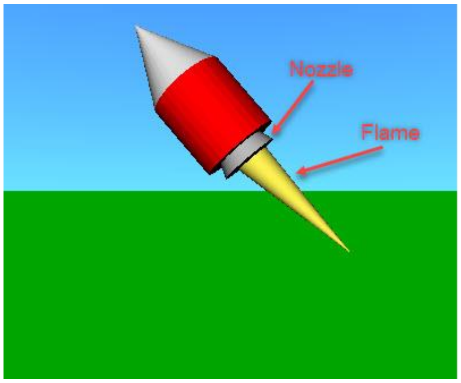
1. Expand the **groundColor** field by clicking the **+** sign, if it is not already expanded.
2. In the numbered list, click a **+** sign to add a gradient.

To add ground angles:

1. Expand the **groundAngle** field by clicking the **+** sign, if it is not already expanded.
2. In the numbered list, click the **+** sign to add an angle.
3. In the corresponding **Value** field, add a value in radians.

Adding an exhaust flame to the rocket

The exhaust flame comes out of the nozzle. For more realistic operation, the flame will be designed to turn on and off based on the value of the signal fed into it during simulation in Embed.

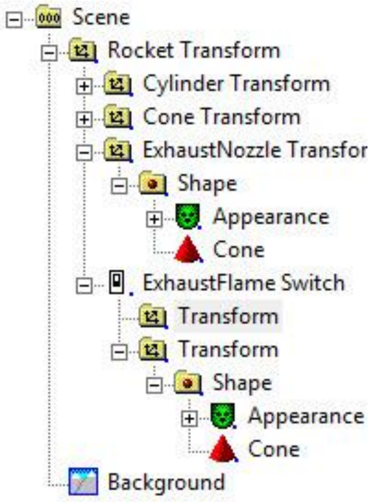


To add a flame:

1. In the **Scene Tree**, select **Rocket Transform**.
2. In the menu, click **Create > Grouping Node > Switch**.
3. Rename the switch to **ExhaustFlame** using the **Edit > DEF** command.
4. Under **ExhaustFlame Switch**, create **two transforms**: one will be **empty**; the other will contain a **cone**, which will be the flame.
    a. To create the empty transform, click **Create > Grouping Node > Transform**.

b. To create a transform that will be the flame, in the **toolbar**, click the **red cone** (▲).

Your Scene Tree will look like this:



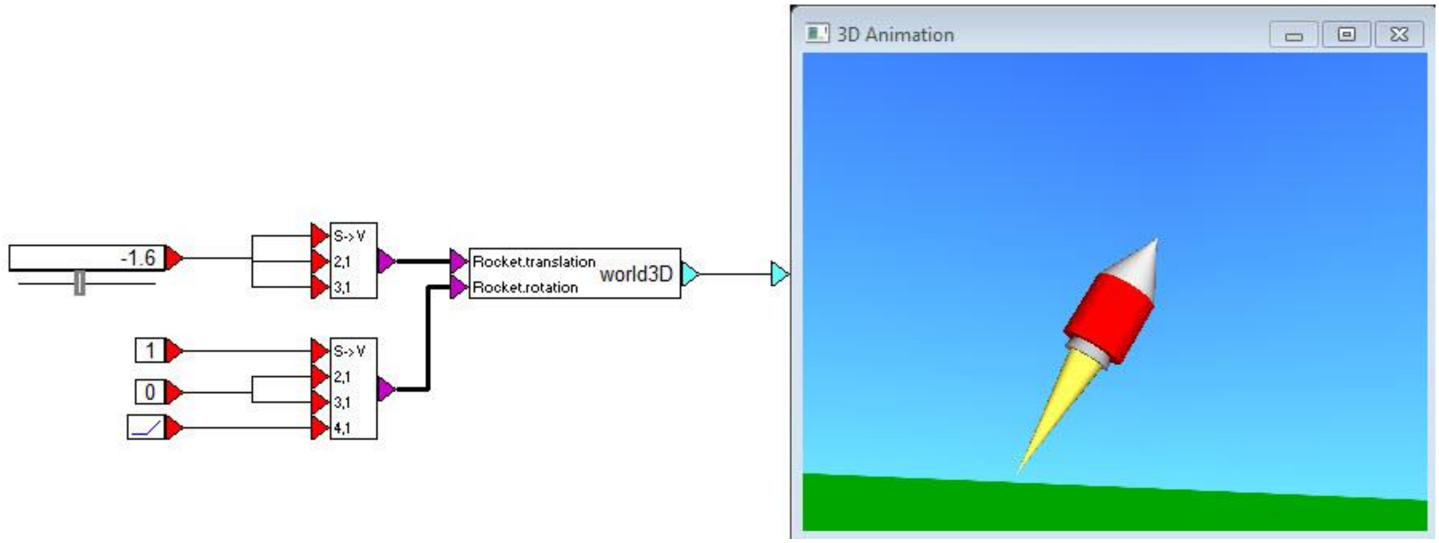
- Scene
  - Rocket Transform
    - Cylinder Transform
    - Cone Transform
    - ExhaustNozzle Transfor
      - Shape
        - Appearance
        - Cone
    - ExhaustFlame Switch
      - Transform
      - Transform
        - Shape
          - Appearance
          - Cone
  - Background

5. The flame cone has been added; however, it cannot be seen because it is inside the cylinder. To make it visible and color it yellow:
   a. In the **Scene Tree**, select **ExhaustFlame Switch** and set **whichChoice** to **1**.
   b. Move the flame cone by following the directions under Moving and rotating an element of the rocket.
   c. Color the flame by following the directions under Changing the color of the rocket.
6. Save your work.

Updating the world3D block with a new rocket model

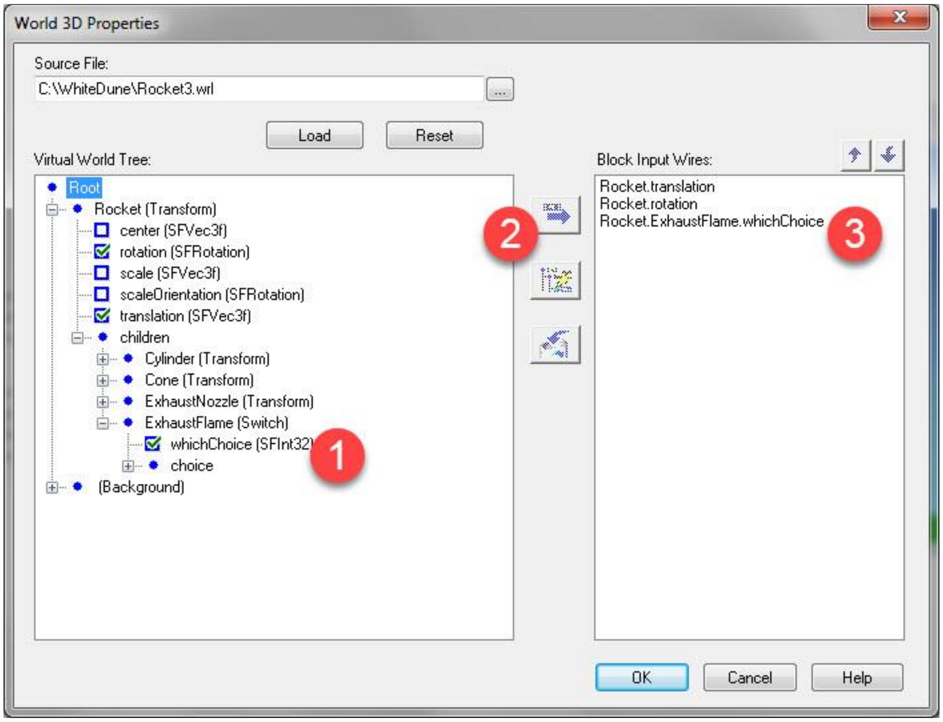
When you edit an existing WRL file that is used in a **world3D** block, Embed automatically updates the **world3D** block with the updated WRL file when you open the diagram. If the diagram is already open, close the diagram and re-open it again for the updated WRL file to take effect.

After adding a background and exhaust flames to the rocket model, the rocket animation appears as follows during simulation in Embed:

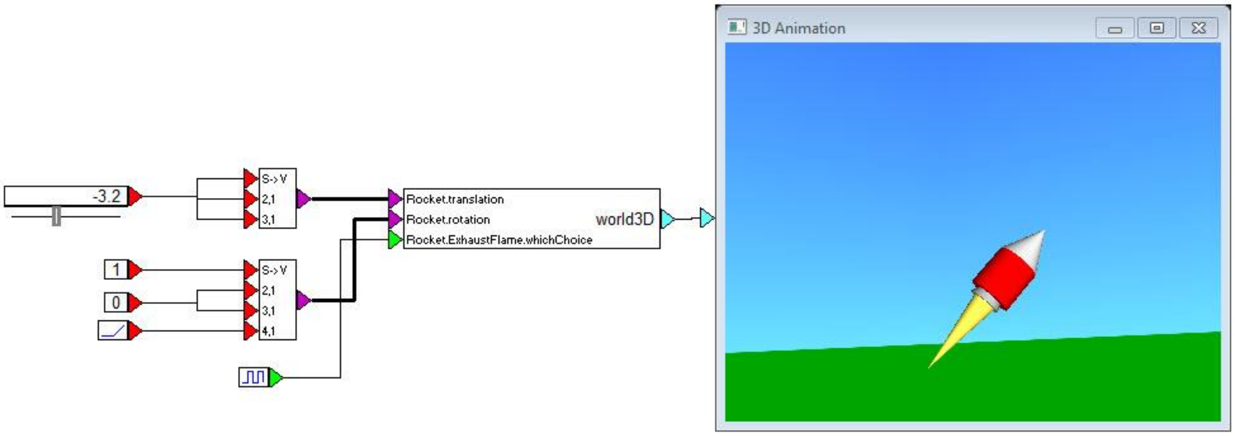


To control the flame:

1. Add the exhaust flame input to the **world3D** block.



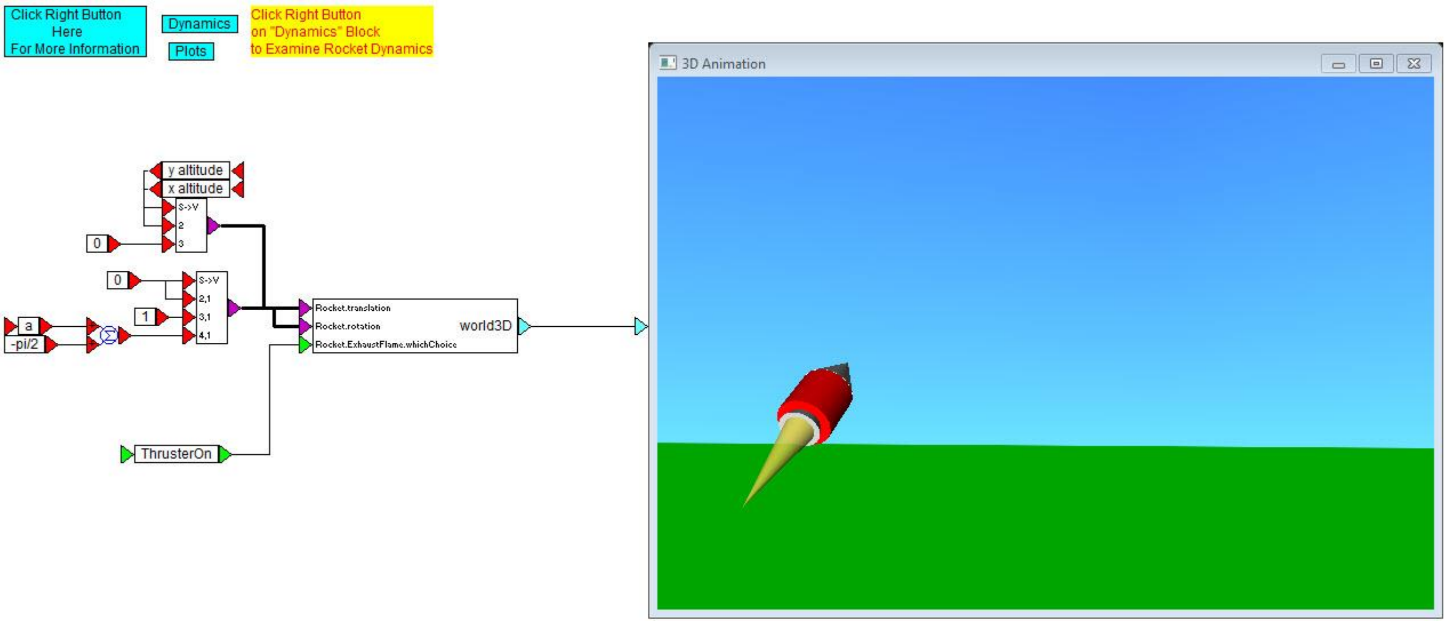
2. Wire a **squareWave** block into the **Rocket.ExhaustFlame.whichChoice** input on the **world3D** block.



Using the default settings in the **squareWave** block, the exhaust flame cycles on and off in one second intervals during simulation.

Applying realistic dynamics to rocket animation

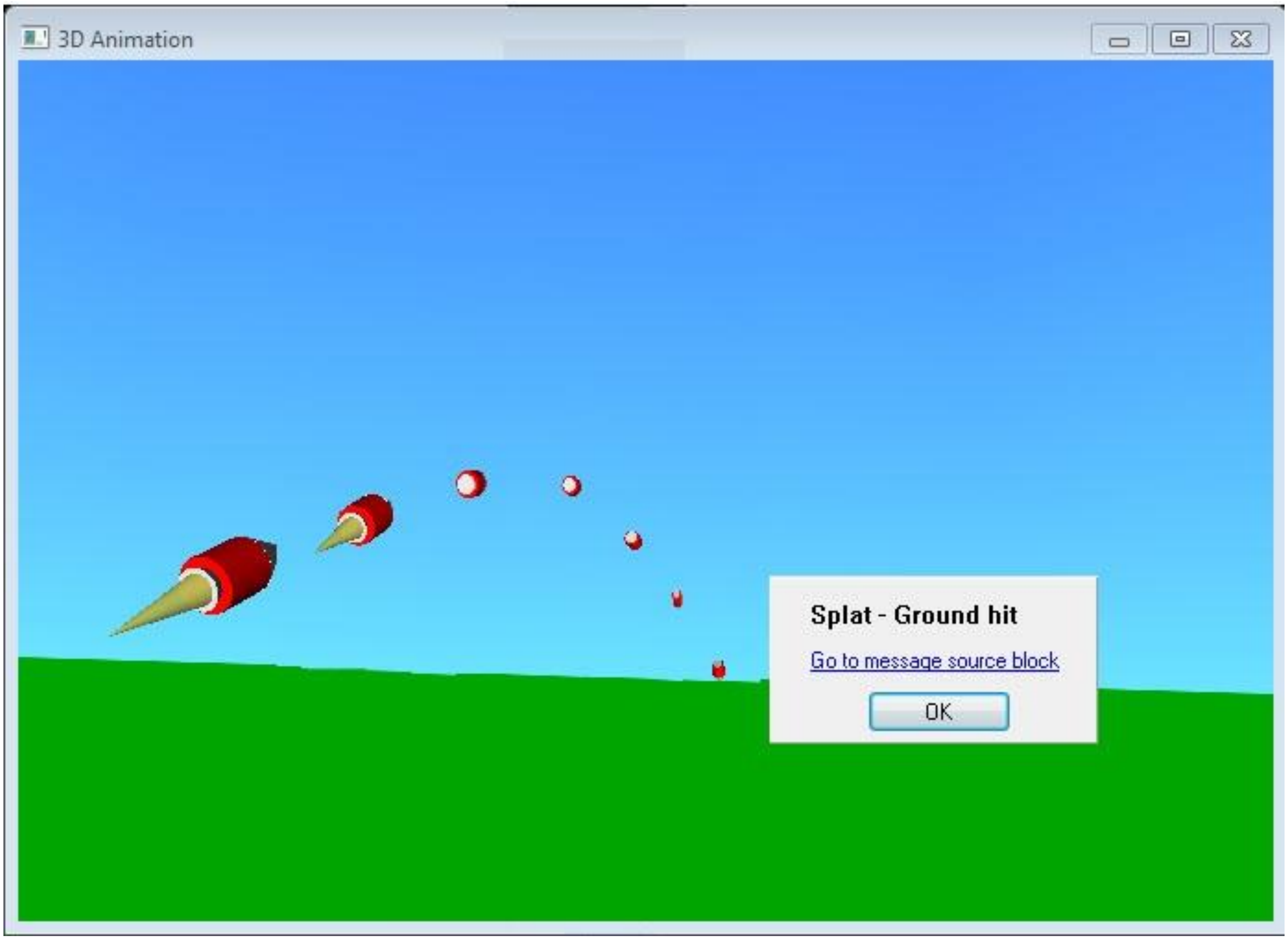
The following diagram applies realistic dynamics to a rocket model similar to the one created in this guide. These dynamics include the effects of gravity, thrust, varying mass, and drag as a function of speed and air density at each altitude. You can access the **RocketFlight** diagram under **Examples > Applications > Animate3D**.


Click Right Button Here For More Information
Dynamics
Plots
Click Right Button on "Dynamics" Block to Examine Rocket Dynamics
y altitude
x altitude
S->V
2
3
0
0
S->V
2,1
3,1
4,1
1
a
-pi/2
Rocket.translation
Rocket.rotation
Rocket.ExhaustFlame.whichChoice
world3D
ThrusterOn
3D Animation

RocketFlight dynamics is divided into five main subsystems. Double-click **Dynamics** in the upper left corner of the diagram to view the subsystems.

- **Y Altitude Calculations and X Altitude Calculations:** Models the rocket's *x* and *y* altitude. The *y* altitude calculation accurately models air density *p* in slugs/ft^3 as a function of *h* in feet.
- **Angle:** Models the rocket flight angle as a function of time. The rocket flies along an initial launch angle during the burn time and then flies an uncontrolled ballistic trajectory. The initial launch angle can be changed in the Dynamics dialog.
- **Drag:** Models the air drag in terms of deceleration versus time as a function of speed and air density *p* at each altitude. The ballistic coefficient *B*, typically 500-2000 lb/ft^2, must be specified. Note that larger *B* corresponds to less air resistance.
- **Mass:** Models the mass of the rocket in lbs as a function of time. The user-definable starting fuel mass, rate of consumption, and thrust strength are specified in the Dynamics dialog.
- **Thrust:** Models the rocket thrust in terms of acceleration as a function of time. The Specific Impulse of the rocket, typically 200-300s, must be specified. Specific Impulse is a concise means of specifying fuel effectiveness. The Fuel and Payload mass are taken from the Mass subsystem.

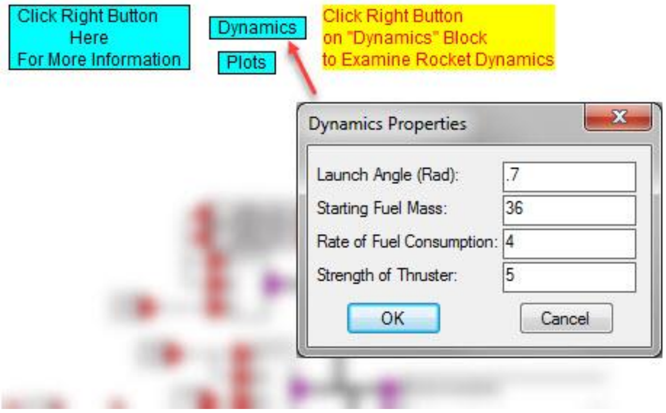
When you simulate **RocketFlight**, the rocket flies along a trajectory defined by parameters set in the Dynamics dialog. Thrust is applied until the fuel tank is empty. When the rocket hits the ground, a user-defined message appears on the screen.



Splat - Ground hit

Changing rocket dynamics

The **Dynamics** dialog lets you examine and change parameter values that affect rocket dynamics. To access the dialog, right-click **Dynamics** in the upper left corner of the diagram.


Click Right Button Here For More Information
Dynamics
Plots
Click Right Button on "Dynamics" Block to Examine Rocket Dynamics
Dynamics Properties

Launch Angle (Rad): .7
Starting Fuel Mass: 36
Rate of Fuel Consumption: 4
Strength of Thruster: 5

You can enter new values, then re-run the simulation to see how the rocket trajectory changes. You may have to increase simulation time for the rocket to complete its course.

