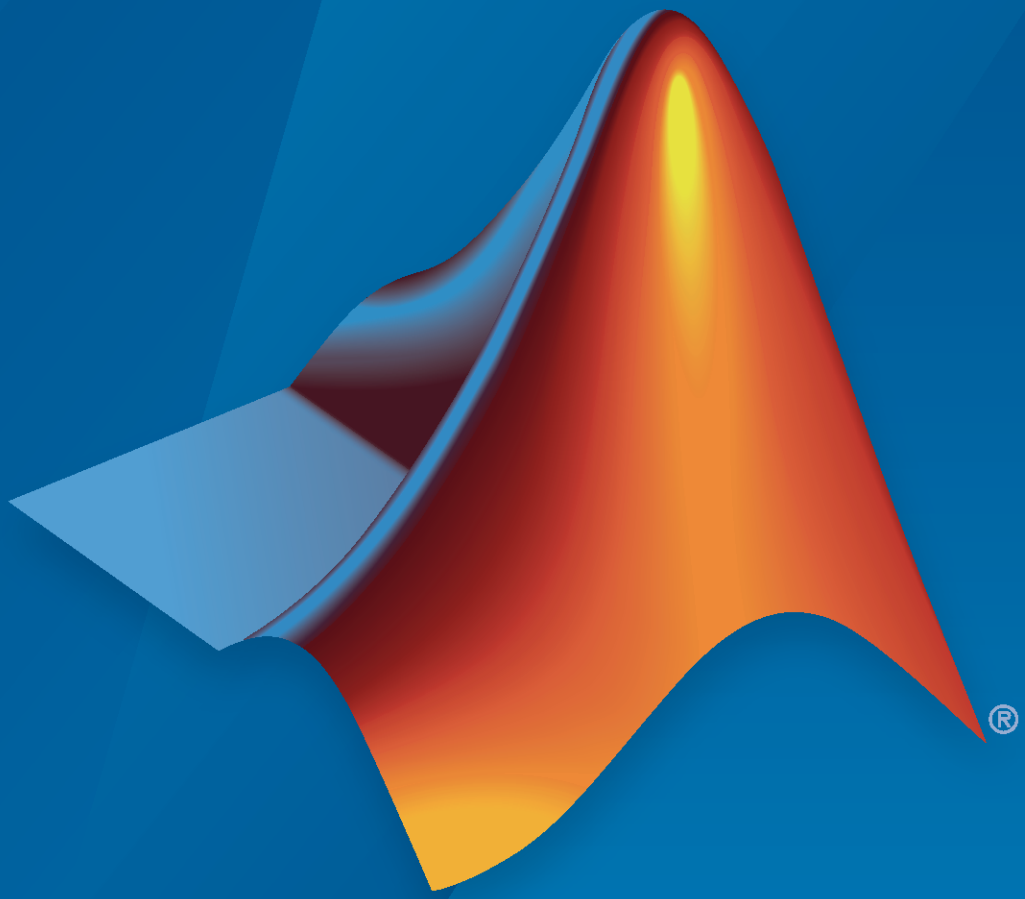
SoC Blockset™

User's Guide



MATLAB® & SIMULINK®

R2021b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

SoC Blockset™ User's Guide

Revision History

March 2019	Online Only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)
March 2020	Online only	Revised for Version 1.2 (Release 2020a)
September 2020	Online only	Revised for Version 1.3 (Release 2020b)
March 2021	Online only	Revised for Version 1.4 (Release 2021a)
September 2021	Online only	Revised for Version 1.5 (Release 2021b)

Contents

1 Product Overview

2 System-on-Chip

3 Processor Software

4 Programmable Logic

5 Memory

6 Peripherals

7 Examples

1

Product Overview

SoC Blockset Product Description

Design, evaluate, and implement SoC hardware and software architectures

SoC Blockset™ provides Simulink® blocks and visualization tools for modeling, simulating, and analyzing hardware and software architectures for ASICs, FPGAs, and systems on a chip (SoC). You can build your system architecture using memory models, bus models, and I/O models, and simulate the architecture together with the algorithms.

SoC Blockset lets you simulate memory and internal and external connectivity, as well as scheduling and OS effects, using generated test traffic or real I/O data. You can quickly explore different system architectures, estimate interface complexity for hardware and software partitioning, and evaluate software performance and hardware utilization.

SoC Blockset exports reference designs for Xilinx® and Intel® FPGA devices and SoC platforms, including Zynq®-7000, UltraScale+™, and Intel SoC FPGAs. These reference designs can be used with Xilinx and Intel design tools.

Supported Third-Party Tools and Hardware

Third-Party Synthesis Tools and Version Support

SoC Blockset supports these third-party FPGA synthesis tools:

- Intel Quartus® Prime Standard Edition 18.1
- Xilinx Vivado® Design Suite 2020.1

To use third-party synthesis tools with SoC Blockset, a supported synthesis tool must be installed, and the synthesis tool executable must be on the system path.

Third-Party Support for Software Generation

SoC Blockset supports this third-party software generation tool:

- Intel SoC FPGA Embedded Development Suite (EDS) 18.0

Supported Xilinx Devices

SoC Blockset supports execution on Xilinx devices shown in this table.

Device Family	Board	Comments
Xilinx Artix®-7	Artix-7 35T Arty FPGA Development Board	
Xilinx Kintex®-7	Kintex-7 KC705	
XilinxZynq	Zynq-7000 ZC706	
	ZedBoard™	
XilinxZynqUltraScale+	Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit	
	Zynq UltraScale+ RFSoC ZCU111 Evaluation Kit	FTDI JTAG not supported on Linux®
	Zynq UltraScale+ RFSoC ZCU216 Evaluation Kit	FTDI JTAG not supported

Supported Intel Devices

SoC Blockset supports execution on Intel devices shown in this table.

Device Family	Board
Intel Arria® 10	Arria 10 SoC Development Kit
Intel Cyclone® V	Cyclone V SoC Development Kit

SoC Board Support Packages

The SoC Blockset support packages contain the definition files for all supported boards. You can download one or more vendor-specific support packages. To generate executables and execute on hardware, download at least one of these packages.

To see the list of SoC Blockset support packages, visit “SoC Blockset Supported Hardware”. To download an SoC Blockset support package, on the MATLAB® **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

See Also

SoC Builder | “Hardware Implementation Pane Overview”

More About

- “Generate SoC Design” on page 2-46
- “SoC Blockset Supported Hardware”

System-on-Chip

SoC Blockset Model Structure

An SoC Blockset model consists of a top model that includes at least one of these reference models.

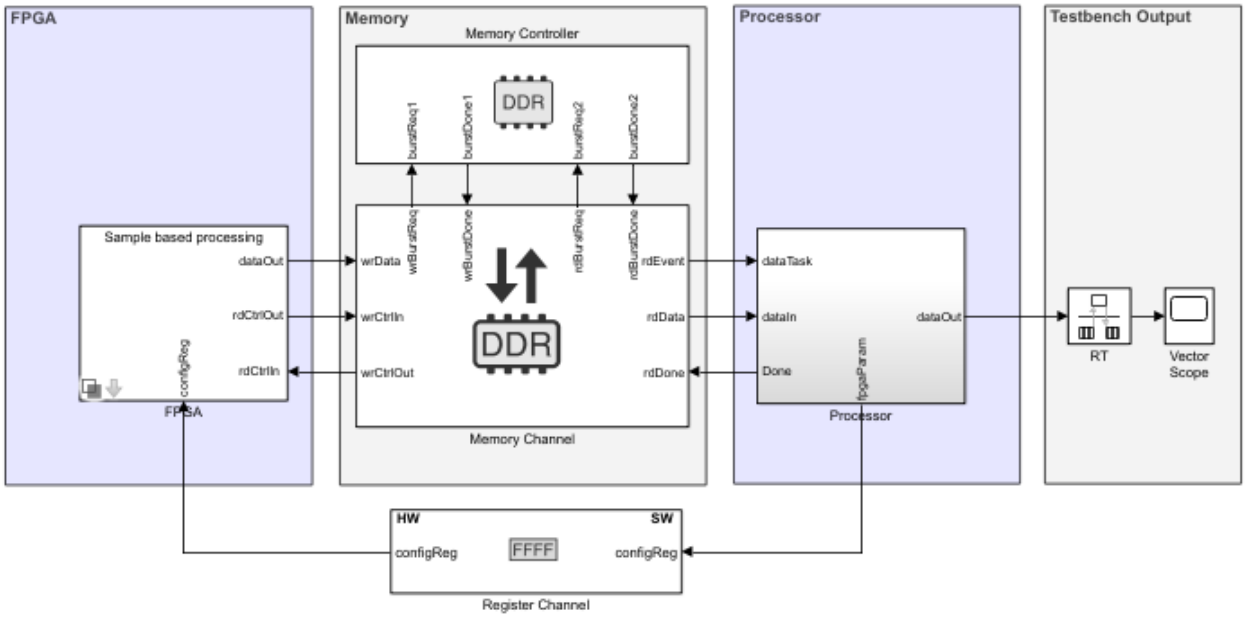
- An FPGA reference model represents the FPGA part of an SoC device. The top model can include at most one FPGA reference model. For information on how to set up an FPGA reference model, see “User Logic on FPGA” on page 2-37.
- A processor reference model represents the processor part of an SoC device. The top model can include one or more processor reference models. For information on how to set up a processor reference model, see “Software and Task Management on Processor” on page 2-34.

The processor and the FPGA subsystems communicate through a Memory Channel, Register Channel or Interrupt Channel block.

In addition to FPGA and processor reference models, the top model can include additional SoC Blockset blocks, such as the blocks listed here, for modeling interfaces and test bench components.

- ADC Interface, PWM Interface, LED, Push Button
- IO Data Sink, IO Data Source, Stream Data Source

The following image shows an SoC Blockset model, with an FPGA reference model, a processor reference model, communicating over a memory channel.



For an example of an SoC Blockset model, see “Streaming Data from Hardware to Software” on page 7-31.

SoC Blockset provides project templates for common SoC use-cases. Use them as a starting point for your design.

See Also

More About

- “Use Template to Create SoC Model” on page 2-4
- “Create an SoC Project Application” on page 2-31
- “Build Error When FPGA or Processor Model Not Detected” on page 2-53

Use Template to Create SoC Model

SoC Blockset model templates provide design patterns and best practices for models intended for simulation, HDL code generation, or SoC deployment. Models created from any one of SoC Blockset model templates have their configuration parameters set up for simulation and code generation.

Create SoC Model Using SoC Blockset Template

To efficiently model hardware for SoC design, create a project by using an SoC Blockset template.

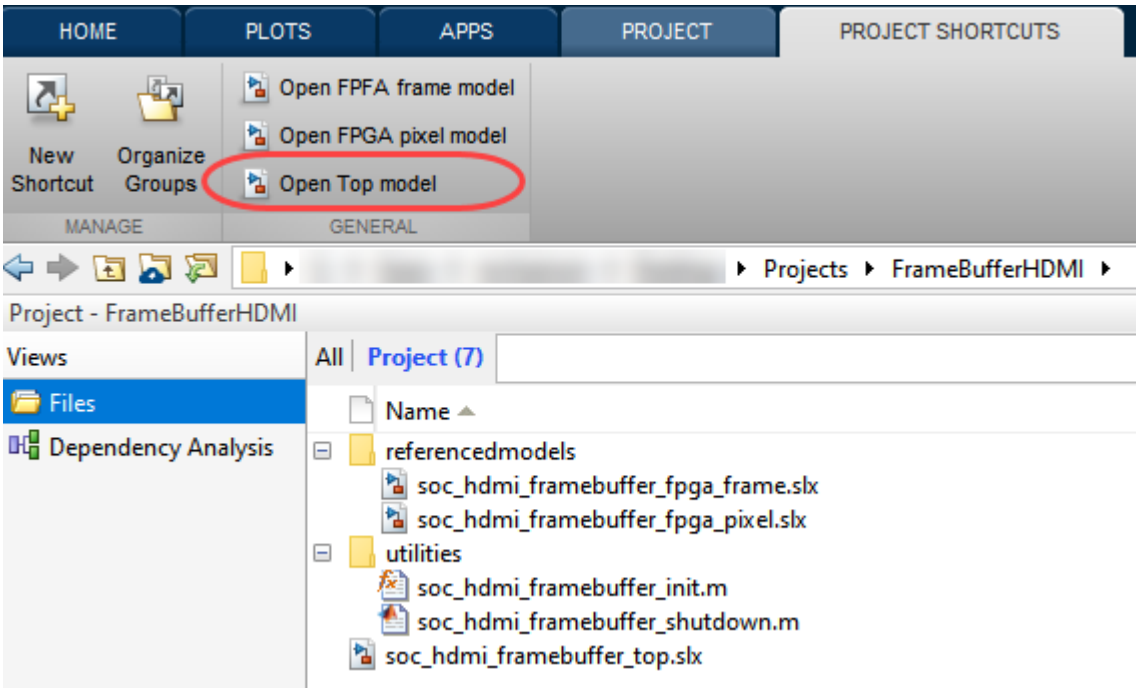
1. In the MATLAB Home tab, select the **Simulink** button. Alternatively, at the command line, enter:

   ```
   simulink
   ```

2. On the Simulink Start Page, scroll down to the **SoC Blockset** section, which contains links to SoC templates for common workflows. Select a template and save the project. A project folder opens in your workspace containing:

   - A model with the name `soc_*_top.slx` – The top-level model for the SoC project.
   - `referencedmodels` – A folder containing the models referenced from the top model. Some templates include an FPGA model and a processor model. Other templates only include one referenced model: an FPGA model or a processor model.
   - `utilities` – A folder containing utility functions or testbench data used by the model.

