Altia Connection for Simulink®

Stateflow® Demonstration

Altia Design 13.3

Updated March 17, 2022

Please Read Before Using this Software

Altia, Inc. © 2022
111 S. Tejon St., Ste 204
Colorado Springs, CO 80903

Altia Design 13.x is distributed through Altia Cloud

Beginning with Altia Design 13, all products are delivered through Altia Cloud software. Altia Launcher automatically downloads and installs your licensed products in the correct versions to support your projects.

Altia Design Requires Visual Studio 2019 Redistributable

Altia Design 13 requires that the Visual Studio 2019 Redistributable package be installed on the host computer. The redistributable package is installed with the Altia Launcher.

Altia Design Compatibility Summary

Design files from Altia Design 12.x, and earlier, can be imported into Altia Design 13 projects using the Resource Manager User Interface. This operation imports a .dsn file along with all resources referenced by the design.

Designs imported from older versions of Altia Design may appear different in Altia Design 13. Some fonts will appear in a different size to comply with the new Altia Design resource system.

Online videos help you get started quickly with Altia Design!

To get started quickly with Altia Design, please visit the Altia Design videos web page:

http://support.altia.com

Table of Contents

List of Figures

1 Introduction

This document accompanies Altia-developed, MATLAB Simulink/Stateflow model files to demonstrate the use of Altia with Stateflow.

NOTE: Using Altia with Stateflow is only supported for the Windows version of MATLAB.

In this document, we do not describe the detailed steps for creating models in Simulink and Stateflow. The Simulink **Help** menu (**?** In R2019b or newer, **Help** in R2019a or older) has options to open Simulink documentation and examples. If a Stateflow chart is open, the **Help** menu has options to open Stateflow documentation and examples. There are also online tutorials available. For example, at the time this document was written, the following URL refers to the MathWorks Videos and Webinars home page:

https://www.mathworks.com/videos.html

On the above web page, do a search for Stateflow, and choose from a selection of videos.

The Altia Design product includes an editor, runtime engine, and numerous libraries of components for quickly creating a human-machine interface (HMI). In addition to using the supplied component libraries to create HMIs, users can make modified versions of these components or create custom components without programming.

In this document, HMI (Human-Machine Interface) may also be referred to as GUI (Graphical User Interface), UI (User Interface), graphical front panel, or simply an Altia design (.dsn) file.

An Altia DeepScreen® code generator is an optional Altia product that provides the ability to generate **C** code to deploy an HMI on one of many different embedded devices.

NOTE: Use Stateflow in conjunction with the Altia **C** API to interface to Altia Runtime on Windows, DeepScreen generated code on Windows, or DeepScreen generated code for deploying on an embedded device.

Continue to the next chapter for instructions on installing this demonstration for use with MATLAB Simulink/Stateflow.

2 MATLAB Installation Requirements

This demonstration requires the following:

- MATLAB for Windows, R2019a or newer, 64-bit. As of January 2022, Altia has tested this demonstration with MATLAB R2019a thru R2021b.
- ***MATLAB must be configured to use Visual Studio 2015, 2017, or 2019 as its compiler.*** Newer versions of Visual Studio may work, but they have not been tested by Altia. To configure the compiler for MATLAB, run the `mex -setup` command at the MATLAB Command Window prompt and follow the instructions.
- To demonstrate creation of an executable to run standalone on Windows, this demonstration uses the MATLAB **Simulink Coder**. You must have a MATLAB **Simulink Coder** license to follow along for this part of the demonstration.
- To demonstrate Stateflow generated code that will be deployed to an embedded device with Altia DeepScreen generated code, this demonstration uses the MATLAB **Embedded Coder**. You must have a MATLAB **Embedded Coder** license to follow along for this part of the demonstration.

In this document, we will refer to the MATLAB user interface that has been active since at least the R2015b release thru R2020a. For these releases, the MATLAB menu ribbon looks like this:

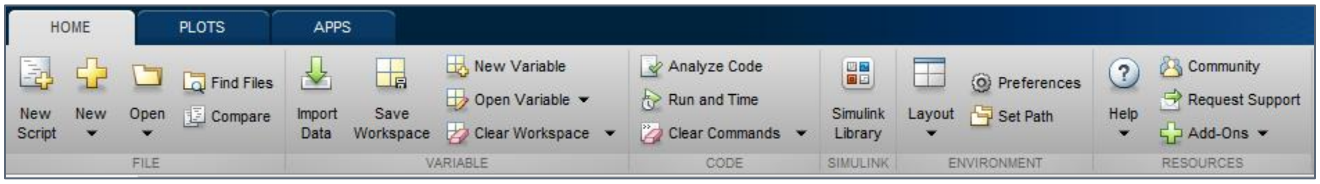


Figure 1: MATLAB R2015b thru R2020a Menu Ribbon

In this document, we will refer to the Simulink user interface that has been active since R2019b. The Simulink menu ribbon looks like this:

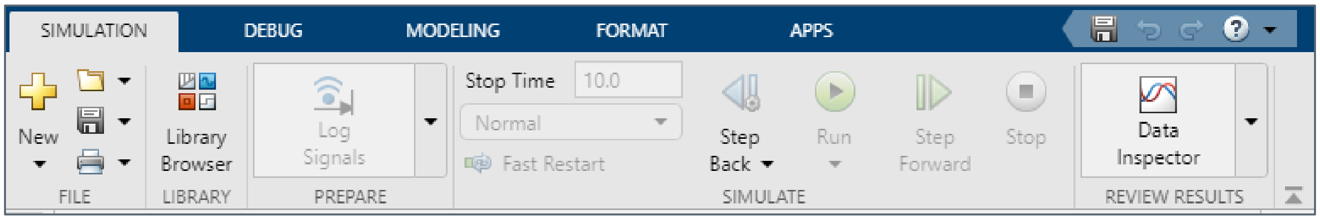


Figure 2: MATLAB R2019b Simulink Menu Ribbon

NOTE: The appearance of dialogs and option selections in Simulink and Stateflow for R2019b and newer is slightly different compared to R2019a and earlier, but the general functionalities of dialogs and option selections remain the same. If you are using Simulink and Stateflow for R2019a or earlier, you must be familiar enough with Simulink and Stateflow functionality to realize how to translate for the differences in appearance.

3 Installing this Demonstration

There are two ways to install the Stateflow demo: as part of the Altia Connection for Simulink toolbox or as a Zip file.

3.1 Altia Connection for Simulink

If you are reading this document as part of installing the Altia Connection for Simulink toolbox or if the Altia Connection for Simulink toolbox is already installed, this demonstration was installed as part of the Altia Connection for Simulink. The files associated with this document are here:

```
C:\Users\<USER_NAME>\AppData\Roaming\MathWorks\MATLAB Add-
Ons\Toolboxes\Altia Connection\demos\AltiaStateflowDemo
```

To easily view/browse your AppData folder, simply enter **%APPDATA%** in the Address bar of Windows Explorer. Copy the **AltiaStateflowDemo** folder into your MATLAB work folder or a suitable user work folder. For example, on Windows 7 or newer, the MATLAB work folder is something like:

```
C:\Users\<USER_NAME>\Documents\MATLAB
```

Start MATLAB and then change MATLAB's Current Folder to the new **AltiaStateflowDemo** folder.

3.2 ZIP file

If you received this demonstration as a ZIP file, unzip it relative to the MATLAB work folder or a suitable user work folder to create a new **AltiaStateflowDemo** sub-folder. Start MATLAB and then change MATLAB's Current Folder to the new **AltiaStateflowDemo** folder.

4 Overview of the Demonstration Model

This demonstration comes with two (2) simple Simulink/Stateflow models to demonstrate use of the Altia Connection with Stateflow: `AltiaStateflowDemo.slx` and `AltiaStateflowNoSimPanel.slx`.

- `AltiaStateflowDemo.slx` – If you have the Altia Connection for Simulink toolbox installed in your MATLAB installation, you can open this model in MATLAB.
    - As shown in the picture below, this Simulink model contains an Altia block `AltiaSimPanel` to provide an Altia interface for exercising the Stateflow chart with input events and data. In this example, `Ignition On/Off` events go into the Stateflow chart from the Altia block and `Speed in` values are delivered to the Stateflow `speed_in` input. The Altia block can also monitor data from the Stateflow chart. In this example, `Speed Out` values go into the Altia block from the Stateflow `speed_out` output.
    - This Simulink model also has a `Signal Generator` for delivering an event at a regular interval to the Stateflow chart. This event triggers the execution of the Stateflow chart (this is described in more detail later in this document).

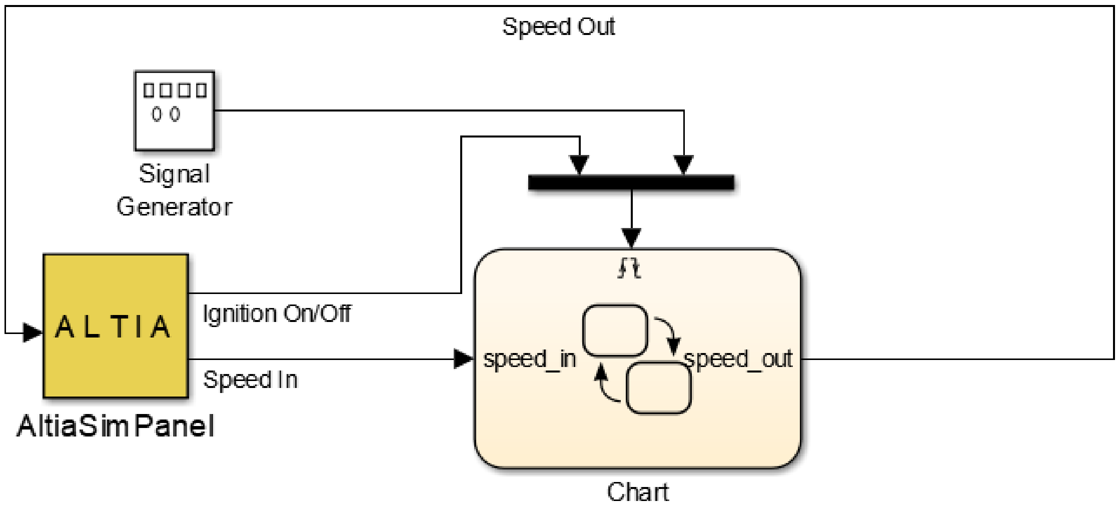


Figure 3: AltiaStateflowDemo.slx

    - The Altia block is optional as is any other content in the Simulink model (such as the `Signal Generator` in this example). The Altia block `AltiaSimPanel` and the other Simulink components make it easier to run in simulation mode within Simulink. We can replace the Altia block with other Simulink components (e.g., a `Ramp` for `Ignition On/Off` and a `Sine Wave` for `Speed In`), but these components do not provide as much flexibility for exercising the Stateflow model in simulation mode.

- **`AltiaStateflowNoSimPanel.slx`** – If you do ***not*** have the Altia Connection for Simulink package installed in your MATLAB installation, or you prefer a model that does not use an Altia block, open this model in MATLAB.

  As shown in the picture below, this model uses a **`Signal Generator`**, **`Ramp`**, and **`Sine Wave`** to simulate input events and data into the Stateflow chart. The output of the Stateflow chart is simply connected to a **`Display`**. The Stateflow chart is identical in both Simulink model files.