Changing your view of the rocket

You can change your view of the rocket in the following ways:

To perform this action	Do this
Move the rocket	CTRL+right-mouse-button and drag
Rotate the rocket	CTRL+left-mouse-button and drag
Zoom in on the rocket	CTRL+Shift+right-mouse-button and drag

Code generation tasks

Arduino: Blinking the built-in LED on an Uno

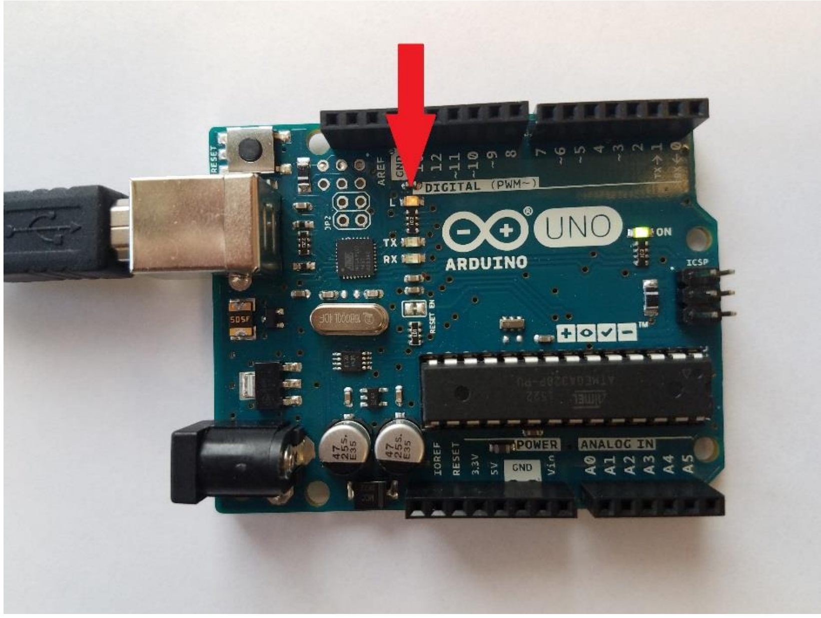
Blinking an Arduino Uno built-it LED is used to introduce basic embedded programming concepts. To blink the LED, follow the step-by-step directions below or watch a similar online video.

What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Arduino Uno	https://store.arduino.cc/usa/arduino-uno-rev3

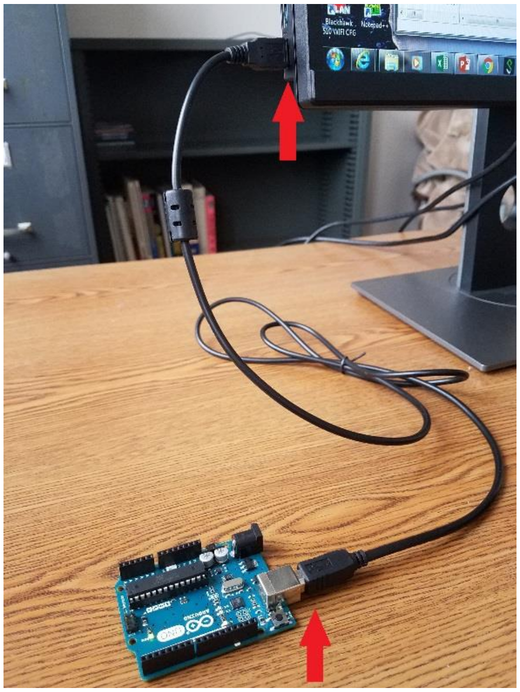
Setting up the Arduino Uno

1. Locate the built-in LED on your Arduino Uno.



The built-in LED is connected to port B, channel 5, which in turn is connected to digital pin 13 on the Arduino Uno.

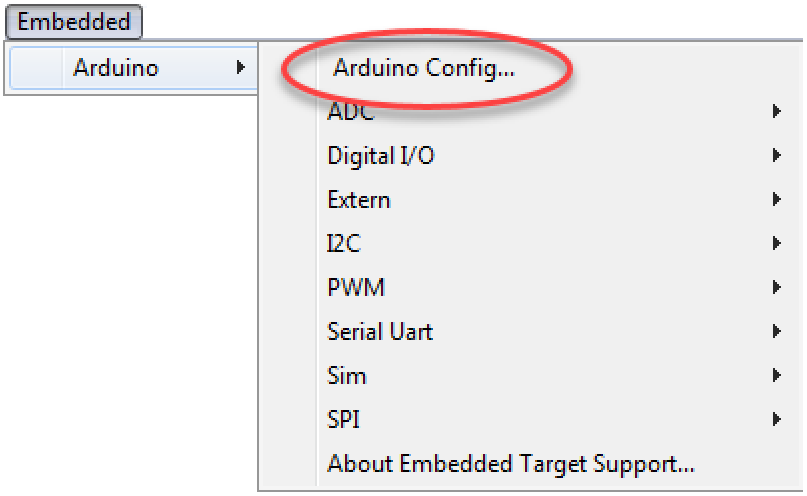
2. Attach the Arduino Uno board to your computer using a USB cable.



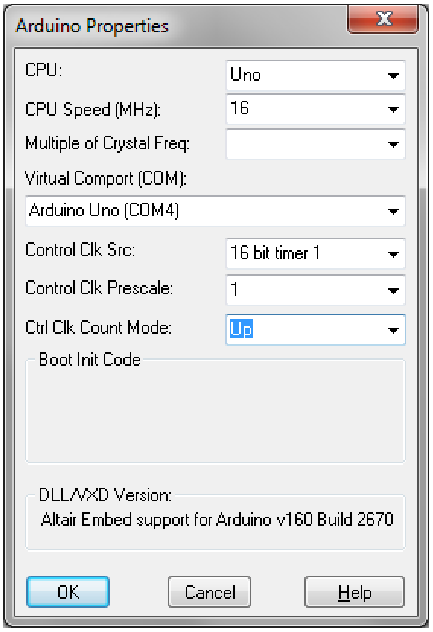
Configuring the diagram for the Arduino Uno

To construct the blink LED diagram, you use an Arduino Config block to set up the diagram.

1. Create a new diagram.
2. Choose **Embedded > Arduino** and click **Arduino Config.**



The Arduino Properties dialog appears.



Arduino Properties

CPU: Uno

CPU Speed (MHz): 16

Multiple of Crystal Freq:

Virtual Comport (COM):

Arduino Uno (COM4)

Control Clk Src: 16 bit timer 1

Control Clk Prescale: 1

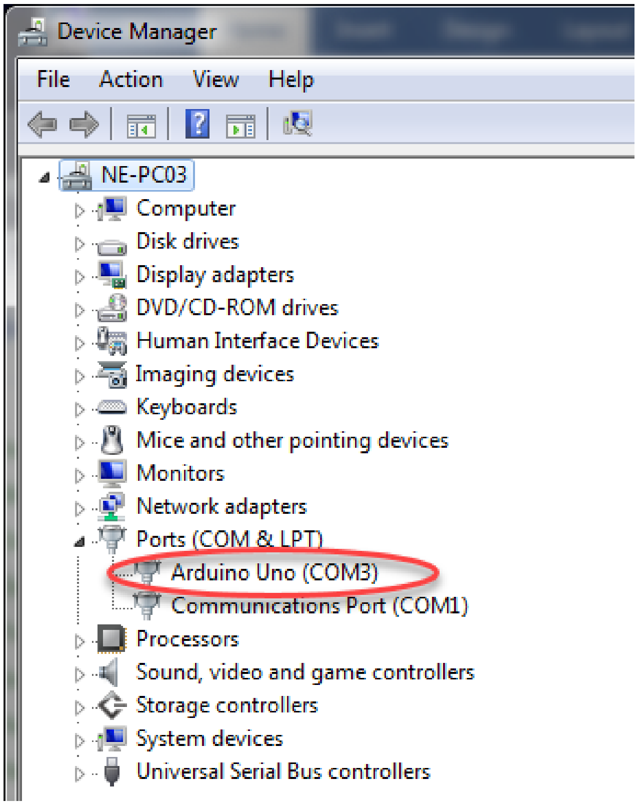
Ctrl Clk Count Mode: Up

Boot Init Code

DLL/VXD Version:
Altair Embed support for Arduino v160 Build 2670

3. Under **Virtual Comport**, select the serial port number for your Arduino.

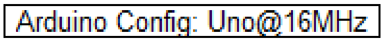
   Note: If you do not know the number, click **Start > Control Panel > Device Manager** and then scroll down and click on **Ports** to find it.



Device Manager

NE-PC03
- Computer
- Disk drives
- Display adapters
- DVD/CD-ROM drives
- Human Interface Devices
- Imaging devices
- Keyboards
- Mice and other pointing devices
- Monitors
- Network adapters
- Ports (COM & LPT)
  - Arduino Uno (COM3)
  - Communications Port (COM1)
- Processors
- Sound, video and game controllers
- Storage controllers
- System devices
- Universal Serial Bus controllers

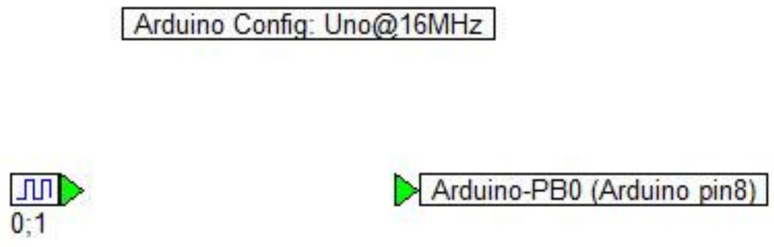
4. The remaining parameters in the Arduino Properties dialog are already correctly set; just click **OK**.
5. Move the pointer to the work area and click to insert the **Arduino Config** block into your diagram.



Arduino Config: Uno@16MHz

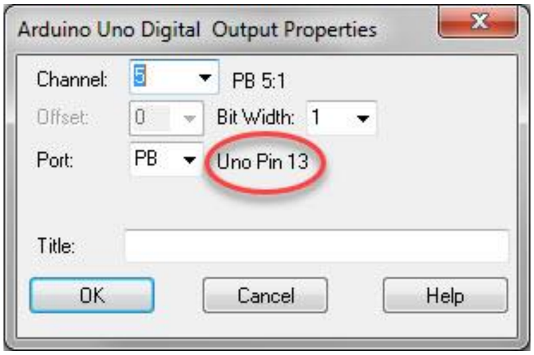
Inserting blocks

To generate a signal that goes from 0 to 1 and back again and connect it to the Arduino Uno, you insert a **squareWave** block (**Blocks > Signal Producer**) and a **Digital Output for Arduino** block (**Embedded > Arduino > Digital I/O**) into your diagram, as shown below. If you need a refresher on inserting blocks, click here.



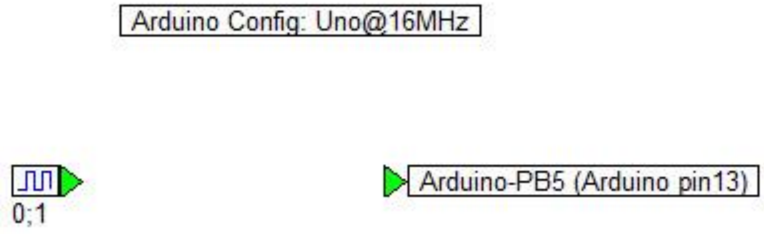
Setting block properties

On the Arduino Uno, the built-in LED is connected to digital pin 13, which is connected to Port B, channel 5, as shown in the Arduino Uno pin mapping schematic. To connect the **Digital Output for Arduino** block to pin 13, make the dialog selections shown below. If you need a refresher on setting block properties, click here.


Arduino Uno Digital Output Properties

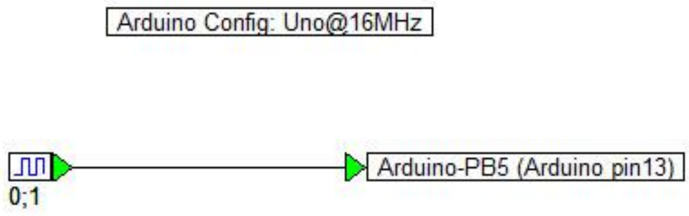
Channel: 5 PB 5:1
Offset: 0 Bit Width: 1
Port: PB Uno Pin 13
Title:

Your diagram will look like this:



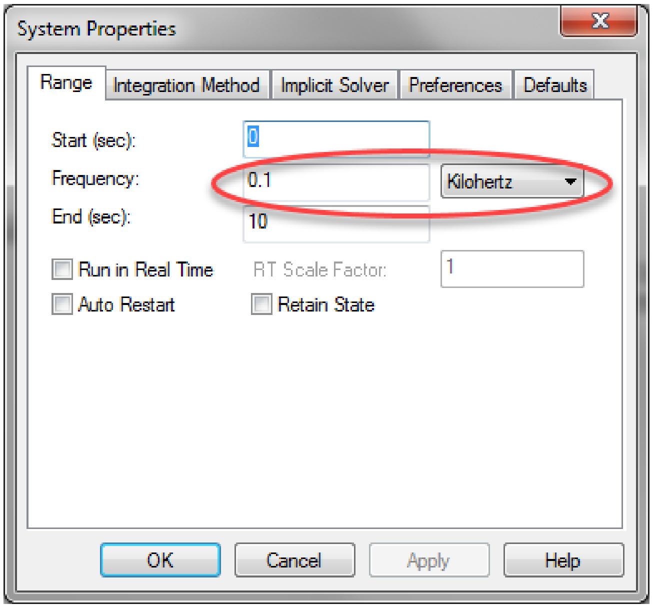
Connecting blocks

Connect the **squareWave** block to the **Digital Output for Arduino** block, as shown below. If you need a refresher on connecting blocks, click here.



Setting simulation properties

In order for the diagram to run at 100Hz, set the simulation frequency to 0.1KHz, as shown below. If you need a refresher on setting simulation properties, click here.


System Properties

Start (sec): 0

Frequency: 0.1 Kilohertz

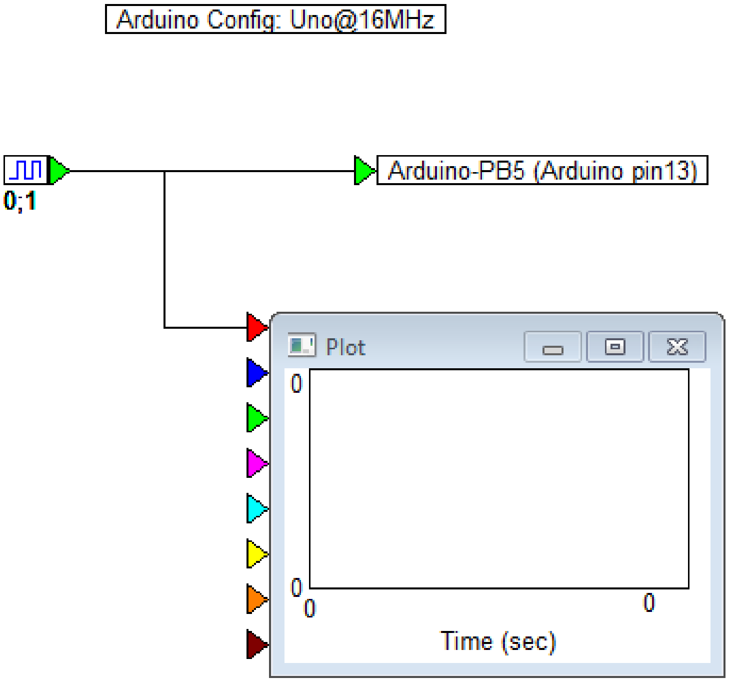
End (sec): 10

□ Run in Real Time RT Scale Factor: 1

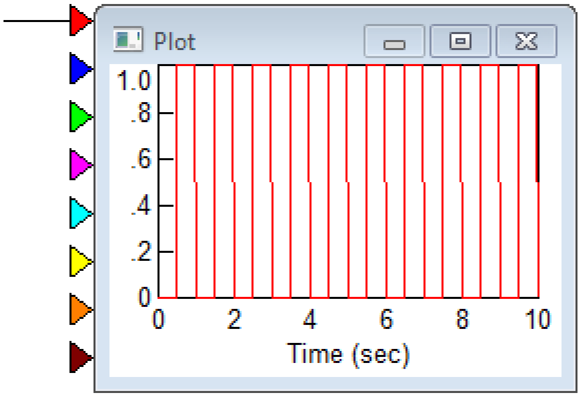
□ Auto Restart □ Retain State

Confirming the signal frequency

Although this diagram is very simple, it is good practice to wire a Signal Consumer block, like a **plot** block, into your diagram to check that the signals you are producing are what you expect.



When you start the simulation, the plot trace shows that the signal correctly cycles between 0 and 1 in one second intervals.



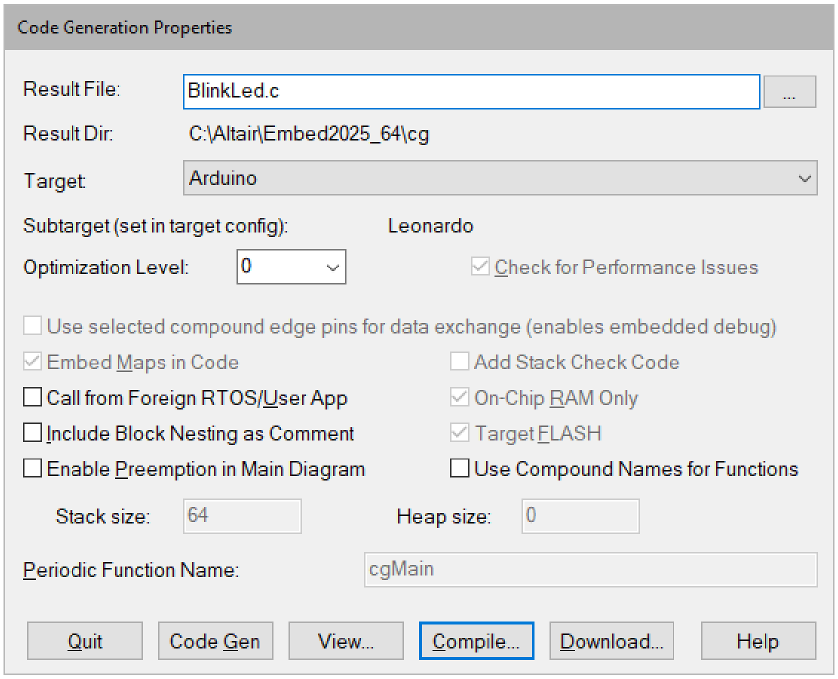
Save your diagram, if you have not already done so.

Compiling and linking your code

You are now ready to generate code to run on the Arduino Uno.

1. Click **Tools > Code Gen**.

   The Code Gen dialog appears.



Code Generation Properties

Result File: BlinkLed.c

Result Dir: C:\Altair\Embed2025_64\cg

Target: Arduino

Subtarget (set in target config): Leonardo

Optimization Level: 0

☑ Check for Performance Issues

□ Use selected compound edge pins for data exchange (enables embedded debug)

☑ Embed Maps in Code

□ Add Stack Check Code

□ Call from Foreign RTOS/User App

☑ On-Chip RAM Only

□ Include Block Nesting as Comment

☑ Target FLASH

□ Enable Preemption in Main Diagram

□ Use Compound Names for Functions

Stack size: 64

Heap size: 0

Periodic Function Name: cgMain

   This dialog provides, among other things, the following information:

   - **Result File:** The name of the generated C file. By default, Embed uses the name of your diagram.
   - **Result Dir:** The name of the directory in which the C file will be placed.
   - **Target:** The target device.
   - **Subtarget:** The CPU that you selected when you configured the diagram.

   For this example, you can ignore the other parameters in the dialog.

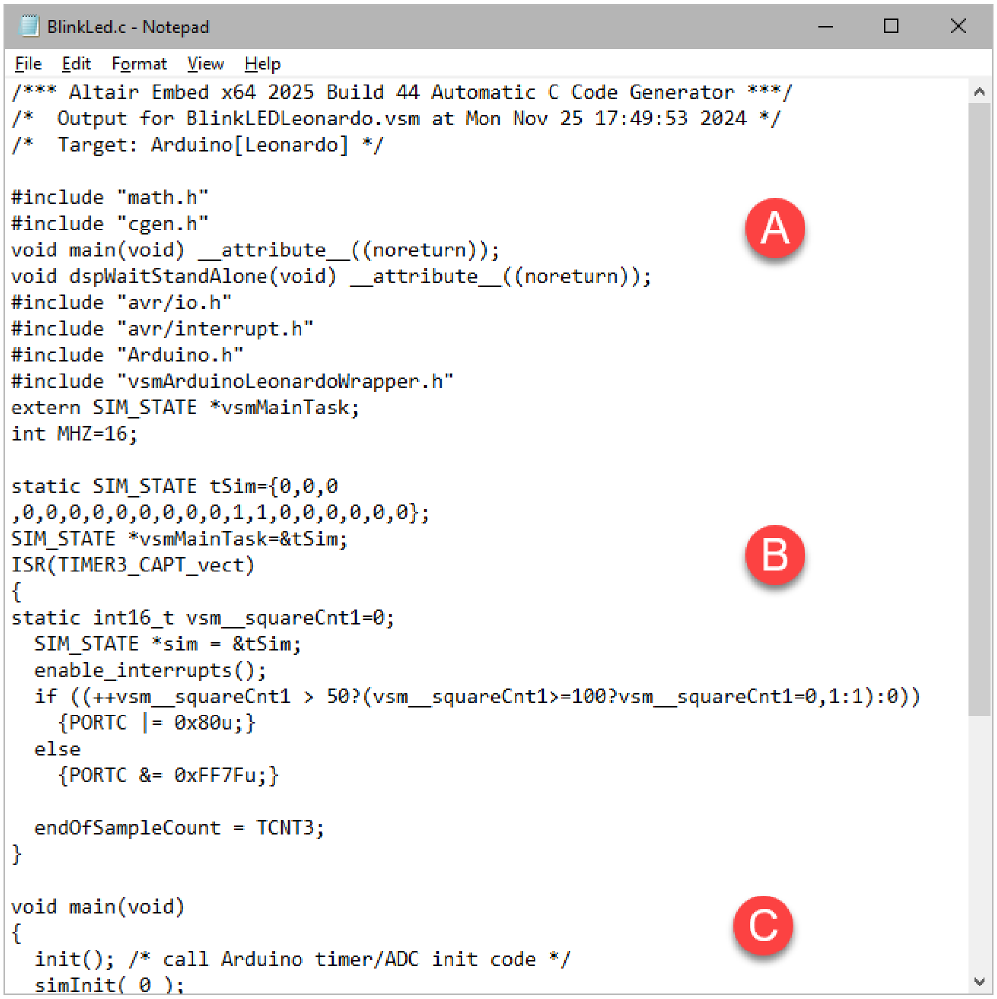
2. Click **Compile**.