   To open the top-level model in Simulink, on the **Project Shortcuts** tab, click **Open Top model**.



3. In each template, navigate to the blocks marked **FPGA Algorithm** in the FPGA model, or **Processor Algorithm** in the processor model. These blocks are highlighted for easy detection. Replace the template blocks with your own algorithm model.

Tip To easily find the algorithm blocks, follow the annotations throughout the model hierarchy.

4 To open the **SoC Blockset** Block Library, select the Library Browser button, then select **SoC Blockset** in the left pane. Alternatively, at the command line, enter:

```
soclib
```

This library includes blocks for creating SoC models and testbenches.

Template Structure

The top model in an SoC Blockset template includes an FPGA subsystem, which represents the logic intended to program the FPGA. The FPGA subsystem includes two Simulink model variants:

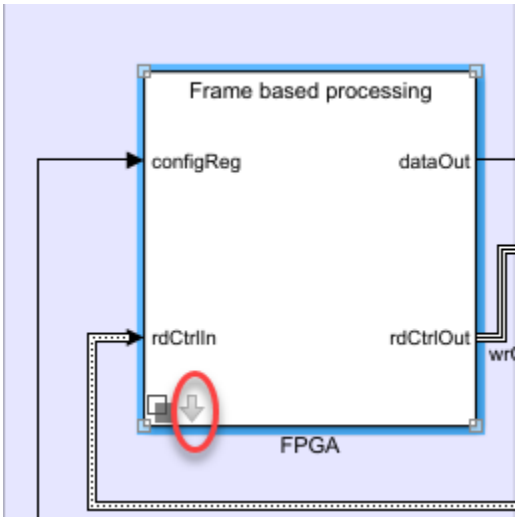
- Frame-based FPGA model – For enhanced simulation performance (not available in the RFSoC template)
- Sample-based FPGA model – For cycle accuracy and code generation

The top model also includes a processor subsystem, which represents the software program intended to run on the SoC processor. Both the FPGA and the top subsystems contain blocks marked as FPGA Algorithm or Processor Algorithm. Replace these algorithms with your own logic. The top model of the template also includes a memory system, with a memory controller and memory channels. These blocks represent the physical memory system on the board. The model often includes a register channel (to enable communication between the processor and FPGA), testbench, or I/O blocks.

Modify Project

Modify the FPGA Model

From the top model, open the FPGA model by clicking the arrow at the bottom left of the FPGA block:



The FPGA model contains two model variants: a frame-based variant and a sample-based variant. Double-click the model variant you want to modify. The FPGA model typically includes two main subsystems for you to modify:

- FPGA Algorithm Wrapper – Double-click to open the model. The algorithm wrapper contains a green-highlighted subsystem named FPGA Algorithm. This block has two inputs and one output

and is implemented as a multiplier. Replace this block with your own FPGA algorithm. Add inputs and outputs as required.

- Test Source Wrapper – This block includes a test source and is intended to generate stimulus as an input to the FPGA algorithm. Modify the test source to your needs, or replace it with an alternative source block. If the input to your FPGA algorithm is routed from an I/O block, such as HDMI or SDR, consider using a specific application template.

Note Not all templates include a Test Source block in the FPGA model.

Modify the Processor Model

The processor model includes a Task Manager block and a processor wrapper. The template implements the processor algorithm as a "pass through" wire. Open the processor algorithm wrapper, and replace the Processor Algorithm block (highlighted in blue) with your desired algorithm.

Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA mode,l or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1. Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.
2. Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

   For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.
3. Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

   For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

   For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.
4. From the top model, wire the new port to the Register Channel block.

See Also

Task Manager | Register Channel | Memory Controller | Memory Channel

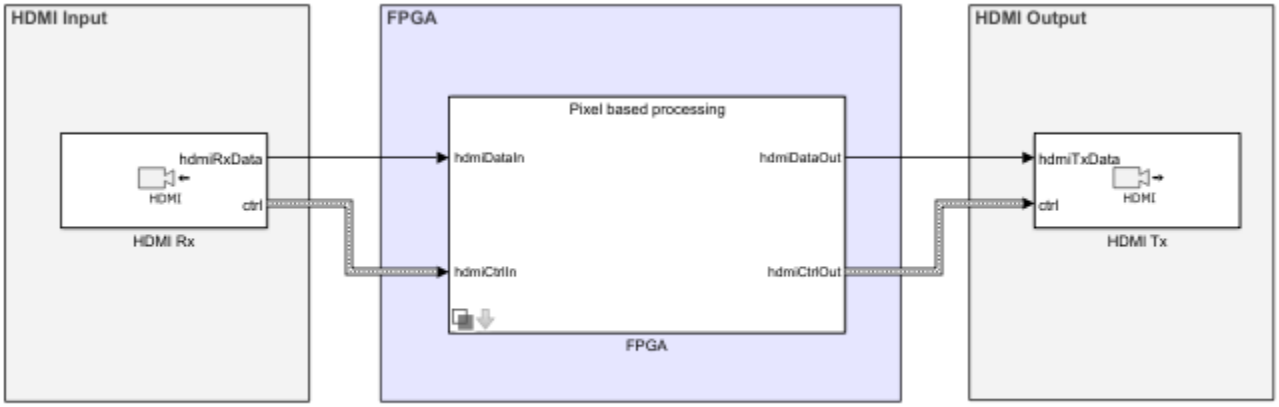
More About

- “Stream from FPGA to Processor Template” on page 2-14

- “Stream from Processor to FPGA Template” on page 2-18
- “SDR Template” on page 2-22
- “HDMI Template” on page 2-8
- “Frame Buffer with HDMI Template” on page 2-11
- “RFSoC Template” on page 2-25

HDMI Template

The High-Definition Multimedia Interface (HDMI) template provides a simulation model for SoC video streaming using SoC Blockset Support Package for Xilinx Devices. Use this template to simulate and analyze the effects of internal and external connectivity, such as HDMI I/O behavior on a vision processing algorithm.



Required Products

- Computer Vision Toolbox™
- Vision HDL Toolbox™
- SoC Blockset Support Package for Xilinx Devices

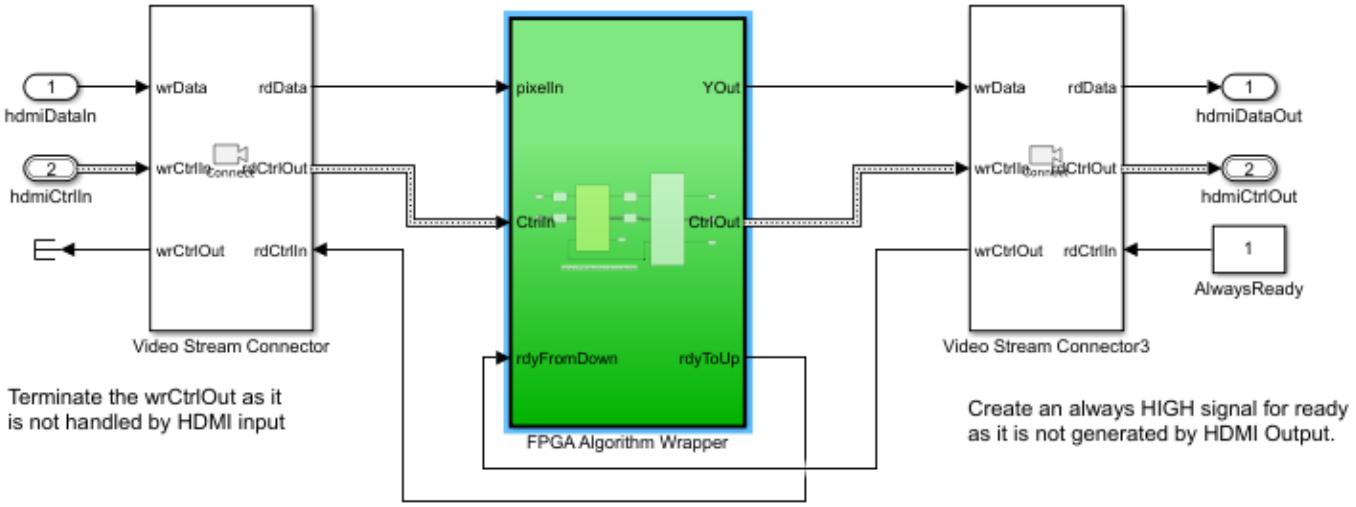
Template Structure

HDMI video streams from an HDMI Rx block into the FPGA, which implements a video data processing algorithm. The processed images stream to the HDMI Tx block.

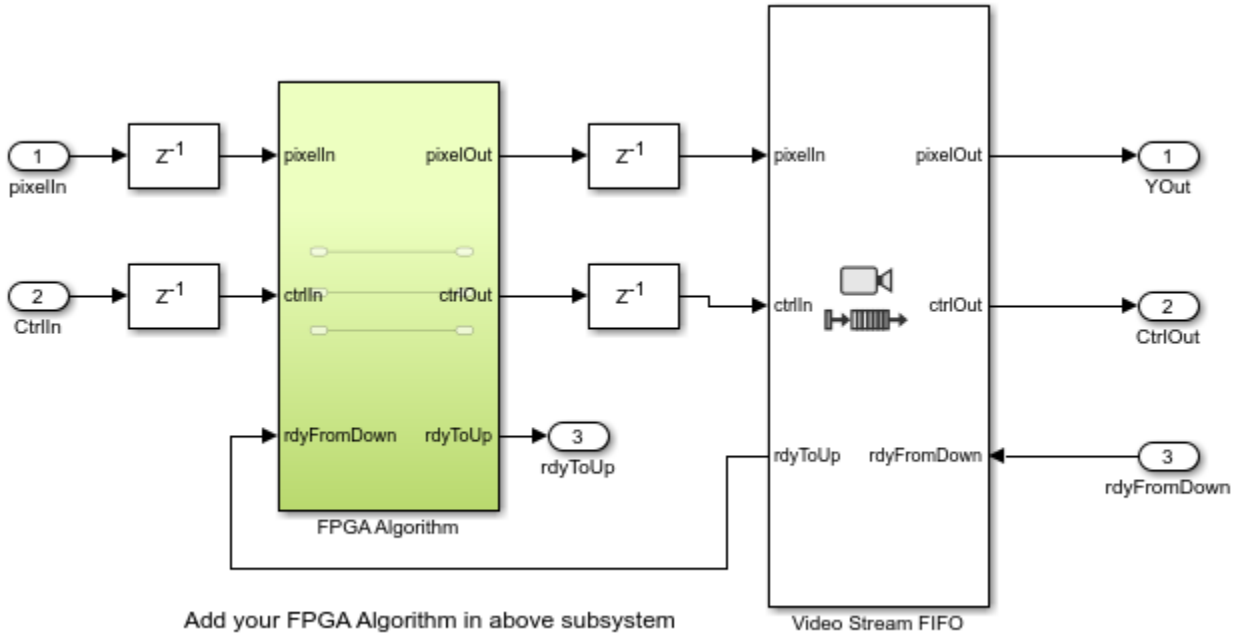
FPGA pixel model uses Video Stream Connector blocks to connect different subsystems and to connect to the HDMI I/O blocks. VideoStream Connector is required to generate each subsystem as a separate IP in the implemented reference design from the model. Since the FPGA frame model is for simulation purposes only and is not used for implementation, the Video stream connector blocks are not modeled.

Modify Project

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA pixel model**. Open the `FPGA Algorithm Wrapper`, as shown highlighted in green.



The `FPGA Algorithm`, also highlighted in green, contains feedthrough ports and signals.



You can modify the content of the FPGA algorithm model to incorporate your desired vision processing algorithm, with complete simulation and code generation of the surrounding video memory system. For pure algorithm design and investigation, click **Open FPGA frame model** in the **Project Shortcuts** tab, and repeat this step.