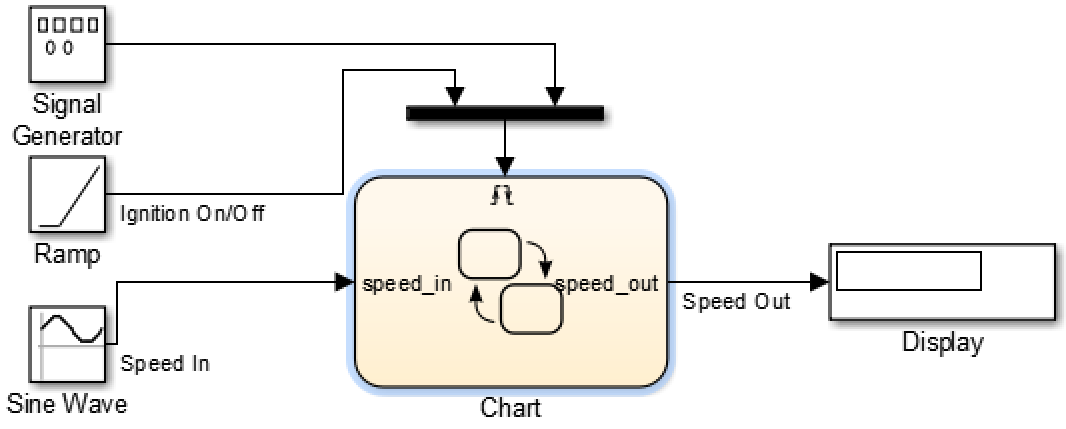


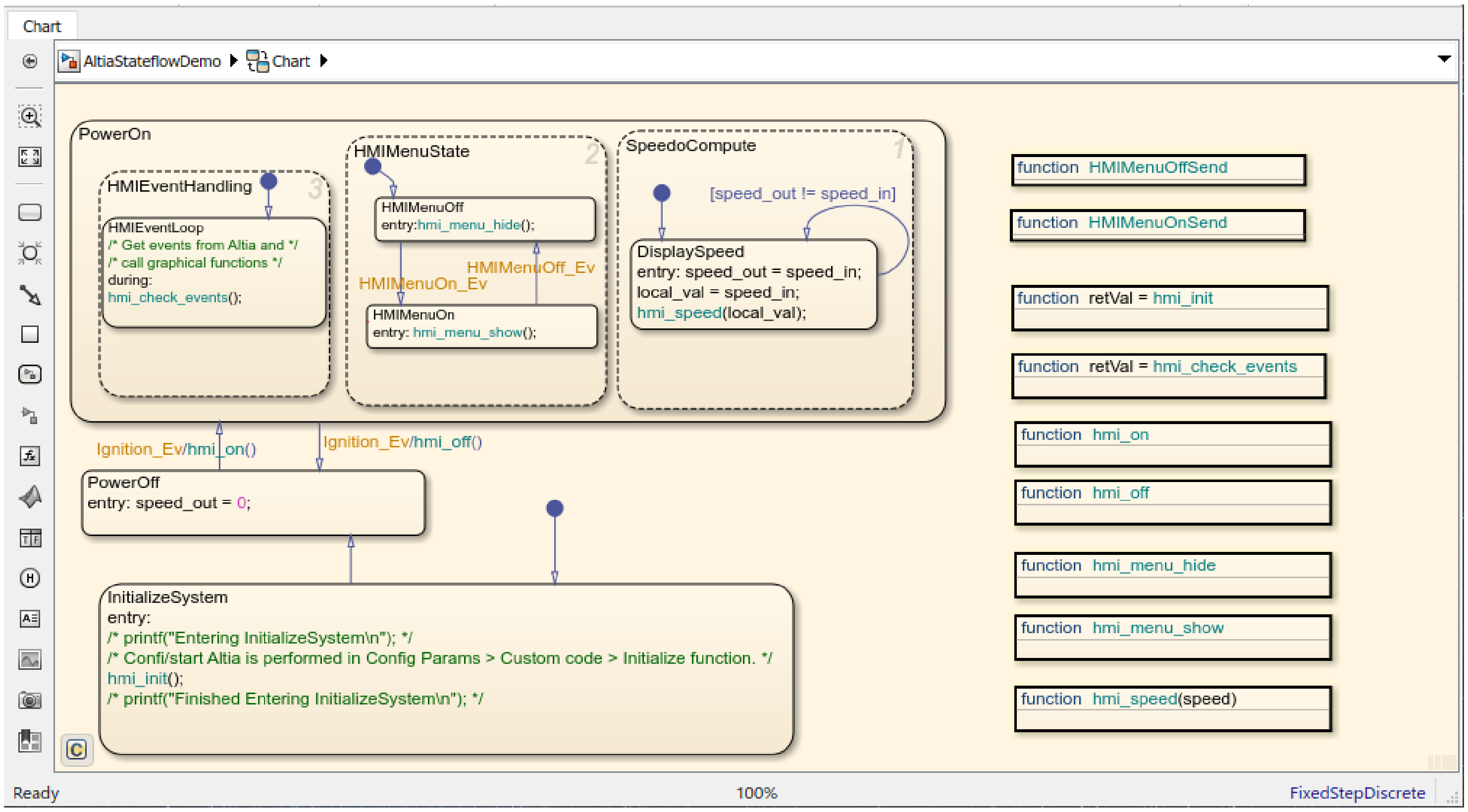
Figure 4: AltiaStateflowNoSimPanel.slx

For these two (2) simple Simulink/Stateflow models, this demonstration shows how to generate code for Windows and compile it using **Simulink Coder**. This process provides an executable version of the Simulink/Stateflow model and Altia HMI on Windows. This executable version does not require installation of MATLAB or Altia Design on the Windows computer.

For transitioning to an embedded target, we will ***not*** generate code for the Simulink portion of the model. The steps of this demonstration ***only*** generate code for the Stateflow chart using Simulink **Embedded Coder**. The Stateflow chart contains logic for interfacing to an entirely independent Altia HMI (Human-Machine Interface) for which Altia DeepScreen code can be generated for an embedded target. For this demonstration, you will generate the code and compile it to run on Windows. The steps are similar to those for generating code to deploy on a real embedded target.

NOTE: The Altia block for Simulink is not portable to embedded systems. Altia recommends using Stateflow® in conjunction with the Altia **C** API to interface to Altia DeepScreen generated code for an embedded target.

The picture below shows the contents of the Stateflow Chart block for our simple Simulink/Stateflow model.



```
HMIEventLoop
/* Get events from Altia and */
/* call graphical functions */
during:
hmi_check_events();

HMIMenuOff
entry:hmi_menu_hide();

HMIMenuOn
entry: hmi_menu_show();

DisplaySpeed
entry: speed_out = speed_in;
local_val = speed_in;
hmi_speed(local_val);

PowerOff
entry: speed_out = 0;

InitializeSystem
entry:
/* printf("Entering InitializeSystem\n"); */
/* Confi/start Altia is performed in Config Params > Custom code > Initialize function. */
hmi_init();
/* printf("Finished Entering InitializeSystem\n"); */

function HMIMenuOffSend
function HMIMenuOnSend
function retVal = hmi_init
function retVal = hmi_check_events
function hmi_on
function hmi_off
function hmi_menu_hide
function hmi_menu_show
function hmi_speed(speed)
```

Figure 5: Stateflow Chart Contents

Important details:

- The chart starts in the `InitializeSystem` state. In this state, the connection to the Altia HMI is already established by code in the **Configuration Parameters > Simulation Target > Insert custom C code in generated: > Initialize function**. You can press `Ctrl+E` to open the **Configuration Parameters** dialog to see this code.
- All Altia API function calls are made through macros. The macro names begin with `MY_` such as `MY_SEND_EVENT()` and `MY_FLUSH_OUTPUT()`. To see some examples, open the Chart `hmi_on` graphical function. The macros are defined in the header file `Include\AltiaHMIMacros.h`.
- The header file `Include\AltiaHMIMacros.h` contains detailed documentation such as an explanation of why macros are used showing usage examples. Open `Include\AltiaHMIMacros.h` in your favorite text editor to learn much more about these macros and how to configure them for different Altia API variations. For example, the macros can be configured for:
  - Unicode or non-Unicode Altia API functions (non-Unicode is the default)
  - Altia Design/Runtime TCP/IP socket (lan) Altia API functions

  - DeepScreen Altia API Server TCP/IP socket (lan) Altia API functions
  - DeepScreen Altia API using animation string names or name ID functions
  - DeepScreen miniGL Altia API which always uses name ID functions
- All sending and receiving of Altia HMI events happen in the Chart **`hmi_`** graphical functions. For example, **`hmi_init()`**, **`hmi_check_events()`**, etc.

5 Files Included in the Demonstration Model

The following chart describes the contents of the Stateflow demo:

File	Description
`AltiaStateflowDemo.slx`	Standalone Stateflow demo project for Altia
`AltiaStateflowNoSimPanel.slx`	Stateflow demo project using Altia blocks
`altiaAPIServer.c`	Main control code that includes Altia API server functionality for Altia DeepScreen
`Include folder`	Required Altia include files such as `altia.h` and `AltiaHMIMacros.h`
`Libs folder`	Required library files include the API TCP/IP socket (lan) library files (`liblan.lib` and `liblanUnicode.lib`), the API DDE library file (`libdde.lib`), and Windows library files (`gdi32.lib`, `user32.lib`, and `winmm.lib`)
`AltiaHMI folder`	Altia GUI project that Stateflow interacts with
`AltiaHMI_3D folder`	Altia GUI project that Stateflow interacts with. This is identical to `AltiaHMI` with added 3D content.
`AltiaSimPanel folder`	Altia project used to control the Stateflow simulation when using the `AltiaStateflowDemo` project

`AltiaHMI.exe`
`AltiaHMI_3D.exe`
`reflash folder`
	Pre-compiled Alita HMI designs and reflash resources

`altiart64.exe`	Altia Runtime executable
`fontModule.dll`	Altia Font Module required for Altia Runtime
`libEGL.dll` `libGLESv2.dll`	OpenGL ES 2.0 emulation DLLs required for Altia Runtime
`Chart_ert_main.c`	Main control code that provides additional logic and data to Stateflow when building an embedded controller
`Chart_ert_main_update.bat`	Batch file used to update `Chart_ert_main.c`, recompile, and run the Stateflow embedded controller

6 Opening a Demonstration Model File for the First Time

Execute the following steps to open the **`AltiaStateflowDemo`** for the first time. This is not required for the **`AltiaStateflowNoSimPanel`** demo.

1. Open the Simulink/Stateflow demonstration model **`AltiaStateflowDemo.slx`** in MATLAB.
2. Make the following change in MATLAB:
   a. At the MATLAB Command Window prompt, execute:
   `altiaLibSetup`
   b. Choose option (2) **LAN**. This configures the Altia block to use the TCP/IP socket (LAN) version of the Altia API. After this step, the Altia block in Simulink indicates it is configured for TCP/IP socket (LAN) communications:

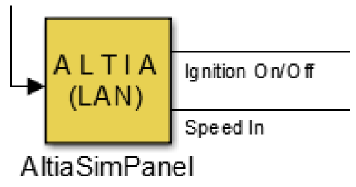


Figure 6: Altia Block Configured for LAN

NOTE: Configuring the Altia block to use the TCP/IP socket (LAN) version of the Altia API is desired for simulating with Altia DeepScreen generated code later in this demonstration. It allows communications with the Altia Runtime executable, and it allows communications with Altia DeepScreen generated code compiled with the DeepScreen Altia API Server TCP/IP socket (LAN) code as a standalone executable.

   c. Save the **`AltiaStateflowDemo.slx`** model file.

7 Running a Demonstration Model in Simulation Mode

1. Open the Simulink/Stateflow demonstration model **`AltiaStateflowDemo.slx`** or **`AltiaStateflowNoSimPanel.slx`** in MATLAB if it is not already opened.

   If you have not performed the steps to opening a demonstration model file for the first time for the model file, do those steps now.

   IMPORTANT: MATLAB's Current Folder must be the folder containing the demonstration model (**`.slx`**) file. Simulink/Stateflow expects to find the required files in MATLAB's current folder when it runs a simulation, generates code, or builds generated code.

   IMPORTANT: For this demonstration, only use Altia API library files (e.g., **`Include\altia.h`** and **`Libs\liblan.lib`**) from the demonstration folder. These files are configured to work properly with the 64-bit version of MATLAB and Microsoft Visual C/C++ 2015 thru 2019. You may copy these Altia API library files to your own project folder for interfacing your own Stateflow model to Altia.

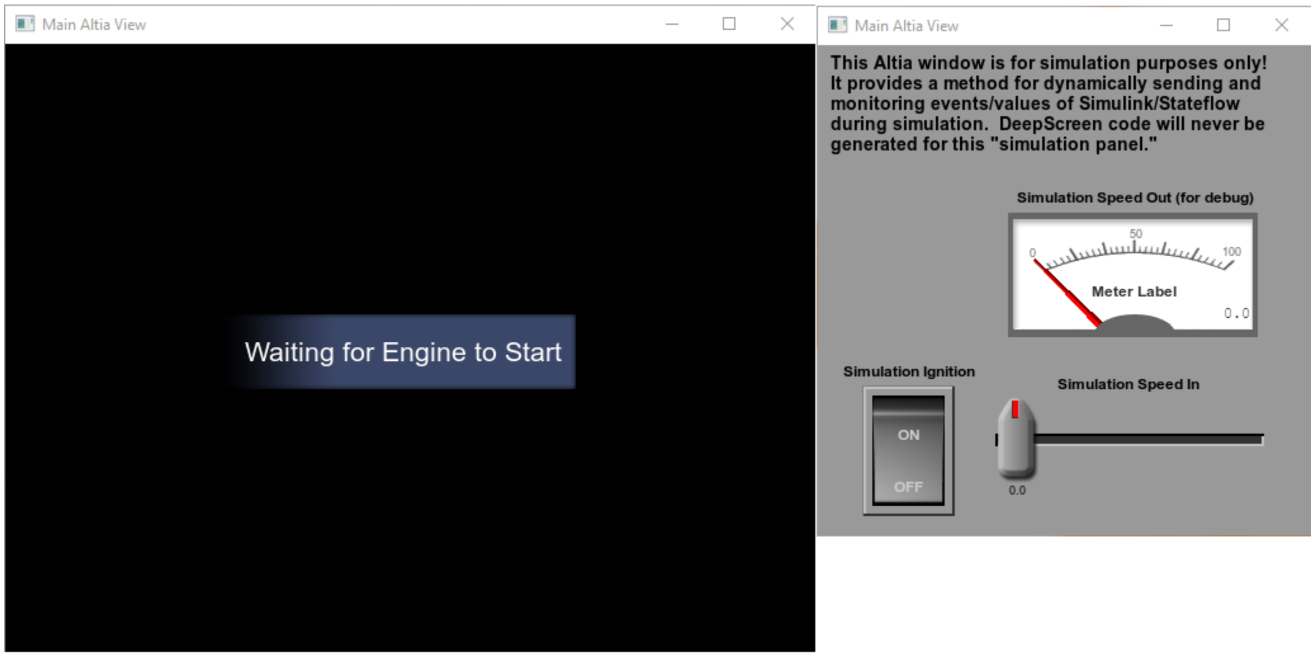
2. From the Simulink **SIMULATION** ribbon, choose **Run**.

   The first run takes extra time to start because Simulink must process the Stateflow chart and build a **`<model_file_name>.mexw64`** for the chart. This file is actually a DLL on Windows. MATLAB uses code generation and Visual Studio to compile and link the file. This compile and link should be successful because the required Altia API library files are in the **`Include`** and **`Libs`** folders with the demonstration model (**`.slx`**) file.