   The following occurs:

   - A **BlinkLed.C** file is generated.

- The target compiler generates a **BlinkLed.ELF** file (the target executable).

3. To examine the BlinkLed.C file, click **View** in the Code Gen dialog.



```
/*** Altair Embed x64 2025 Build 44 Automatic C Code Generator ***/
/*  Output for BlinkLEDLeonardo.vsm at Mon Nov 25 17:49:53 2024 */
/*  Target: Arduino[Leonardo] */

#include "math.h"
#include "cgen.h"
void main(void) __attribute__((noreturn));
void dspWaitStandAlone(void) __attribute__((noreturn));
#include "avr/io.h"
#include "avr/interrupt.h"
#include "Arduino.h"
#include "vsmArduinoLeonardoWrapper.h"
extern SIM_STATE *vsmMainTask;
int MHZ=16;

static SIM_STATE tSim={0,0,0
,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0};
SIM_STATE *vsmMainTask=&tSim;
ISR(TIMER3_CAPT_vect)
{
static int16_t vsm__squareCnt1=0;
  SIM_STATE *sim = &tSim;
  enable_interrupts();
  if ((++vsm__squareCnt1 > 50?(vsm__squareCnt1>=100?vsm__squareCnt1=0,1:1):0))
    {PORTC |= 0x80u;}
  else
    {PORTC &= 0xFF7Fu;}

  endOfSampleCount = TCNT3;
}

void main(void)
{
  init(); /* call Arduino timer/ADC init code */
  simInit( 0 );
```

The code is separated into three sections:

A Includes the necessary header files for the code to run on the Arduino Uno.

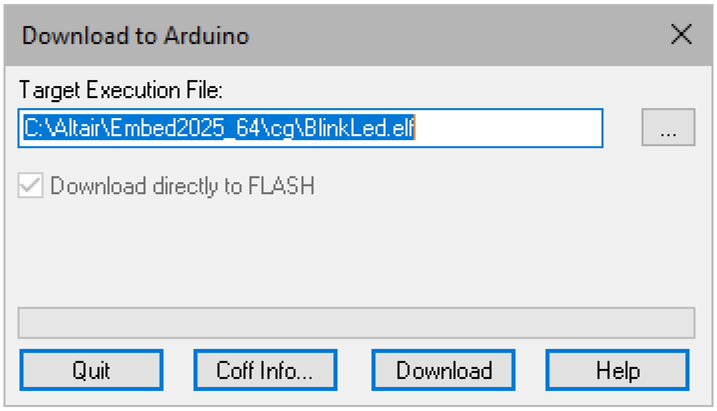
B Sets the target interface to run at the rate specified in the System Properties dialog, which is 100Hz, and creates 1Hz blink (50 counts ON and 50 counts OFF).

C Generates interrupts at a 100Hz rate.

Downloading and executing the code on the Arduino Uno

1. To download BlinkLed.ELF to the Arduino Uno, click **Download** in the Code Gen dialog.

   The Download to Arduino dialog appears.



Download to Arduino

Target Execution File: C:\Altair\Embed2025_64\cg\BlinkLed.elf

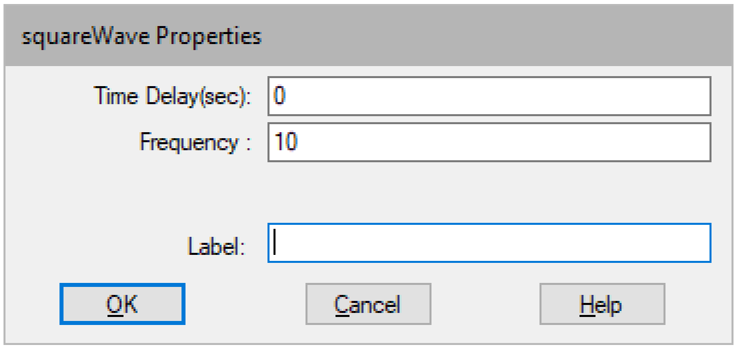
☑ Download directly to FLASH

2. Click **Download**.

The built-in LED on the Arduino Uno starts blinking at one second intervals.

Changing the blink frequency

You can easily change the blink rate by right-clicking the **squareWave** block and editing the **Frequency** parameter.



squareWave Properties

Time Delay(sec): 0

Frequency : 10

Label:

In this case, the Frequency has been set to 10Hz. After you save the diagram, and re-compile and download the code to the Arduino, the built-in LED blinks at a more rapid rate.

Arduino: Using serial monitor to debug code

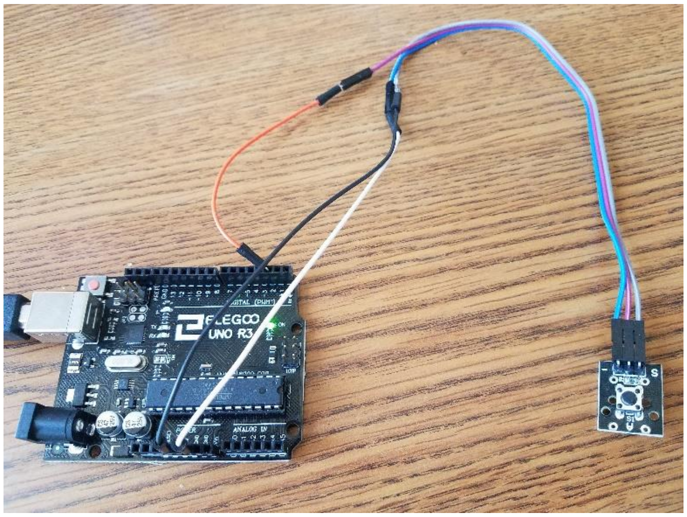
This example demonstrates how to use the Arduino IDE serial monitor to debug a defective diagram using the serial monitor in the Arduino IDE. The diagram is supposed to generate code that — when loaded onto an Arduino Uno R3 — causes the built-in LED to blink when a pushbutton sensor attached to the Uno is pressed.

What you'll need

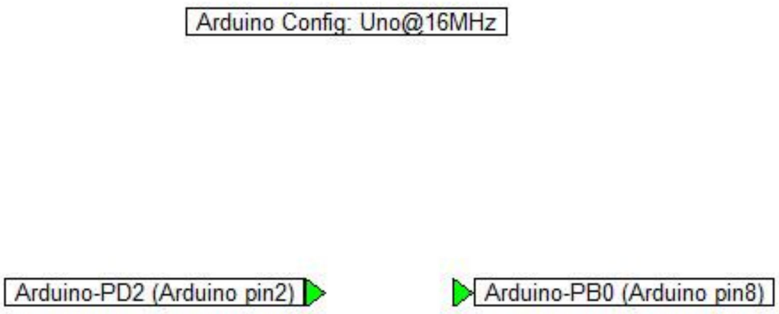
Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Arduino Uno	https://store.arduino.cc/usa/arduino-uno-rev3

Configuring the hardware and the diagram

1. Set up the hardware by attaching the pushbutton sensor to the GRND, 3.3 V and digital input 2 pins on the Arduino Uno R3.



2. Attach the Uno to your computer with a USB cable.
3. Start **Embed** and click **File > New**.
4. Save the diagram as **BlinkLEDwithPushButton.vsm**.
5. Add an **Arduino Config** block, **Digital Input for Arduino** block, and **Digital Output for Arduino** block to your diagram.


Arduino Config: Uno@16MHz
Arduino-PD2 (Arduino pin2)
Arduino-PB0 (Arduino pin8)

6. Wire the **Digital Input for Arduino** block into the **Digital Output for Arduino** block and make sure:
   - The **Arduino Config** is set to the proper **COMM port**.
   - The **Digital Input for Arduino** is set to **channel 2** and **port PD**.
   - The **Digital Output for Arduino** is set to **channel 2** and **port PB**.

7. Generate code to run on the Arduino.

8. After the code has been downloadeded to the Arduino Uno R3, click the **pushbutton** on the sensor. The built-in LED fails to respond when pushing the sensor button.

The next several sections step you through how to debug the code using the Arduino serial monitor.

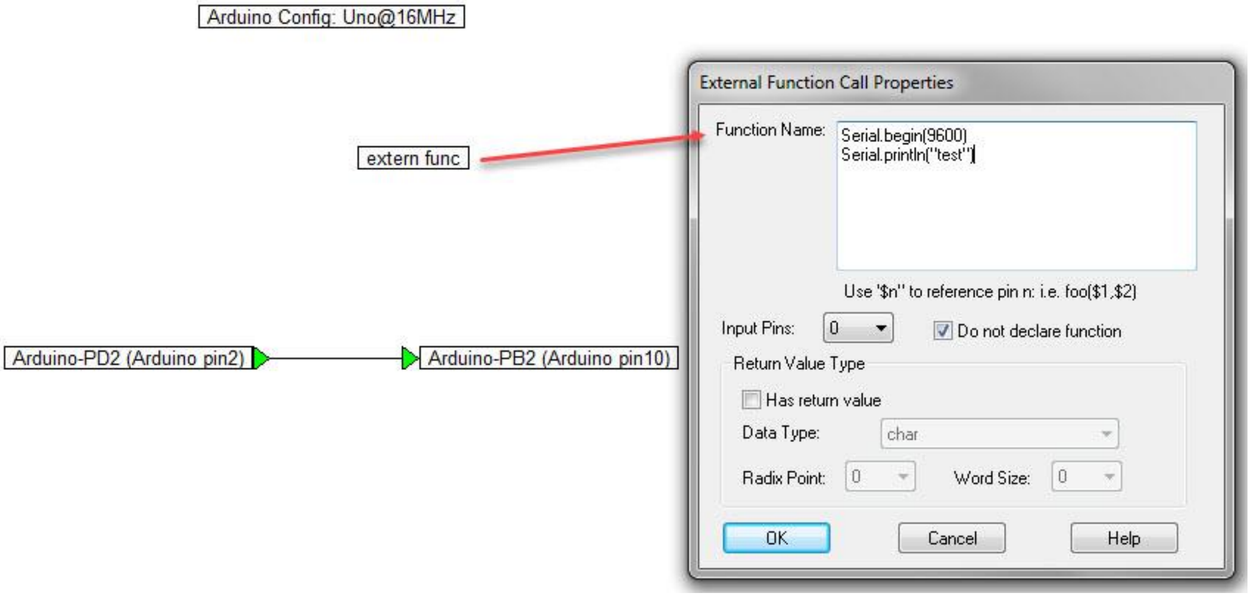
Confirming that data can be printed to the serial monitor

1. Go to **C:\Program Files (x86) > Arduino** and click **Arduino.exe**.

2. Return to the **BlinkLEDwithPushButton** diagram.

3. Check if data can be printed o the serial monitor by doing the following:

   a. Add an **Extern Function** block to the diagram and call the functions **Serial.begin(9600); Serial.println(“test”)**.


Arduino Config: Uno@16MHz
extern func
Arduino-PD2 (Arduino pin2)
Arduino-PB2 (Arduino pin10)
External Function Call Properties
OK
Cancel
Help

Function Name:
```
Serial.begin(9600)
Serial.println("test")
```
Use '$n" to reference pin n: i.e. foo($1,$2)

Input Pins: 0

☑ Do not declare function

Return Value Type

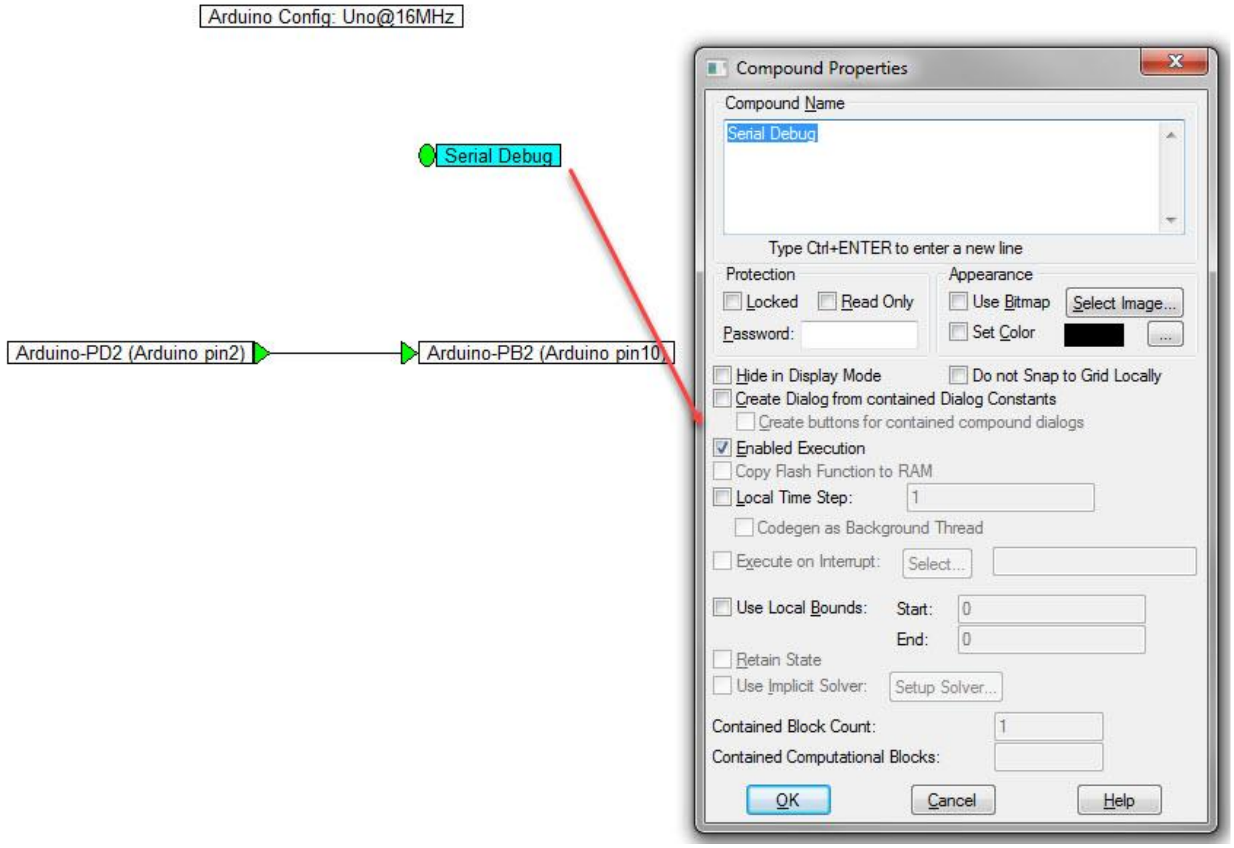
□ Has return value

Data Type: char

Radix Point: 0

Word Size: 0

   b. Encapsulate the **Extern Function block** in a compound block named **Serial Debug** and activate **Enabled Execution**.



Compound Name

Serial Debug

Type Ctrl+ENTER to enter a new line

Protection

□ Locked □ Read Only

Password:

Appearance

□ Use Bitmap

□ Set Color

□ Hide in Display Mode □ Do not Snap to Grid Locally

□ Create Dialog from contained Dialog Constants

□ Create buttons for contained compound dialogs

☑ Enabled Execution

□ Copy Flash Function to RAM

□ Local Time Step: 1

□ Codegen as Background Thread

□ Execute on Interrupt:

□ Use Local Bounds: Start: 0 End: 0

□ Retain State

□ Use Implicit Solver:

Contained Block Count: 1

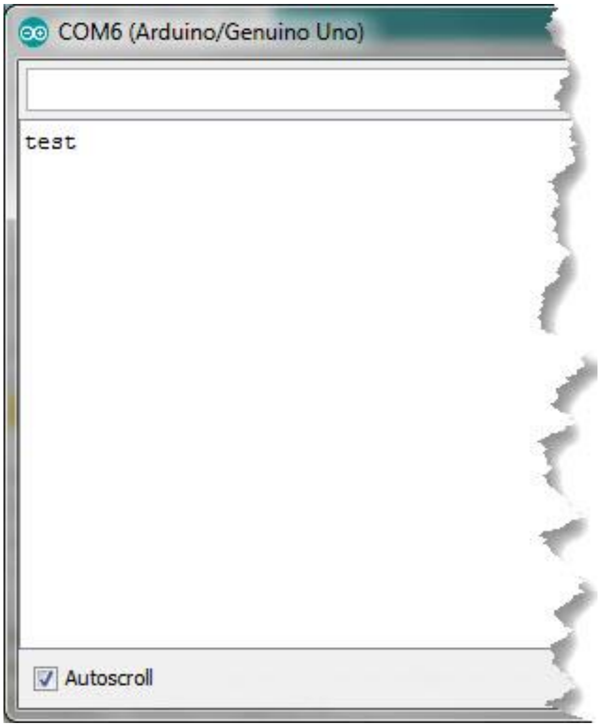
Contained Computational Blocks:

c. Wire a **variable** block set to **$firstPass** into the **Serial Debug** block.

4. Generate code to run on the Arduino.

5. Switch to the **Arduino IDE** and click **Tools > Serial Monitor**.

   The monitor window displays the word *test*, which shows that data communication is working.



COM6 (Arduino/Genuino Uno)

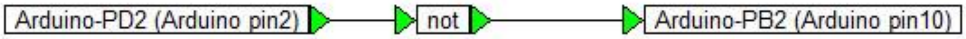
```
test
```

☑ Autoscroll

6. Close the **serial monitor**.

Confirming that the pushbutton is working

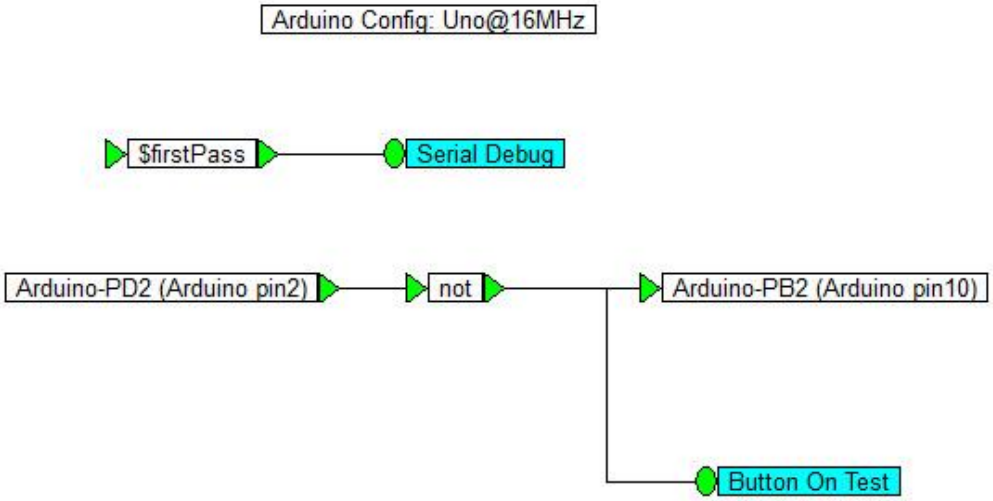
1. Wire a **Boolean not** block into the diagram:



2. Add an **Extern Function** block that calls the function **Serial.println("ON")**.

3. Encapsulate the **Extern Function** block in a compound block named **Button On Test** and activate **Enabled Execution**.

4. Wire the **Boolean not** to the **Button On Test** block.

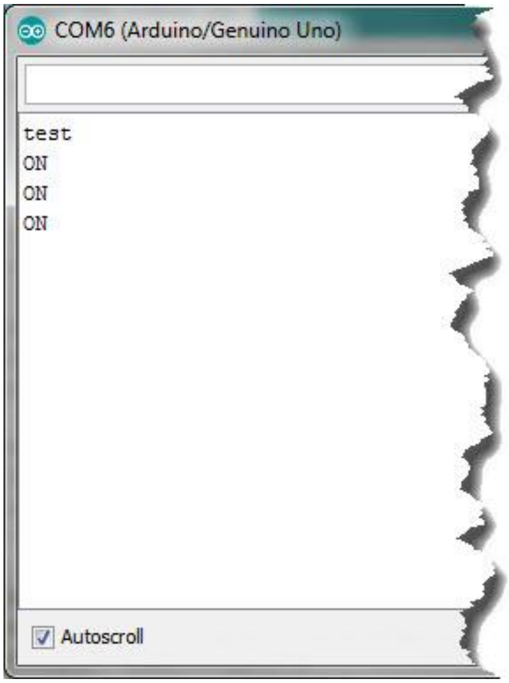

Arduino Config: Uno@16MHz
$firstPass
Serial Debug
Arduino-PD2 (Arduino pin2)
not
Arduino-PB2 (Arduino pin10)
Button On Test

5. Generate code to run on the Arduino

6. Switch to the **Arduino IDE** and click **Tools > Serial Monitor**.

7. Press the **pushbutton** on the sensor.

   The word *ON* is displayed in the serial monitor after each press, which confirms that the pushbutton is working correctly.



```
test
ON
ON
ON
```

8. Close the **serial monitor**.

Checking diagram parameters

1. Right-click each block and check that the parameter values are set correctly.

   The Channel parameter for the **Digital Output for Arduino** block is incorrectly set to **2**. Set it to **5**, which corresponds to **Uno Pin 13** (the built-in LED).