See Also

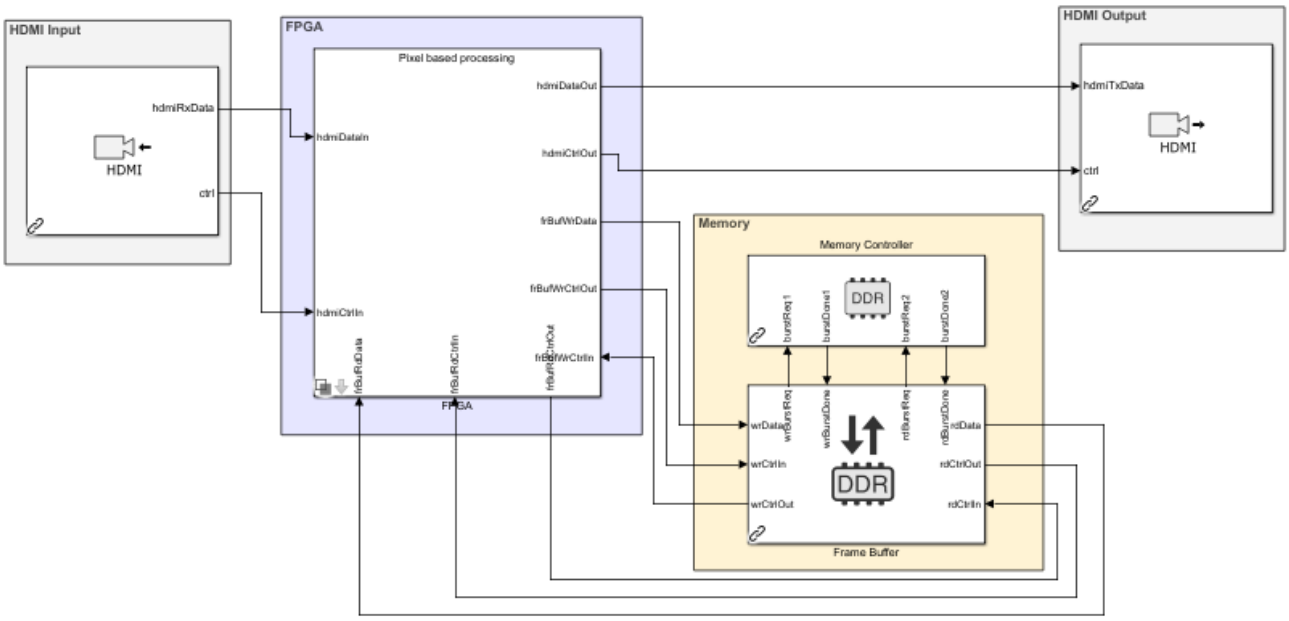
“Use Template to Create SoC Model” on page 2-4 | “Create a New Project Using Templates”

More About

- “What Are Projects?”

Frame Buffer with HDMI Template

The Frame Buffer with High-Definition Multimedia Interface (HDMI) template creates a Simulink project with models to simulate and generate a video application with external memory frame buffer. This template forms the base for the “Histogram Equalization Using Video Frame Buffer” on page 7-21 example. Use this template to simulate the full reference design of a video processing application on an FPGA with HDMI I/O and connection to an external memory frame buffer for advanced image processing designs.



Required Products

- Vision HDL Toolbox
- Computer Vision Toolbox
- SoC Blockset Support Package for Xilinx Devices

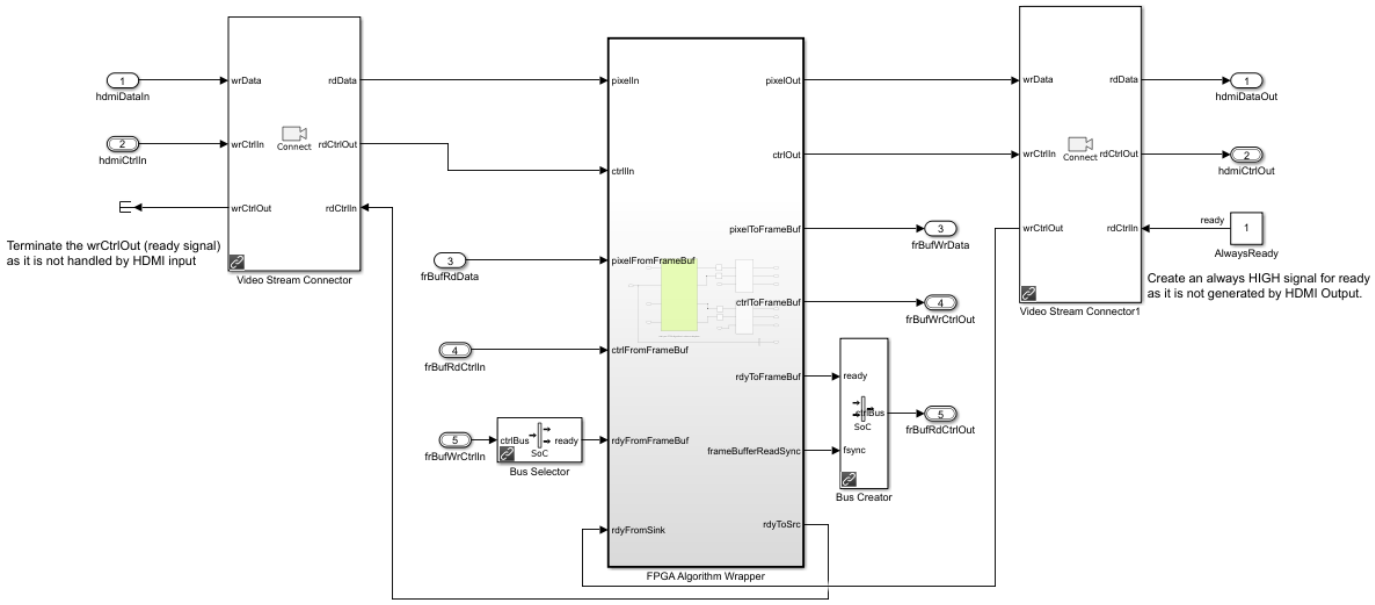
Template Structure

HDMI video streams video data from an HDMI Rx block into the FPGA. The FPGA implements a color-space transformation and your image processing algorithm. The processed images then undergo the inverse color-space transformation and stream to the HDMI Tx block. The FPGA algorithm is connected to the external memory frame buffer Memory Channel block configured in `AXI4-Stream Video Frame Buffer` mode.

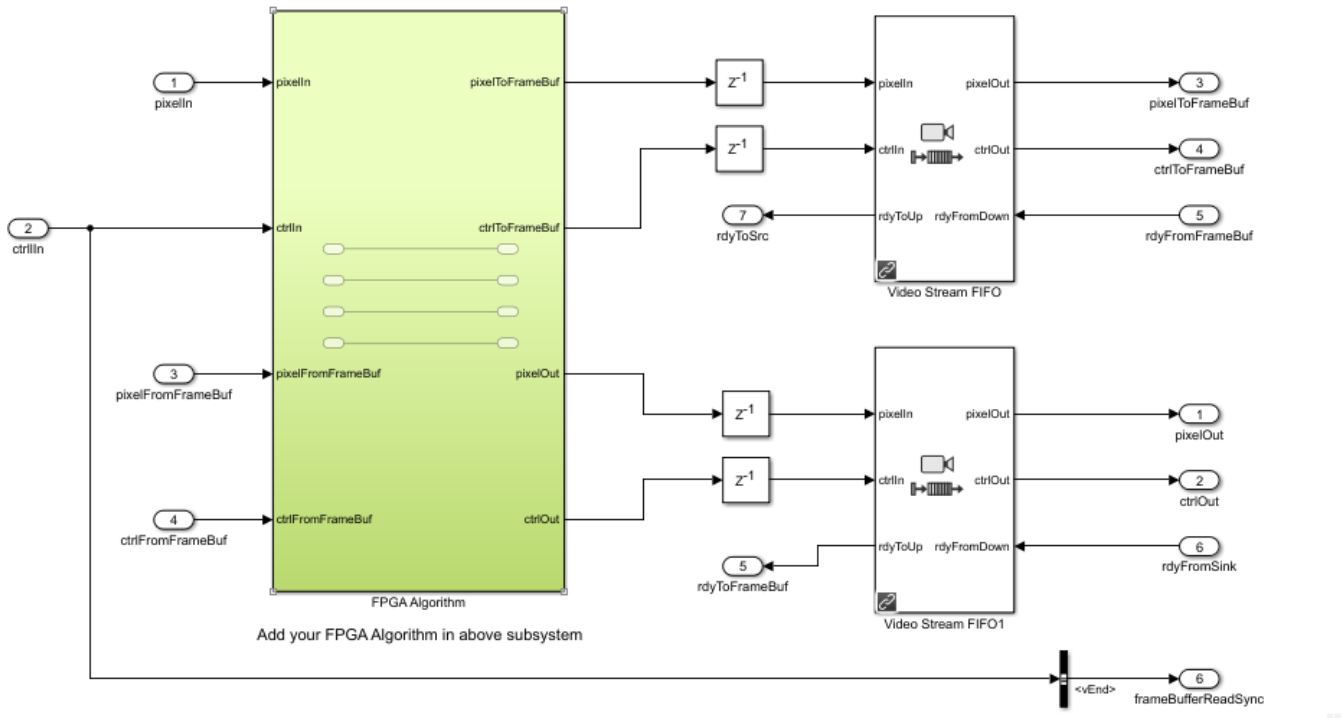
The FPGA pixel model uses Video Stream Connector blocks to connect different subsystems and to connect to HDMI I/O blocks. This is required to be able to generate each subsystem as a separate IP in the implemented reference design from the model. Since the FPGA frame model is for simulation purposes only and is not used for implementation, the Video Stream Connector blocks are not modeled.

Modify Project

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA pixel model**. Double-click to open the `FPGA Algorithm Wrapper`.



The `FPGA Algorithm`, highlighted in green, contains feedthrough ports and signals.



Modify the content of the `FPGA Algorithm` subsystem to incorporate your desired vision processing algorithm, with complete simulation and code generation of the surrounding video memory system.

The **pixelToFrameBuf** and **pixelFromFrameBuf** ports provide access to the external memory channel, `Frame Buffer`. For pure algorithm design and investigation, in the **Project Shortcuts** tab, click **Open FPGA frame model**, and repeat this step.

See Also

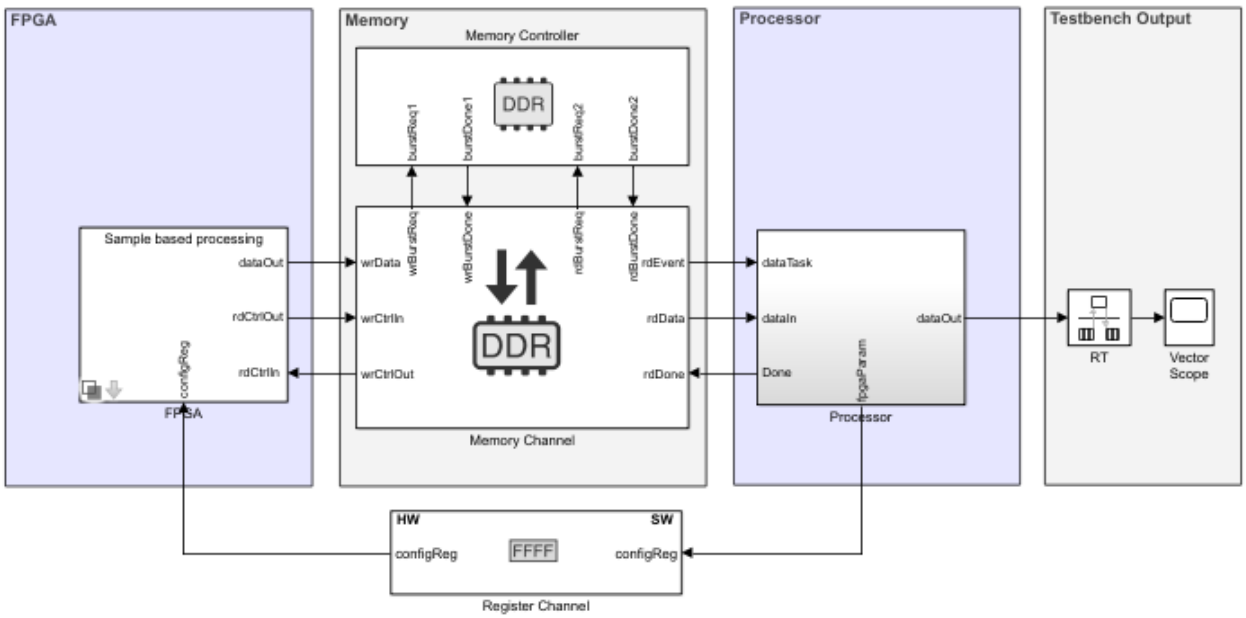
“Use Template to Create SoC Model” on page 2-4 | “Create a New Project Using Templates”

More About

- “What Are Projects?”
- “Histogram Equalization Using Video Frame Buffer” on page 7-21

Stream from FPGA to Processor Template

Use the *Stream from FPGA to Processor* template to create an SoC Blockset model for designing a datapath from hardware (FPGA) to software (Processor). To create a project using the "Stream to Processor" template, follow the steps to “Create SoC Model Using SoC Blockset Template” on page 2-4.



