

Using Model-Based Design to Develop and Deploy a Video Processing Application

BY HOUMAN ZARRINKOUB

According to the U.S. National Highway Traffic Safety Administration, single-vehicle road departures result in many serious accidents each year. To reduce the likelihood of a vehicle's straying out of lane, automotive engineers have developed lane tracking and departure warning systems that use a small camera to transmit video information about lane markings and road conditions to a microprocessor unit installed on the vehicle.

In this article, we show how Model-Based Design with Simulink® and the Video and Image Processing Blockset can be used to design a lane-detection and lane-departure warning system, implement the design on a Texas Instruments DSP, and verify its on-target performance in real time.

The core element of Model-Based Design is an accurate system model—an executable specification that includes all software and hardware implementation requirements, including fixed-point and timing behavior. You use the model to automatically generate code and test benches for final system verification and deployment.

This approach makes it easy to express a design concept, simulate the model to verify the algorithms, automatically generate the code to deploy it on a hardware target, and verify exactly the same operation on silicon.

Building the System Model

Using Simulink, the Signal Processing Blockset, and the Video and Image Processing Blockset, we first develop a floating-point model of the lane-detection system. We model lane markers as line seg-

ments, detected by maximizing the Hough transform of the edges in a video frame.

We input a video stream to the simulation environment using the From Multimedia File block from the Video and Image Processing

Blockset. During simulation, the video data is processed in the Lane Marker Detection and Tracking subsystem, which outputs the detection algorithm results to the To Video Display block for computer visualization (Figure 1).

Lane Detection and Visualization

Figure 2 shows the main subsystem of our Simulink model. The sequence of steps in the lane marker detection and tracking algorithm maps naturally to the sequence of subsystems in the model.

We begin with a preprocessing step in which we define a relevant field of view and filter the output of this operation to reduce image noise. We then determine the edges of the image using the Edge Detection block in the Video and Image Processing Blockset. With this block we can use the Sobel, Prewitt, Roberts, or Canny methods to output a binary

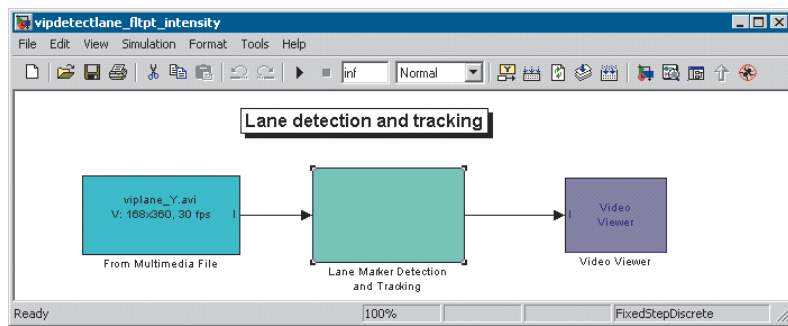


Figure 1. Lane-detection model.

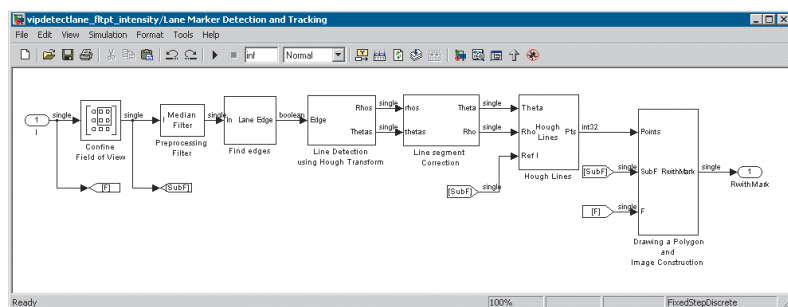


Figure 2. Floating-point model: the Lane Marker Detection and Tracking subsystem.

image, a matrix of Boolean values corresponding to edges.

Next, we detect lines using the Hough Transform block, which maps points in the Cartesian image space to curves in the Hough parameter space using the following equation:

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

The block output is a parameter space matrix whose rows and columns correspond to the ρ and θ values, respectively. Peak values in this matrix represent potential lines in the input image.

Our lane marker detection and tracking subsystem uses a feedback loop to further refine the lane marker definitions. We post-process the Hough Transform output, using line segment correction to deal with image boundary outliers, and then compute the Hough lines. The Hough Lines block in the Video and Image Processing Blockset finds the Cartesian coordinates of line end-points by locating the intersections between the lines, characterized by the θ and ρ parameters and the boundaries of the reference image.

The subsystem then uses the computed end-points to draw a polygon, and reconstructs the image. The sides of the polygon correspond to the detected lanes, and the polygon is overlaid onto the original video. We simulate the model to verify the lane detection and tracking design (Figure 3).

Figure 3. Lane tracking simulation results, with a trapezoidal figure marking the lanes in the video image.