2. Generate code to run on the Arduino.

3. Press the **pushbutton** on the sensor.

The built-in LED now blinks each time you press the pushbutton.

Arduino: Importing an Arduino library that displays text on an Adafruit SSD1306

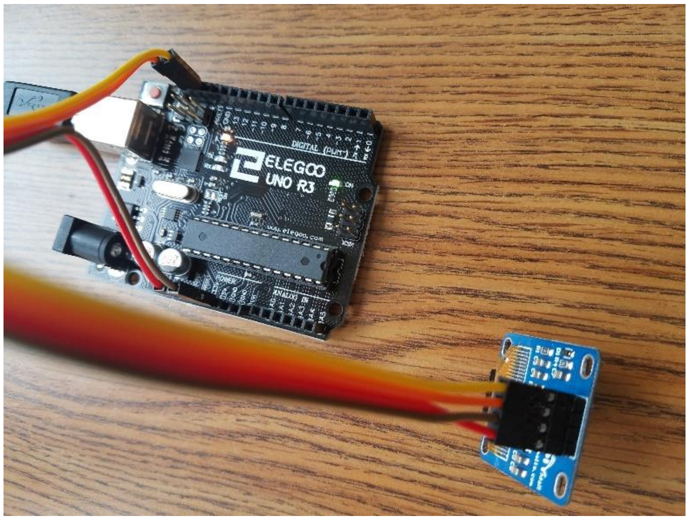
This example describes how to use the Adafruit SSD1306 driver along with the Adafruit GFX general-purpose graphics software to print “Hello, world!” on the SSD1306 on an Arduino Uno coupled with a 128x32-bit display connected via I2C. You can easily modify the steps for a 64-bit display or SPI connection.

What you’ll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Arduino Uno	https://store.arduino.cc/usa/arduino-uno-rev3
Adafruit SSD1306	https://www.adafruit.com/product/326
Adafruit GFX general-purpose graphics software	https://github.com/adafruit/Adafruit-GFX-Library

Setting up the Arduino Uno

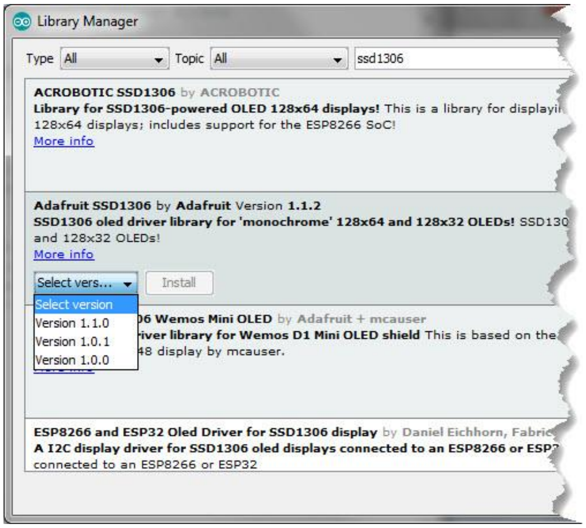
1. Attach the SSD1306 OLED to the Arduino Uno board as shown below. For wiring instructions, go to https://learn.adafruit.com/monochrome-oled-breakouts/wiring-128x32-spi-oled-display.



2. Start the Arduino IDE.

3. Click **Sketch > Include Library > Manage Libraries** and do the following:

   a. In the **Search box,** enter **ssd1306**.



Library Manager

Type All Topic All ssd1306

ACROBOTIC SSD1306 by ACROBOTIC
Library for SSD1306-powered OLED 128x64 displays! This is a library for displayi
128x64 displays; includes support for the ESP8266 SoC!
More info

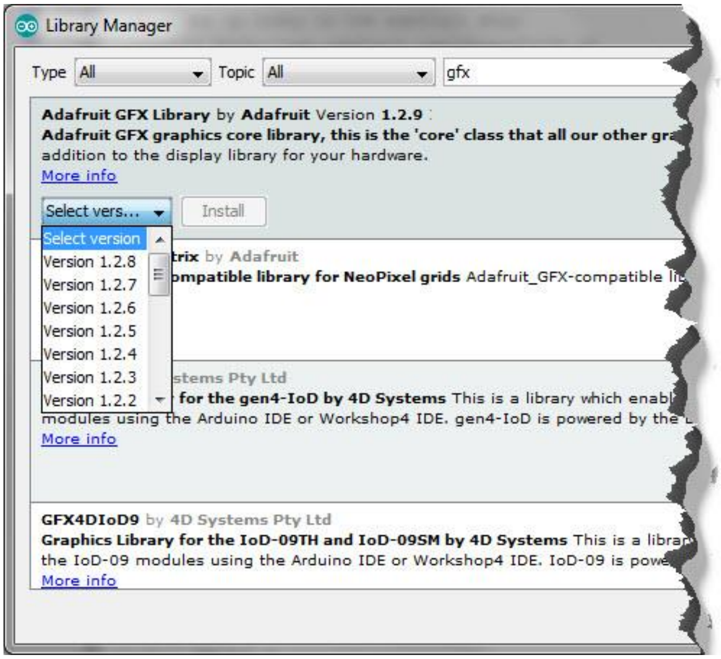
Adafruit SSD1306 by Adafruit Version 1.1.2
SSD1306 oled driver library for 'monochrome' 128x64 and 128x32 OLEDs! SSD130
and 128x32 OLEDs!
More info

)6 Wemos Mini OLED by Adafruit + mcauser
river library for Wemos D1 Mini OLED shield This is based on the
48 display by mcauser.

ESP8266 and ESP32 Oled Driver for SSD1306 display by Daniel Eichhorn, Fabric
A I2C display driver for SSD1306 oled displays connected to an ESP8266 or ESP
connected to an ESP8266 or ESP32

b. Under **Adafruit SSD1306**, select the most recent version and click **Install**.

c. Repeat steps a - b but this time search for and install the most recent version of **Adafruit GFX library**.



Library Manager

Type All Topic All gfx

Adafruit GFX Library by Adafruit Version 1.2.9
Adafruit GFX graphics core library, this is the 'core' class that all our other gra
addition to the display library for your hardware.
More info

trix by Adafruit
ompatible library for NeoPixel grids Adafruit_GFX-compatible li

stems Pty Ltd
for the gen4-IoD by 4D Systems This is a library which enab
modules using the Arduino IDE or Workshop4 IDE. gen4-IoD is powered by the
More info

GFX4DIoD9 by 4D Systems Pty Ltd
Graphics Library for the IoD-09TH and IoD-09SM by 4D Systems This is a librar
the IoD-09 modules using the Arduino IDE or Workshop4 IDE. IoD-09 is powe
More info

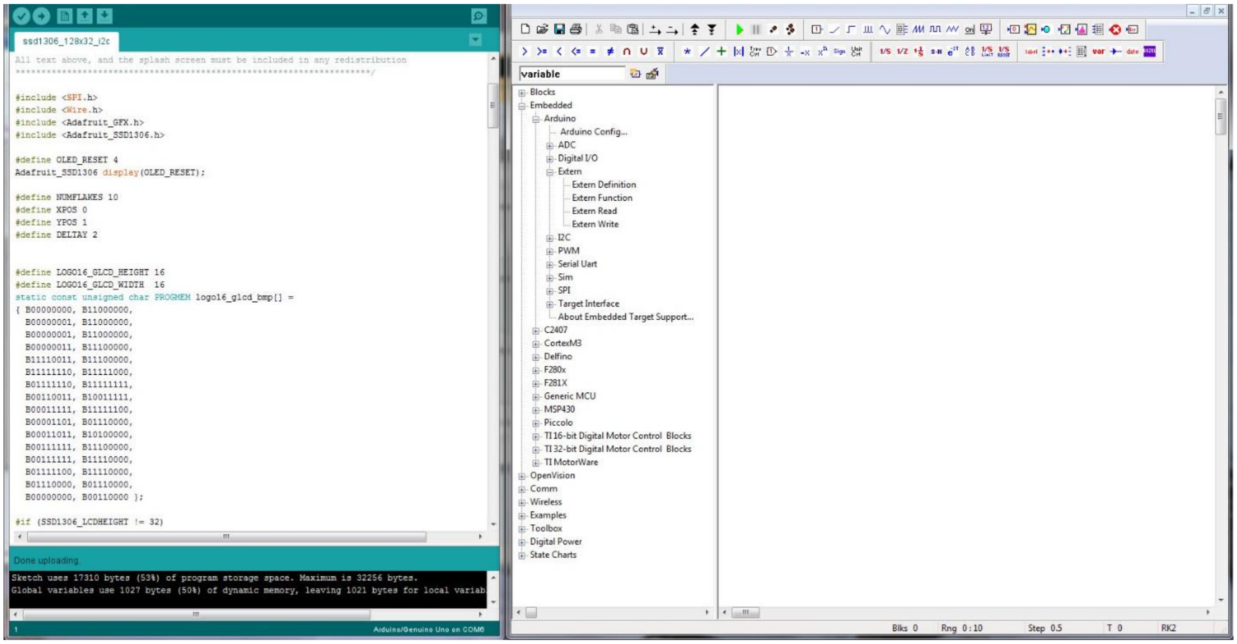
4. Click **File > Open > Examples > ssd1306_128x32_i2c** and select **ssd1306_128x32_i2c.ino**.

5. To verify that the library modules have been installed correctly, compile the code from the Arduino IDE by clicking the **checkmark** in the upper left corner of the Arduino window.

6. To verify that the hardware is connected properly and works as expected, click the **right arrow** in the upper left corner of the Arduino window to upload and run the code on your Arduino.

Setting up your Embed diagram and importing the Adafruit libraries

1. Start Embed and position it next to the Arduino window.



```
All text above, and the splash screen must be included in any redistribution
*********************************************************************/

#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RESET);

#define NUMFLAKES 10
#define XPOS 0
#define YPOS 1
#define DELTAY 2


#define LOGO16_GLCD_HEIGHT 16
#define LOGO16_GLCD_WIDTH  16
static const unsigned char PROGMEM logo16_glcd_bmp[] =
{ B00000000, B11000000,
  B00000001, B11000000,
  B00000001, B11000000,
  B00000011, B11100000,
  B11110011, B11100000,
  B11111110, B11111000,
  B01111110, B11111111,
  B00110011, B10011111,
  B00011111, B11111100,
  B00001101, B01110000,
  B00011011, B10100000,
  B00111111, B11100000,
  B00111111, B11110000,
  B01111100, B11110000,
  B01110000, B01110000,
  B00000000, B00110000 };

#if (SSD1306_LCDHEIGHT != 32)
```

Done uploading.

```
Sketch uses 17310 bytes (53%) of program storage space. Maximum is 32256 bytes.
Global variables use 1027 bytes (50%) of dynamic memory, leaving 1021 bytes for local variab
```

1

Arduino/Genuino Uno on COM6

2. Create a new diagram and save it as **OLED2.vsm**.

3. Insert the following blocks into your diagram:

   - **Embedded > Arduino** > **Arduino Config**block. Make sure it is configured for an **Uno** and the **Comm port** is set correctly.

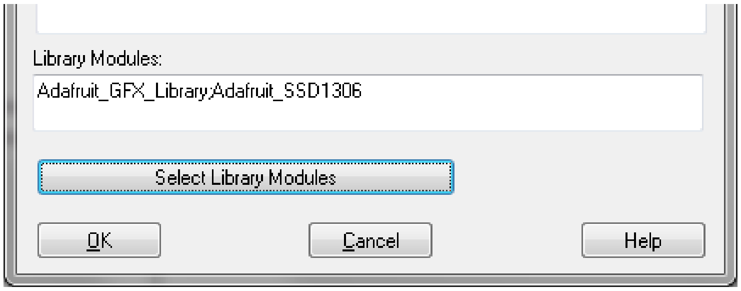
   - **Embedded > Arduino > Extern** > **Extern Definition**block in your diagram.

4. Right-click the **Extern Definition** block to access its Properties dialog.

5. Click **Select Library Modules**.

6. In the **External Library Selection** dialog, select **Adafruit_SSD1306** and **Adafruit_GFX_Library**, then click **OK**.

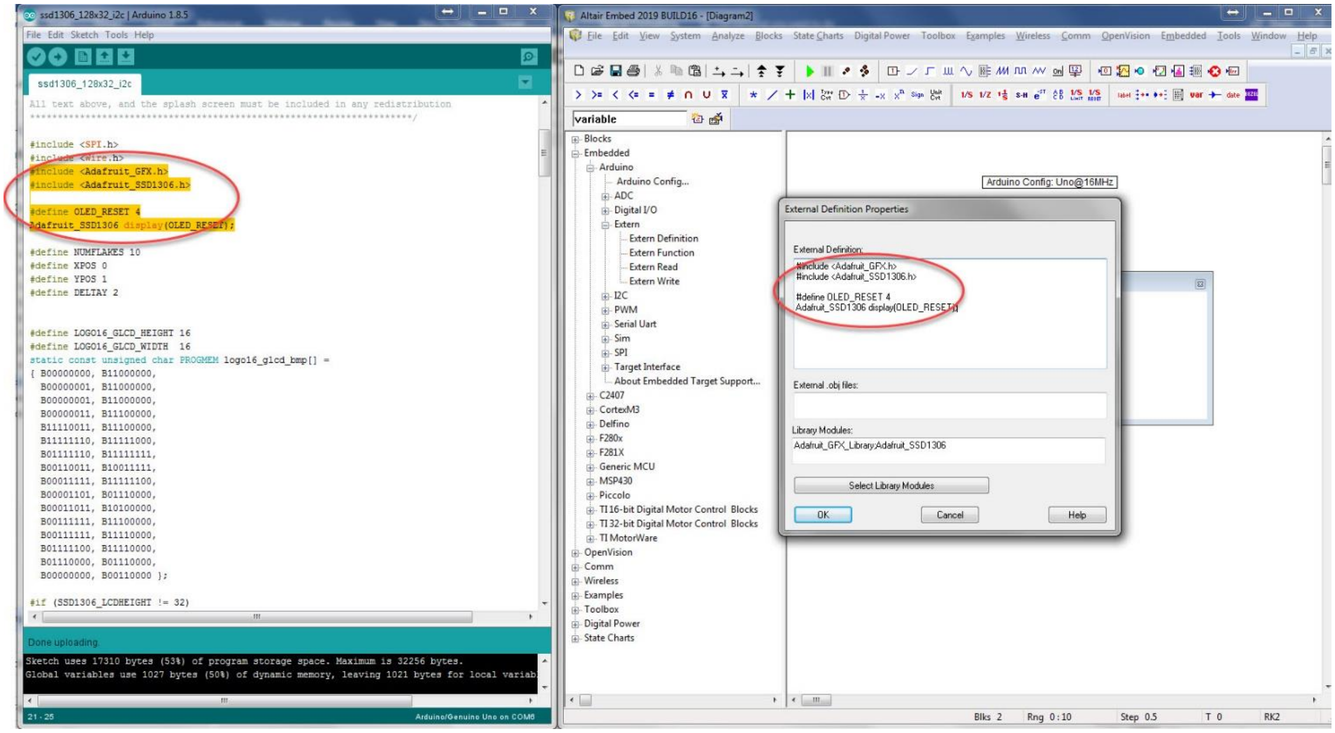
7. The **Extern Definition** dialog displays the selected libraries under **Library Modules**.



Library Modules:

Adafruit_GFX_Library;Adafruit_SSD1306

8. With the Arduino window and Embed window side-by-side, copy the **#include**, **#define**, and **instantiation declarations** from the Arduino sketch into the External Definition window.


ssd1306_128x32_i2c | Arduino 1.8.5
File Edit Sketch Tools Help
ssd1306_128x32_i2c
Altair Embed 2019 BUILD16 - [Diagram2]
File Edit View System Analyze Blocks State Charts Digital Power Toolbox Examples Wireless Comm OpenVision Embedded Tools Window Help
variable
Blocks
Embedded
Arduino
Arduino Config...
ADC
Digital I/O
Extern
Extern Definition
Extern Function
Extern Read
Extern Write
I2C
PWM
Serial Uart
Sim
SPI
Target Interface
About Embedded Target Support...
C2407
CortexM3
Delfino
F280x
F281X
Generic MCU
MSP430
Piccolo
TI 16-bit Digital Motor Control Blocks
TI 32-bit Digital Motor Control Blocks
TI MotorWare
OpenVision
Comm
Wireless
Examples
Toolbox
Digital Power
State Charts
Arduino Config: Uno@16MHz
External Definition Properties
Select Library Modules
OK
Cancel
Help
Blks 2
Rng 0 : 10
Step 0.5
T 0
RK2

```
All text above, and the splash screen must be included in any redistribution
*********************************************************************/

#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RESET);

#define NUMFLAKES 10
#define XPOS 0
#define YPOS 1
#define DELTAY 2


#define LOGO16_GLCD_HEIGHT 16
#define LOGO16_GLCD_WIDTH  16
static const unsigned char PROGMEM logo16_glcd_bmp[] =
{ B00000000, B11000000,
  B00000001, B11000000,
  B00000001, B11000000,
  B00000011, B11100000,
  B11110011, B11100000,
  B11111110, B11111000,
  B01111110, B11111111,
  B00110011, B10011111,
  B00011111, B11111100,
  B00001101, B01110000,
  B00011011, B10100000,
  B00111111, B11100000,
  B00111111, B11110000,
  B01111100, B11110000,
  B01110000, B01110000,
  B00000000, B00110000 };

#if (SSD1306_LCDHEIGHT != 32)
```

Done uploading.

```
Sketch uses 17310 bytes (53%) of program storage space. Maximum is 32256 bytes.
Global variables use 1027 bytes (50%) of dynamic memory, leaving 1021 bytes for local variab
```

21 - 25 Arduino/Genuino Uno on COM6

External Definition:

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RESET);
```

External .obj files:

Library Modules: Adafruit_GFX_Library;Adafruit_SSD1306

9. In the Extern Definition window, rename the **Adafruit_GFX.h** and **Adafruit_SSD1306.h** to **Adafruit_GFX.cpp** and **Adafruit_SSD1306.cpp.** The CPP files contain all the driver logic.

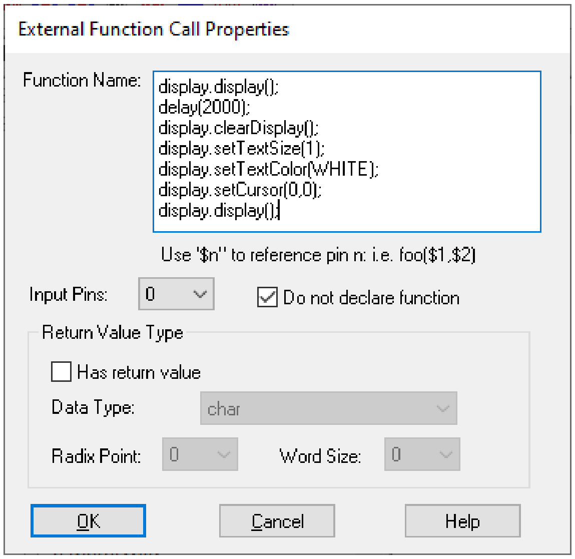
10. Click **OK**.

11. Integrate a setup loop in the diagram using the **Extern Function**block.

    a. Insert an **Extern Function** block into the diagram beneath the **Extern Definition** block.

    b. In the Arduino sketch, copy the following code into the **Extern Function** block under **Function Name**:

    ```
    display.begin(SSD1306_SWITCHCAPVCC, 0x3C);

    display.display();

    delay(2000);

    display.clearDisplay();

    display.setTextSize(1);

    display.setTextColor(WHITE);

    display.setCursor(0,0);

    display.display();
    ```



External Function Call Properties

Function Name:

```
display.display();
delay(2000);
display.clearDisplay();
display.setTextSize(1);
display.setTextColor(WHITE);
display.setCursor(0,0);
display.display();
```

Use '$n'' to reference pin n: i.e. foo($1,$2)

Input Pins: 0

☑ Do not declare function

Return Value Type

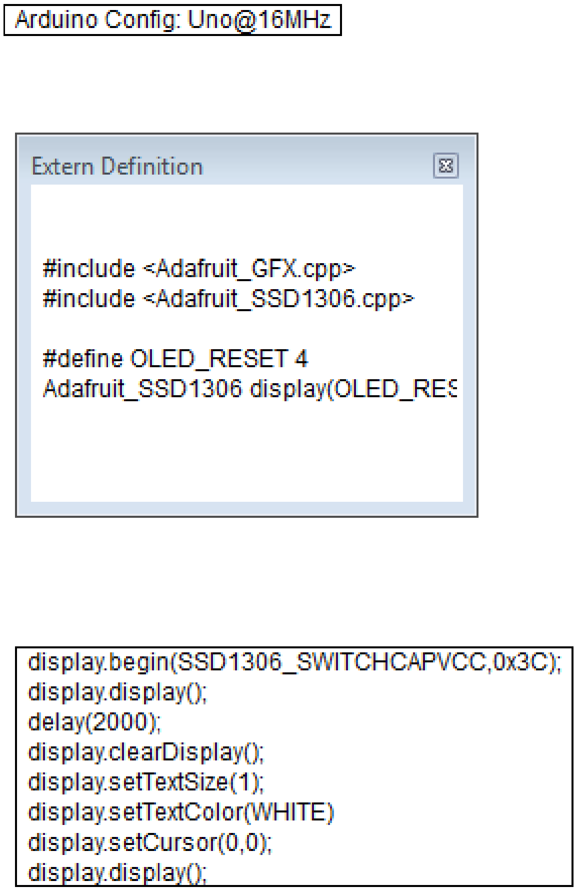
☐ Has return value

Data Type: char

Radix Point: 0

Word Size: 0

Your diagram will look like this:



Arduino Config: Uno@16MHz

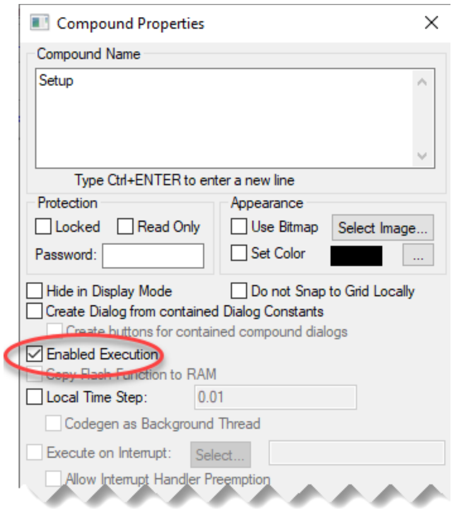
Extern Definition

```
#include <Adafruit_GFX.cpp>
#include <Adafruit_SSD1306.cpp>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RES
```

```
display.begin(SSD1306_SWITCHCAPVCC,0x3C);
display.display();
delay(2000);
display.clearDisplay();
display.setTextSize(1);
display.setTextColor(WHITE)
display.setCursor(0,0);
display.display();
```

c. Encapsulate the **Extern Function** block in a **compound block** and name it **Setup**.

d. CTRL-right-click over **Setup** and activate **Enabled Execution** and click **OK**.



Compound Properties

Compound Name

Setup

Type Ctrl+ENTER to enter a new line

Protection

☐ Locked ☐ Read Only

Password:

Appearance

☐ Use Bitmap   Select Image...

☐ Set Color

☐ Hide in Display Mode ☐ Do not Snap to Grid Locally

☐ Create Dialog from contained Dialog Constants

☐ Create buttons for contained compound dialogs

☑ Enabled Execution

☐ Copy Flash Function to RAM

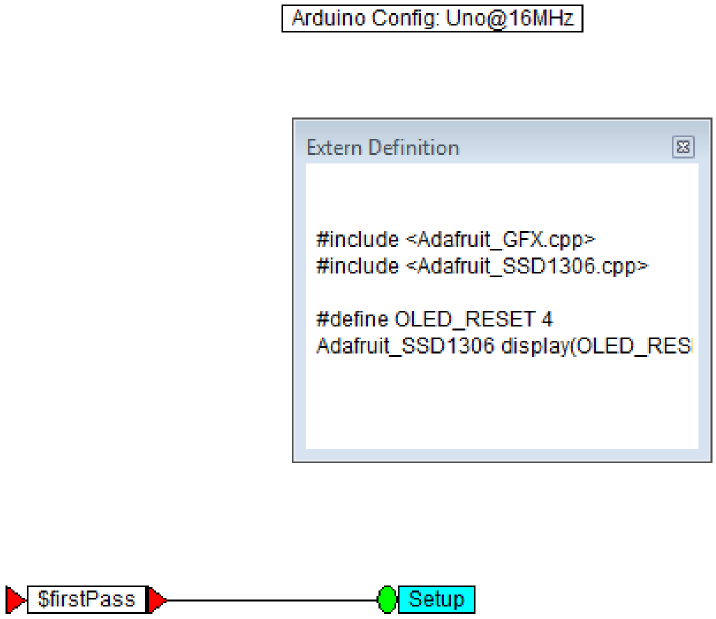
☐ Local Time Step: 0.01

☐ Codegen as Background Thread

☐ Execute on Interrupt: Select...

☐ Allow Interrupt Handler Preemption

e. Wire a **variable block** into **Setup** and set the **variable** block to **$firstPass**. Because **Setup** runs only once at boot, the **$firstPass** flag is used to control the enabled compound to run once at boot.



Arduino Config: Uno@16MHz

Extern Definition

```
#include <Adafruit_GFX.cpp>
#include <Adafruit_SSD1306.cpp>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RES
```

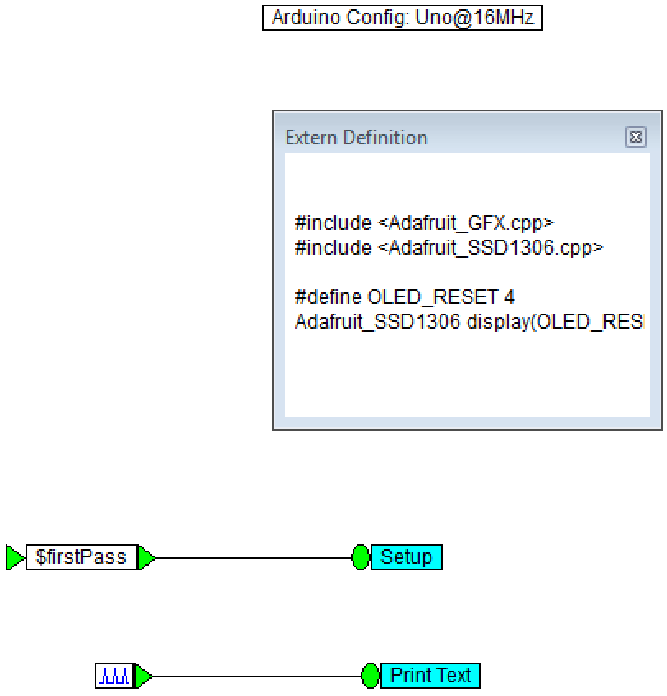
12. Add one more **Extern Function** block to the top level of your diagram.
13. In the Arduino sketch, copy the following code into the **Extern Function** block under **Function Name:**

```
Display.println(“Hello, world!”);
display.display();
```

14. Encapsulate the **Extern Function** block in a **compound block** and name it **Print Text**.
15. CTRL-right-click over **Print Text** and in the dialog, activate **Enabled Execution** and click **OK**.

16. Wire a **pulseTrain** block into **Print Text** and set the **Time Delay** to 2s and the **Time Between Pulses** to 1. The pulseTrain block sets the frequency at which to print Hello, world!.

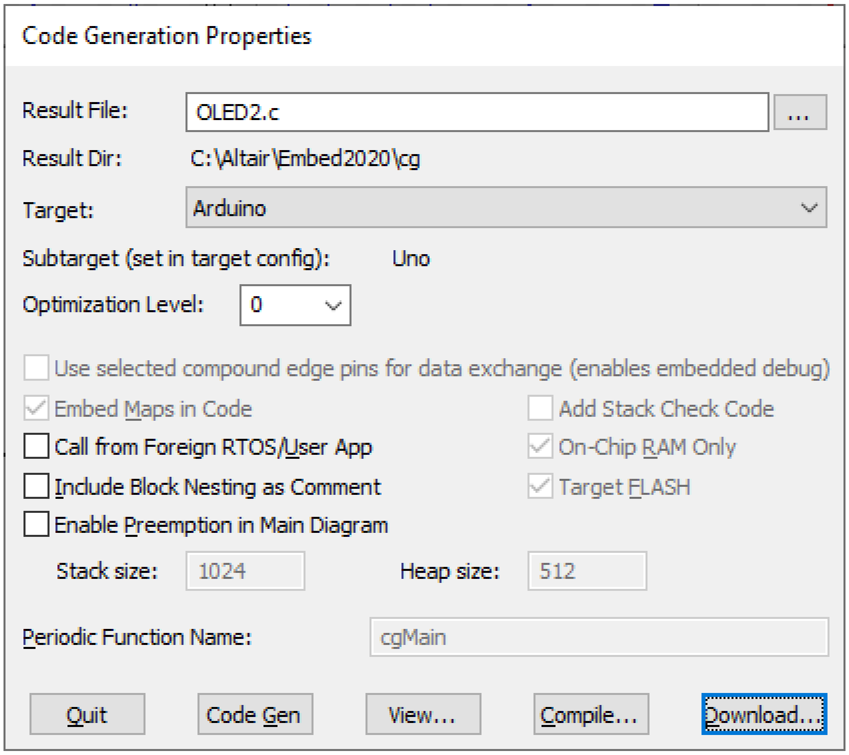
    Your diagram will look like this:



```
#include <Adafruit_GFX.cpp>
#include <Adafruit_SSD1306.cpp>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RES
```

Compiling, linking, and downloading the code to the Arduino Uno

1. To compile the code for the Arduino, click **Tools > Code Gen**.



Code Generation Properties

Result File: OLED2.c

Result Dir: C:\Altair\Embed2020\cg

Target: Arduino

Subtarget (set in target config): Uno

Optimization Level: 0

□ Use selected compound edge pins for data exchange (enables embedded debug)

☑ Embed Maps in Code

□ Add Stack Check Code

□ Call from Foreign RTOS/User App

☑ On-Chip RAM Only

□ Include Block Nesting as Comment

☑ Target FLASH

□ Enable Preemption in Main Diagram

Stack size: 1024

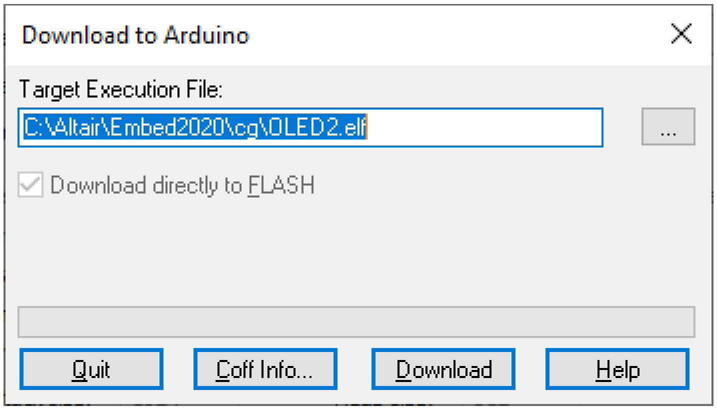
Heap size: 512

Periodic Function Name: cgMain

2. In the **Code Generation Properties** dialog, click **Compile**.

3. The code is compiled in a DOS window. When the compilation completes, click **Download** in the Code Gen dialog.

4. In the **Download to Arduino** dialog, click **Download.**



Download to Arduino

Target Execution File: C:\Altair\Embed2020\cg\OLED2.elf

☑ Download directly to FLASH

The text **Hello, world!** is displayed on the **SSD1306**.

Raspberry Pi: Controlling GPIO with an iPhone

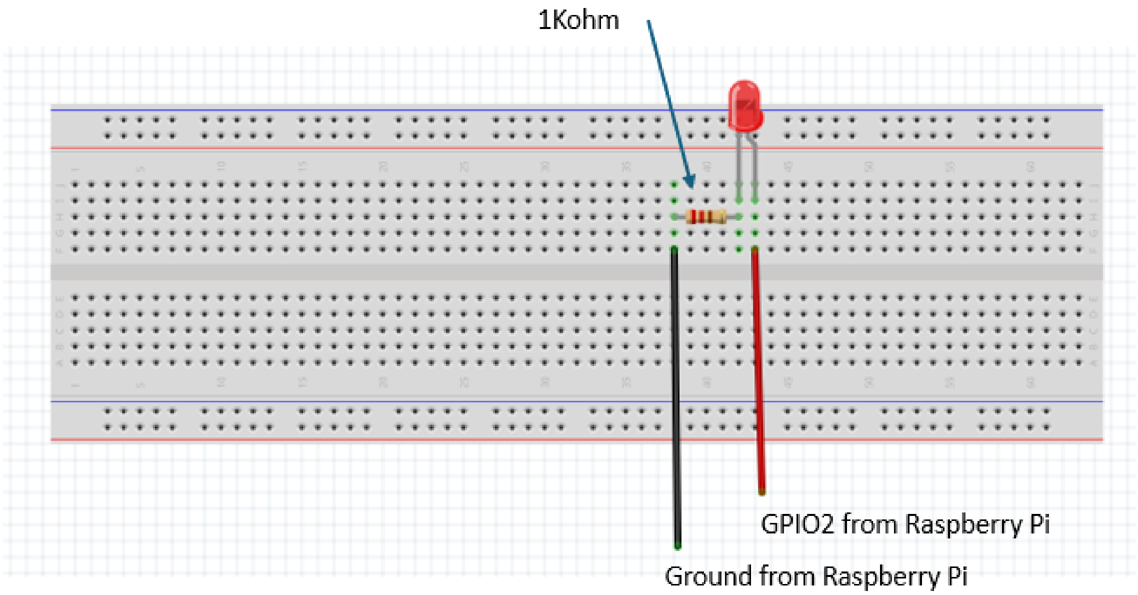
What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Raspberry Pi 4B	tbs

Setting up

On your iPhone, go to the **App Store** and install **RaspController** on your iPhone

Connect the Raspberry Pi 4B to the following LED circuit:.



Launch the **Raspberry Pi 4B** and make sure it is connected to your WiFi network.

Launch RaspController on your iPhone, make sure your iPhone is on the same WiFi network, and execute the following sequence of four steps to control the blink of the LED.

STMicroelectronics: HIL testing with an imported block from Twin Activate

The **Twin Activate HIL Example** diagram builds on the Twin Activate SIM diagram. It is a good idea to follow the procedure there before starting this example.

What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
STM32410RB	https://www.st.com/en/microcontrollers-microprocessors/stm32f410rb.html
Twin Activate	https://altair.com/twin-activate ; you only need Twin Activate if you want to open the Twin Activate diagram, you don't need it to follow along with the example

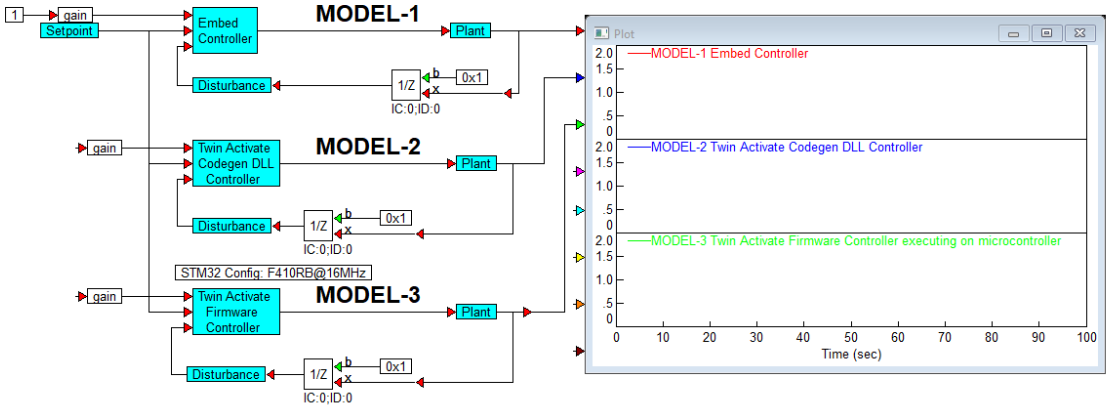
Setting up the diagram

In the HIL diagram, Embed converts the **Twin Activate Codegen DLL Controller** to firmware configured to execute on an STMicroelectronics STM32F410RB device.

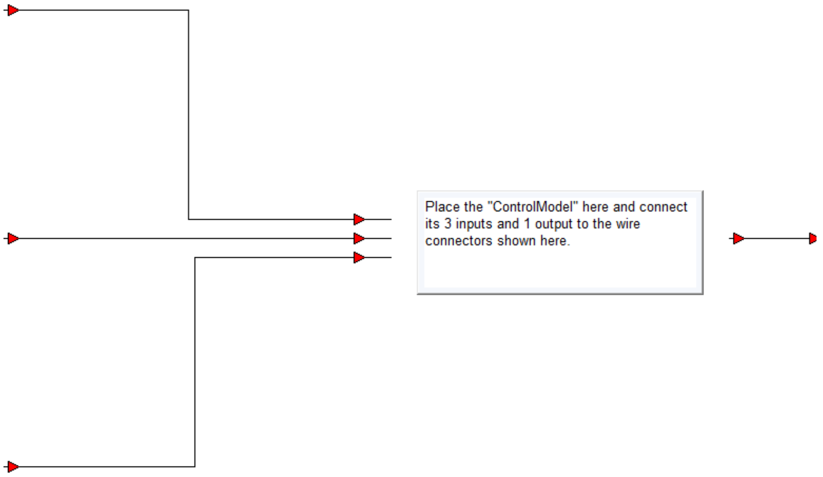
1. If you have not yet performed **steps 1a – 1e** in the Twin Activate SIM Example diagram example, do so now.

2. Add the **imported** block to the **Twin Activate HIL Example** diagram.

   a. Click **Examples > Blocks > Extensions > Imported Blocks > Twin Activate > Twin Activate HIL Example**.

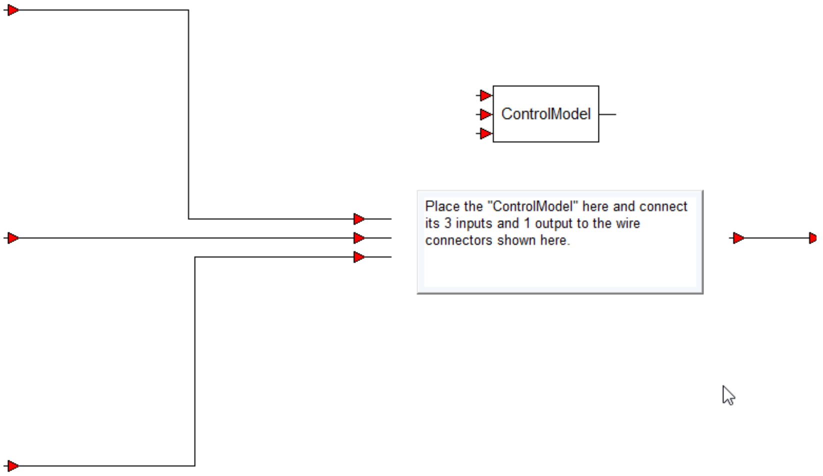


   b. Under **MODEL-2**, right-click **Twin Activate Codegen DLL Controller** compound block to dive into the next lower level of the block.



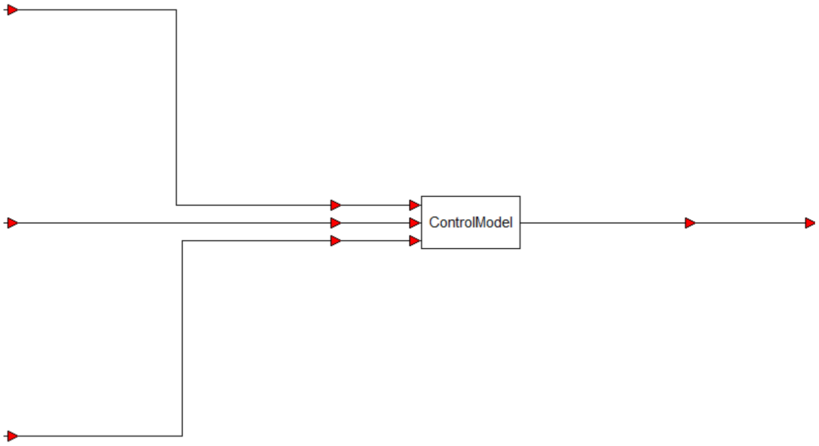
Place the "ControlModel" here and connect its 3 inputs and 1 output to the wire connectors shown here.

   c. Click **Imported Blocks > Twin Activate > ControlModel** and insert the **block** into the diagram.



Place the "ControlModel" here and connect its 3 inputs and 1 output to the wire connectors shown here.

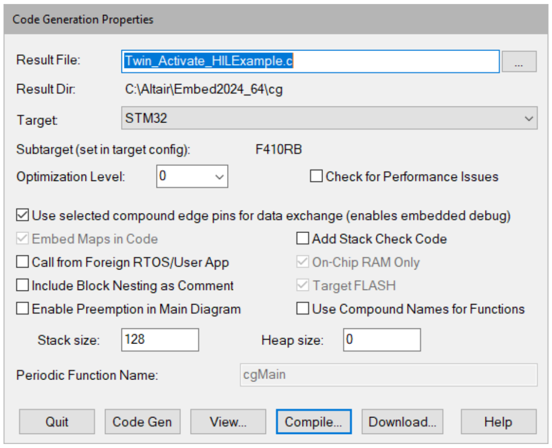
d. Replace the **comment** block with the **ControlModel** block and wire it into the diagram.



e. Right-click on empty screen space to return to the top level of the diagram.

Generating firmware

1. Select the **Twin Activate Codegen DLL Controller** block. The block is highlighted in red.
2. Click **Tools > Code Gen**.



Code Generation Properties

Result File: Twin_Activate_HILExample.c

Result Dir: C:\Altair\Embed2024_64\cg

Target: STM32

Subtarget (set in target config): F410RB

Optimization Level: 0

□ Check for Performance Issues

☑ Use selected compound edge pins for data exchange (enables embedded debug)

☑ Embed Maps in Code

□ Add Stack Check Code

□ Call from Foreign RTOS/User App

☑ On-Chip RAM Only

□ Include Block Nesting as Comment

☑ Target FLASH

□ Enable Preemption in Main Diagram

□ Use Compound Names for Functions

Stack size: 128

Heap size: 0

Periodic Function Name: cgMain

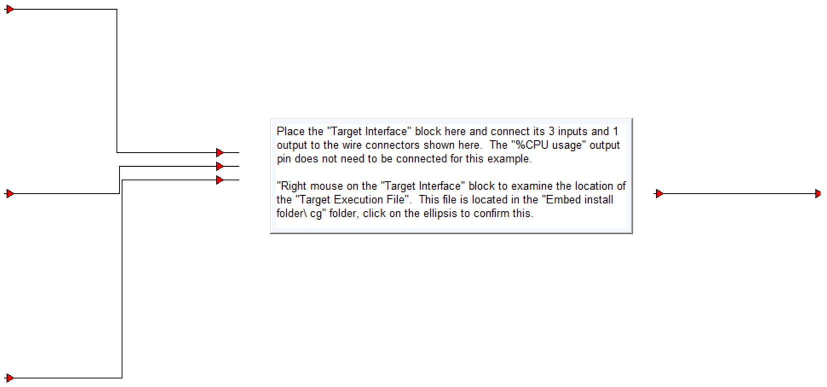
3. Make sure the **Use selected compound edge pins for data exchange** is activated. Then click **Compile**.

   The resulting file **Twin_Activate_HILExample.c** configured for an **STM32F410RB** is placed in the **C:\Altair\Embed2025.2_64\cg** directory.

4. Click **Quit**.

Setting up the HIL

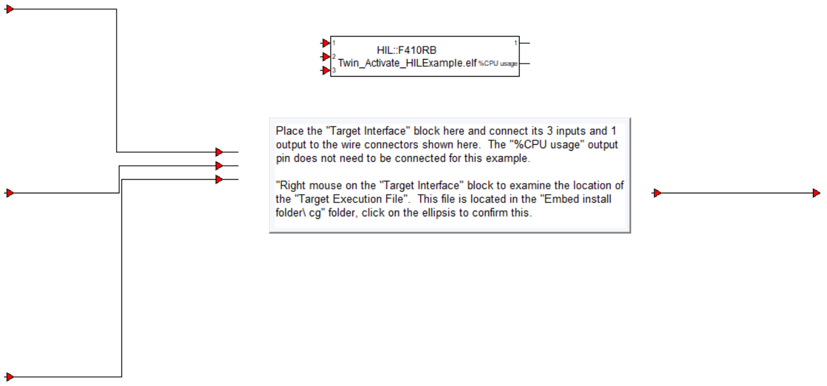
1. Under **MODEL 3**, right-click **Twin Activate Firmware Controller** compound block to dive into the next lower level of the block.



Place the "Target Interface" block here and connect its 3 inputs and 1 output to the wire connectors shown here. The "%CPU usage" output pin does not need to be connected for this example.

"Right mouse on the "Target Interface" block to examine the location of the "Target Execution File". This file is located in the "Embed install folder\ cg" folder, click on the ellipsis to confirm this.

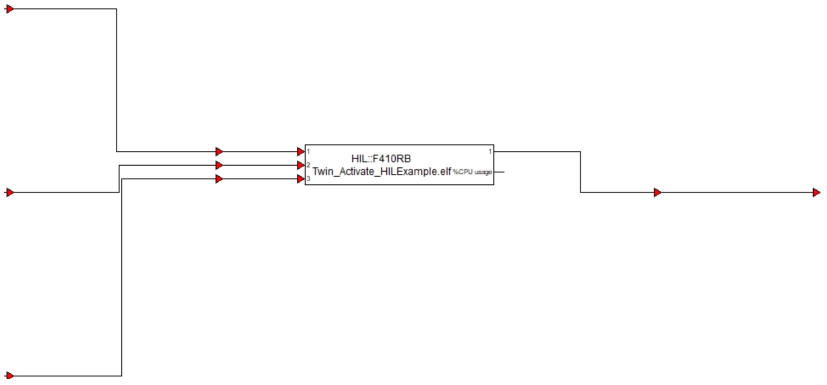
2. Click **Embedded > STM32 > Target Interface** and insert a **Target Interface** block into the diagram.



Place the "Target Interface" block here and connect its 3 inputs and 1 output to the wire connectors shown here. The "%CPU usage" output pin does not need to be connected for this example.

"Right mouse on the "Target Interface" block to examine the location of the "Target Execution File". This file is located in the "Embed install folder\ cg" folder, click on the ellipsis to confirm this.

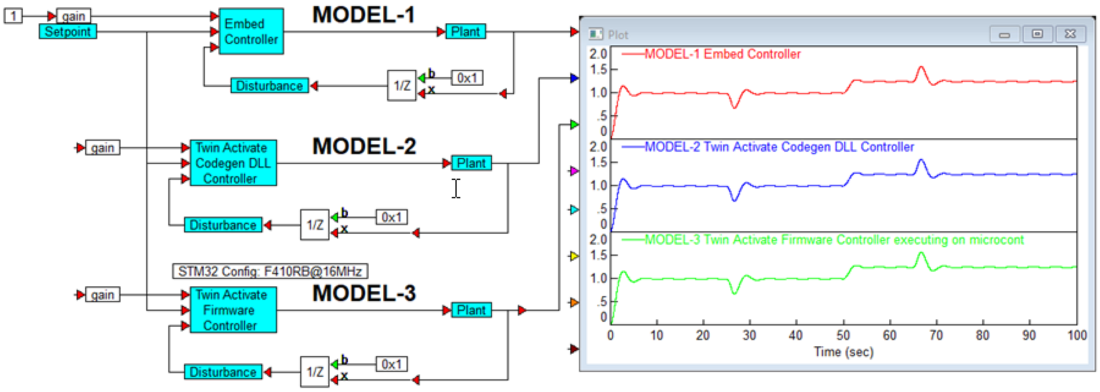
3. Replace the **comment** block with the **HIL-F410RB Target Interface** block and wire it into the diagram.



The **CPU% usage outpin pin** does not need to be connected.

4. Right-click on **empty screen space** to return to the top level of the diagram.
5. Click **System > System Properties > Range** and activate **Run in Real Time**.

6. Click the **Go** (▶) toolbar button to simulate the diagram and execute the code on the target device as shown in the lower plot below:



1
gain
Setpoint
Embed Controller
MODEL-1
Plant
Disturbance
1/Z
b
x
0x1
gain
Twin Activate Codegen DLL Controller
MODEL-2
Plant
Disturbance
1/Z
b
x
0x1
STM32 Config: F410RB@16MHz
gain
Twin Activate Firmware Controller
MODEL-3
Plant
Disturbance
1/Z
b
x
0x1
Plot
MODEL-1 Embed Controller
MODEL-2 Twin Activate Codegen DLL Controller
MODEL-3 Twin Activate Firmware Controller executing on microcont
2.0
1.5
1.0
.5
0
0
10
20
30
40
50
60
70
80
90
100
Time (sec)

Texas Instruments: Blinking the built-in LED on an F28069M

To blink the LED, follow the step-by-step directions in the Blink the LED using Altair Embed online video.

What you'll need

To perform these steps at your computer, you will need the following products:

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Texas Instruments F28069M	https://www.ti.com/tool/LAUNCHXL-F28069M

Texas Instruments: Measuring temperature on an F28069M

The **Chip Temp on F28069** diagram measures the temperature in centigrade of a Texas Instruments Piccolo F28069 device. The ADC channel 5 is redirected from an external pin to the on-chip temperature sensor. The compound block **Turn On Ch 5 Temp Conversion** performs the redirection.

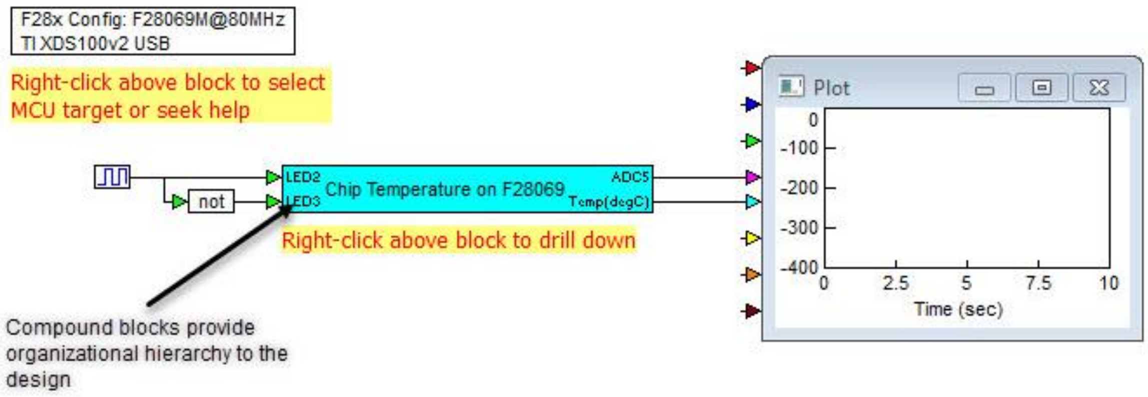
What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Texas Instruments F28069M	https://www.ti.com/tool/LAUNCHXL-F28069M

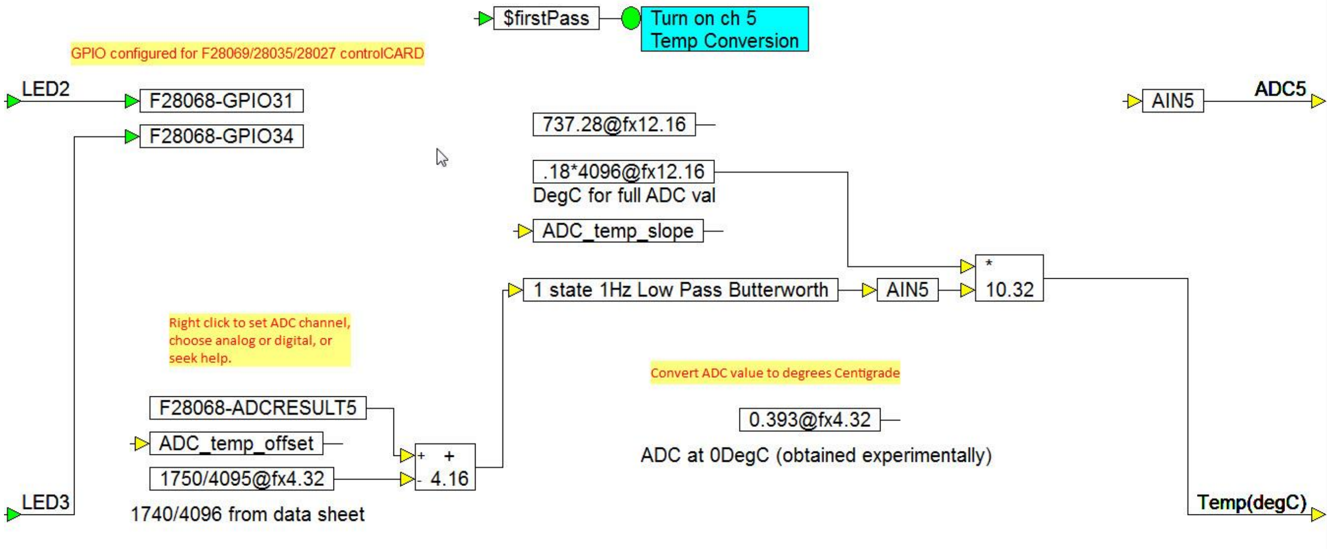
Opening and exploring Chip Temp on F28069 diagram

1. Click **Examples > Embedded > Piccolo > ADC**.
2. Select **Chip Temp on F28069**.

   The following diagram appears:

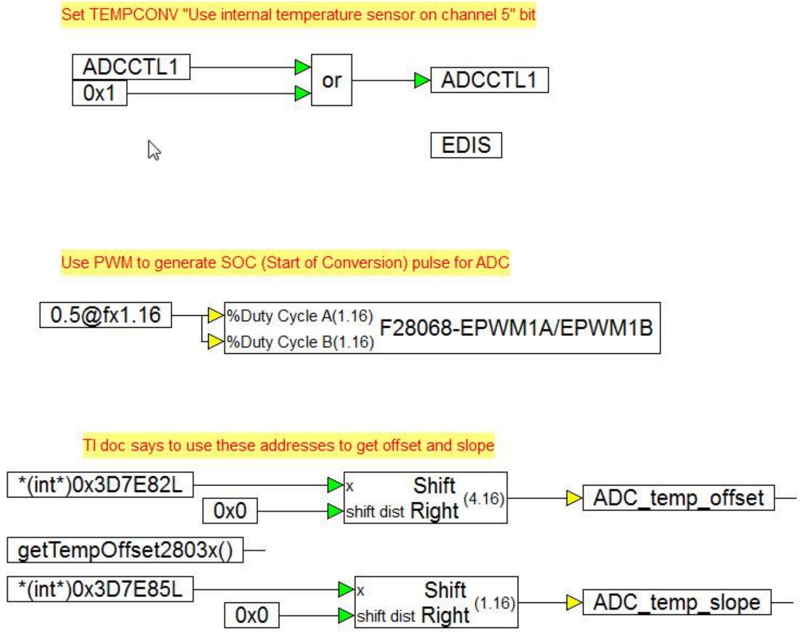

F28x Config: F28069M@80MHz
TI XDS100v2 USB
Right-click above block to select MCU target or seek help
not
LED2
LED3
Chip Temperature on F28069
ADC5
Temp(degC)
Right-click above block to drill down
Compound blocks provide organizational hierarchy to the design
Plot
0
-100
-200
-300
-400
0
2.5
5
7.5
10
Time (sec)

3. Right-click **Chip Temperature on F28069** to move down a level of hierarchy in the diagram.


$firstPass
Turn on ch 5 Temp Conversion
GPIO configured for F28069/28035/28027 controlCARD
LED2
F28068-GPIO31
F28068-GPIO34
AIN5
ADC5
737.28@fx12.16
.18*4096@fx12.16
DegC for full ADC val
ADC_temp_slope
*
10.32
1 state 1Hz Low Pass Butterworth
AIN5
Right click to set ADC channel, choose analog or digital, or seek help.
Convert ADC value to degrees Centigrade
F28068-ADCRESULT5
ADC_temp_offset
1750/4095@fx4.32
1740/4096 from data sheet
+
+
-
4.16
0.393@fx4.32
ADC at 0DegC (obtained experimentally)
LED3
Temp(degC)

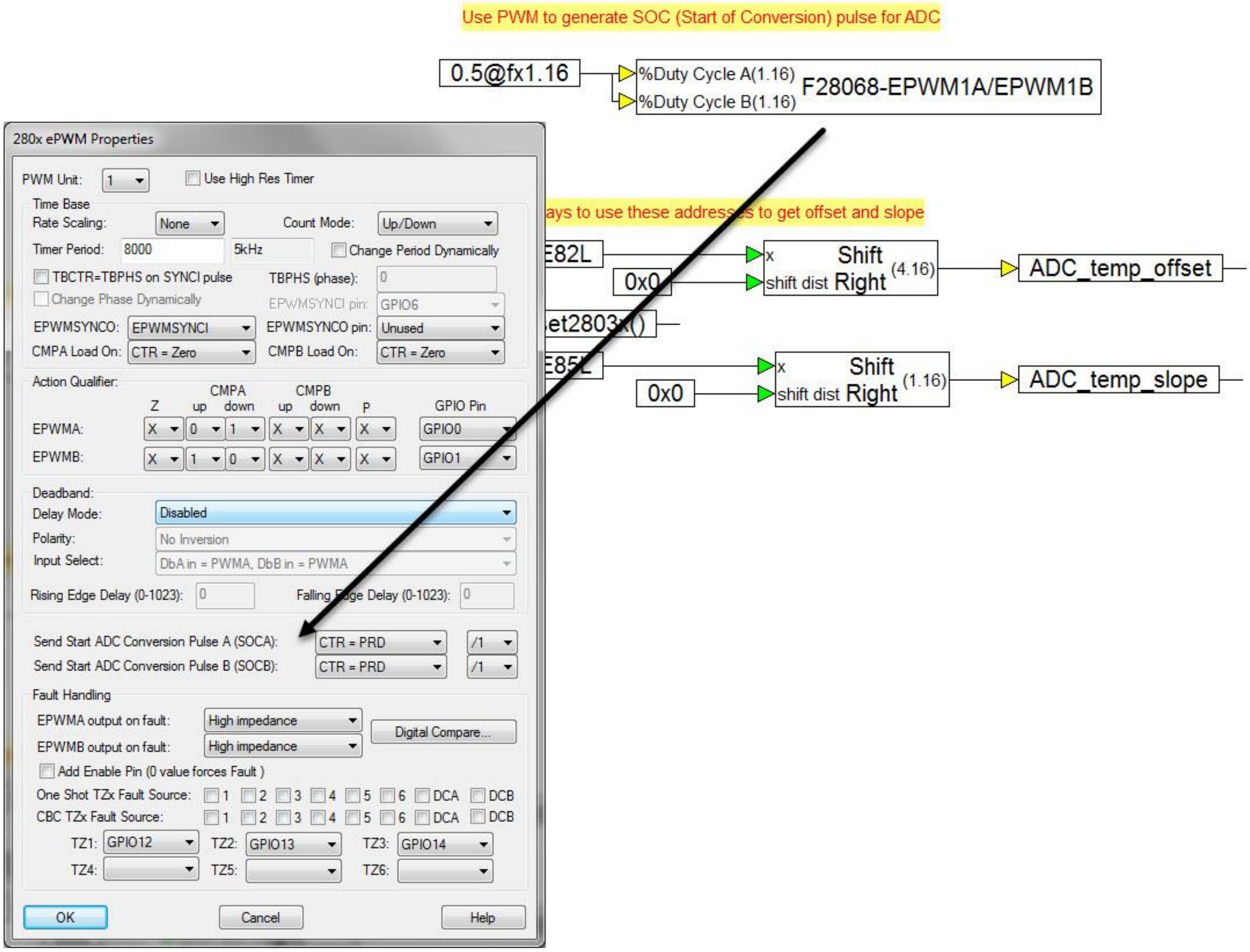
The above compound block reads ADC channel 5 and applies an offset and gain to convert the reading to degrees centigrade. It also executes the code contained in **Turn on Ch 5 Temp Conversion**, which switches ADC 5 from an external pin to the internal temperature sensor. The **Turn on Ch 5 Temp Conversion** is triggered by the built-in variable **$firstPass**. This means that the compound block and its contents are executed once at boot time.

4. Right-click **Turn on Ch 5 Temp Conversion** to move down another level of hierarchy in the diagram.



The above compound block enables the internal temperature sensor on ADC A5. The **Extern Read (*(int*)0x3D7E82L)** and **Extern Write (*(int*)0x3D7E85L)** blocks write directly to the hardware registers. To enforce the order of execution, Embed executes parallel flows in top-down order. The **ePWM** block sends **Start of Conversion** pulses to the **ADC A5**.

This code also enables the **ePWM** block to send **Start of Conversion** pulses to the **ADC A5**.



280x ePWM Properties

PWM Unit: 1 □ Use High Res Timer

Time Base
Rate Scaling: None Count Mode: Up/Down
Timer Period: 8000 5kHz □ Change Period Dynamically
□ TBCTR=TBPHS on SYNCI pulse TBPHS (phase): 0
□ Change Phase Dynamically EPWMSYNCI pin: GPIO6
EPWMSYNCO: EPWMSYNCI EPWMSYNCO pin: Unused
CMPA Load On: CTR = Zero CMPB Load On: CTR = Zero

Action Qualifier:

		CMPA		CMPB			
	Z	up	down	up	down	P	GPIO Pin
EPWMA:	X	0	1	X	X	X	GPIO0
EPWMB:	X	1	0	X	X	X	GPIO1

Deadband:
Delay Mode: Disabled
Polarity: No Inversion
Input Select: DbA in = PWMA, DbB in = PWMA
Rising Edge Delay (0-1023): 0 Falling Edge Delay (0-1023): 0

Send Start ADC Conversion Pulse A (SOCA): CTR = PRD /1
Send Start ADC Conversion Pulse B (SOCB): CTR = PRD /1

Fault Handling
EPWMA output on fault: High impedance
EPWMB output on fault: High impedance
□ Add Enable Pin (0 value forces Fault)
One Shot TZx Fault Source: □ 1 □ 2 □ 3 □ 4 □ 5 □ 6 □ DCA □ DCB
CBC TZx Fault Source: □ 1 □ 2 □ 3 □ 4 □ 5 □ 6 □ DCA □ DCB
TZ1: GPIO12 TZ2: GPIO13 TZ3: GPIO14
TZ4: TZ5: TZ6:

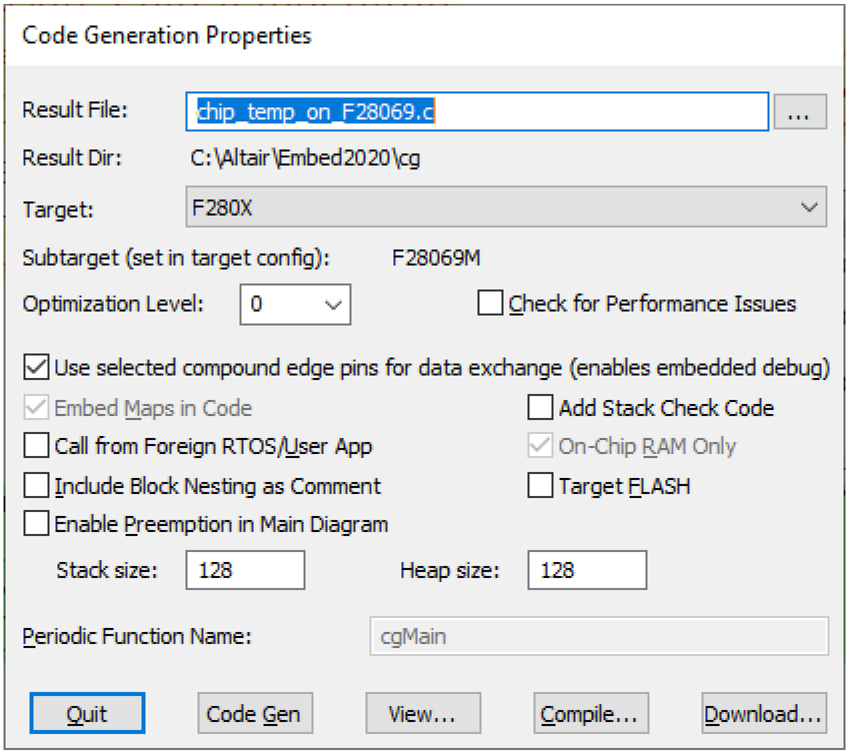
Compiling the source diagram

1. Move back up to the top level of the diagram by right-clicking on empty screen space.
2. Select the compound block **Chip Temperature on F28069**.

   The compound block turns red.

3. Click **Tools > Code Gen**.

   The C Code Generation dialog appears.



Code Generation Properties

Result File: chip_temp_on_F28069.c

Result Dir: C:\Altair\Embed2020\cg

Target: F280X

Subtarget (set in target config): F28069M

Optimization Level: 0

□ Check for Performance Issues

☑ Use selected compound edge pins for data exchange (enables embedded debug)

☑ Embed Maps in Code

□ Add Stack Check Code

□ Call from Foreign RTOS/User App

☑ On-Chip RAM Only

□ Include Block Nesting as Comment

□ Target FLASH

□ Enable Preemption in Main Diagram

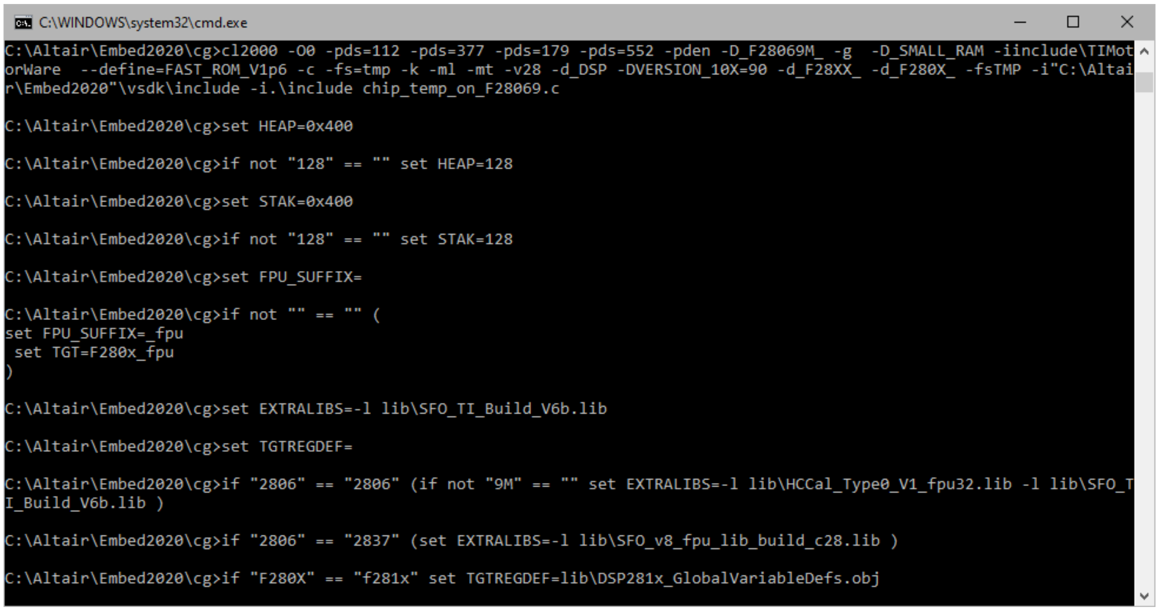
Stack size: 128

Heap size: 128

Periodic Function Name: cgMain

4. Activate **Use selected compound edge pins for data exchange**. This lets you debug the target executable.
5. Click **Compile** to generate C code and compile it with Code Composer.

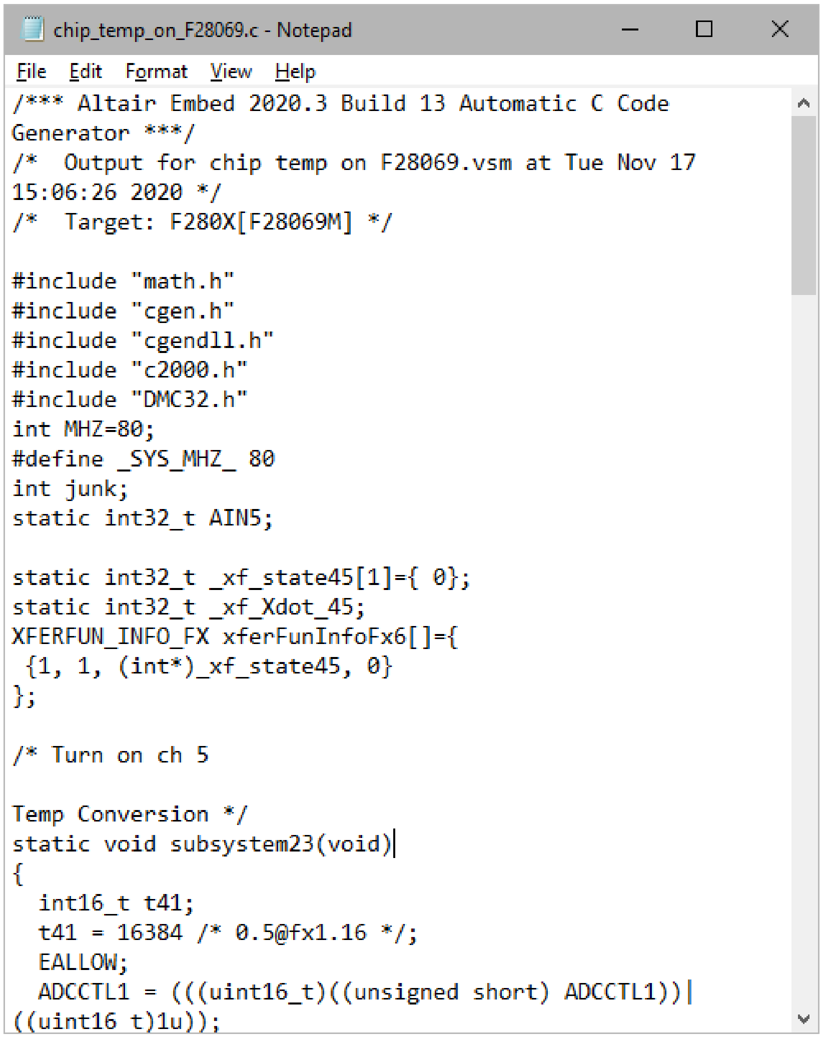
   The following DOS window appears.