Required Products

For *sample-based* processing, no additional products are required.

For *frame-based* processing, DSP System Toolbox™ is required.

Template Structure

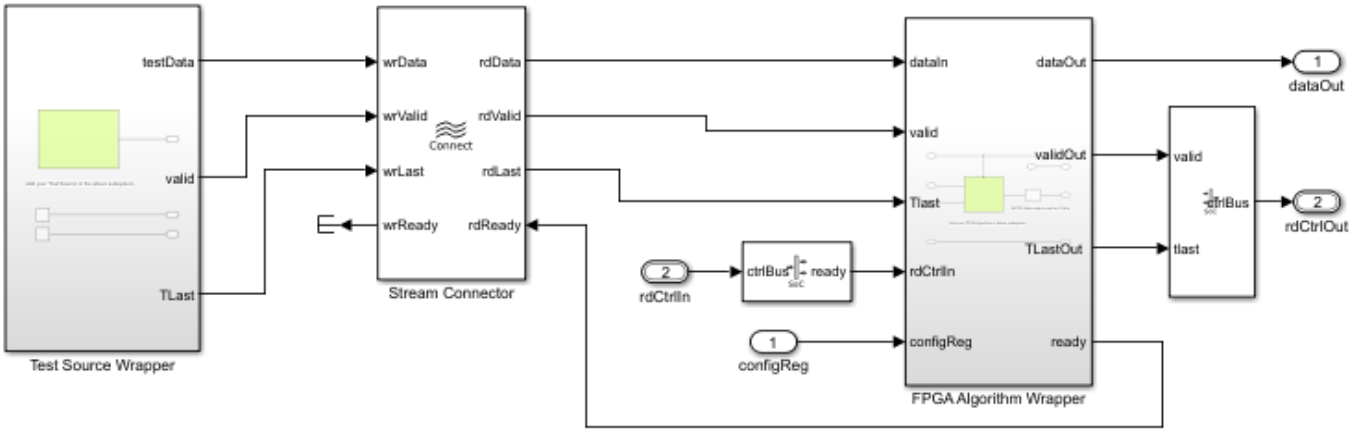
This template models a counter as the test data source and minimal logic for the FPGA and processor algorithms. Use this template as a guide and replace the FPGA algorithm and Processor algorithm with your own functionality. The FPGA algorithm is a simple multiplication performed on input data from the test source and from a **configReg** parameter. The processor writes the **configReg**. This parameter is modeled using the Register Channel block. Data from the FPGA algorithm is passed to the processor through a Memory Channel block. The memory **Channel Type** parameter is set to `AXI4-Stream to Software via DMA`, which models the DMA data transfer through shared external memory.

The processor reads the computed data from the memory and performs additional computing, which is implemented in the template as a pass-through wire. You can view the simulation results by double-clicking the Vector Scope block in the testbench sink.

Modify Project

Modify the FPGA Model

In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open FPGA sample model** to open the FPGA model. In the model, two areas are highlighted green, which represents user code: one in the FPGA Algorithm Wrapper block and one in the Test Source Wrapper block.



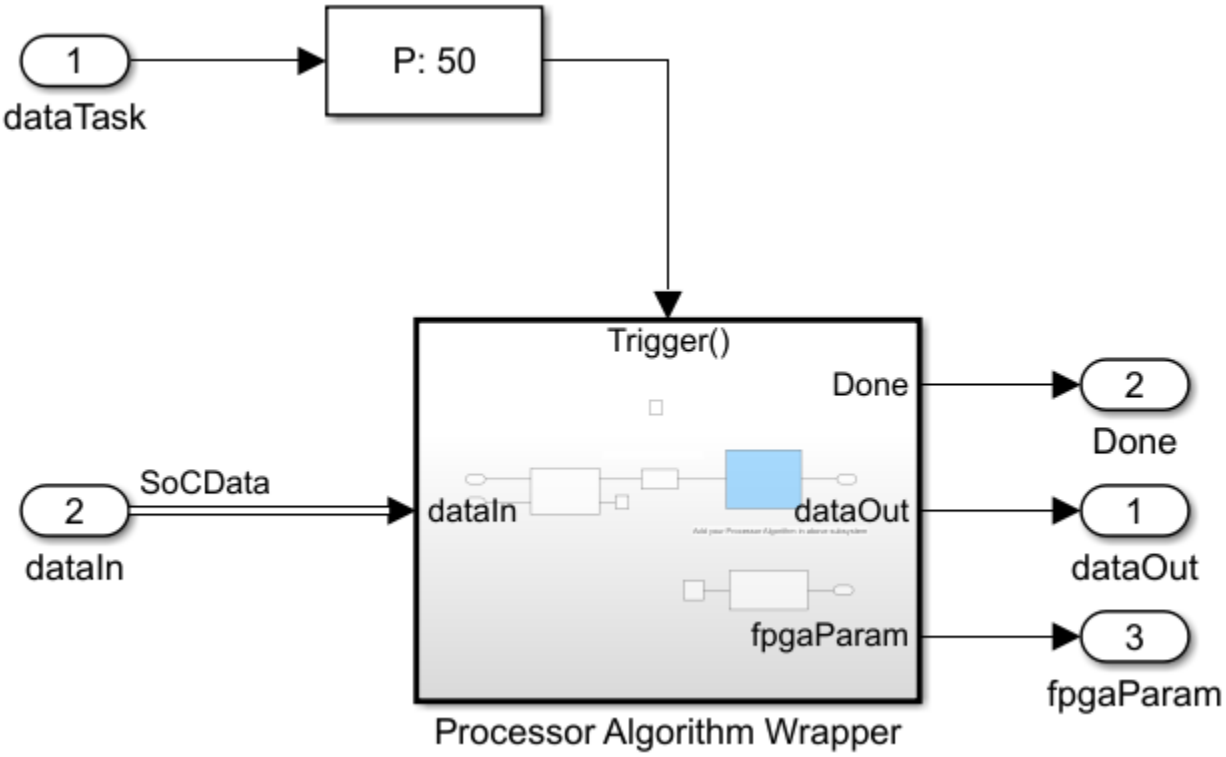
- FPGA Algorithm Wrapper – Double-click to open the model. The algorithm wrapper contains a green-highlighted subsystem named FPGA Algorithm. This block has two inputs and one output and is implemented as a multiplier. Replace this block with your own FPGA algorithm. Add inputs and outputs as required.
- Test Source Wrapper – This block includes a test source and is intended to generate stimulus as an input to the FPGA algorithm. This block is implemented as a counter in this template. Modify the test source to your needs, or replace it with an alternative source block.

Tip When your FPGA model includes more than one IP, you must define each IP as a subsystem and connect the subsystems using a Stream Connector or Video Stream Connector block. For additional information, see “Considerations for Multiple IPs in FPGA Model” on page 4-4.

To enable consistent simulation behavior, click **Open FPGA frame model** in the **Project Shortcuts** tab and repeat this step. To simulate frame-based processing, you must have a DSP System Toolbox license.

Modify the Processor Model

In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open Processor model**. The processor wrapper contains a blue highlighted subsystem representing the user code for the processor algorithm. Open the Processor Algorithm wrapper and replace the Processor Algorithm block with your desired algorithm.



Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA model, or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1. Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.
2. Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

   For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.
3. Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

   For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.

4 From the top model, wire the new port to the Register Channel block.

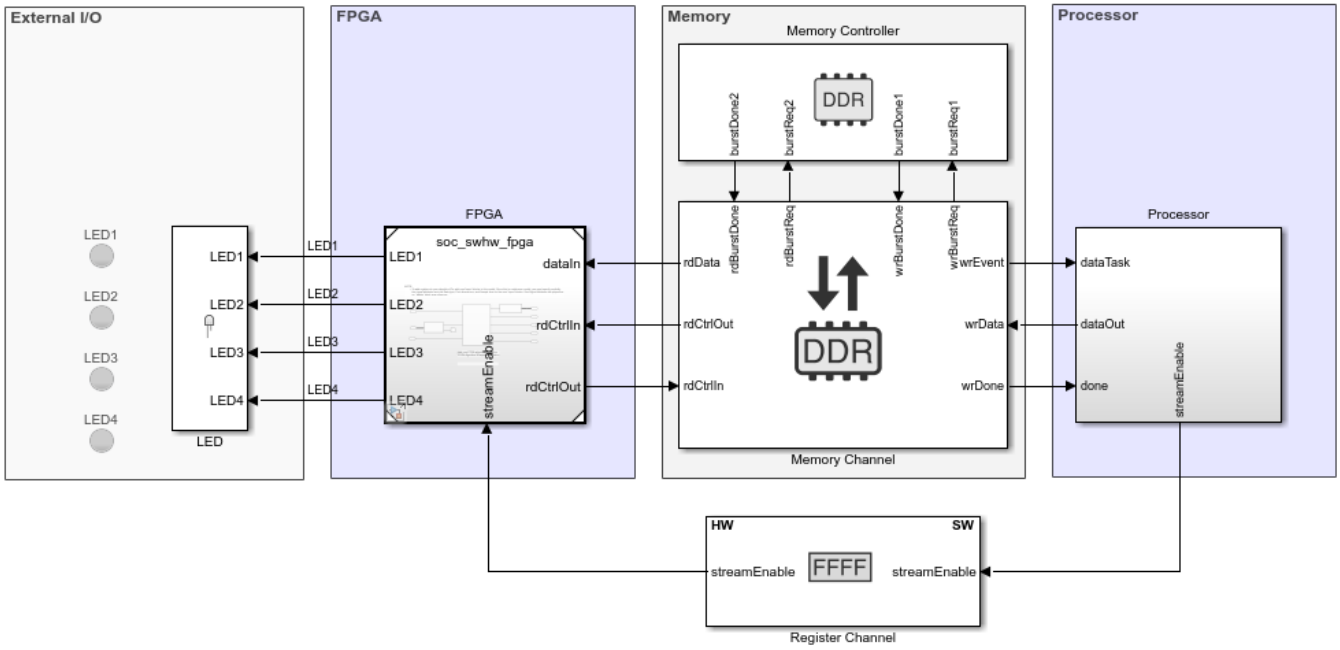
See Also

More About

- “Use Template to Create SoC Model” on page 2-4

Stream from Processor to FPGA Template

Use the Stream from Processor to FPGA template to create an SoC Blockset model for designing a datapath from software (Processor) to hardware (FPGA). To create a project using the Stream from Processor to FPGA template, follow the steps in “Create SoC Model Using SoC Blockset Template” on page 2-4. Then, add your FPGA algorithm in the FPGA subsystem and your processor algorithm in the Processor subsystem.



Template Structure

The Stream from Processor to FPGA template comprises three models: the Top model, the FPGA model, and the Processor model. This template models a counter as the test data source and minimal logic for the FPGA and processor algorithms. Use this template as a guide, replacing the FPGA algorithm and processor algorithm with your own functionality.

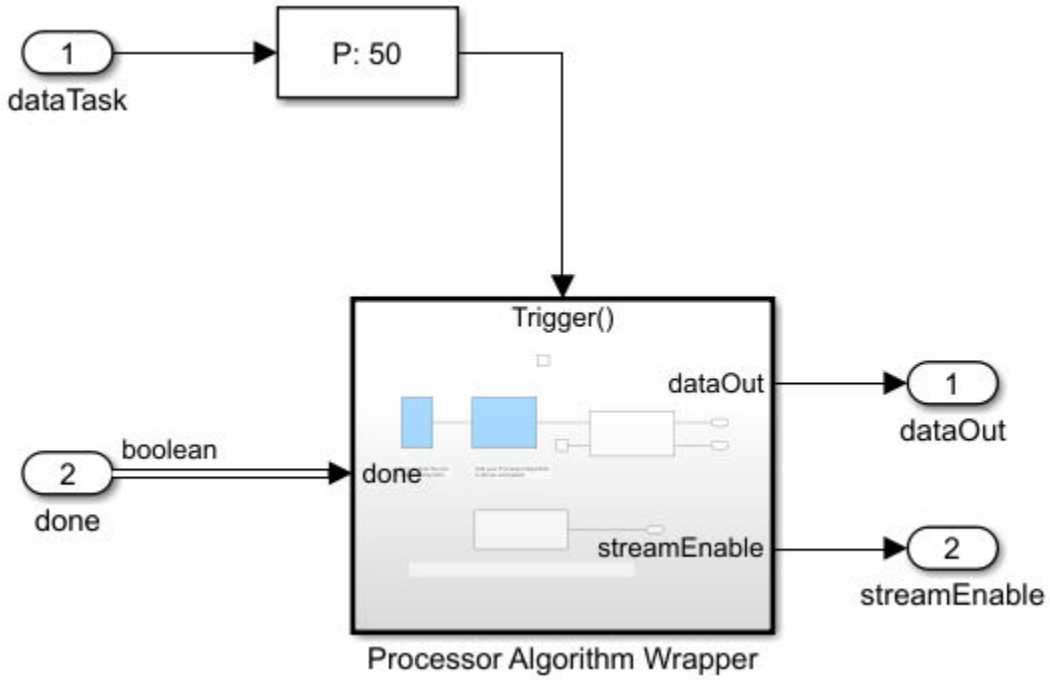
The processor controls the output **ready** signal in the FPGA Algorithm Wrapper subsystem by using the **streamEnable** port. The processor passes the data to the FPGA through a Memory Channel block. The **Channel Type** parameter in the Memory Channel block is set to `Software to AXI4-Stream via DMA` to model the direct memory access (DMA) data transfer through shared external memory.