   When the simulation begins execution, code for the Simulink Altia block starts an Altia Runtime session with the **`AltiaSimPanel`** Altia project, and code generated for the Stateflow chart starts a completely independent Altia Runtime session with the **`AltiaHMI`** Altia project.

   NOTE: If the **`AltiaSimPanel`** Altia project is already opened in the Altia Design editor from double-clicking on the Altia block in Simulink, the simulation will ***not*** connect to it. This is a side-effect of setting up the Altia block to use the TCP/IP socket (LAN) version of the Altia API. When the Altia block is using LAN, it always starts an Altia Runtime window for the Altia block when the simulation starts. Configuring the Altia block to use LAN is required for simulating with Altia DeepScreen generated code later in this demonstration.

The Altia Runtime windows open as shown in the picture below. If you are simulating for the `AltiaStateflowNoSimPanel.slx` model file, there is no Altia Runtime session for the `AltiaSimPanel` Altia project. Only the Altia Runtime window for the `AltiaHMI` Altia project exists, and it immediately displays a speedometer and **Press Here for More…** text. The speedometer needle is cycling automatically because it is connected to a Simulink Sine Wave block.



This Altia window is for simulation purposes only! It provides a method for dynamically sending and monitoring events/values of Simulink/Stateflow during simulation. DeepScreen code will never be generated for this "simulation panel."

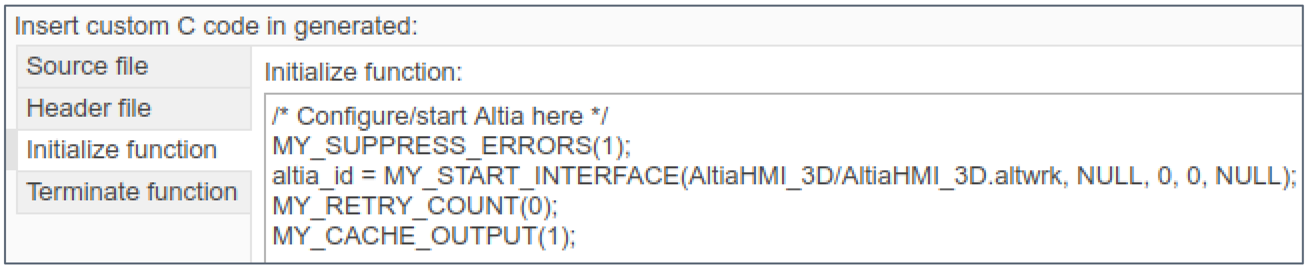
Figure 7: Altia Runtime Windows in Simulation Mode

3. Click the Simulation Ignition **ON/OFF** toggle button in the Altia Simulation-only Panel window.

   The Altia Runtime window for `AltiaHMI` displays a speedometer and **Press Here for More…** text.

   - Changing the Simulation **Speed In** slider changes the position of the needle in the speedometer.
   - Clicking on the **Press Here for More…** text in the Altia Runtime `AltiaHMI` window changes the state of the Altia HMI to show a menu with selectable items (they just highlight).
   - Clicking on the **Press Here to Exit** item closes the menu.
   - If the Stateflow chart content is showing in Simulink, you will see the chart animate to indicate state changes and currently active states. This Stateflow chart animation is a powerful feature of simulation mode. It helps with debugging issues in the Stateflow chart logic prior to generating standalone code.

4. To stop the simulation, choose **SIMULATION > Stop** from the Simulink or Stateflow editor window.
    - The Altia Runtime `AltiaHMI` window closes automatically when the simulation is stopped because the Terminate function of the model has instructions to do this (as described in a later chapter about Simulink Configuration Parameters).
    - The Altia Runtime `AltiaSimPanel` window must be closed manually (from the **X** button in the top-right corner of the window).
5. Now have some fun! Press `Ctrl+E` to open the **Configuration Parameters** dialog and select **Simulation Target** on the left side. In the **Insert custom C code in generated:** section, select the **Initialize function**. Edit the `MY_START_INTERFACE()` line to open `AltiaHMI_3D/AltiaHMI_3D.altwrk` like this:



```
/* Configure/start Altia here */
MY_SUPPRESS_ERRORS(1);
altia_id = MY_START_INTERFACE(AltiaHMI_3D/AltiaHMI_3D.altwrk, NULL, 0, 0, NULL);
MY_RETRY_COUNT(0);
MY_CACHE_OUTPUT(1);
```

Figure 8: `MY_START_INTERFACE` *Line to Start the 3D Altia Design Simulation*

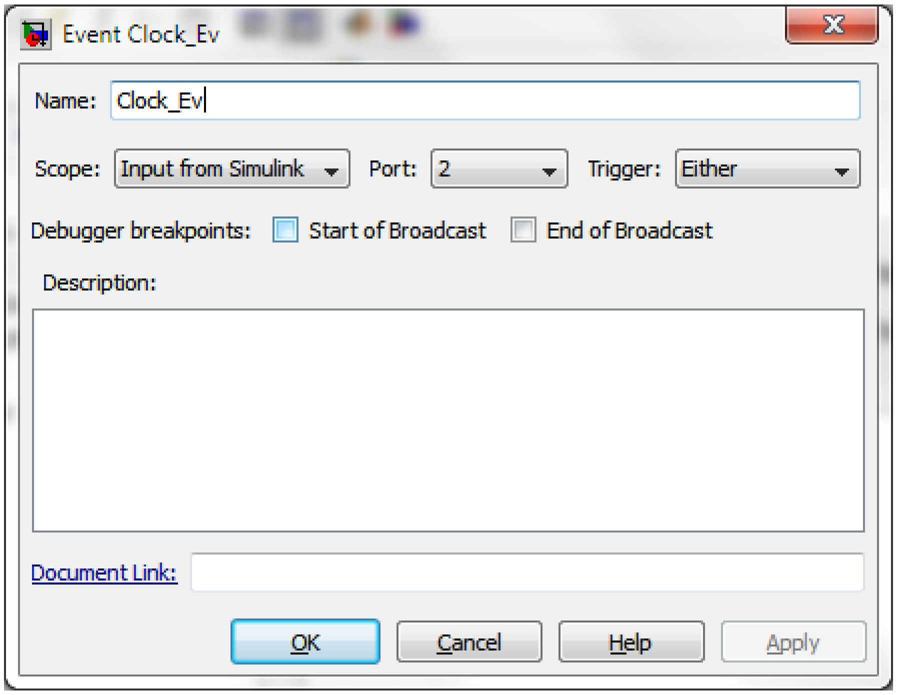
6. Press the **OK** button in the **Configuration Parameters** dialog to apply the changes and it also closes the dialog.
7. From the Simulink **SIMULATION** ribbon, choose **Run**. There will be a few seconds delay before the simulation starts because a rebuild of the `.mexw64` is required by the change to the Stateflow chart.
8. When the simulation starts running, the Altia Runtime window shows the `AltiaHMI_3D` Altia project. It has some simple 3D content added, a blue cube, red sphere, and green pyramid. This is just to demonstrate 3D objects in the Altia HMI project.

8 Configuring Stateflow for Use in a Simulink Model

Stateflow has no notion of time, so we need a periodic event to trigger it for simulation mode. In this demo, we will use a **Signal Generator** block in Simulink.

The Signal Generator block is connected to the `Clock_Ev` event in the Stateflow model.

1. To display the **Event Clock_Ev** properties dialog, open the **Model Explorer** from Simulink or Stateflow.
2. Select the **Chart** in the **Model Hierarchy** pane.
3. Right click on the `Clock_Ev` event in the **Contents of:** pane, and choose **Properties...**



Event Clock_Ev

Name: Clock_Ev

Scope: Input from Simulink

Port: 2

Trigger: Either

Debugger breakpoints: ☐ Start of Broadcast ☐ End of Broadcast

Description:

Document Link:

Figure 9: Event Clock_Ev Properties

4. Change the event properties to match the figure above. The **Scope:** must be **Input from Simulink** arriving on **Port: 2** of the Stateflow Chart block.

   Port 1 of the Stateflow Chart block is associated with the event `Ignition_Ev`. In Simulink, it is driven by an output from the Altia block. For both events (`Clock_Ev` and `Ignition_Ev`), the Trigger type is `Either`, so a rising value from <= 0 to positive triggers the event or a falling value from >= 0 to negative also triggers the event.

A Stateflow block in Simulink only has one (1) input pin for incoming events. If there is more than one (1) incoming event (such as in our demonstration), use a `Mux` block in Simulink to collect the multiple input events into an array for the Stateflow block.

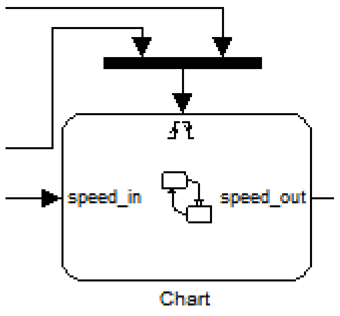


Figure 10: Stateflow Block Input

9 Getting Events from the Altia HMI Design into Stateflow

For a touchscreen HMI, we want to receive events from the Altia HMI design. For this demonstration, receiving and processing events from the HMI is implemented in the Stateflow graphical function `hmi_check_events`. A graphical function is added to a Stateflow chart with the ***fx*** tool.

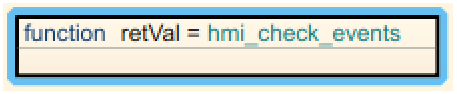


Figure 11: Stateflow Graphical Function hmi_check_events

Open the `hmi_check_events` graphical function to see the implementation details.

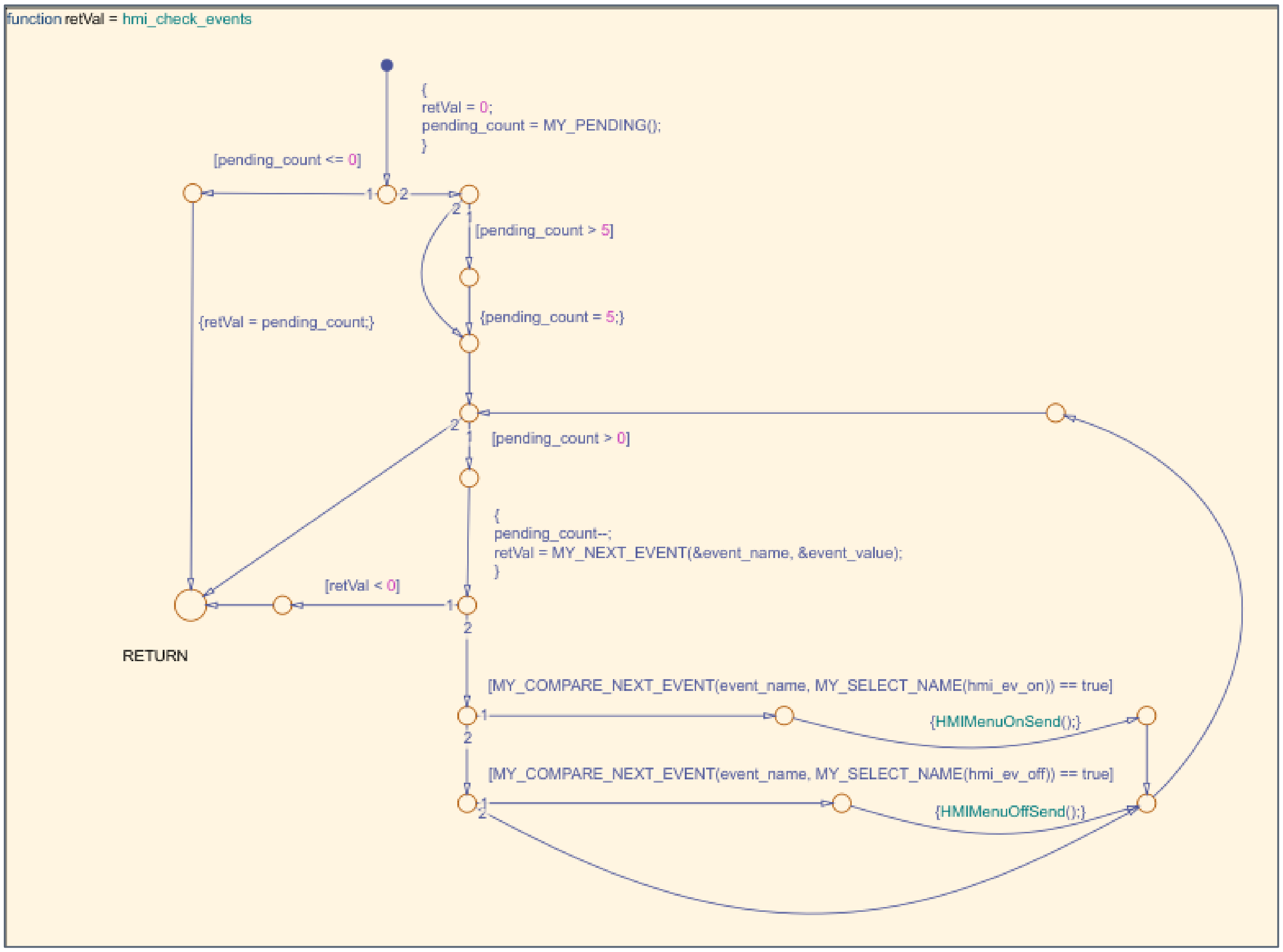

function retVal = hmi_check_events
{
retVal = 0;
pending_count = MY_PENDING();
}
[pending_count <= 0]
[pending_count > 5]
{pending_count = 5;}
{retVal = pending_count;}
[pending_count > 0]
{
pending_count--;
retVal = MY_NEXT_EVENT(&event_name, &event_value);
}
[retVal < 0]
RETURN
[MY_COMPARE_NEXT_EVENT(event_name, MY_SELECT_NAME(hmi_ev_on)) == true]
{HMIMenuOnSend();}
[MY_COMPARE_NEXT_EVENT(event_name, MY_SELECT_NAME(hmi_ev_off)) == true]
{HMIMenuOffSend();}

Figure 12: Graphical Function `hmi_check_events` *Processes Events from the Altia HMI*

This graphical function uses `Include\AltiaHMIMacros.h` macros to check for pending events with macro `MY_PENDING()`. It gets the next available event with `MY_NEXT_EVENT()`. It compares the next event with expected events `hmi_ev_on` and `hmi_ev_off` using `MY_COMPARE_NEXT_EVENT()`. Open `Include\AltiaHMIMacros.h` in your favorite text editor for macro descriptions and usage examples.