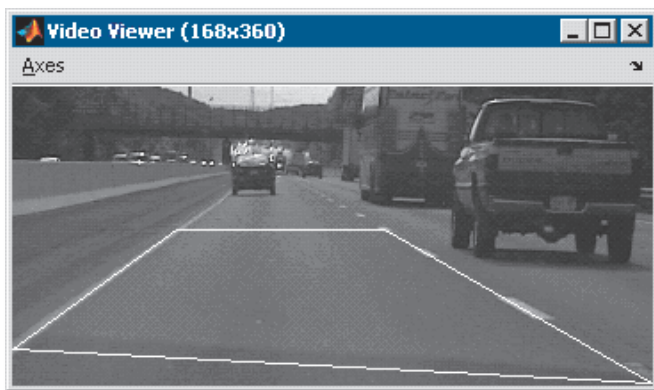
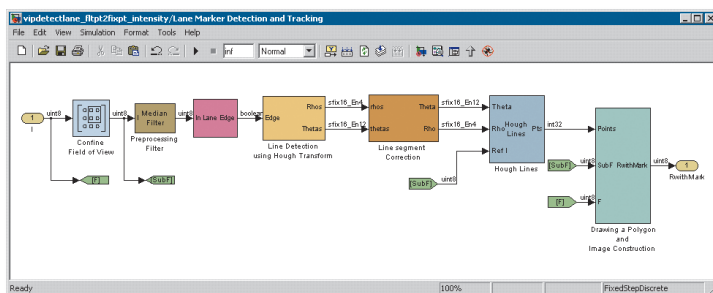


Figure 4. Fixed-point model: main subsystem.



Converting the Design from Floating Point to Fixed Point

To implement this system on a fixed-point processor, we convert the algorithm to use fixed-point data types. In a traditional design flow based on C programming, this conversion would require major code modification. Conversion of the Simulink model involves three basic steps:

1. Change the source block output data types. During automatic data type propagation, Simulink displays messages indicating the need to change block parameters to ensure data type consistency in the model.
2. Set the fixed-point attributes of the accumulators and product outputs using Simulink Fixed Point tools, such as Min-max and Overflow logging.
3. Examine blocks whose parameters are sensitive to the pixel values to ensure that these parameters are consistent with the input signal data type. (The interpretation of pixel values depends on the data type. For example, the maximum intensity of a pixel is denoted by a value of 1 in floating point and by a value of 255 in an unsigned 8-bit integer representation.)

Figure 4 shows the resulting fixed-point model. During simulation, the flexibility and generality provided by fixed-point

operators as they check for overflows and perform scaling and saturations can cause a fixed-point model to run slower than a floating-point model. To speed up the simulation, we can run

the fixed-point model in Accelerator mode. The Simulink Accelerator can substantially improve performance for larger Simulink models by generating C code for the model, compiling the code, and generating a single executable for the model that is customized to a model's particular configuration. In Accelerator mode, the simulation for the fixed-point model runs at the speed of compiled C code.

Implementing and Verifying the Application on TI Hardware

Using Real-Time Workshop® and Real-Time Workshop Embedded Coder, we automatically generate code and implement our embedded video application on a TI C6400™ processor using the Embedded Target for TI C6000™ DSP. To verify that the implementation meets the original system specifications, we use Link for Code Composer Studio™ to perform real-time hardware-in-the-loop validation and visualization of the embedded application. Before implementing our design on a TI C6416DSK evaluation board, we must convert the fixed-point, target-independent model to a target-specific model. For this task we use Real-Time Data eXchange (RTDX), a TI real-time communications protocol that enables the transfer of data to and from the host. RTDX blocks let us ensure that the same test bench used to validate the design in simulation is used in implementation.

Creating the target-specific model involves three steps:

1. Replace the source block of the target-independent model with the From RTDX block and set its parameters.
2. Replace the Video Viewer block of the target-independent model with the To RTDX block and set its parameters.

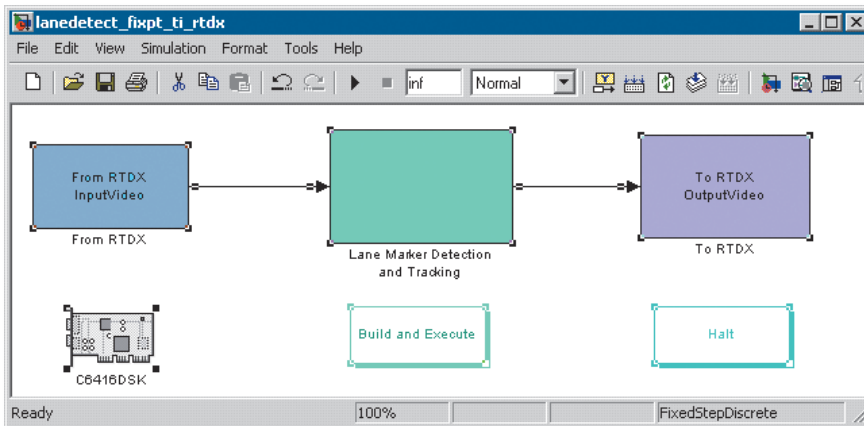


Figure 5. The TI C6416 DSK block automatically sets up all Real-Time Workshop targeting parameters based on the configuration of the TI board and Code Composer Studio installed locally.