The processor generates test data and performs additional computing. The additional computation is implemented in the template as a pass-through wire. Then, the processor writes the computed data to the memory.

Modify Project

Modify the Processor Model

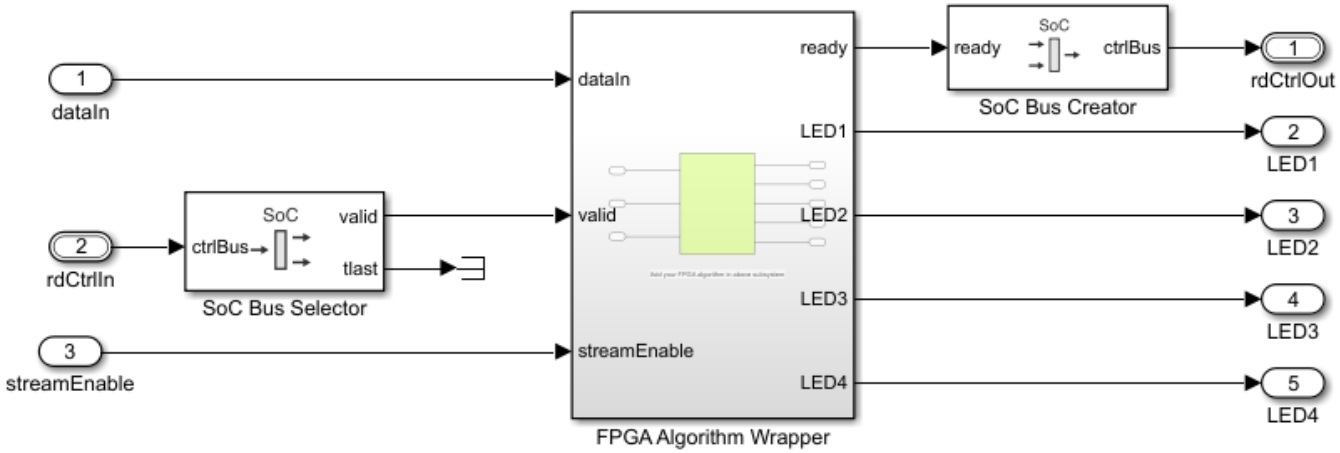
In the MATLAB Toolstrip, on the **Project Shortcuts** tab, click **Open Processor model** to open the processor model. In the Processor Algorithm Wrapper subsystem, two areas are highlighted blue (which represents user code) as shown in this figure. One highlighted area appears in the Processor Algorithm block, and the other highlighted area appears in the Test Source block.



- Processor Algorithm block – Replace the internal Processor Algorithm block (highlighted in blue) with your desired algorithm.
- Test Source block – This block generates a ramp signal. Modify the test source to your needs or replace it with an alternative source block.
- Stream Enable for DUT block – This block contains a control logic to ensure that the memory is primed before continuous streaming begins. In the control logic, the **streamEnable** signal is asserted high only after available buffers in the memory channel are filled completely.

Modify the FPGA Model

In the MATLAB Toolstrip, on the **Project Shortcuts** tab, click **Open FPGA model** to open the FPGA model. In the FPGA Algorithm Wrapper subsystem, the FPGA Algorithm block is highlighted green (which represents user code).



Double-click the FPGA Algorithm Wrapper subsystem to open the model. The FPGA algorithm extracts four bits from the input data to drive the LEDs on the hardware. The status of these LEDs indicates that the processor is writing stream data to the FPGA. Replace this block with your own FPGA algorithm. Add inputs and outputs as required.

Tip When your FPGA model includes more than one IP, define each IP as a subsystem and connect the subsystems using a Stream Connector or Video Stream Connector block. For additional information, see “Considerations for Multiple IPs in FPGA Model” on page 4-4.

To enable consistent simulation behavior, on the **Project Shortcuts** tab, click **Open FPGA model** tab and repeat this step.

Modify the Register Channel

The top model of the template includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA model or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers, modify the Register Channel block parameters, the FPGA algorithm, and the processor algorithm by following these steps.

1. Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see the Register Channel block.
2. Add ports to the Processor model – Navigate to the Processor Algorithm Wrapper subsystem. To navigate to the Processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click the Processor Algorithm Wrapper subsystem to modify it.

   For write registers, add an output port and logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.
3. Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper subsystem. To navigate to the FPGA model, click **Open FPGA model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper subsystem to modify it.

For write registers, add an input port and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

4 From the top model, wire the new port to the Register Channel block.

See Also

“Use Template to Create SoC Model” on page 2-4

See Also

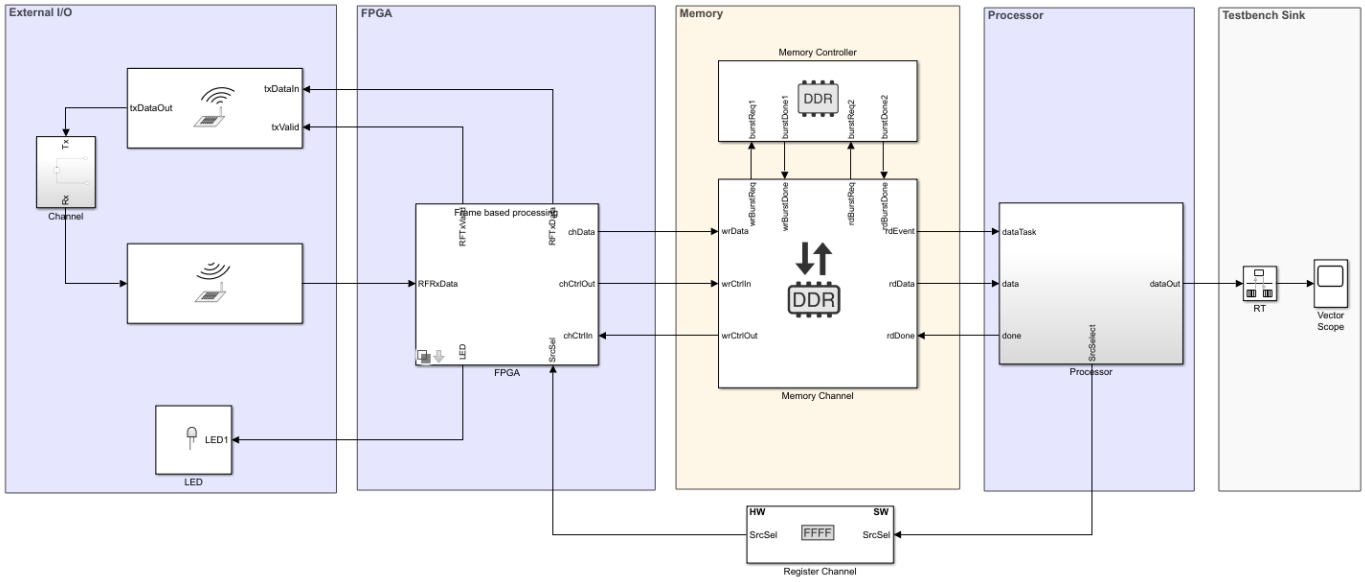
Related Examples

- “Streaming Data from Software to Hardware” on page 7-43

SDR Template

The software defined radio (SDR) template provides a simulation model for an SoC reference design available from Communications Toolbox™ Support Package for Xilinx Zynq-Based Radio. Use this template to simulate the full reference design and analyze the effects of internal and external connectivity on and SDR algorithm, such as memory behavior and Radio Frequency (RF) I/O behavior.

To get started with SoC Blockset model for designing an SDR system, follow the steps to “Create SoC Model Using SoC Blockset Template” on page 2-4.



Required Products

- Communications Toolbox
- SoC Blockset Support Package for Xilinx Devices

Template Structure

This template models an SDR transceiver composed of AD9361 transmitter and receiver blocks. The transceiver connects an RF channel to the FPGA, which implements a receiver and a transmitter algorithm. The FPGA algorithm includes a Test Source block, which generates a sinusoid signal and drives it to the transmitter. The FPGA algorithm also includes a Tx algorithm, implemented as simple pass-through wires, and an Rx algorithm, implemented as a gain block. A configuration register **srcSel** is modeled in the FPGA to select the source of data for the Rx algorithm. The processor writes the **srcSel** register to select either the test source from the FPGA or RF data from the transceiver. This register is modeled using the Register Channel block. Data from the FPGA algorithm is passed to the processor through a Memory Channel block.

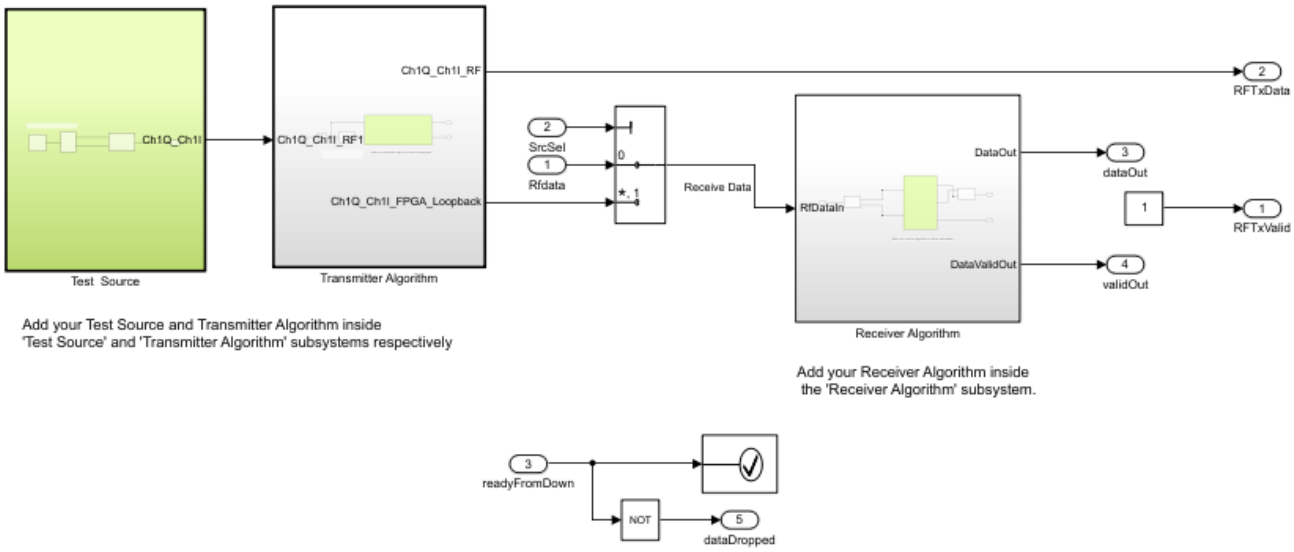
Use this template as a guide and replace the Rx Algorithm and Tx Algorithm in the FPGA and the Processor Algorithm in the processor with your own functionality. The memory **Channel Type** parameter is set to `AXI4-Stream to software via DMA`, which models the direct memory access (DMA) data transfer through shared external memory.

The processor reads the computed data from the memory, and performs additional computing (implemented in the template as a pass-through wire). You can view the simulation results by double-clicking the Vector Scope block in the testbench sink.

Modify Project

Modify the FPGA Model

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA sample model**. Then, open the FPGA Tx-Rx Alg Wrapper. Notice three areas highlighted in green. These areas represent user code and are located in the Receiver Algorithm block, in the Transmitter Algorithm block, and the Test Source block.