NOTE: For an Altia HMI design with many events, the above logic could get very complicated. It may be more elegant to use a different approach. Please consult with Stateflow experts from MathWorks for the best approach.

10 Configuring Simulink Parameters for Simulation Execution with Altia

For our Stateflow chart to execute in Simulation mode, Stateflow must:

- Resolve the `MY_` macro usages.
- Have definitions for variables **`altia_id,`** **`event_value`**, and **`event_name`**.
- Link with the Altia API object libraries and extra Visual Studio system libraries.

To accomplish these items, there are required settings in the **Configuration Parameters > Simulation Target** fields. Later, we will see that these same settings also apply for using Simulink Coder to create an executable to run standalone on Windows.

1. Open the **Configuration Parameters** dialog from a Simulink or Stateflow editor window with **Ctrl+E**.
2. Select **Simulation Target** from the left-side pane.
3. Review and apply the following settings as appropriate:

IMPORTANT: For the macros from **`AltiaHMIMacros.h`** to work successfully, the following checkbox near the top-right of the dialog must be **disabled**.

- For R2018a and newer: **Import custom code** must be ***disabled*** as shown in this picture:

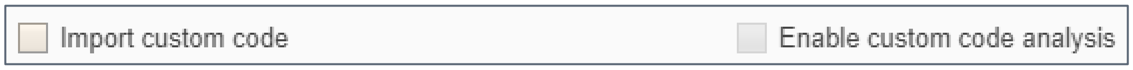

Import custom code
Enable custom code analysis

Figure 13: Disable Configuration Parameters > Import custom code

- For R2017b, maybe R2017a, and R2016a/b: **Parse custom code symbols** must be ***disabled***.

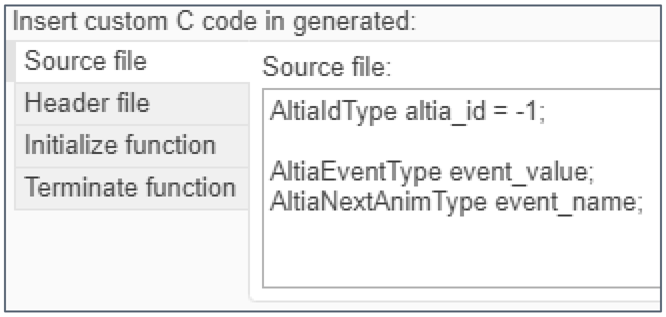
  If one of these options is enabled, the parser for Simulink/Stateflow coder will ***NOT*** permit a name like **`hmi_power`** in **`MY_SEND_EVENT(hmi_power, 1)`** because **`hmi_power`** does not have a definition. The parser is not able to apply the macro definitions from **`AltiaHMIMacros.h`** during its parse, causing errors like the following during a build:

```
Unresolved data ' hmi_power' in '{Something);}'.
```

- Required custom settings for **Simulation Target > Insert custom C code in generated: > Source file**:

```
AltiaIdType altia_id=-1;
```

```
AltiaEventType event_value;
AntiaNextAnimType event_name;
```



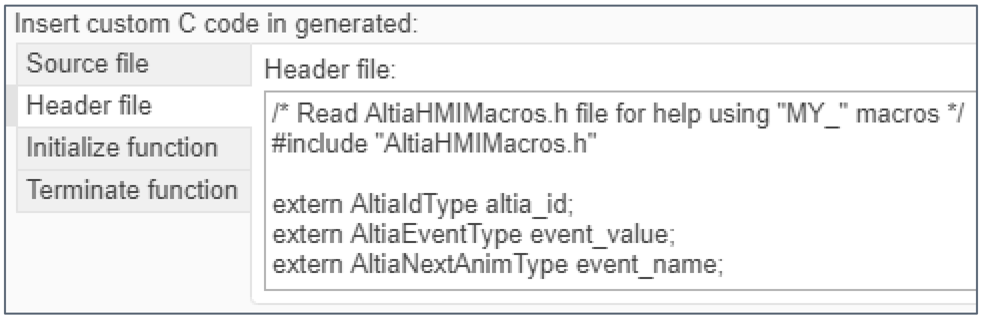
```
AltiaIdType altia_id = -1;

AltiaEventType event_value;
AltiaNextAnimType event_name;
```

Figure 14: Simulation Custom C Code Source File Content

These are variables referenced in **`AltiaHMIMacros.h`** and also in the Stateflow model states and graphical functions. The content in the Source file section becomes **C** code in one of the Simulink/Stateflow generated **C** files.

- Required custom settings for **Simulation Target > Insert custom C code in generated: > Header file**:

```
/* Read AltiaHMIMacros.h file for help using "MY_" macros */
#include "AltiaHMIMacros.h"

extern AltiaIDType altia_id;
extern AltiaEventType event_value;
extern AltiaNextAnimType event_name;
```



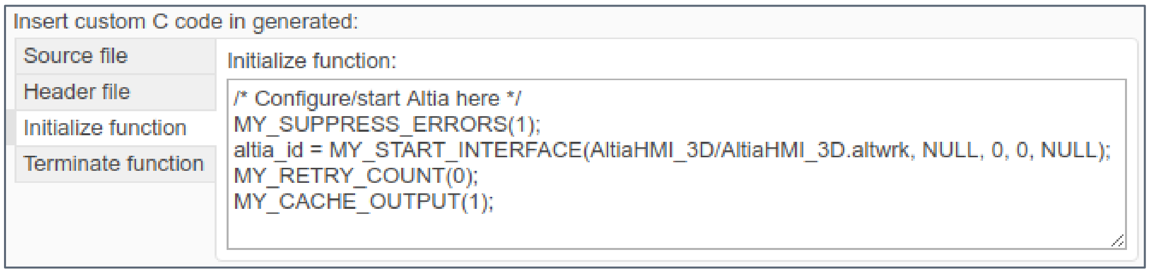
```
/* Read AltiaHMIMacros.h file for help using "MY_" macros */
#include "AltiaHMIMacros.h"

extern AltiaIdType altia_id;
extern AltiaEventType event_value;
extern AltiaNextAnimType event_name;
```

Figure 15: Simulation Custom C Code Header File Content

This is where we include **`AltiaHMIMacros.h`** and provide extern declarations for the variables defined in the Source file section. The content in the Header file section is included as a header file in each Simulink/Stateflow generated **C** file, which allows us to use the macros and variables in the Stateflow model without compiler warnings or errors.

- Required custom settings for the **Simulation Target > Insert custom C code in generated: > Initialize function** section (using `AltiaHMI_3D` project in this example):

```
/* Configure/start Altia here */
MY_SUPPRESS_ERRORS(1);
altia_id = MY_START_INTERFACE(AltiaHMI_3D/AltiaHMI_3D.altwrk, NULL,
0, 0, NULL);
MY_RETRY_COUNT(0);
MY_CACHE_OUTPUT(1);
```

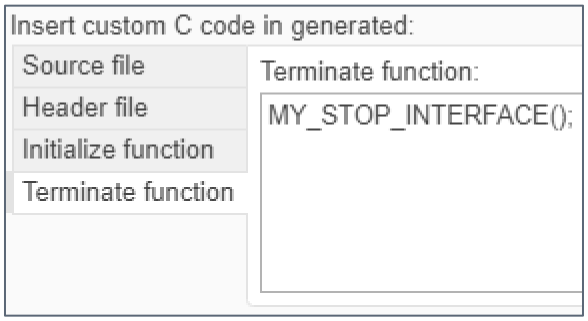


```
/* Configure/start Altia here */
MY_SUPPRESS_ERRORS(1);
altia_id = MY_START_INTERFACE(AltiaHMI_3D/AltiaHMI_3D.altwrk, NULL, 0, 0, NULL);
MY_RETRY_COUNT(0);
MY_CACHE_OUTPUT(1);
```

Figure 16: Simulation Custom C Code Initialize Function Content

- Required custom settings for **Simulation Target > Insert custom C code in generated: > Terminate function**:

```
MY_STOP_INTERFACE();
```



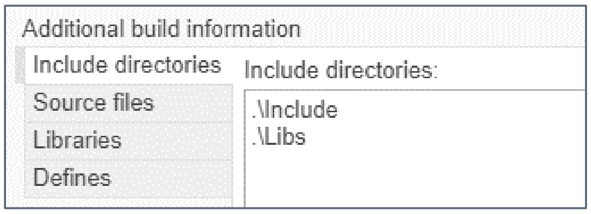
```
MY_STOP_INTERFACE();
```

Figure 17: Simulation Custom C Code Terminate Function Content

`MY_STOP_INTERFACE()` is a macro call to stop the Altia HMI. This call closes the Altia HMI window when a simulation is stopped. This macro call is recommended for typical models, but not absolutely required.

- Required custom settings for **Simulation Target > Additional build information: > Include directories**:

```
.\Include
.\Libs
```

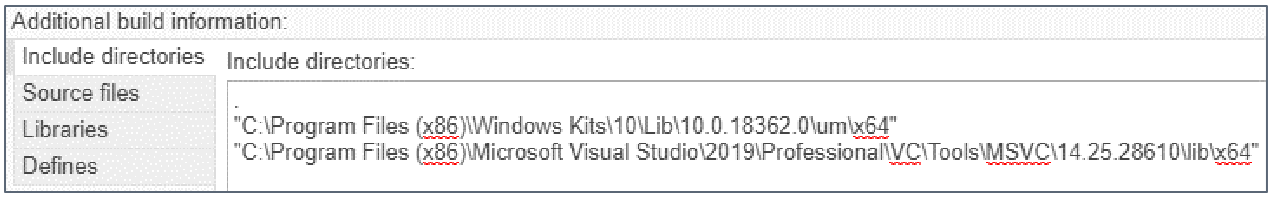

Additional build information
Include directories | Source files | Libraries | Defines

Include directories:
```
.\Include
.\Libs
```

Figure 18: Simulation Additional Build Info Include Directories

In the above picture, the settings instruct a Simulink/Stateflow build to search the `.\Include` and `.\Libs` folders in the MATLAB current folder for header files, such as `AltiaHMIMacros.h` and `altia.h`, and for library files, such as `liblan.lib` and `user32.lib`.

NOTE: The **Include directories** list is also a list of directory paths to object libraries that are referenced in the **Libraries** section. For this demonstration, all object libraries in the **Libraries** section are in the demonstration `.\Libs` folder. Other directory paths can be added. For example, if Windows 10 SDK 10.0.18362.0 and Visual Studio 2019 are installed and you would like a build to search for Windows specific object libraries in the Windows 10 SDK 64-bit folder and Visual Studio 2019 libs 64-bit folder, the **Include** directories list could look like the next picture. Please note the required double-quotes (") around the extra directories because they have spaces in folder names.