```
C:\Altair\Embed2020\cg>cl2000 -O0 -pds=112 -pds=377 -pds=179 -pds=552 -pden -D_F28069M_ -g  -D_SMALL_RAM -iinclude\TIMot
orWare  --define=FAST_ROM_V1p6 -c -fs=tmp -k -ml -mt -v28 -d_DSP -DVERSION_10X=90 -d_F28XX_ -d_F280X_ -fsTMP -i"C:\Altai
r\Embed2020"\vsdk\include -i.\include chip_temp_on_F28069.c

C:\Altair\Embed2020\cg>set HEAP=0x400

C:\Altair\Embed2020\cg>if not "128" == "" set HEAP=128

C:\Altair\Embed2020\cg>set STAK=0x400

C:\Altair\Embed2020\cg>if not "128" == "" set STAK=128

C:\Altair\Embed2020\cg>set FPU_SUFFIX=

C:\Altair\Embed2020\cg>if not "" == "" (
set FPU_SUFFIX=_fpu
 set TGT=F280x_fpu
)

C:\Altair\Embed2020\cg>set EXTRALIBS=-l lib\SFO_TI_Build_V6b.lib

C:\Altair\Embed2020\cg>set TGTREGDEF=

C:\Altair\Embed2020\cg>if "2806" == "2806" (if not "9M" == "" set EXTRALIBS=-l lib\HCCal_Type0_V1_fpu32.lib -l lib\SFO_T
I_Build_V6b.lib )

C:\Altair\Embed2020\cg>if "2806" == "2837" (set EXTRALIBS=-l lib\SFO_v8_fpu_lib_build_c28.lib )

C:\Altair\Embed2020\cg>if "F280X" == "f281x" set TGTREGDEF=lib\DSP281x_GlobalVariableDefs.obj
```

The above window displays the output of the Code Composer compiling and linking the diagram.

6. You can check to make sure the compile (cl2000) and link (link2000) are error free, and then press ***any key*** to continue.
7. Click **View** in the C Code Generation dialog if you want to examine the generated C code.



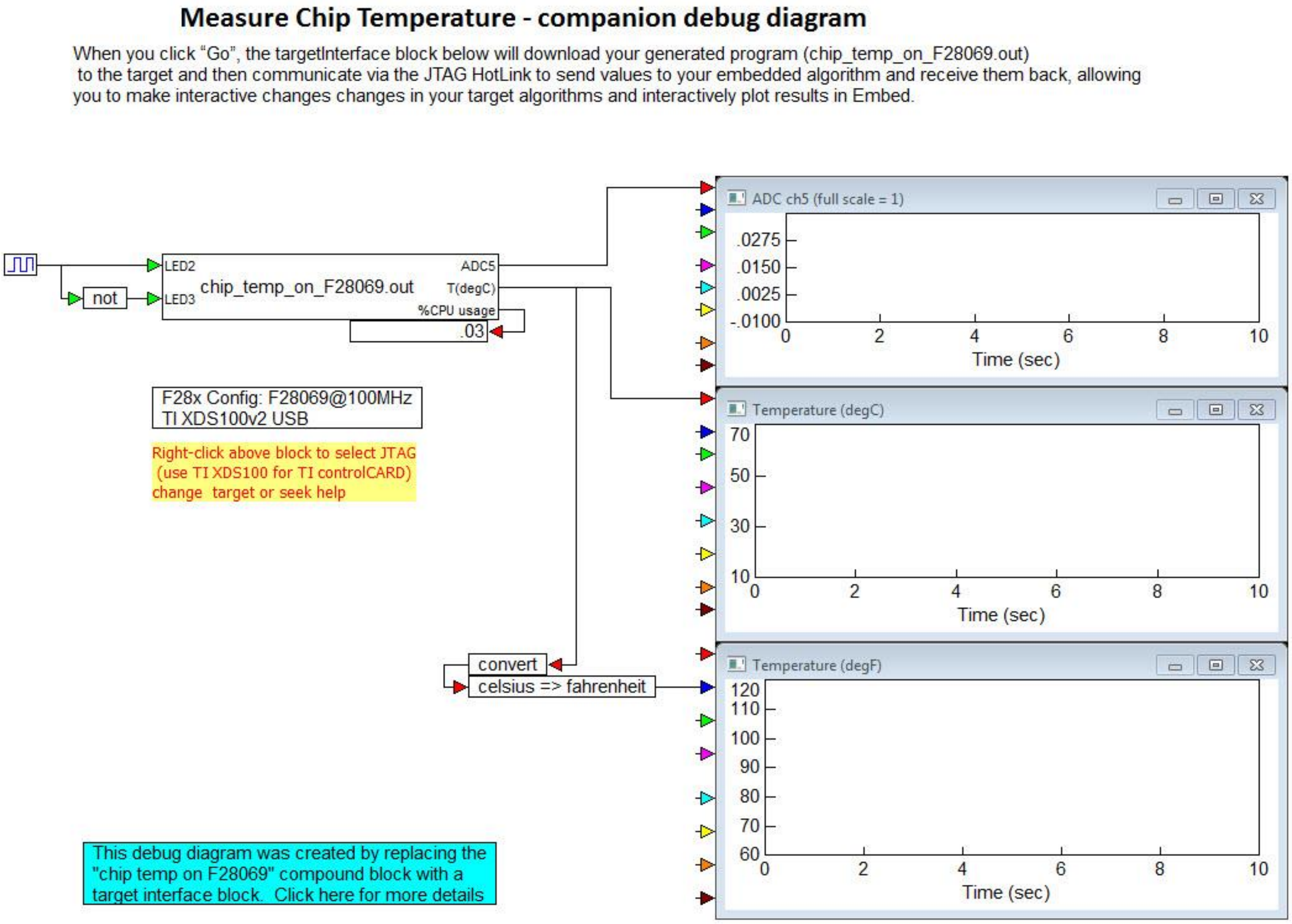
```
/*** Altair Embed 2020.3 Build 13 Automatic C Code
Generator ***/
/*  Output for chip temp on F28069.vsm at Tue Nov 17
15:06:26 2020 */
/*  Target: F280X[F28069M] */

#include "math.h"
#include "cgen.h"
#include "cgendll.h"
#include "c2000.h"
#include "DMC32.h"
int MHZ=80;
#define _SYS_MHZ_ 80
int junk;
static int32_t AIN5;

static int32_t _xf_state45[1]={ 0};
static int32_t _xf_Xdot_45;
XFERFUN_INFO_FX xferFunInfoFx6[]={
 {1, 1, (int*)_xf_state45, 0}
};