3. Set up Real-Time Workshop target-specific preferences by dragging a block specific to our target board from the C6000 Target Preferences library into the model. Figure 5 shows the resulting target-specific model.

To automate the process of building the application and verifying accurate real-time behavior on the hardware, we create a script, using Link for Code Composer Studio to perform the following tasks:

1. Invoke the Link for Code Composer Studio IDE to automatically generate the Link for Code Composer Studio project.
2. Compile and link the generated code from the model.
3. Load the code onto the target.
4. Run the code: Send the video signal to the target-specific model from the same input

file used in simulation and retrieve the processed video output from the DSP.

5. Plot and visualize the results in a MATLAB figure window.

Figure 6 shows the script used to automate embedded software verification for TI DSPs from MATLAB. Link for Code Composer Studio provides several functions that can be invoked from MATLAB to parameterize and automate the test scripts for embedded software verification.

Figure 7 shows the results of the automatically generated code executing on the target DSP. We observe that the application running on the target hardware properly detects the lane markers, and we verify that the application meets the requirements of the original model. After running our application on the target, we may find that our algorithm does

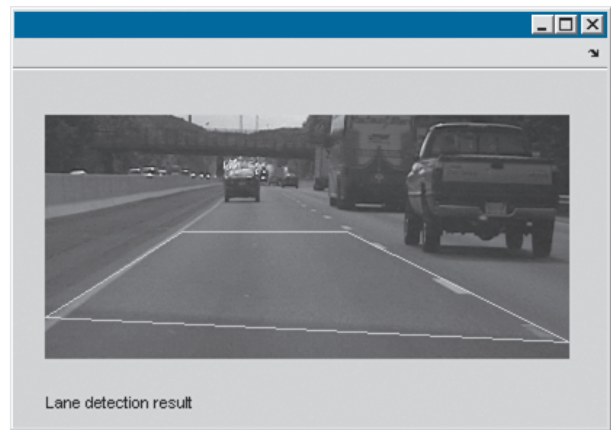


Figure 7. Automatically generated code executing on the target DSP verifies that the application correctly detects the lane markers.

not meet the real-time hardware requirements. In Model-Based Design, simulation and code generation are based on the same model, and so we can quickly conduct multiple iterations to optimize the design. For example, we can use the profiling capabilities in Link for Code Composer Studio to identify the most computation-intensive segments of our algorithm. Based on this analysis, we can change the model parameters, use a more efficient algorithm, or even replace the general-purpose blocks used

in the model with target-optimized blocks supplied with the Embedded Target for TI C6000. Such design iterations help us optimize our application for the best deployment on the hardware target. ◀

```

43 hVideo = videoSetup(sz);
44
45 % Configure RTDX
46 r = CCS_Obj.rtdx;
47 r.disable;
48 r.open('InputVideo', 'w', ...
49       'OutputVideo', 'r');
50 r.configure(32768,4,'continuous');
51 CCS_Obj.run; % Run application
52 r.enable('all');
53 r.enable; % Master RTDX enable
54
55 stopLoop = 0;
56 TestVidLoop = 0; % test video loop counter
57 TxCnt = 0;
58 while ~stopLoop,
59     % Get next test video frame and send to target
60     TestVidLoop = 1+rem(TestVidLoop,VideoLoopFrames);
61     inFrame = hcx(TestVidLoop).cdata;
62     %inFrame = vq3(:, :,TestVidLoop);
63     r.writemsg('InputVideo', inFrame);
64     TxCnt = TxCnt + 1;
65
66     % Process received frames
67     while ~stopLoop && (r.msgcount('OutputVideo') > 0),
68         outFrame = r.readmsg('OutputVideo', 'uint8',h2_size);
69         videoUpdate(inFrame, outFrame, TxCnt, 0,0, hVideo);
70     end

```

Figure 6. Link for Code Composer Studio IDE script.

RESOURCES

- ▶ **Video and Image Processing Blockset**
www.mathworks.com/res/viprocessing
- ▶ **Model-Based Design for Signal Processing and Communication Systems**
www.mathworks.com/res/dsp_comm
- ▶ **Webinar: Using Simulink for Video and Image Processing**
www.mathworks.com/res/vipwebinar
- ▶ **Webinar: Design and Implementation of Video Applications on TI C6400 DSPs with Simulink**
www.mathworks.com/res/video webinar
- ▶ **Book: Video Processing and Communications**
www.mathworks.com/res/book2613