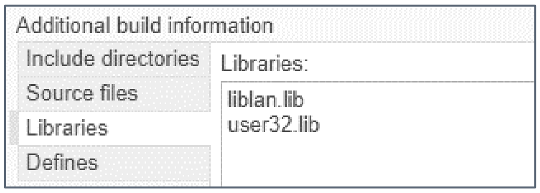

Additional build information:
Include directories | Source files | Libraries | Defines

Include directories:
```
.
"C:\Program Files (x86)\Windows Kits\10\Lib\10.0.18362.0\um\x64"
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.25.28610\lib\x64"
```

Figure 19: Simulation Example Include Directories with Extra Library Paths

- There are no required custom settings for the **Simulation Target > Additional build information: > Source files** section.
- Required custom settings for **Simulation Target > Additional build information: > Libraries**:

```
liblan.lib
user32.lib
```


Additional build information

Include directories | Source files | Libraries | Defines

Libraries:

liblan.lib
user32.lib

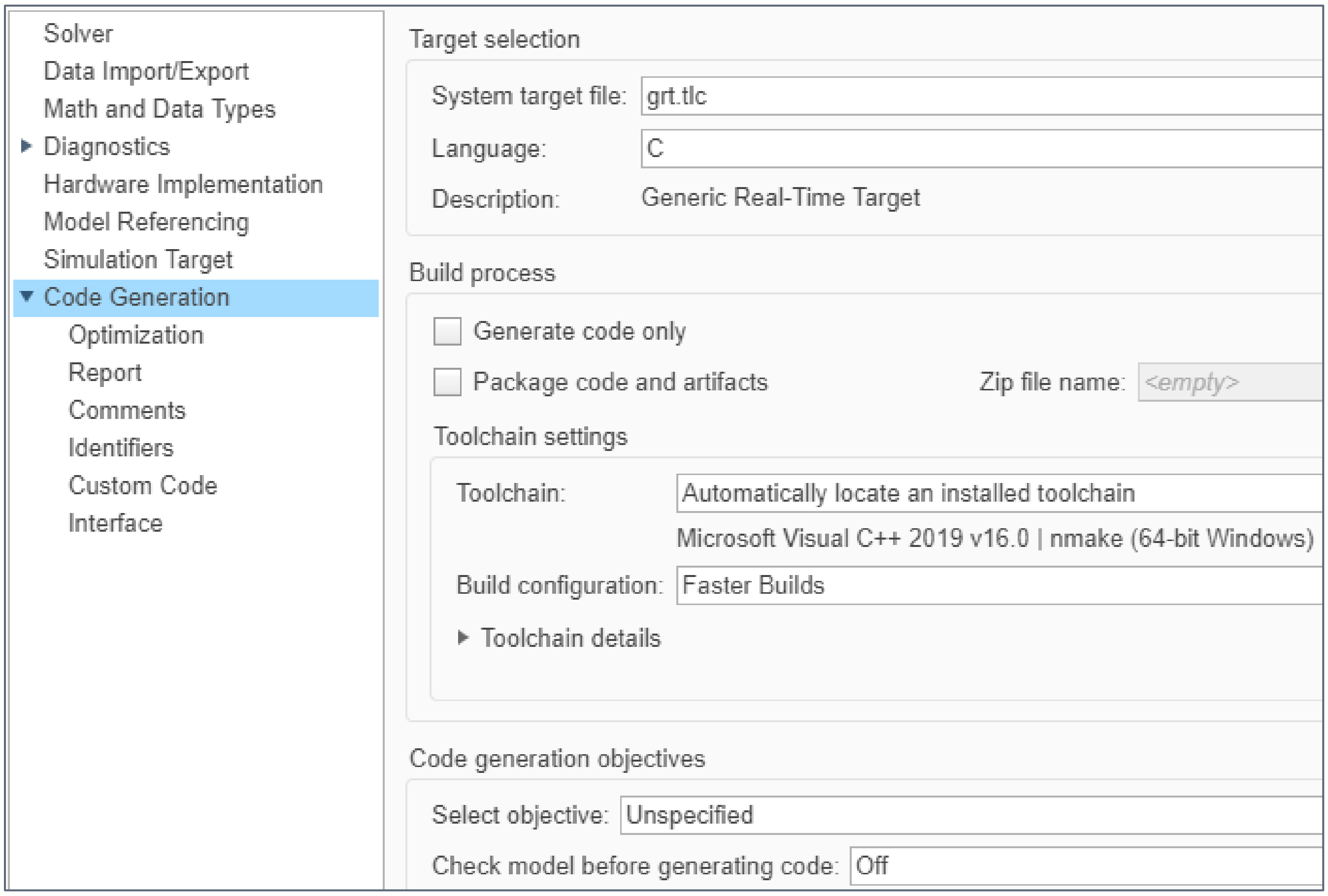
Figure 20: Simulation Additional Build Info Libraries

The `liblan.lib` file is the TCP/IP socket (lan) version of the Altia API library. It provides the Windows pre-compiled `C` code for Altia API functions used by the macros in `AltiaHMIMacros.h`. The `AltiaHMIMacros.h` file includes `altia.h`, which is the `C` header file for `liblan.lib`.

IMPORTANT: For this demonstration, only use Altia API library files from the demonstration `.\Include` and `.\Libs` folders. See the next chapter for complete details.

In the **Libraries** section, `user32.lib` is a Microsoft Visual Studio system library compatible with Visual Studio 2015 thru 2019. To make the demonstration more portable, there are copies of these libraries in the `.\Libs` folder of the demonstration. As a result, only the library names are required, not the full Visual Studio installation path for the libraries.

4. While the **Configuration Parameters** dialog is still open, select **Code Generation** from the left-side pane.
5. Verify the proper **Target Selection**, **Build process**, and **Code Generation objectives**. The settings in the figure below are for MATLAB R2020a using Visual Studio 2019. Your dialog may look slightly different for a different version of MATLAB and/or a different version of Visual Studio.


Solver
Data Import/Export
Math and Data Types
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Optimization
Report
Comments
Identifiers
Custom Code
Interface

Target selection

System target file: grt.tlc
Language: C
Description: Generic Real-Time Target

Build process

Generate code only
Package code and artifacts Zip file name: <empty>

Toolchain settings

Toolchain: Automatically locate an installed toolchain
Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
Build configuration: Faster Builds
Toolchain details

Code generation objectives

Select objective: Unspecified
Check model before generating code: Off

Figure 21: Configuration Parameters Code Gen Settings

6. Choose **Code Generation > Custom Code** from the left-side pane.
7. Confirm the option **Use the same custom code settings as Simulation Target** is set.

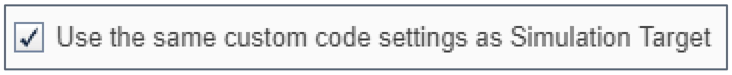
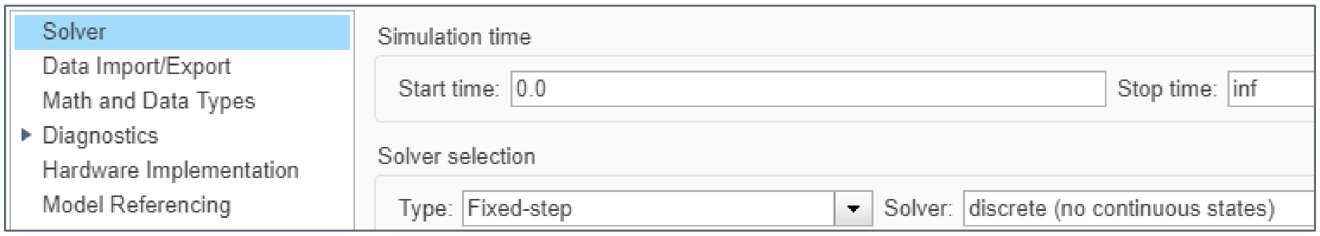

☑ Use the same custom code settings as Simulation Target

Figure 22: Configuration Parameters Code Gen > Custom Code Settings

8. While the **Configuration Parameters** dialog is still open, select **Solver** from the left-side pane.
9. Verify the following settings on the Solver screen:
   - **Start time:** is **0.0**
   - **Stop time:** is **inf**
   - **Solver selection Type:** is **Fixed-step**
   - **Solver section Solver:** is **discrete (no continuous states)**



Simulation time

Start time: 0.0 Stop time: inf

Solver selection

Type: Fixed-step Solver: discrete (no continuous states)

Figure 23: Configuration Parameters Solver Settings

11 Only Use Altia API Library Files from the Demonstration Folder

IMPORTANT: For this demonstration, only use Altia API library files from the demonstration `.\Include` and `.\Libs` folders.

These demonstration folders contain integer Altia API files (such as `altia.h` and `liblan.lib`) configured to work properly with 64-bit versions of MATLAB and Microsoft Visual Studio C/C++ 2010 through 2019. You may copy these integer Altia API library files to your own project folder for interfacing your own Stateflow model to Altia.

If the Simulink model contains an Altia block (as it does in this demonstration), the `.\Include` and `.\Libs` folders must refer to locations containing an integer version of the Altia API header file `altia.h` and Altia API library `liblan.lib` (as is the case for this demonstration). Assuming you want to use Simulink Coder (i.e., Code Generation) to generate `C` source code from the model, the Stateflow generated code must be compatible with the Simulink generated code. The generated code for the Altia block only supports compiling and linking with the integer version of the Altia API. This requires that the Stateflow chart use the integer version of the Altia API header file and library. If there is no Altia block in the Simulink model, this is not a restriction.

The Stateflow chart can use a float version of the Altia API header file and library if there is no Altia block in the Simulink model, but float versions are not provided with this demonstration. Float versions are available from the `bin\libfloat\ms64` (64-bit) folder of the Altia Connection for Simulink installation. To view this installation folder, start by entering `%APPDATA%` in the Address bar of Windows Explorer and then browse to `MathWorks\MATLAB Add-Ons\Toolboxes\Altia Connection\bin\libfloat\ms64`.

12 Transitioning to a C Code Build from Simulation Mode

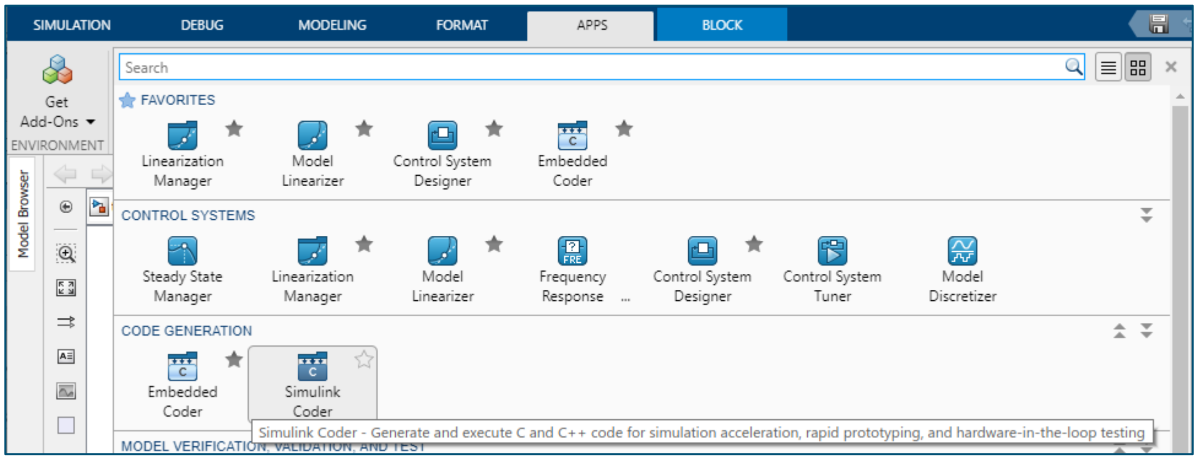
It is typical to migrate from simulation mode to a Windows desktop executable. This is accomplished by doing a **C** Code Build using Simulink Coder.

The benefit of this **C** Code Build is that it is much like running in simulation mode (it opens the Altia simulation panel window as well as the Altia HMI window), and it is portable for running on a different Windows computer that does not have MATLAB Simulink/Stateflow installed. Even if a computer has MATLAB Simulink/Stateflow installed, there is no need to start it.

NOTE: For Simulink code compilation, the Windows computer primary drive (usually the C: drive) must be configured to support short file names (also referred to as 8.3 file names). If it is not, there will be a compile error similar to:
`NMAKE : fatal error U1073: don't know how to make 'Add-Ons\Toolboxes\Altia'`
This error is caused by spaces in the folder path `%APPDATA%\MathWorks\MATLAB Add-Ons\Toolboxes\Altia Connection`. The MathWorks Help Center article https://www.mathworks.com/help/rtw/ug/enable-build-process-for-folder-names-with-spaces.html describes how to enable short file names. Especially the section ***Troubleshooting Errors When Folder Names Have Spaces*** is important. The `MATLAB Add-Ons` folder and its contents must be recreated after enabling short file names. An easy way to do this is to make a copy of the `MATLAB Add-Ons` folder (`Ctrl+C`, then `Ctrl+V`), delete the original folder, rename the copy to `MATLAB Add-Ons`.

To generate a **C** Code Build:

1. In a Simulink or Stateflow window, choose the **APPS** ribbon.
2. From the APPS dropdown list, choose the **Simulink Coder** option.



Simulink Coder - Generate and execute C and C++ code for simulation acceleration, rapid prototyping, and hardware-in-the-loop testing

*Figure 24: Choose Simulink Coder from **APPS** Dropdown List*

3. Select the new **Simulink C CODE** ribbon if it is not already selected:

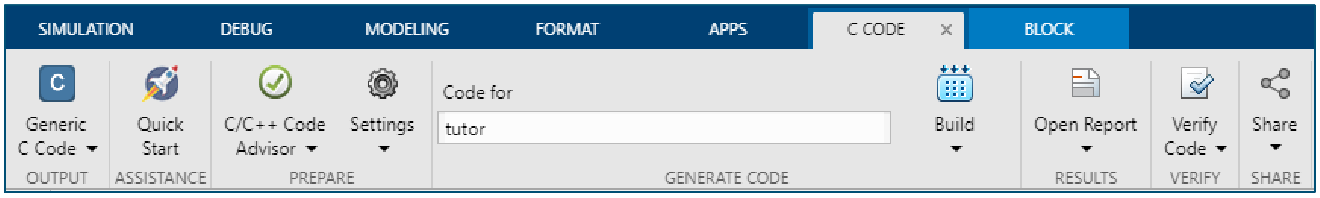


Figure 25: Simulink C CODE Ribbon

4. Choose **Build > Build** to generate code and build an executable.

   NOTE: If switching between building the AltiaStateflow and AltiaStateflowNoSimPanel projects, delete the `slprj` folder before building.

5. During this step, Simulink opens an Altia Runtime window for each Altia block in the model. For example, a **Main Altia View** window opens for the `AltiaSimPanel` block in `AltiaStateflowDemo.slx`. Close the window from the close icon **X** in the top-right corner of the window.

6. If the code generation or build fails, Simulink is good at displaying diagnostic information. Failures show in a **Diagnostic Viewer** window. Resolve them and try again.

   After the code generates and compiles successfully, a new executable file `<Model_Name>.exe` resides in the current project folder (for our demonstration, the file is `AltiaStateflowDemo.exe` or `AltiaStateflowNoSimPanel.exe`). The project folder is the folder containing the Simulink model file and Altia HMI files.