/* Turn on ch 5

Temp Conversion */
static void subsystem23(void)
{
  int16_t t41;
  t41 = 16384 /* 0.5@fx1.16 */;
  EALLOW;
  ADCCTL1 = (((uint16_t)((unsigned short) ADCCTL1))|
((uint16_t)1u));
```

Downloading and debugging

After compiling the source diagram, you can download and debug it using the companion debug diagram **Chip Temp on F28069-d**.

1. Click **Examples > Embedded > Piccolo > ADC**.
2. Select **Chip Temp on F28069 –d**.

   The following diagram appears:



Measure Chip Temperature - companion debug diagram

When you click “Go”, the targetInterface block below will download your generated program (chip_temp_on_F28069.out) to the target and then communicate via the JTAG HotLink to send values to your embedded algorithm and receive them back, allowing you to make interactive changes changes in your target algorithms and interactively plot results in Embed.

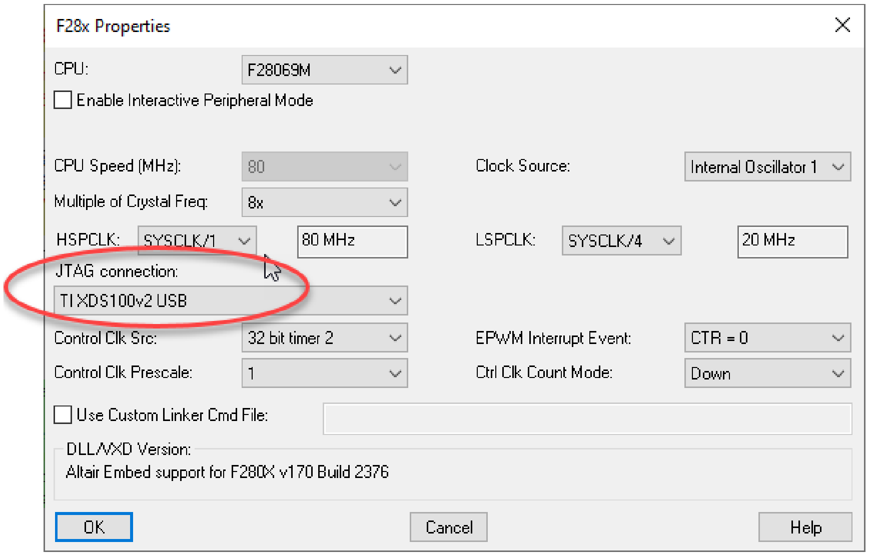
F28x Config: F28069@100MHz
TI XDS100v2 USB

Right-click above block to select JTAG (use TI XDS100 for TI controlCARD) change target or seek help

This debug diagram was created by replacing the "chip temp on F28069" compound block with a target interface block. Click here for more details

3. Right-click **F28x Config**.

   The F28x Properties dialog appears.



F28x Properties

CPU: F28069M

□ Enable Interactive Peripheral Mode

CPU Speed (MHz): 80

Clock Source: Internal Oscillator 1

Multiple of Crystal Freq: 8x

HSPCLK: SYSCLK/1 80 MHz

LSPCLK: SYSCLK/4 20 MHz

JTAG connection: TI XDS100v2 USB

Control Clk Src: 32 bit timer 2

EPWM Interrupt Event: CTR = 0

Control Clk Prescale: 1

Ctrl Clk Count Mode: Down

□ Use Custom Linker Cmd File:

DLL/VXD Version:
Altair Embed support for F280X v170 Build 2376

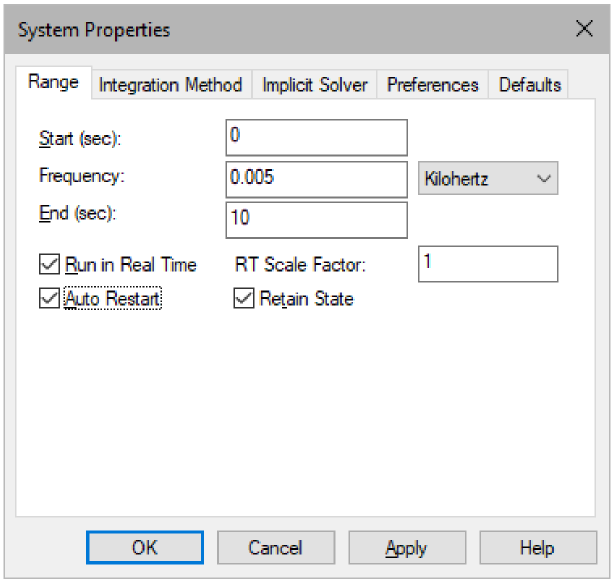
4. Make sure that the proper JTAG linkage is selected. This example uses **XDS100**.

Setting simulation properties

You set the main run rate of the diagram for both simulation and generated code for the target in the System Properties dialog. For simulation purposes, you can also set the integration algorithm and duration of the simulation.

1. Choose **System > System Properties**.

   The System Properties dialog appears.



System Properties

Start (sec): 0

Frequency: 0.005 Kilohertz

End (sec): 10

☑ Run in Real Time

RT Scale Factor: 1

☑ Auto Restart

☑ Retain State

2. The above dialog is for the Chip Temp on F28069-d debug diagram. Notice the options used in the debug diagram:

   - **Frequency/Time Step:** 0.005 provides a 200Hz update rate to data and plots.
   - **End:** Creates a 10 sec interval on plots.
   - **Run in Real Time:** Executes the diagram in real time so that Embed runs in sync with the target.
   - **Auto Restart:** Runs continuously until you stop it.
   - **Retain State**: Refrains from initializing blocks on restart and prevents reloading the OUT file.

Running the diagram and viewing results

When you simulate the diagram, Embed downloads and runs the OUT file you created when you compiled the source diagram. After it starts running on the target, Embed provides the following:

- **Interactive inputs:** The 1Hz square wave to the GPIOs blinks the on-board LEDs on the Texas Instruments controlSTICK or controlCARD
- **Interactive plots of on-chip outputs:** The raw ADC A5 reading and adjusted temperature in centigrade

The simulation runs until you click **Stop** in the toolbar.

Texas Instruments: Implementing fixed-point controllers and control logic on target hardware

From a methodology point of view, the main concept that is crucial in embedded system prototyping is the principle of adequate and complete encapsulation. That is, all control, logic, input-output (I/O,) and other subsystems that must run on the embedded target, must be contained in a single compound block.

With Embed, you can quickly and easily implement fixed-point controllers and logic on embedded targets. The embedded code can be exercised and tested by changing set points using slider blocks and observing output in plots while the actual control code is executed on the embedded target.

What you'll need

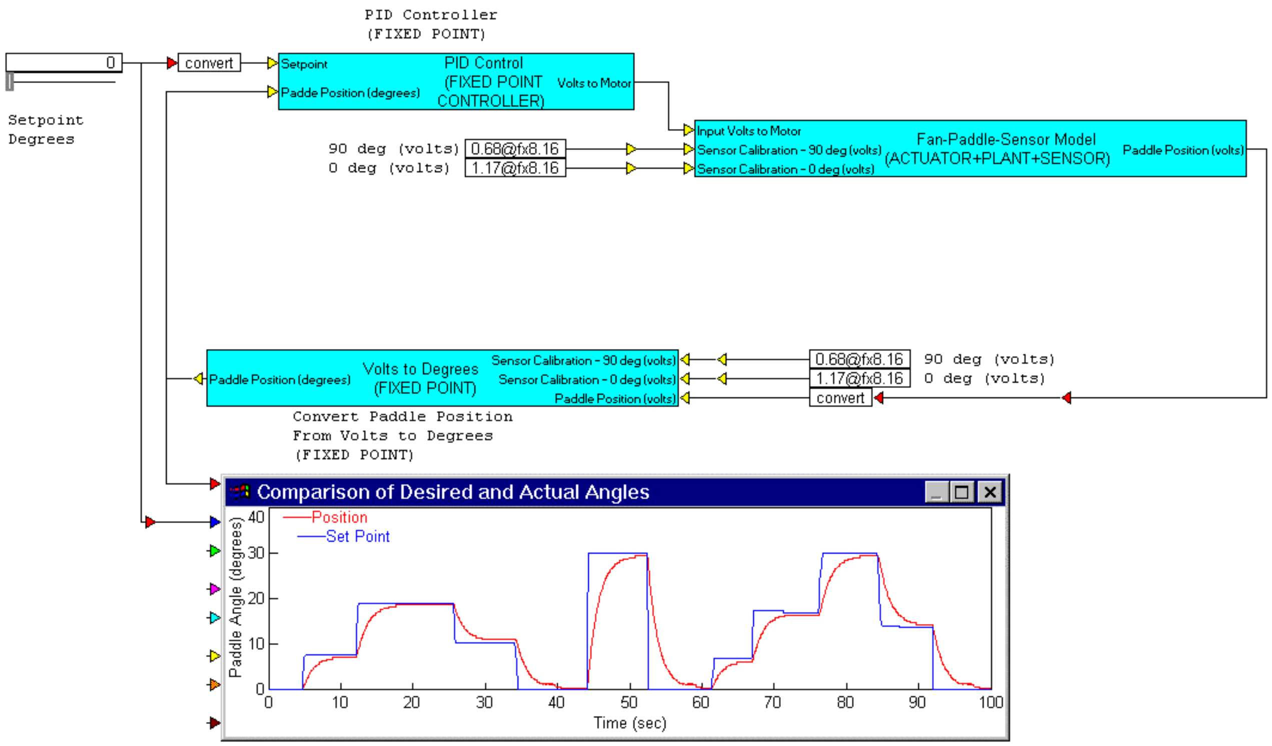
This example explores the **position_control_embedded** diagram under Examples > Fixed Point. This diagram builds on the **position_control_fixpoint** diagram.

Configuring the system for implementation on an embedded target

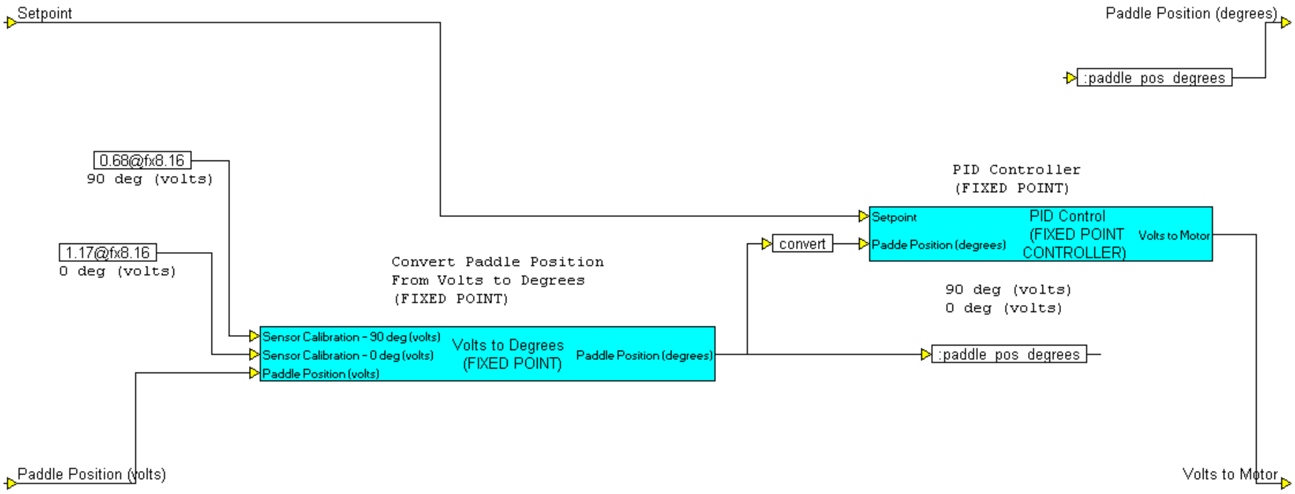
Once the controller and control logic are simulated in scaled fixed-point, and the design issues are addressed satisfactorily, the next step is the actual implementation of the controller and control logic on target hardware.

In the fixed-point version of the fan-paddle position control system, the **PID Controller (FIXED POINT CONTROLLER)** and **Volts to Degrees (FIXED POINT)** are the control and logic functions for this system. To ready this system for direct implementation on an embedded target, it follows that all you need to do is collapse these two compound blocks into a single compound block.

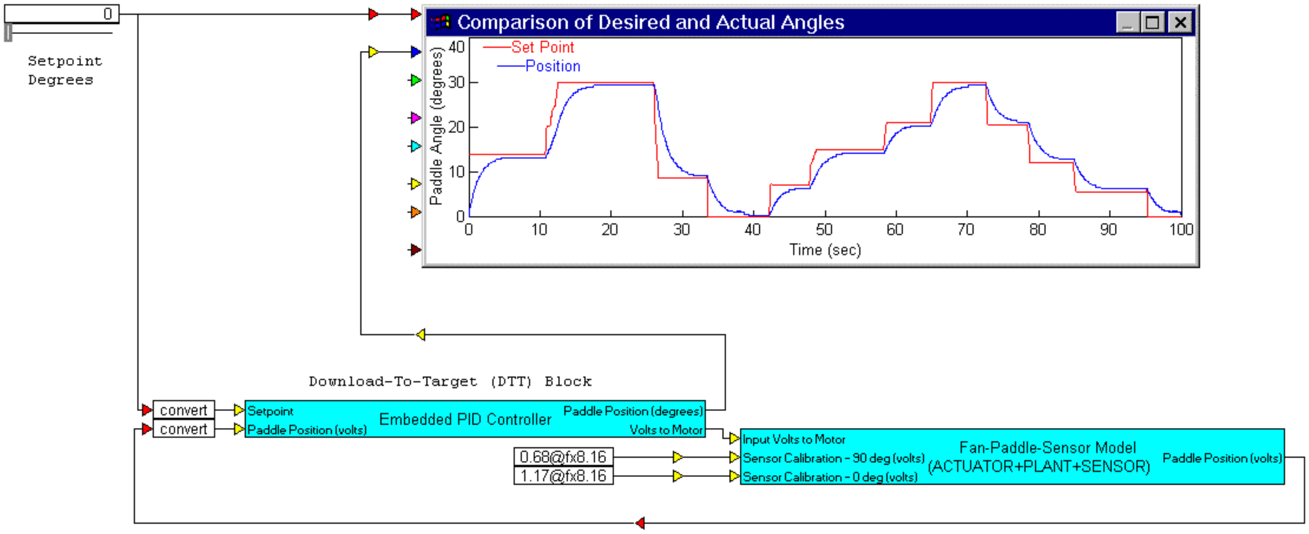
To prepare for encapsulation, begin by duplicating the 0° and 90° calibration value constant, so that **Fan-Paddle-Sensor** and **Volts to Degrees Converter** each has its own set of constants.



Next, encapsulate **PID Controller** and **Volts to Degrees** in a single compound block named **Embedded PID Controller**.



A new local variable called *paddle_pos_degrees* is created and wired to an additional output tab to bring this value out and provide external access to it for monitoring purposes. The complete system representation is as follows:



In this form, **Embedded PID Controller** can generate, compile, link, and download embeddable C code to supported targets and perform HIL system prototyping and validation.

When performing HIL validation, the use of analog and digital inputs and outputs is quite common. Embed provides analog and digital I/Os that can be configured like any other Signal Producer or Consumer block and placed inside the **Embedded PID Controller** block. Embed supports automatic programming of on-board analog and digital I/O, as well as all the important peripherals. The parameters entered in the configuration of the I/O points and peripherals, such as channel number, and range, are extracted and placed in the embedded control code for the **Embedded PID Controller** block and automatically downloaded to the target.

Texas Instruments: HIL testing with an imported block from PSIM

The **PSIM HIL Example** diagram builds on the PSIM SIM diagram. It is a good idea to follow the procedure there before starting this example.

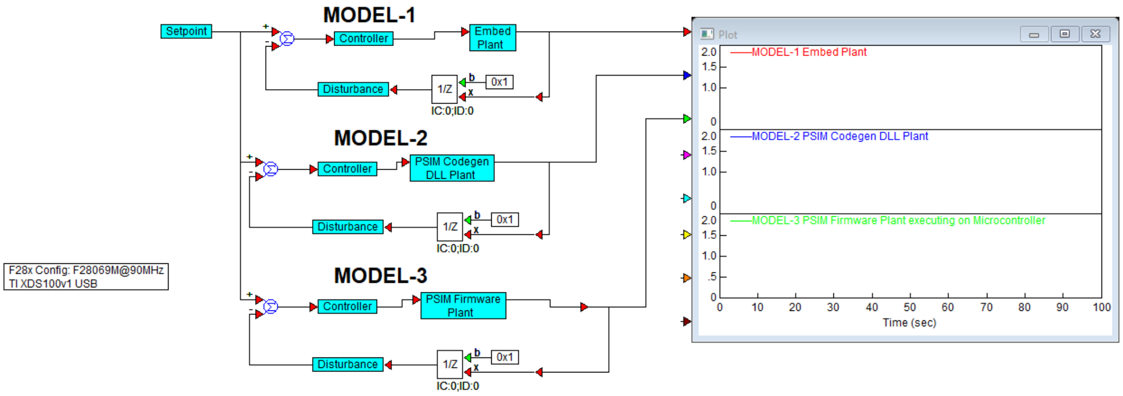
What you'll need

Product	Where to get it
Embed Pro or Embed Personal	https://www.altair.com/embed/ ; https://web.altair.com/embed-personal-edition
Texas Instruments F28069M	https://www.ti.com/tool/LAUNCHXL-F28069M
PSIM	https://altair.com/psim ; you only need PSIM if you want to open the PSIM schematic, you don't need it to follow along with the example

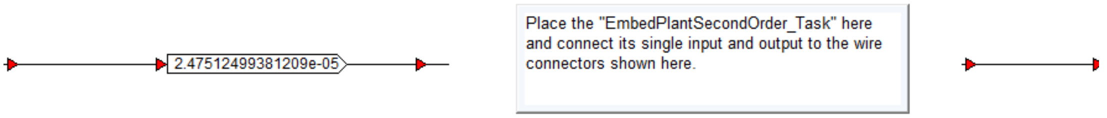
Setting up the diagram

In the HIL diagram, Embed converts the **PSIM Codegen Plant** to firmware configured to execute on a Texas Instruments F28069M device.

1. If you have not yet performed **steps 1a – 1e** in the PSIM SIM Example diagram example, do so now.
2. Add the **imported** block to the **PSIM HIL Example** diagram.
    a. Click **Examples > Blocks > Extensions > Imported Blocks > PSIM > PSIM HIL Example**.

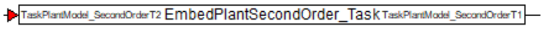


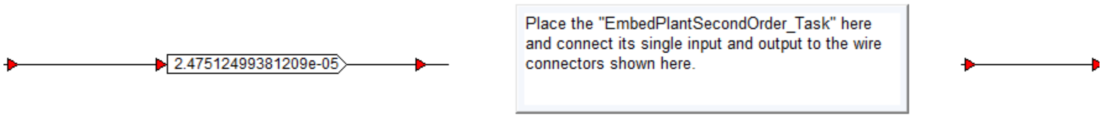
    b. Under **MODEL-2**, right-click **PSIM Codegen DLL Plant** compound block to dive into the next lower level of the block.



Place the "EmbedPlantSecondOrder_Task" here and connect its single input and output to the wire connectors shown here.

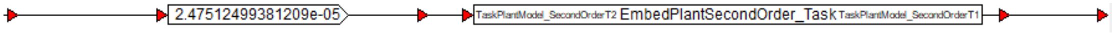
    c. Click **Imported Blocks > PSIM > Blocks for EmbedPlantSecondOrder** and insert the **block** into the diagram.


TaskPlantModel_SecondOrderT2
EmbedPlantSecondOrder_Task
TaskPlantModel_SecondOrderT1



Place the "EmbedPlantSecondOrder_Task" here and connect its single input and output to the wire connectors shown here.

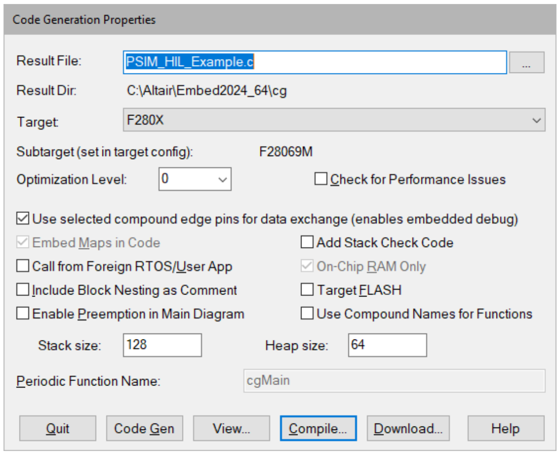
    d. Replace the **comment** block with the **EmbedPlantSecondOrder_Task** block and wire it into the diagram.


2.47512499381209e-05
TaskPlantModel_SecondOrderT2
EmbedPlantSecondOrder_Task
TaskPlantModel_SecondOrderT1

    e. Right-click on empty screen space to return to the top level of the diagram.

Generating firmware

1. Select the **PSIM Codegen DLL Plant** block. The block is highlighted in red.
2. Click **Tools > Code Gen**.



Code Generation Properties

Result File: PSIM_HIL_Example.c ...

Result Dir: C:\Altair\Embed2024_64\cg

Target: F280X

Subtarget (set in target config): F28069M

Optimization Level: 0 □ Check for Performance Issues

☑ Use selected compound edge pins for data exchange (enables embedded debug)

☑ Embed Maps in Code □ Add Stack Check Code

□ Call from Foreign RTOS/User App ☑ On-Chip RAM Only

□ Include Block Nesting as Comment □ Target FLASH

□ Enable Preemption in Main Diagram □ Use Compound Names for Functions

Stack size: 128 Heap size: 64

Periodic Function Name: cgMain

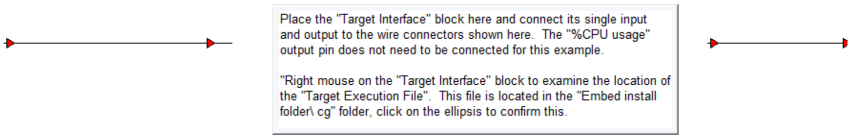
3. Make sure the **Use selected compound edge pins for data exchange** is activated. Then click **Compile**.

   The resulting file **PSIM_HIL_Example.c** configured for an **TIF28069M** is placed in the ***<Embed-installation-directory>*\cg** directory.

4. Click **Quit**.

Setting up the HIL

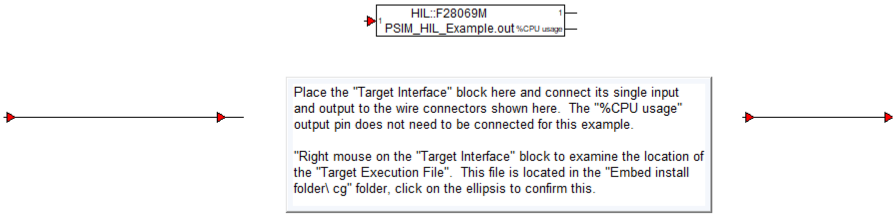
1. Under **MODEL 3**, right-click **PSIM Firmware Plant** compound block to dive into the next lower level of the block.



Place the "Target Interface" block here and connect its single input and output to the wire connectors shown here. The "%CPU usage" output pin does not need to be connected for this example.

"Right mouse on the "Target Interface" block to examine the location of the "Target Execution File". This file is located in the "Embed install folder\ cg" folder, click on the ellipsis to confirm this.

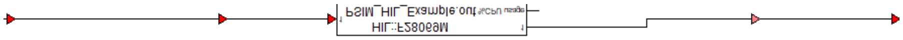
2. Click **Embedded > F280x > Target Interface** and insert a **Target Interface** block into the diagram.



Place the "Target Interface" block here and connect its single input and output to the wire connectors shown here. The "%CPU usage" output pin does not need to be connected for this example.

"Right mouse on the "Target Interface" block to examine the location of the "Target Execution File". This file is located in the "Embed install folder\ cg" folder, click on the ellipsis to confirm this.

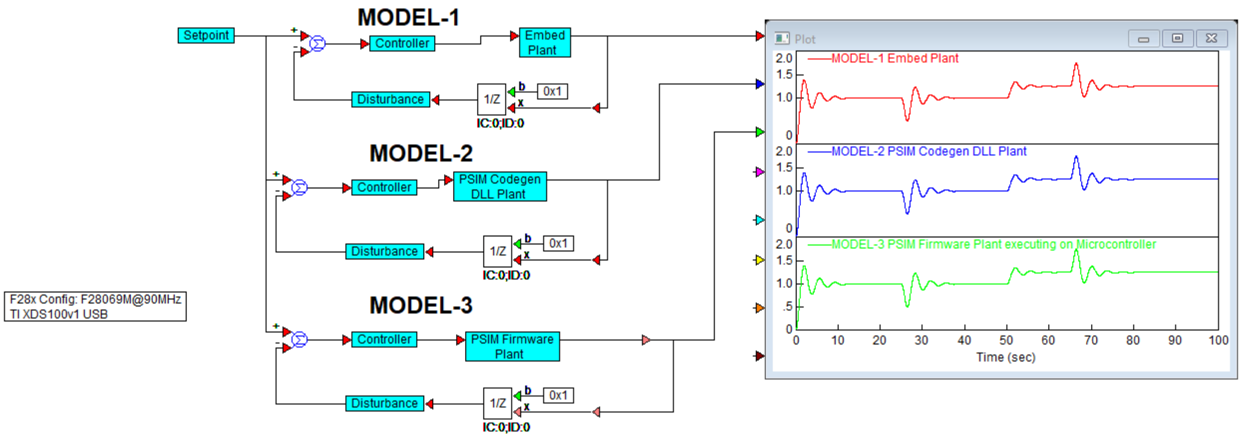
3. Replace the **comment** block with the **HIL-F28069M Target Interface** block and wire it into the diagram.



   The **CPU% usage outpin pin** does not need to be connected.

4. Right-click on **empty screen space** to return to the top level of the diagram.
5. Click **System > System Properties > Range** and activate **Run in Real Time**.

6. Click the **Go** (▶) toolbar button to simulate the diagram and execute the code on the target device as shown in the lower plot below:



Where to go from here

Sample diagrams

Embed provides hundreds of fully documented sample diagrams in the Examples menu. These diagrams illustrate both simple and complex diagrams spanning a broad range of engineering disciplines, including aerospace, biophysics, chemical engineering, control design, dynamic systems, electromechanical systems, environmental systems, HVAC, motion control, process control, and signal processing.

Videos

The online videos offer quick and easy ways to learn basic concepts in Embed. Each video focuses on a specific Embed feature. If you are a new Embed user, a good video to start with is Introduction to Altair Embed.

Online forum

Embed has an active online community forum where you can post questions, get answers, and promote conversations with other Embed users.

Training services

The Embed Solutions group offers training sessions for learning and gaining expertise in Embed and the Embed family of add-on products. Training sessions are conducted at Altair in-house training facilities, as well as at customer sites and as online webinars.