The FPGA model includes the following sections for you to modify (highlighted in green):

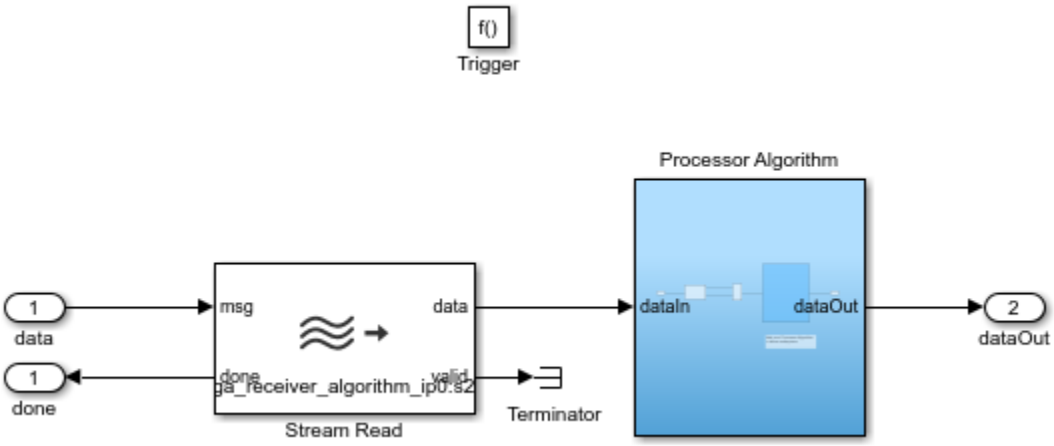
- Test Source block – This block generates a 10-kHz sinusoid signal and drives it to the transmitter algorithm. Modify the test source to your needs or replace it with an alternative source block.
- Receiver Algorithm subsystem – Inside the green-highlighted subsystem named Rx Algorithm, there is a block labeled Algorithm. The algorithm takes I/Q data as input and output with a valid signal. Replace this block with your own Rx algorithm.
- Transmitter Algorithm – Inside the green-highlighted subsystem named Tx Algorithm, the algorithm has an input from the test source and two output signals: one to the RF channel and one to the FPGA. Replace this block with your own Tx algorithm.

To enable consistent simulation behavior, in the **Project Shortcuts** tab, click **Open FPGA frame model** and repeat this step.

Modify the Processor Model

In MATLAB, on the **Project Shortcuts** tab, click **Open processor model**. The subsystem highlighted in blue represents the user code for the processor algorithm. Open the Processor Algorithm Wrapper

and replace the internal Processor Algorithm block (also highlighted in blue) with your desired algorithm.



Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA model or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1. Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.
2. Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

   For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.
3. Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

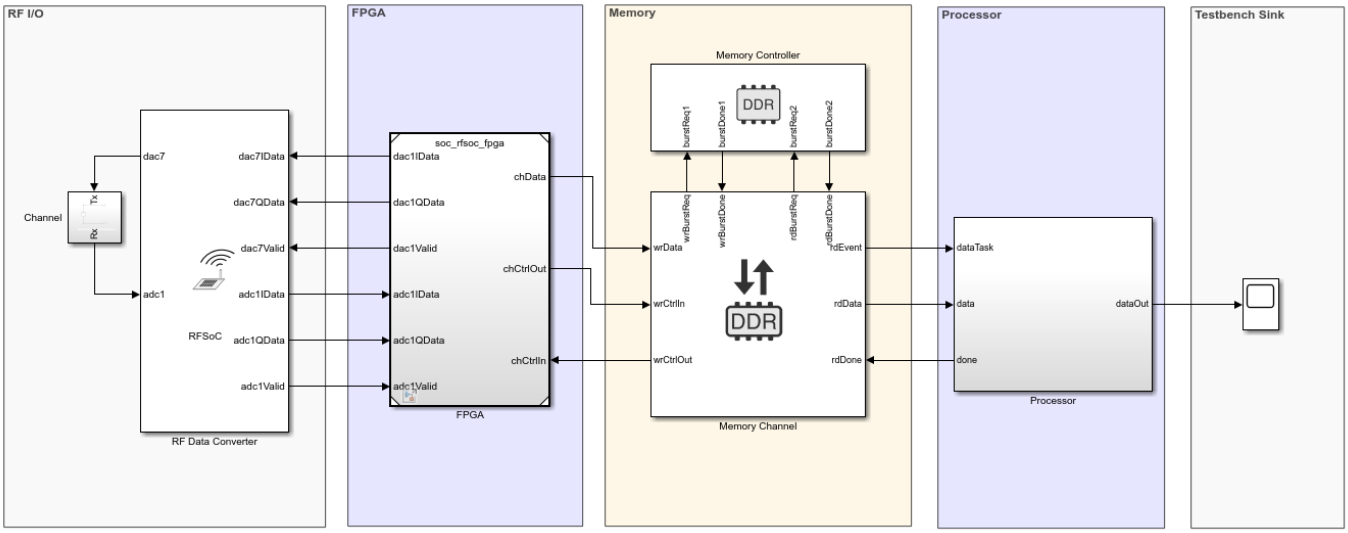
   For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

   For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.
4. From the top model, wire the new port to the Register Channel block.

RFSoC Template

The RFSoC template provides a simulation model for an SoC reference design available from SoC Blockset Support Package for Xilinx Devices. Use this template to simulate the full reference design. Analyze the effects of internal and external connectivity on transmit and receive communication algorithms, such as memory behavior and Radio Frequency (RF) input/output (I/O) behavior.

To get started with the SoC Blockset model for designing an RFSoC-enabled wireless communication system, follow the steps in “Create SoC Model Using SoC Blockset Template” on page 2-4. Then, add your transmitter and receiver algorithms in the FPGA subsystem and your processor algorithm in the Processor subsystem.



Required Products

- DSP System Toolbox
- SoC Blockset Support Package for Xilinx Devices

Template Structure

The RFSoC template comprises three models: the Top model, the FPGA model, and the Processor model. In the Top model, the RF Data Converter block connects to the FPGA block and provides an RF I/O interface. The FPGA block implements receiver and transmitter algorithms. The FPGA algorithm includes a Test Source block, which generates a sinusoidal signal and drives it to the digital-to-analog converter (DAC) through the Transmitter Algorithm subsystem. The Transmitter Algorithm subsystem is implemented as simple pass-through wires, and the Receiver Algorithm subsystem is implemented using down-sampler logic. The configuration register **SrcSelReg** is modeled in the FPGA to select the data source for the Receiver Algorithm subsystem. The processor writes the **SrcSelReg** register to select either the test source from the FPGA or the RF data from the analog-to-digital converter (ADC) in the RF Data Converter block. Data from the FPGA subsystem is passed to the Processor subsystem through a Memory Channel block. The **Channel Type** parameter in the Memory Channel block is set to `AXI4-Stream to software via DMA`, which models the direct memory access (DMA) data transfer through shared external memory.

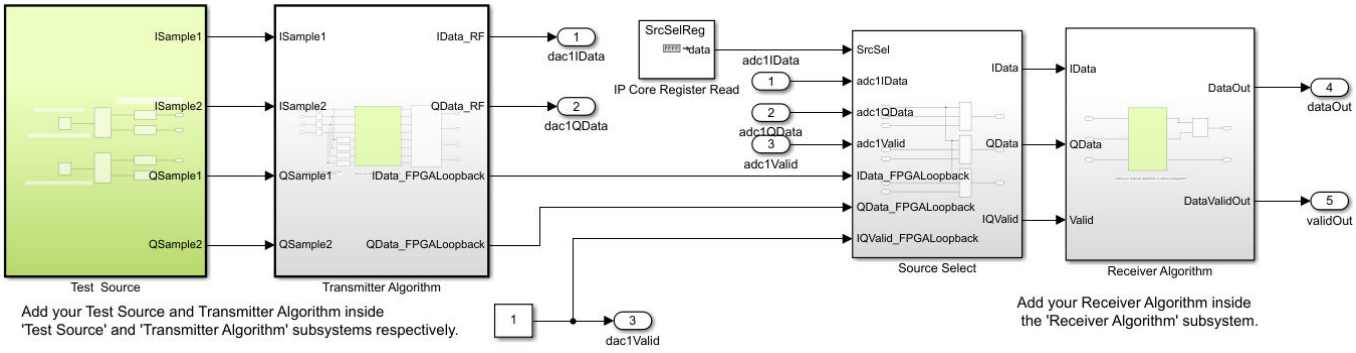
Use this template as a guide, replacing the Receiver Algorithm and Transmitter Algorithm subsystems in the FPGA model and the Processor Algorithm subsystem in the Processor model with your own functionality. In this template, the RF Data Converter block is configured with a custom RF interface that has one DAC (DAC7) and one ADC (ADC1) and an I/Q digital interface. The number of samples per clock cycle is set to 2. The RF path in this template is looped back. If you want to use different data for the transmitter and receiver, you can remove the loop back and can provide your desired input data to the ADC.

The processor reads the computed data from the memory and performs additional computing (implemented in the template as a pass-through wire). You can view the simulation results by double-clicking the Time Scope block in the Testbench Sink subsystem.

Modify Project

Modify the FPGA Model

In the MATLAB Toolstrip, on the **Project Shortcuts** tab, click **Open FPGA model**. Then, open the FPGA Tx-Rx Alg Wrapper subsystem. Three areas are highlighted in green, as shown in this figure. These areas represent user code and are located in the Test Source block, the Transmitter Algorithm subsystem, and the Receiver Algorithm subsystem.



The FPGA model includes these sections (highlighted in green) for you to modify.

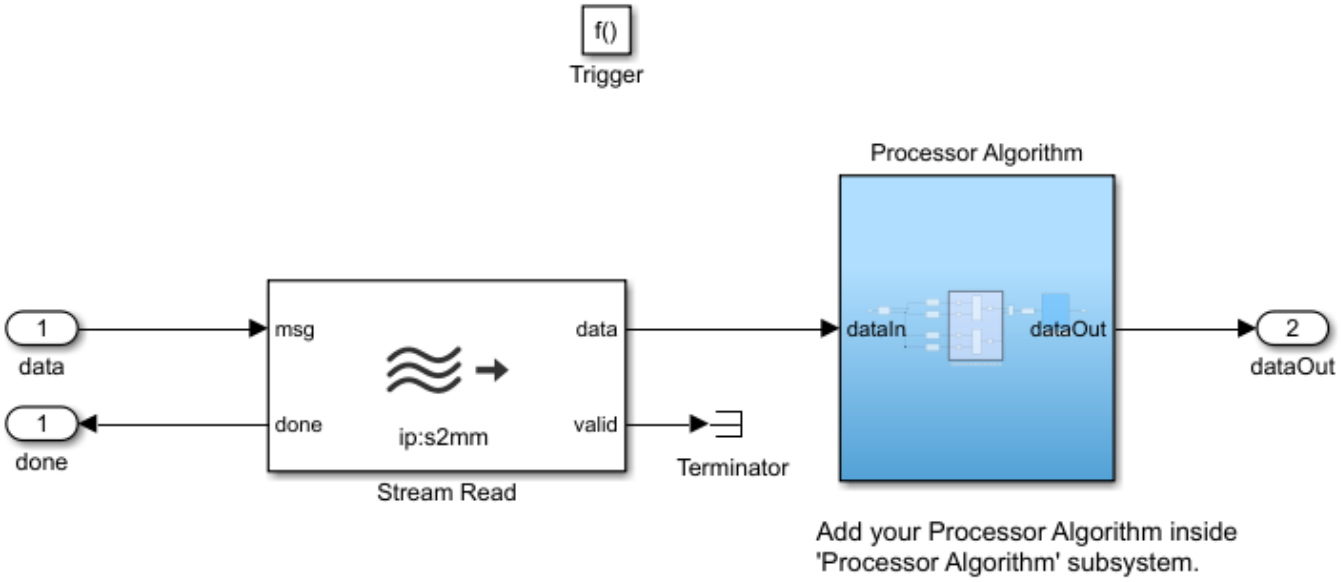
- Test Source block – This block generates a 500 kHz sinusoidal signal and drives it to the Transmitter algorithm subsystem. Modify the test source to your needs or replace it with an alternative source block.
- Transmitter Algorithm subsystem – Contains Tx Algorithm subsystem highlighted in green. Replace the Tx Algorithm subsystem with your own transmitter algorithm.
- Receiver Algorithm subsystem – Contains Rx Algorithm subsystem highlighted in green. Replace the Rx Algorithm subsystem with your own receiver algorithm.
- IP Core Register Read blocks – Inside the design under test (DUT), add these blocks to add registers to your algorithm IPs. Initialize this block using a corresponding Register Write block in the Processor model.

To enable consistent simulation behavior, on the **Project Shortcuts** tab, click **Open FPGA model** and repeat this step.

Modify the Processor Model

In the MATLAB Toolstrip, on the **Project Shortcuts** tab, click **Open Processor model**. The Processor model contains the Processor Algorithm Wrapper and the Initialize Function subsystems.

The Processor Algorithm Wrapper subsystem is highlighted in blue, which represents the user code for the processor algorithm. Open the Processor Algorithm Wrapper subsystem and replace the internal Processor Algorithm subsystem (also highlighted in blue) with your desired algorithm. Open the Initialize Function subsystem and add a Register Write block for each IP Core Register Read block added in the FPGA model.