7. This executable runs standalone from Simulink. Open a Windows Explorer window, browse to the project folder, and double-click on the executable (e.g., `AltiaStateflowDemo.exe`).

   The executable opens a Windows Command Prompt window because it is compiled as a console application. Very soon after the Command Prompt window opens, the Altia Runtime windows open. If executing `AltiaStateflowDemo.exe`, there is an Altia Runtime **Main Altia View** window for the `AltiaSimPanel` Altia project and another for the `AltiaHMI` (or `AltiaHMI_3D`) Altia project. You can interact with these windows the same way you interact with them in Simulation mode.

   NOTE: Closing one or both Altia Runtime windows does not stop the executable. It continues to run until the Command Prompt window is closed by clicking on the **X** button in its top-right corner. Similarly, closing the Command Prompt window does not close any Altia Runtime windows. You must manually close an Altia Runtime window by clicking on the **X** button in its top-right corner.

You can copy the standalone files to a different folder on the current Windows computer or a folder on a different Windows computer and run the simulation standalone. For a `C` Code Build of `AltiaStateflowDemo.slx`, the standalone files are:

```
AltiaStateflowDemo.exe
AltiaHMI or AltiaHMI_3D directory
AltiaSimPanel directory
altiart64.exe
fontModule.dll
libEGL.dll
libGLESv2.dll
```

13 Using Altia DeepScreen Generated Code with Simulation and C Code Build

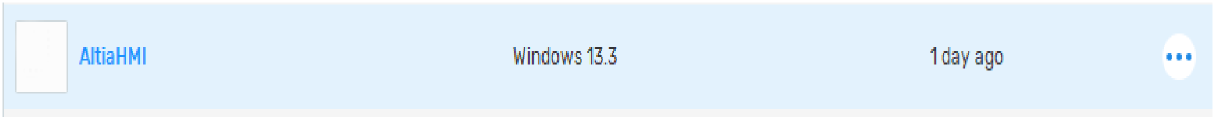
For developing an Altia HMI to run on an embedded device, it is typical to run the Altia HMI as a DeepScreen generated code executable on Windows instead of Altia Runtime. This allows for verification of the Altia HMI DeepScreen generated code with the Stateflow model on Windows. Issues can be much easier to debug and resolve on Windows prior to generating code, compiling, downloading, and running on an embedded target.

NOTE: Altia DeepScreen code generation for Windows requires a license for Altia Design 13.2 or newer and a license for the DeepScreen Windows Intel x86 Port for OpenGL ES 2.0 Target. If you do not have the required licenses, you can still follow along using the pre-compiled DeepScreen executables provided with this demonstration. Skip the DeepScreen code generation and build steps of this chapter and go immediately to the section for changing the Stateflow model to start a DeepScreen executable.

13.1 Generate and Build DeepScreen Windows Code for the Altia HMI

1. Open the `AltiaHMI` or `AltiaHMI_3D` project in Altia Design from Windows Explorer by double-clicking on the `.altwrk` file in the respective project folder. ***If the project does not open, read the following NOTE for help***.

   NOTE: The `AltiaHMI.altwrk` and `AltiaHMI_3D.altwrk` projects were originally saved in a project version that might be different from the project version(s) you are licensed to use. For example, the `.altwrk` might be a **Windows 13.3** project, but you might only have a license for **Windows 13.2**. If this happens, a project is created in the Altia Launcher, but you cannot open it. Here is an example picture of an `AltiaHMI` project for **Windows 13.3** in the Altia Launcher.

   AltiaHMI   Windows 13.3   1 day ago   •••

   To change this `AltiaHMI` project to a project version that you are licensed to use, click on the 3 ellipses menu at the far right •••, choose the **Reassign Template...** option, select an available **Windows** project template from the list, and press the **Assign** button. The list only shows project templates you are licensed to use. Now you can double-click on the `AltiaHMI` project in the Altia Launcher **Projects** pane or double-click on the `AltiaHMI.altwrk` file from Windows Explorer to open the project in Altia Design. You must choose a **Windows 13.X** or **Windows Cloud 13.X** project template for this demonstration. A **Windows MiniGL 13.X**, **Windows MinGL Cloud 13.X**, or an embedded target template is not supported for this demonstration.

2. From the Altia Design **Code Generation** ribbon, choose **Build Prototype**. Several windows pop up and then close to generate and build the Windows executable.
   - For `AltiaHMI`, the executable is `AltiaHMI.exe` in the `AltiaHMI\out` folder.
   - For `AltiaHMI_3D`, the executable is `AltiaHMI_3D.exe` in the `AltiaHMI_3D\out` folder.
3. As a sanity check, double-click on the new executable `AltiaHMI.exe` or `AltiaHMI_3D.exe` from a Windows Explorer window.

   The executable starts and opens a window for the Altia HMI to show in its initial state. If you click in the window, there is no response because the Simulink/Stateflow logic code is not running.
4. Click on the **X** button in the window's top-right corner. This closes the window and stops the executable.

13.2 Use the Altia API Server Code for DeepScreen Code in a Simulation or C Code Build

The Altia HMI projects take advantage of the ability to include a custom `main()` function when building a DeepScreen executable. This is done by including a `C` source code file by the same name as the Alita project in the `<ALTIA_project>\code` folder. For example, for the `AltiaHMI` project, the source code file is `AltiaHMI\code\AltiaHMI.c`.

For the Stateflow demo, the `AltiaHMI.c` and `AltiaHMI_3D.c` files are copies of the Altia API Server code. This is Altia application `C` code that opens a TCP/IP socket (lan) as a server. It can send/receive messages with other programs that are linked with a TCP/IP socket (lan) version of the Altia API, such as `liblan.lib` in the demonstration folder. The other programs can be Simulink/Stateflow in simulation mode or a Simulink Coder `C` Code Build standalone executable. The other programs send/receive events to/from the DeepScreen executable in the same way they send/receive events with Altia Runtime.

For interfacing the DeepScreen generated Windows code to Simulink/Stateflow simulations and Simulink Coder `C` Code Build standalone executables, it is best practice to compile and link the DeepScreen generated Windows code with the Altia API Server code, so it runs on its own. If there is an Altia block in the Simulink model, the Altia block is not able to open its associated Altia Runtime window for a Simulink Coder `C` Code Build executable if the DeepScreen generated

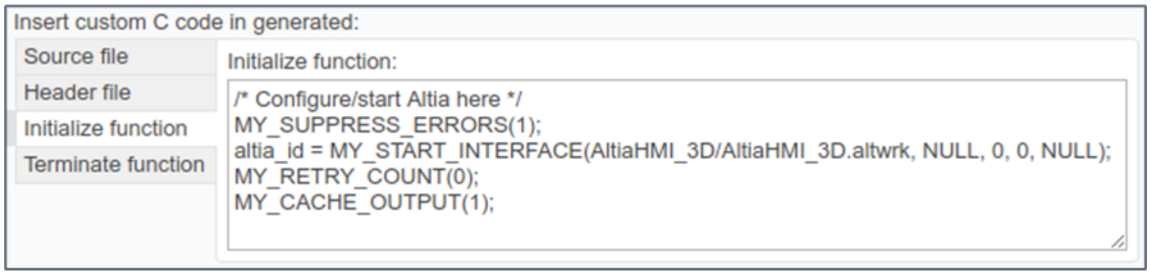
Windows code is linked directly into the executable. In our demonstration, the Altia block opens the Altia Simulation-only Panel window for the **`AltiaSimPanel`** Altia project.

To interface DeepScreen generated Windows code to Simulink/Stateflow simulations and/or Simulink Coder **`C`** Code Builds, copy the **`C`** file **`altiaAPIServer.c`** to the **`<ALTIA_project>\code`** folder, rename it to match the same name of the Altia HMI project, and then, rebuild the Altia project.

13.3 Change the Stateflow Model to Start a DeepScreen Executable

After completing the steps in Section 13.3, the **`<Altia_project>\out`** folder now has a new **`AltiaHMI.exe`** or **`AltiaHMI_3D.exe`** executable file and an **`<Altia_project>\out\reflash`** folder. If you do not have Altia Design 13.2 or newer or a DeepScreen Intel x86 OpenGL ES 2.0 Windows code generator, you can continue by using the pre-built versions of **`AltiaHMI.exe`**, **`AltiaHMI_3D.exe`**, and the **`reflash`** folder, that already exist in the Stateflow demo folder.

1. Copy the DeepScreen executable and the reflash folder from **`<Altia_project>\out`** to the Simulink project folder. For example, for the **`AltiaHMI`** project, copy **`AltiaHMI\out\AltiaHMI.exe`** to **`AltiaHMI.exe`** and **`AltiaHMI\code\reflash`** to **`reflash`** in the Stateflow demo folder.
2. Open the Simulink/Stateflow demonstration model (**`.slx`**) file in MATLAB if it is not already open.
3. Press **`Ctrl+E`** to open the **Configuration Parameters** dialog and select **Simulation Target** on the left side. In the **Insert custom C code in generated:** section, select the **Initialize function**. The connection to the Altia HMI is established by the **`MY_START_INTERFACE()`** call as shown in the picture below.



```
/* Configure/start Altia here */
MY_SUPPRESS_ERRORS(1);
altia_id = MY_START_INTERFACE(AltiaHMI_3D/AltiaHMI_3D.altwrk, NULL, 0, 0, NULL);
MY_RETRY_COUNT(0);
MY_CACHE_OUTPUT(1);
```

Figure 26: Preparing to change `MY_START_INTERFACE()` to Start DeepScreen

   a. The **`AltiaHMIMacros.h`** file defines the **`MY_START_INTERFACE()`** macro to the Altia API library function **`AtStartInterface()`**.

   b. The `MY_START_INTERFACE()` in the picture starts the Altia Runtime executable `altiart64.exe` and passes it the `AltiaHMI_3D/AltiaHMI_3D.altwrk` filename as an argument. Note the use of forward slash (/). It is safer in C code than PC slash (\).

   c. The executable `altiart64.exe` opens `AltiaHMI_3D/AltiaHMI_3D.altwrk` and displays it in a window.

4. To start a DeepScreen executable, change the parameters for `MY_START_INTERFACE()`.
   - For starting `AltiaHMI.exe`, change the `MY_START_INTERFACE()` line to the following (changes are highlighted in yellow):

     ```
     altia_id = MY_START_INTERFACE(AltiaHMI.exe, NULL, 3, 0, NULL);
     ```

   - For starting `AltiaHMI_3D.exe`, change the line to the following:

     ```
     altia_id = MY_START_INTERFACE(AltiaHMI_3D.exe, NULL, 3, 0, NULL);
     ```

NOTE: In the above changes for `MY_START_INTERFACE()`, ***no*** double-quotes around the executable name is correct. ***Never*** double-quote this parameter. The `MY_START_INTERFACE()` macro definition takes care of this for us.

13.4 Run in Simulation Mode with a DeepScreen Executable

Now you are ready to run in Simulink/Stateflow Simulation mode. See the earlier chapter Running a Demonstration Model in Simulation Mode for details.

The only change you will notice is the title of the Altia HMI window. When Altia Runtime was opening the `AltiaHMI` or `AltiaHMI_3D` Altia project, the window title was **Main Altia View**. When running the DeepScreen executable `AltiaHMI.exe` or `AltiaHMI_3D.exe`, the window title is **AltiaHMI** or **AltiaHMI_3D**. The look, feel, and behavior are identical because the DeepScreen generated code duplicates the look, feel, and behavior of Altia Runtime.

13.5 Generate a C Code Build and Run Standalone with a DeepScreen Executable

The demonstration model is ready for a C Code Build using Simulink Coder. The steps for generating a C Code Build are like those in the earlier chapter Transitioning to a C Code Build from Simulation Mode. Below are the steps, but without pictures:

1. In a Simulink or Stateflow window, choose the **APPS** ribbon.
2. From the **APPS** dropdown list, choose the **Simulink Coder** option.
3. Select the new Simulink **C CODE** ribbon if it is not already selected.

4. Choose **Build > Build** to generate code and build an executable.
5. During this step, Simulink opens an Altia Runtime window for each Altia block in the Simulink model if any exist. Close the window from the **X** button in the top-right corner of the window.
6. If the code generation or build fails, Simulink is good at displaying diagnostic information. Failures show in a **Diagnostic Viewer** window. Resolve them and try again.

   After the code generates and compiles successfully, a new executable file `<Model_Name>.exe` resides in the current project folder (for our demonstration, the file is `AltiaStateflowDemo.exe` or `AltiaStateflowNoSimPanel.exe`). The project folder is the folder containing the Simulink model file and Altia HMI projects.
7. This executable file runs standalone from Simulink. Open a Windows Explorer window, browse to the project folder, and double-click on the executable (e.g., `AltiaStateflowDemo.exe`).

   The executable opens a Windows Command Prompt window because it is compiled as a console application. Very soon after the Command Prompt window opens, the Altia windows open. If executing `AltiaStateflowDemo.exe`, this is the Altia Runtime window **Main Altia View** for the `AltiaSimPanel` Altia project and the DeepScreen window **AltiaHMI** or **AltiaHMI_3D** for the DeepScreen executable `AltiaHMI.exe` or `AltiaHMI_3D.exe`. You can interact with these windows the same way you interact with them in Simulation mode.