See Also

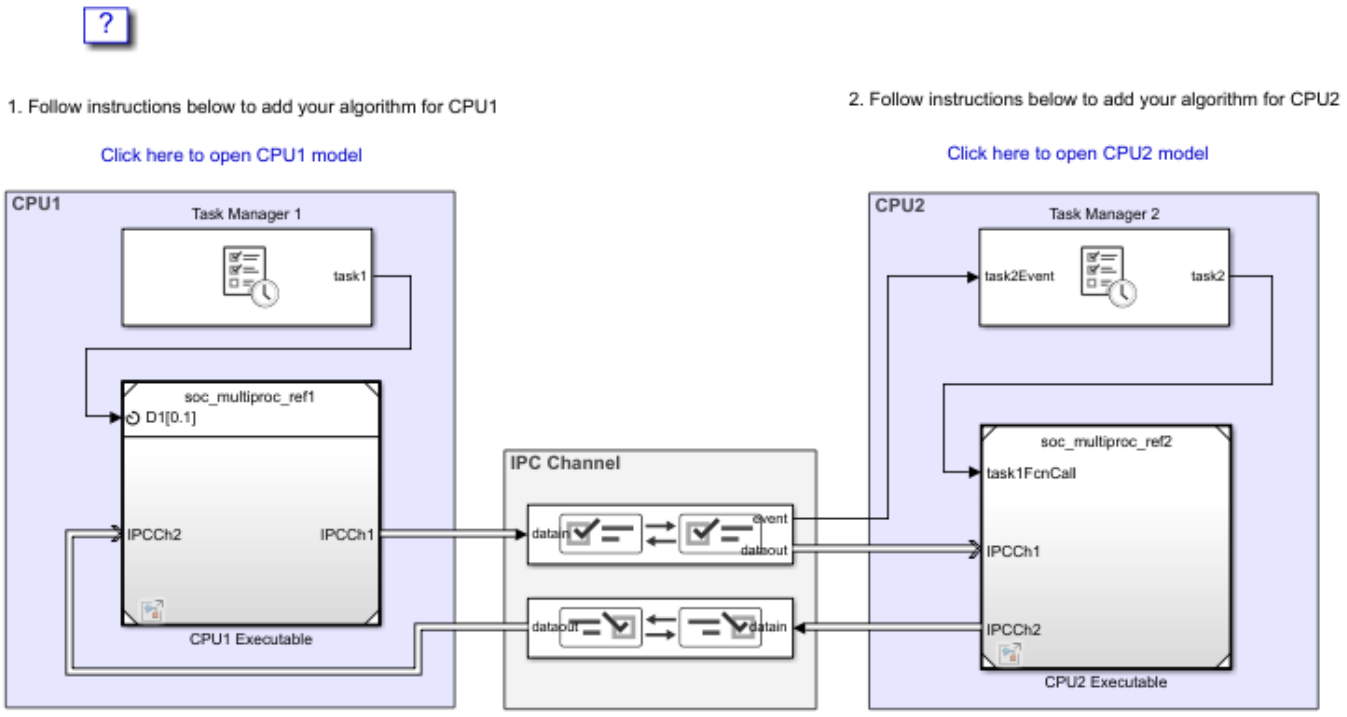
“Use Template to Create SoC Model” on page 2-4

Related Examples

- “Transmit and Receive a Tone Using Xilinx RFSoC Device - Part 1 System Design” (SoC Blockset Support Package for Xilinx Devices)
- “Transmit and Receive a Tone Using Xilinx RFSoC Device - Part 2 Deployment” (SoC Blockset Support Package for Xilinx Devices)

Multiprocessor Architecture Template

To create an SoC Blockset model for designing a system with two CPUs connected by Interprocess Data Channel blocks, use the Multiprocessor Architecture template. To create a project using the "Multiprocessor Architecture" template, follow the steps in the topic “Create SoC Model Using SoC Blockset Template” on page 2-4.



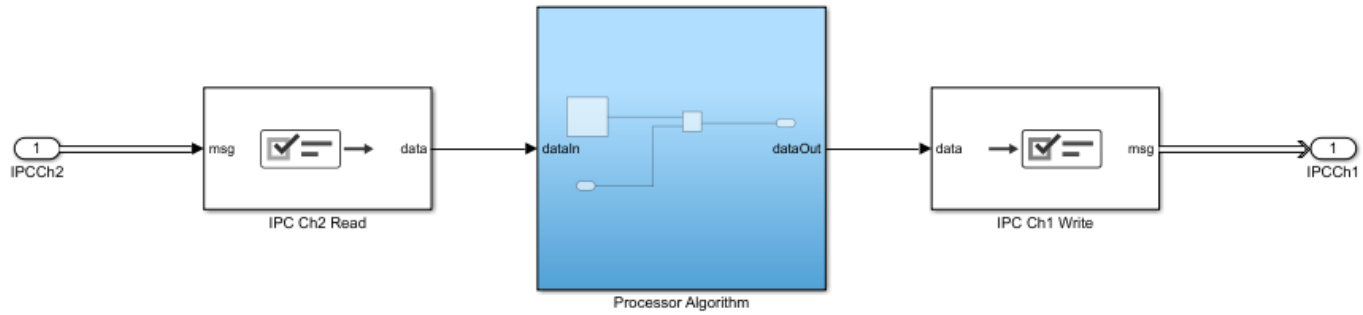
Template Structure

This template models two CPUs that are connected with a pair of interprocess communication channels. Use this template as a guide and replace the algorithms in the reference models. The Task Manager block in CPU1 executes a timer-driven task to run the CPU1 Executable reference model with the output, **IPCCh1**, sent to the Interprocess Data Channel block. The Task Manager block in CPU2 executes an event-driven task to run the CPU2 Executable reference model after using the input, **IPCCh1**, which is the output from CPU1. When the event-driven task completes, it outputs data, **IPCCh2**, to the Interprocess Data Channel block to return the result to CPU1.

Modify Project

Modify CPU1 Processor Model

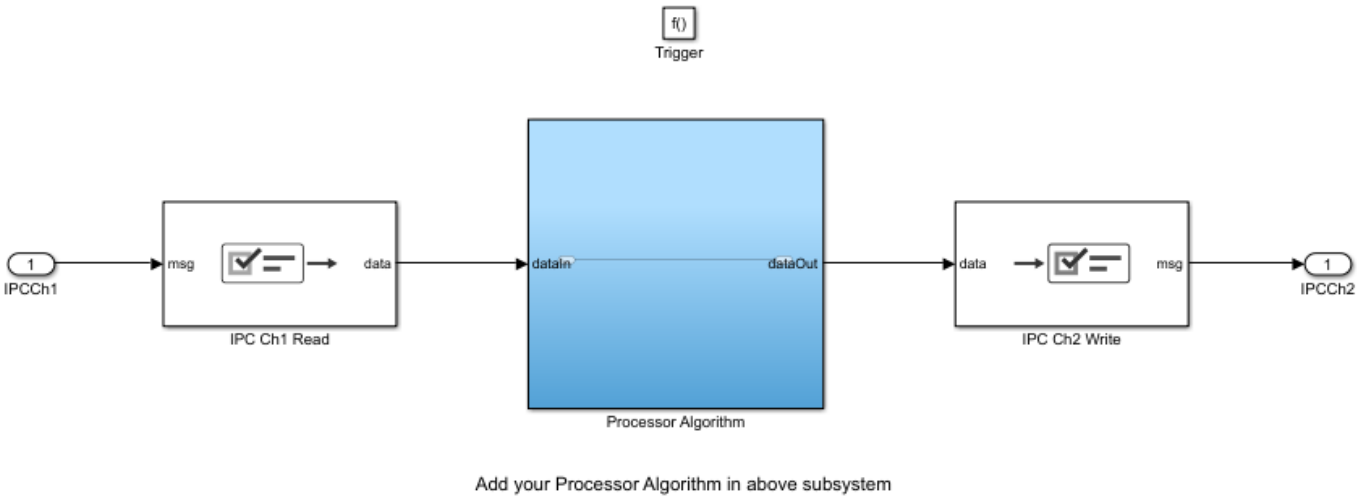
In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open CPU1 Processor model**. The processor wrapper contains a blue highlighted subsystem representing the user code for the processor algorithm. Open the Processor Algorithm subsystem and replace the Processor Algorithm block with your desired algorithm.



- `Processor Algorithm` — This block has one input and one output, implementing an increment operation. Replace this block with your own processor algorithm. Add inputs and outputs as required.
- `IPC Ch2 Read` — This Interprocess Data Read block reads available data from the Interprocess Data Channel block sent from CPU2.
- `IPC Ch1 Write` — This Interprocess Data Write block writes data to the Interprocess Data Channel block to be read by CPU2.

Modify CPU2 Processor Model

In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open CPU2 Processor model**. Double-click the Task1 model block to open the task. The processor wrapper contains a blue highlighted subsystem representing the user code for the processor algorithm. Open the Processor Algorithm wrapper and replace the Processor Algorithm block with your desired algorithm.



- `Processor Algorithm` — This block has one input and one output, implementing an increment operation. Replace this block with your own processor algorithm. Add inputs and outputs as required.
- `IPC Ch1 Read` — This Interprocess Data Read block reads available data from the Interprocess Data Channel block sent from CPU1.

- `IPC Ch2 Write` — This Interprocess Data Write block writes data to the Interprocess Data Channel block to be read by CPU1.

Modify IPC Channel

The top model of this template also includes two Interprocess Data Channel blocks, which create a bidirectional communication path between CPU1 and CPU2. If you need to transfer more data between the two CPUs, you can add additional Interprocess Data Channel blocks or bundle data into the existing Interprocess Data Channel blocks.

See Also

Interprocess Data Channel | Task Manager

More About

- “Use Template to Create SoC Model” on page 2-4
- “Interprocess Data Communication via Dedicated Hardware Peripheral” on page 3-31

Create an SoC Project Application

A system-on-chip (SoC) project developed using the SoC Blockset typically contains many diverse systems that make up a the complete application. These systems can include:

- Embedded processors with timer-driven and event-driven tasks.
- FPGAs with custom IP logic and timing.
- External memory systems with interaction to embedded processors and FPGAs.
- I/O device interaction, such as TCP/IP and UDP connections.

This example shows the steps to create an SoC application, using the features of the SoC Blockset, as a Simulink project. To begin, see “Project and Top-Level Model” on page 2-32.

Note This project is equivalent to the project automatically created by the “Stream from FPGA to Processor Template” on page 2-14. Templates are the recommended and preferred method for creating new projects. This example should be used for information purposes only.

See Also

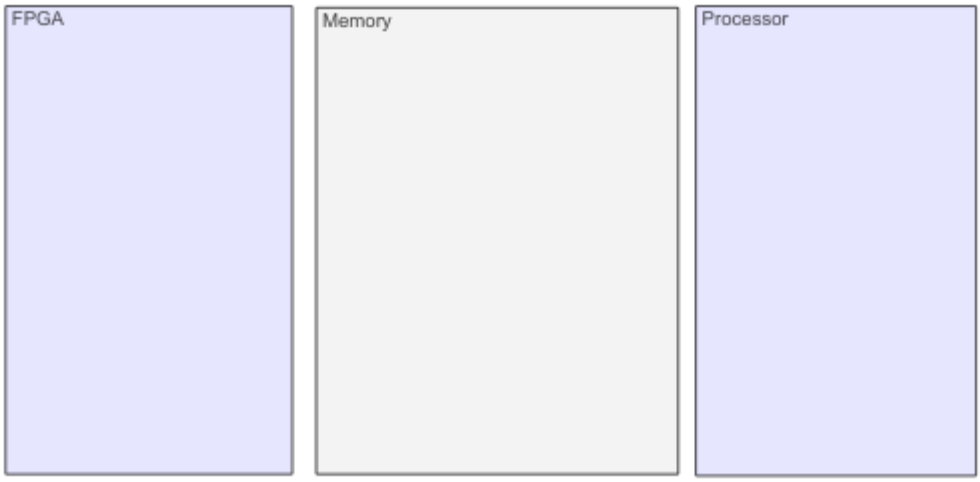
“Use Template to Create SoC Model” on page 2-4 | “Stream from FPGA to Processor Template” on page 2-14

Project and Top-Level Model

An SoC application model developed using the SoC Blocksetcombines multiple subsystems and reference models. Each subsystem and reference model maps to a particular feature of an SoC device. Organization of the models and shared configuration settings requires a Simulink project.

1. Create a new SoC Blockset project named `SampleSoCApplication`. Creating a new project automatically creates a new project folder with the same name. For more information on creating projects, see “Create a New Project From a Folder”.
2. Open a new Simulink model. Save the model as `soc_hwsw_top.slx` into the project folder.
3. In MATLAB, on the **Project** tab, in the **Tools** section, select **Run Checks** > **Add Files** and add the `soc_hwsw_top.slx` model file to the project.
4. In Simulink, configure the `soc_hwsw_top.slx` model to as an SoC application. On the **Apps** tab, under **Setup to Run on Hardware**, click **System on Chip (SoC)**.
5. In the **System on Chip (SoC)** pop-up window, select **Hardware Board** > **Xilinx Zynq ZC706 evaluation kit**. Click **Finish**.

   Note You can optionally choose any of the available hardware boards based to suit your system requirements.
6. On the **System on Chip** tab, click **Hardware Settings**. On the **Configuration Parameters** dialog box, in the **Solver** tab, set **Solver selection** > **Type** to `Variable-step`. Click **OK**.
7. Create three box areas and label them as `FPGA`, `Memory`, and `Processor`. For more information on creating box areas, see “Box and Label Areas of a Model”. In the following sections, these areas are populated for various aspects of your SoC application.