   NOTE: Closing the **Main Altia View** window or the **AltiaHMI_3D** window does not stop the executable. It continues to run until the Command Prompt window is closed by clicking on the **X** button in its top-right corner. Similarly, closing the Command Prompt window does not close any Altia windows. You must manually close an Altia window by clicking on the **X** button in its top-right corner.

   You can copy the standalone files to a different folder on the current Windows computer or a folder on a different Windows computer and run the simulation standalone.

   For a C Code Build of `AltiaStateflowDemo.slx` or `AltiaStateflowNoSimPanel.slx`, the required standalone files are listed below.

`AltiaStateflowdemo.exe or AltiaStateflowNoSimPanel.exe`	Simulink Coder C Code Build executable. Start this executable to run standalone.
`AltiaHMI.exe or AltiaHMI_3D.exe libEGL.dll`	DeepScreen executable using the Altia API Server code to serve a TCP/IP socket (lan). The DeepScreen executable also requires these two

`libGLESv2.dll`
`reflash folder`
	(2) DLL files for OpenGL ES 2.0 emulation on Windows and the reflash folder.

`altiart64.exe`
`fontModule.dll`
	Altia Runtime executable and support files.
	`AltiaStateflowDemo.exe`
	Requires these files to open Altia Runtime window for `AltiaSimPanel` Altia block.
	`AltiaStateflowNoSimPanel.exe`
	Does not require these files because it does not have an `AltiaSimPanel` Altia block.

`AltiaSimPanel folder`	`AltiaStateflowDemo.exe`
	Requires these files to open Altia Runtime window for `AltiaSimPanel` Altia block.
	`AltiaStateflowNoSimPanel.exe`
	Does not require these files because it does not have an `AltiaSimPanel` Altia block.

14 Transitioning to Embedded Coder Deployable Code

The main purpose of this demonstration is to show the use of Altia with Stateflow.

The Simulink blocks in our demonstration models (for example, Altia block and Signal Generator block) only exist to make it easier to interact with the Stateflow Chart in Simulation mode and a Simulink Coder **C** Code Build executable for Windows.

To transition to deployable code for an embedded system, we will only generate code for the Stateflow subsystem and then write **C** code to send/receive data or events to/from the Stateflow generated code. In a real embedded system, the **C** code could monitor external devices for data and send that data to the Stateflow generated code. It could also monitor Stateflow outputs and send data to external devices.

Simulink's Embedded Coder produces the most efficient code for deploying to an embedded system. One of the Embedded Coder toolchains is Microsoft Visual C++, so we can generate and build an executable for Windows for demonstration purposes.

Until now, we compiled the Altia HMI DeepScreen generated code with the Altia API Server code into its own standalone executable. This approach allowed us to easily interface with Altia HMI DeepScreen generated code in Simulation mode, as well as build and run a Simulink Coder **C** Code Build executable. We did both without changing any build configuration parameters.

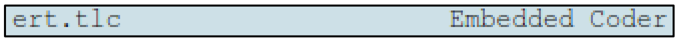
Now we want to create a Windows executable that closely matches an embedded system build. This chapter shows how to do Embedded Coder code generation and compile and link it ***directly*** with Altia HMI DeepScreen generated code.

NOTE: To complete the steps in this chapter, you must have a Simulink Embedded Coder license and a license for Altia Design 13.2 or newer and DeepScreen Intel x86 OpenGL ES 2.0 Windows code generator.

14.1 Set Configuration Parameters for Embedded Coder

1. Open the Simulink/Stateflow demonstration model (`.slx`) file in MATLAB if it is not already opened.
2. Open the **Configuration Parameters** dialog from a Simulink or Stateflow editor window with **Ctrl+E**.
3. In the left-side pane of the **Configuration Parameters** dialog, select **Code Generation**.
4. In the **Target selection > System target file:** field, press the **Browse…** button.

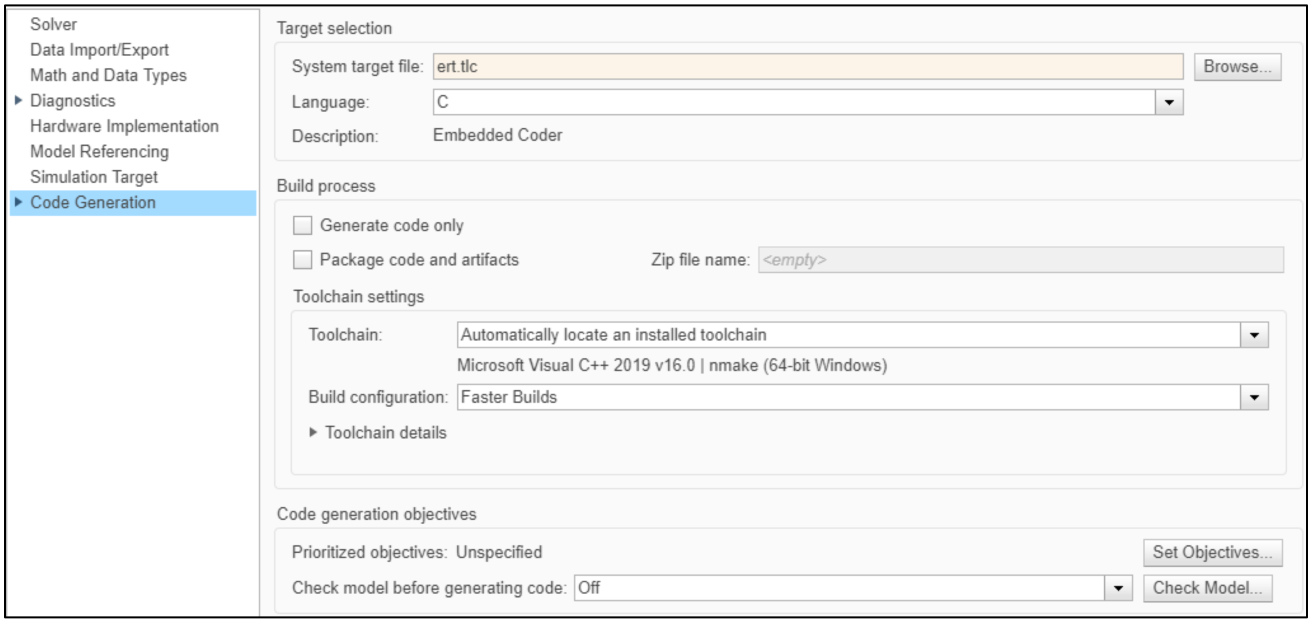
5. In the new **System Target File Browser** window, highlight the **System Target File** option `ert.tlc` that has the description of Embedded Coder, and click the **OK** button.



```
ert.tlc                    Embedded Coder
```

Figure 27: Selecting `ert.tlc` *System target file for Embedded Coder*

Now the **Code Generation** fields of the **Configuration Parameters** dialog should look very similar to the picture below. The version of the Microsoft Visual C++ in the Toolchain settings may be different if your MATLAB is configured for a different version of Visual Studio.



Target selection

System target file: ert.tlc

Language: C

Description: Embedded Coder

Build process

☐ Generate code only

☐ Package code and artifacts   Zip file name: <empty>

Toolchain settings

Toolchain: Automatically locate an installed toolchain

Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)

Build configuration: Faster Builds

Toolchain details

Code generation objectives

Prioritized objectives: Unspecified

Check model before generating code: Off

Figure 28: Code Generation Settings for Embedded Coder with Microsoft Visual C++

6. Expand **Code Generation** in the left-side pane and select **Custom Code**.

7. Disable the **Use the same custom code settings as Simulation Target** option.

8. Use the filenames below to ***replace all current filenames*** in the **Additional build information: > Libraries** section.

```
altiaWinLib.lib
altiaAPIlib.lib
libEGL.lib
libGLESv2.lib
libpng64.lib
zlib64.lib
user32.lib
gdi32.lib
winmm.lib
```

a. Highlight and copy (**Ctrl+C**) the filenames above into the Windows clipboard.

b. Then, highlight the current filenames in the **Additional build information: > Libraries** section, and paste (**Ctrl+V**) to replace the current filenames with the filenames in the Windows clipboard.

NOTE: The `altiaWinLib.lib`, `altiaAPIlib.lib`, `libpng64.lib`, `zlib64.lib`, `libEGL.lib`, and `libGLESv2.lib` files are created from generating DeepScreen code and compiling it. If you have not already generated and compiled DeepScreen code, see the instructions for doing so in an earlier chapter of this document.

NOTE: The `user32.lib`, `gdi32.lib`, and `winmm.lib` libraries are Microsoft Visual Studio system libraries compatible with Visual Studio 2015 thru 2019. For convenience, these files are included with the demonstration in the `Libs` folder. As a result, only the library names, not the full Visual Studio installation path for the libraries, are required.

9. Add the following directories to the Include directories in the **Additional build information: > Include directories** section. If DeepScreen code was generated for `AltiaHMI_3D`, use `AltiaHMI_3D` in the directory paths.

```
.\AltiaHMI\out\libs
.\AltiaHMI\out\libs\api\int
```

10. Add the following define in the **Additional build information: > Defines** section.

```
DEEPSCREEN
```

11. After making the above changes to the **Code Generation > Custom Code** settings in the **Configuration Parameters** dialog, press **Apply** to only apply the changes and leave the dialog open, or press **OK** to apply the changes and close the dialog.

14.2 Build This Subsystem

1. Return to the Simulink editor window. Change to the top-level of the model to show the Stateflow chart and the other Simulink components.
2. Right-click on the **Stateflow Chart** block, and choose the option **C/C++ Code > Build this Subsystem** from the menu.

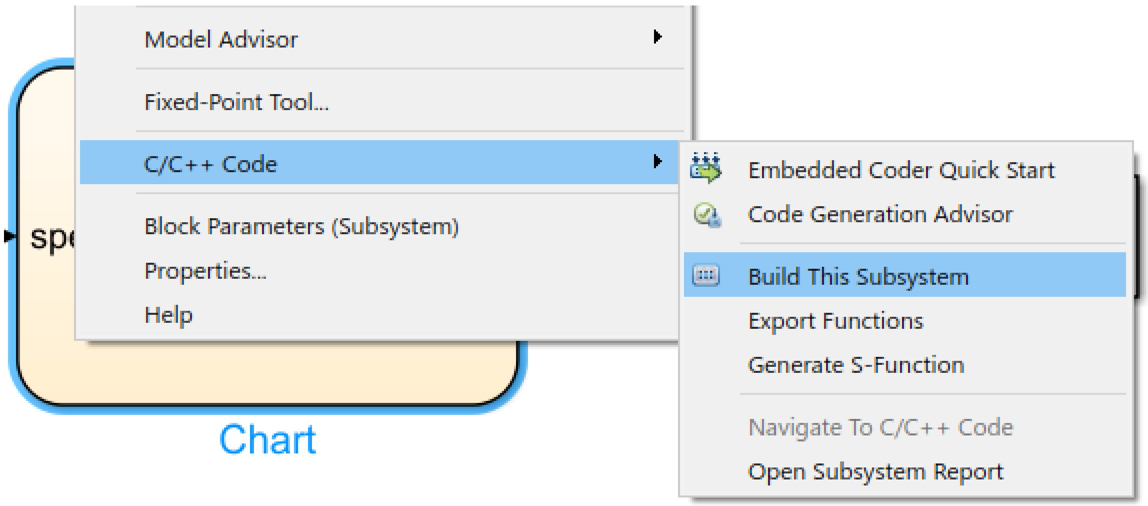
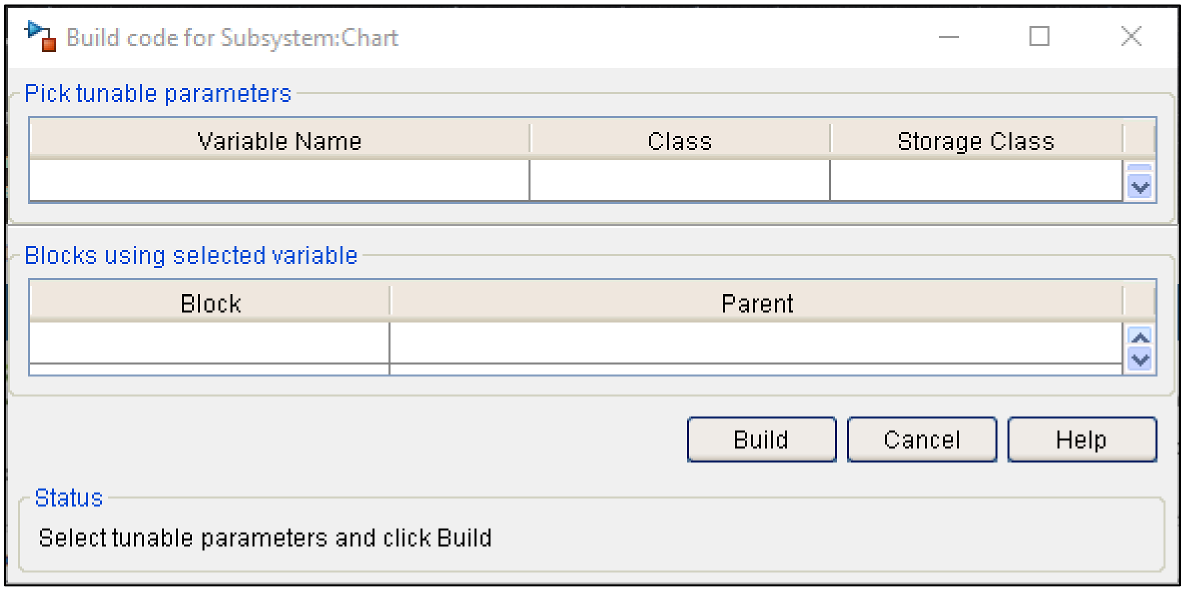


Figure 29: Embedded Coder > Build This Subsystem

3. When the **Build code for Subsystem:Chart** dialog appears, press the **Build** button.



Select tunable parameters and click Build

*Figure 30: Press **Build** in the **Build code for Subsystem:Chart** dialog*

The Build code for `Subsystem:Chart` dialog closes when the code generation and build complete. The Simulink editor window shows a new pane for the generated `Chart.c` source code.

4. You can study the source code or just close the pane from the **X** button in the top-right corner of the pane.

5. Open a Windows Explorer window and browse to the demonstration folder.

6. Notice the new sub-folder named `Chart_ert_rtw`. This is the folder where Embedded Coder put the generated `C` source code for the Stateflow Chart block.

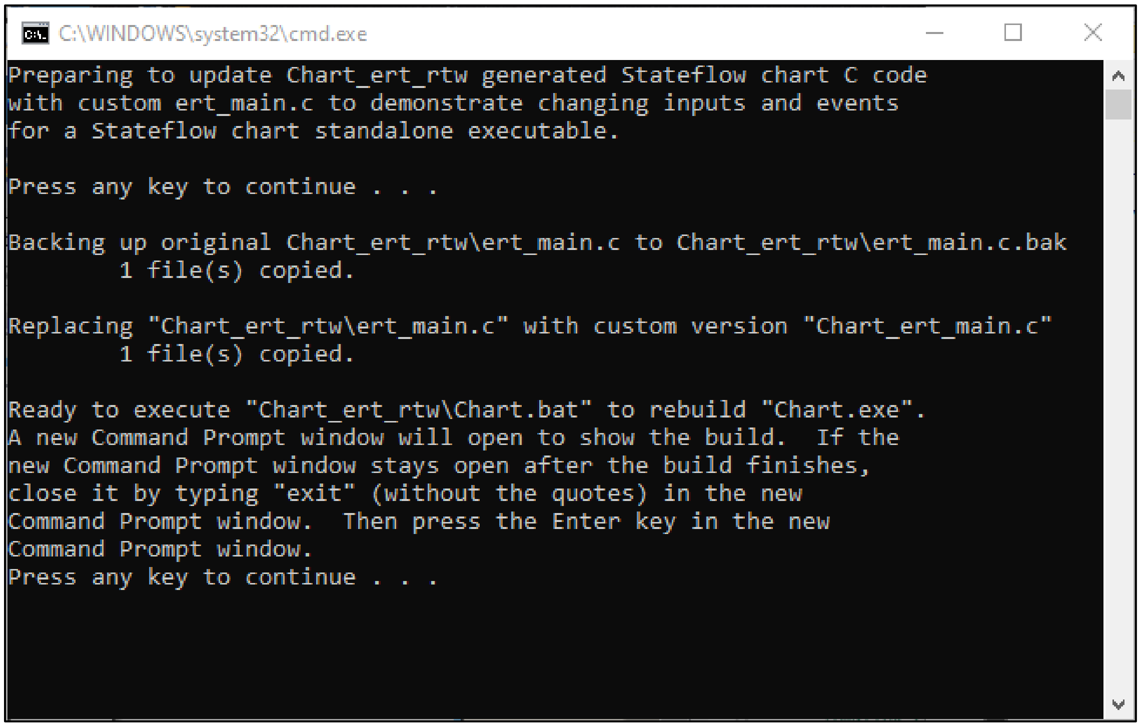
7. Scroll down in the Windows Explorer window to find `Chart.exe`. Embedded Coder compiled the `C` source code in `Chart_ert_rtw` to create the `Chart.exe` executable.
8. Double-click on `Chart.exe` to run it.
9. It runs, but it only opens a Command Prompt window. Close the Command Prompt window from the **X** button in the top-right corner.

   To make `Chart.exe` more interesting, we must modify its main function to provide additional logic to step the Stateflow chart code while inputting some interesting data values. The file `Chart_ert_main.c` is provided with the demonstration to replace the generated `Chart_ert_rtw\ert_main.c` file.

10. Double-click on the script `Chart_ert_main_update.bat` to copy `Chart_ert_main.c` to `Chart_ert_rtw\ert_main.c`, and then rebuild `Chart.exe`.

    The script displays a Command Prompt window that shows messages for each step it performs. It waits for you to **Press any key to continue …** before it performs the next step. If you want to see how `Chart_ert_main_update.bat` works, open it in a simple text editor such as Notepad.

    When the script runs, it displays the text below before it is ready to rebuild `Chart.exe`.



C:\WINDOWS\system32\cmd.exe

```
Preparing to update Chart_ert_rtw generated Stateflow chart C code
with custom ert_main.c to demonstrate changing inputs and events
for a Stateflow chart standalone executable.

Press any key to continue . . .

Backing up original Chart_ert_rtw\ert_main.c to Chart_ert_rtw\ert_main.c.bak
        1 file(s) copied.

Replacing "Chart_ert_rtw\ert_main.c" with custom version "Chart_ert_main.c"
        1 file(s) copied.

Ready to execute "Chart_ert_rtw\Chart.bat" to rebuild "Chart.exe".
A new Command Prompt window will open to show the build.  If the
new Command Prompt window stays open after the build finishes,
close it by typing "exit" (without the quotes) in the new
Command Prompt window.  Then press the Enter key in the new
Command Prompt window.
Press any key to continue . . .
```

Figure 31: Executing the Chart_ert_main_update.bat script

11. Please read the messages before pressing any key to continue.

Depending on the version of MATLAB, the new Command Prompt window that opens to perform the **`Chart.exe`** build either stays open after the build finishes or immediately closes after the build finishes.

12. After the **`Chart_ert_main.bat`** script executes **`Chart_ert_rtw\Chart.bat`** to rebuild **`Chart.exe`**, the script is ready to run the new version of **`Chart.exe`**. You can also just double-click on **`Chart.exe`** from any Windows Explorer window to run it.

    When **`Chart.exe`** runs, it now shows the Altia HMI DeepScreen window with the speedometer needle continuously cycling through its range. This new behavior is a result of the new version of the **`Chart_ert_rtw\ert_main.c`** **C** code file.

13. From a Windows Explorer window, open the **`Chart_ert_rtw`** folder, and notice these two files:

    ```
    ert_main.c
    ert_main.c.bak
    ```

    The **`ert_main.c`** is our new version and **`ert_main.c.bak`** is the original version generated by Embedded Coder.

14. Compare the two files with your favorite file comparing utility (such as WinMerge). You can see the simple changes made to **`ert_main.c`** to input data into the Stateflow chart generated code.

    MathWorks has support pages and videos that show other ways to interface custom **C** code to Embedded Coder generated **C** code.

    The **`Chart.exe`** executable contains both the Embedded Coder generated code and the Altia HMI DeepScreen generated code. The **`Chart.exe`** and several required DLL files are easily copied to a different folder on the current Windows computer or a folder on a different Windows computer to run standalone.

    The files required to run **`Chart.exe`** standalone are shown below:

`Chart.exe`	Single executable containing both the Embedded Coder generated Stateflow chart **C** code and the Altia HMI DeepScreen generated **C** code.
`libEGL.dll` **`libGLESv2.dll`**	The DeepScreen generated **C** code requires these two (2) DLL files for OpenGL ES 2.0 emulation on Windows.
`reflash folder`	The DeepScreen code requires the resources in this folder.

15 Configuring Unicode Simulations and Code Builds

Altia users often use international text in HMIs. Our demonstration Altia HMI has no international text. If it did, it would need to display wide character strings to support non-Latin international text, such as Arabic, Chinese, Japanese, Korean, etc. The wide character values must follow the Unicode standard. We refer to the combination of using wide character strings containing Unicode character values as just *Unicode* for short.

To support non-Latin international text, we must configure these elements for Unicode:

- DeepScreen code generation
- Simulink Simulation mode
- Simulink Coder
- Embedded Coder

This chapter describes how to configure each of these elements for Unicode.

15.1 Generate and Build DeepScreen Code for Unicode

1. An earlier section in this document describes the steps to Generate and Build DeepScreen Windows Code. Complete these steps to enable Unicode in the AltiaHMI or AltiaHMI_3D Altia project.
    a. In Altia Design navigate to the **Code Generation** tab and click on **CodeGen Options**.
    b. In the **Code Generation Options** dialog, click on the **Override locked presets…** button. Click on **Override**.
    c. Click the checkbox next to **Unicode Font Characters** to enable Unicode.
    d. Click **OK** to close the **Code Generation Options** dialog.
    e. Save the Altia project.
2. Click on the **Build Prototype** button to generate code and build the executable. The resulting `AltiaHMI.exe` or `AltiaHMI_3D.exe` and the associated `altiaWinLib.lib`, `altiaAPIlib.lib`, and `altiaAPIlibfloat.lib` object libraries are now all Unicode versions.
3. Close Altia Design.
4. If your Stateflow model is configured to start the DeepScreen executable, copy the DeepScreen executable from `<Altia_project>\out` to the Simulink project folder. For example, for the `AltiaHMI` project, copy `AlitaHMI\out\AltiaHMI.exe` to `AltiaHMI.exe` in the Simulink project folder.

15.2 Configure Simulink Simulations for Unicode

1. Open the Simulink **Configuration Parameters** dialog with **Ctrl+E**.
2. Select **Simulation Target** in the left-side pane.
3. Select **Additional build information: > Libraries**.
4. Replace `liblan.lib` with `liblanUnicode.lib.`
5. Select **Additional build information: > Defines**.
6. Enter these two (2) defines:

   ```
   UNICODE
   ALTIAUNICODEAPI
   ```

7. Press **OK** to apply and close the dialog, or press **Apply** to apply the changes and keep the dialog open.
8. For each Altia block in the Simulink model:
   a. Right click on the Altia block and choose **Block Parameters (S-Function)**.
   b. In the **S-function modules:** field, replace `liblan.lib` with `liblanUnicode.lib`.
   c. Press **OK** to close the Block Parameters dialog.
9. Save the Simulink project.
10. Go to the Simulink or Stateflow editor window, and **Run** a simulation as described in the previous chapter Running a Demonstration Model in Simulation Mode.

    For our Simulink/Stateflow model and Altia HMI, there will be no visual difference after configuring for Unicode. However, the above changes made to the Simulation Target rebuilt the Stateflow chart `<model_file_name>_sfun.mexw64` for Unicode.

15.3 Configure Simulink Coder C Code Build and Run for Unicode

1. Perform the previous Configure Simulink Simulations for Unicode steps if they have not already been performed.
2. Perform a Simulink Coder **C** Code build and run. These earlier chapters describe how:
   a. Transitioning to a C Code Build from Simulation Mode
   b. Using Altia DeepScreen Generated Code with Simulation and C Code Build

15.4 Configure Embedded Coder Build and Run for Unicode

1. Perform the previous Generate and Build DeepScreen Code for Unicode steps if they have not already been performed.
2. If you have not already done so, follow the steps in the earlier chapter Transitioning to Embedded Coder Deployable Code to learn how to build and run a `Chart.exe`.
3. Open the Simulink **Configuration Parameters** dialog with **Ctrl+E**.
4. Select **Code Generation** in the left-side pane.
5. Confirm the **Target selection > System target file:** is set to `ert.tlc`. If it is not, go back to the earlier chapter Transitioning to Embedded Coder Deployable Code for instructions on how to set it to `ert.tlc`.
6. Expand **Code Generation** in the left-side pane and select **Custom Code**.
7. Confirm the **Use the same custom code settings as Simulation Target** option is already **disabled**. If it is enabled, you may not have completed all the steps in the earlier chapter Transitioning to Embedded Coder Deployable Code.
8. Select **Additional build information: > Defines**.
9. Add these three (3) defines:

```
UNICODE
ALTIAUNICODEAPI
DEEPSCREEN
```

10. Press **OK** to apply and close the dialog, or press **Apply** to apply the changes and keep the dialog open.
11. Return to the earlier chapter Transitioning to Embedded Coder Deployable Code for the steps to build and run `Chart.exe`.

    For our Altia HMI and Stateflow chart, there is no visual difference in the `Chart.exe` execution with or without Unicode. However, the generation and build of the DeepScreen code for Unicode and the additions to the **Code Generation > Custom Code > Additional build information: > Defines** resulted in a `Chart.exe` that is now using Unicode strings for the Altia HMI.

16 Stateflow Demonstration Summary

For this demo, we completed the following:

- Studied a Simulink model containing a Stateflow chart that uses the Altia API to control an Altia HMI.
- Ran the Simulink/Stateflow model in Simulation mode. We interacted with the Altia HMI running in the Altia Runtime executable.
- Used Simulink Coder for the Simulink/Stateflow model to generate and build a standalone executable. We ran the standalone executable, and it interacted with the Altia HMI running in the Altia Runtime executable.
- Generated Altia DeepScreen code for the Altia HMI and compiled it as its own executable. In Simulation mode and from a Simulink Coder standalone executable, we interacted with the Altia HMI running in the DeepScreen executable. We saw the behavior of the Altia HMI in the DeepScreen executable was identical to the behavior in the Altia Runtime executable.
- Switched from Simulink Coder to Embedded Coder to build a single executable containing the generated Stateflow **C** code and the generated DeepScreen **C** code. The **C** code in this single executable is at the quality level of deployable embedded system code.
- Learned how to make the necessary changes to support Unicode strings in Simulation mode, a Simulink Coder build and run, and an Embedded Coder build and run.