8. Create a new MATLAB function to initialize variables used throughout the project.

```
function soc_hwsw_init
% Initialize the model wide variables and set them in base workspace.

SourceSTime = 1e-7;
```

```
FrameSize = 1000;
ProcSTime = SourceSTime*FrameSize;
FPGASTime = SourceSTime;
FPGAFrameSize = 1;

assignin('base','ProcSTime',ProcSTime);
assignin('base','FPGASTime',FPGASTime);
assignin('base','SourceSTime',SourceSTime);
assignin('base','FPGAFrameSize',FPGAFrameSize);
assignin('base','FrameSize',FrameSize);

end
```

In the project folder, save the file as `soc_hwsw_init.m` in a new subfolder, `utilities` and add the file to project.

See Also

“Software and Task Management on Processor” on page 2-34

More About

- “Create a New Project From a Folder”
- “Box and Label Areas of a Model”

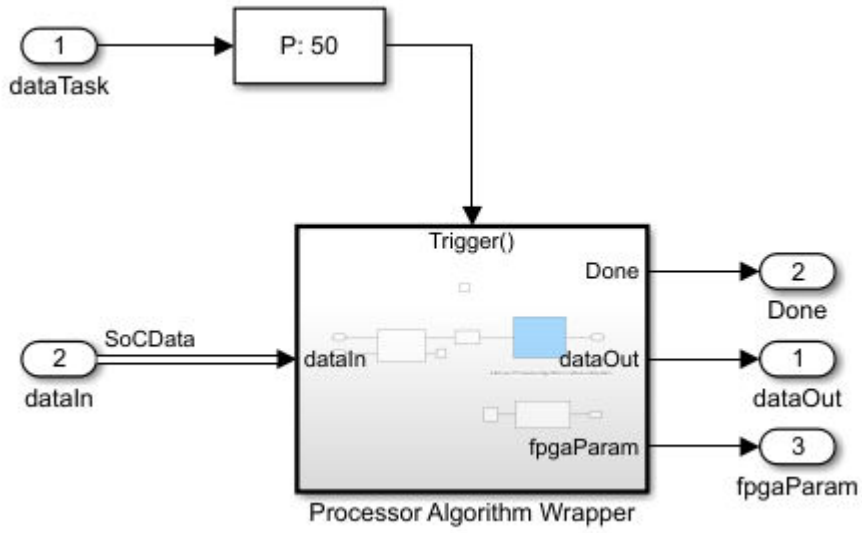
Software and Task Management on Processor

The processor system in this SoC application reads data from the external memory following a write from the FPGA to that memory. Since FPGA writes and interaction with external memory are asynchronous, the processor uses an event-driven task to read from memory. The software also manages a register on the FPGA that specifies a multiplication factor to be used in the FPGA algorithm.

Processor Model

1. Open a new Simulink model. Save the model as `soc_hwsw_proc.slx` into a new subfolder, named `processor`, in the project folder. Add the `soc_hwsw_proc.slx` model to the project.
2. In Simulink, configure the `soc_hwsw_top.slx` model to as an SoC application. On the **Apps** tab, under **Setup to Run on Hardware**, click**System on Chip (SoC)**.
3. In the **System on Chip (SoC)** pop-up window, select **Hardware Board > Xilinx Zynq ZC706 evaluation kit**. Click **Finish**.

   Note The processor model must use the same hardware board and solver configuration parameter settings as the top level model.
4. In the model, using a Function-Call Subsystem block, Asynchronous Task Specification block, Inport block, and Outport blocks, create the following system.



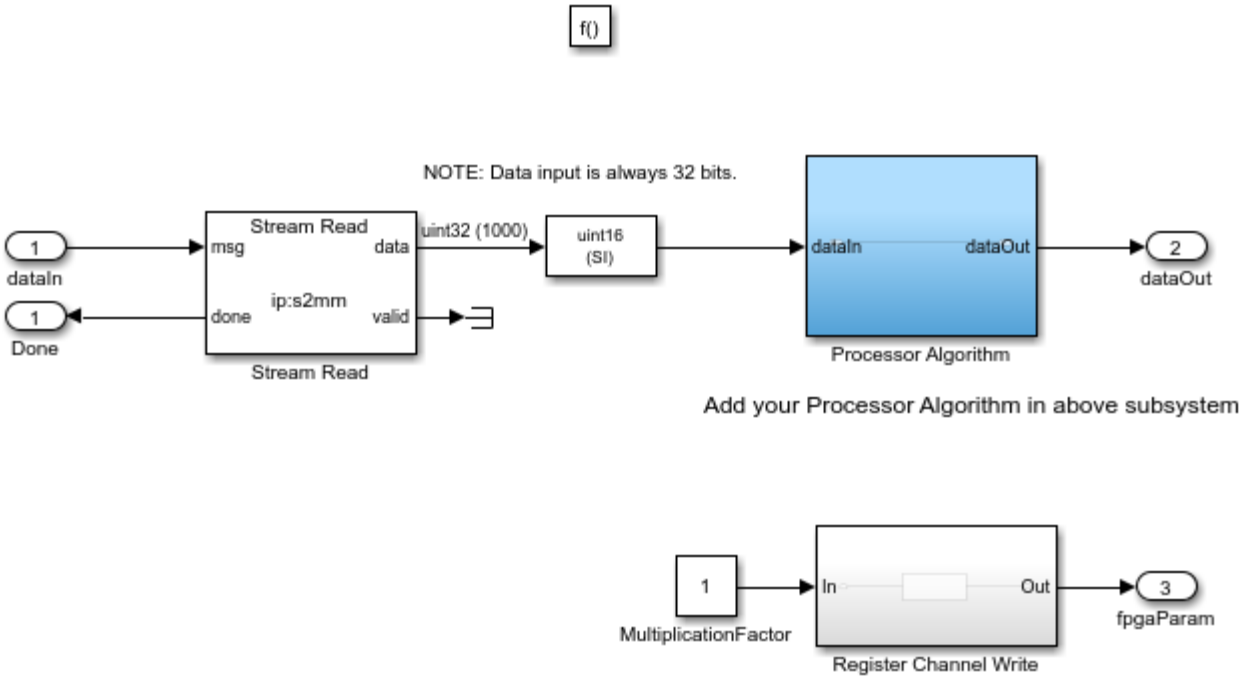
5. In the `dataTask` block dialog mask, check **Signal Attributes > Output function call** to expose a function call port on the outside model.
6. In the Asynchronous Task Specification block dialog mask, set **Task priority** to `50`.

   Note The task priority of the Asynchronous Task Specification block must match the priority of task in the Task Manager block driving this task.

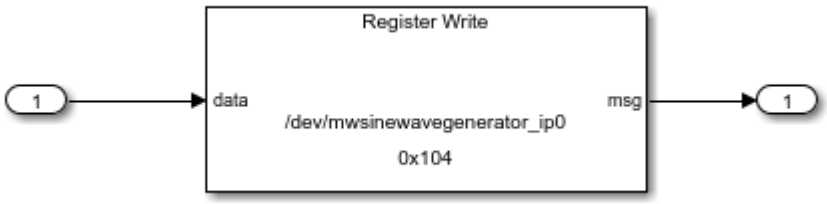
Task Processing

The `Processor Algorithm Wrapper` subsystem reads data from the external memory only after each write to the external memory by the FPGA.

1. Open the `Processor Algorithm Wrapper` block.
2. Using a Stream Read block, Constant block, Data Type Conversion block, and Subsystem blocks, create the following model.



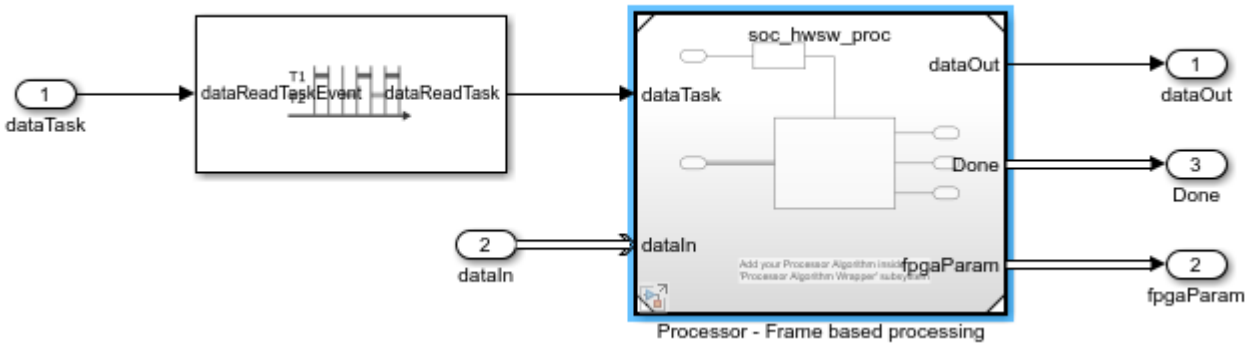
3. Open the Stream Read block dialog mask. Set **Number of buffers** to `6`.
4. Open the Data Type Conversion block dialog mask and set **Output data type** to `uint16`.
5. The `Processor Algorithm` subsystem serves as a base to develop your own processing algorithm.
6. Open the `Register Channel Write` subsystem block.
7. Add a Register Write block to create the following model.



8. Open the Register Write block dialog mask. Set **Device name** to `/dev/mwsinewavegenerator_ip0` and **Offset address** to `hex2dec('100')`.

Top Model

1. In the project folder, open the model `soc_hwsw_top.slx`.
2. Add a Subsystem block into the `Processor` area and label the block `Processor`.
3. In the `Processor` subsystem, using the Task Manager block and Model block, create the following system.



4. Open the Model block dialog mask and set **Model name** to `soc_hwsw_proc.slx`.
5. Open the Task Manager block dialog mask. Set the task **Name** to `dataReadTask` and set the **Priority** to `50`. In the **Simulation** tab, set the **Mean**, **Min**, and **Max** to `8e-05`. Click **OK**.

See Also

Task Manager

More About

- “What is Task Execution?” on page 3-2
- “Event-Driven Tasks” on page 3-8
- “Task Duration” on page 3-13

User Logic on FPGA

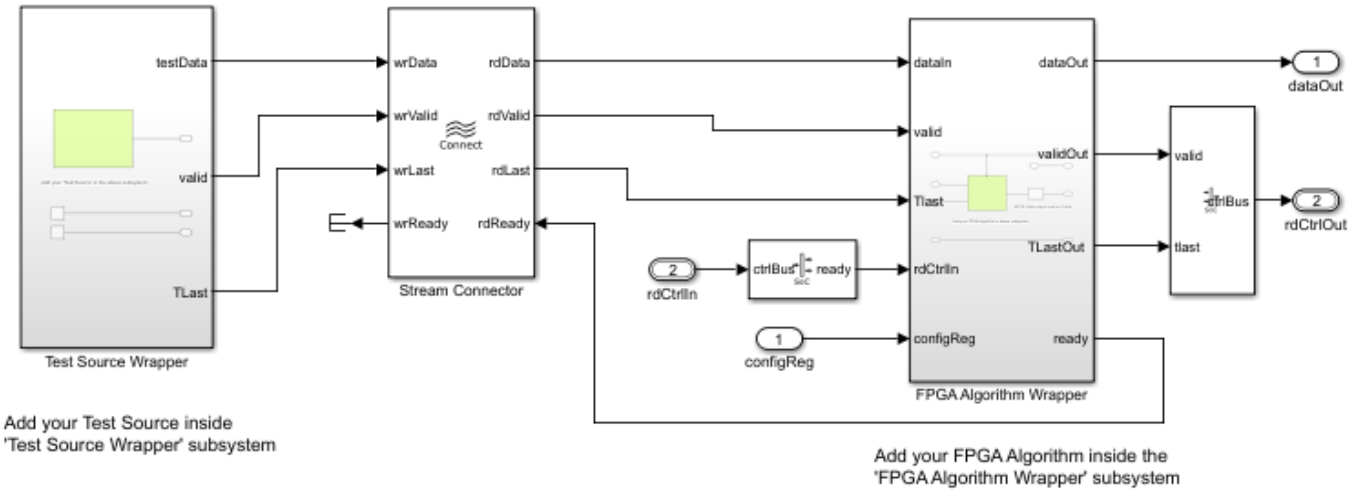
In this SoC project example, the FPGA generates test data and process it in FPGA algorithm before passing it to processor using shared memory.

Sample Based Model

1. Open a new Simulink model. Save the model as `soc_hwsw_fpga_sample.slx` into the subfolder, named `referencedmodels`, in the project folder.
2. On the **Modelling** tab, click **Model Settings**. On the **Configuration parameters** window, in the **Hardware Implementation** panel, set **Hardware board** to `None` and set **Device vendor** to `ASIC/FPGA`. In the **Solver** panel, set **Solver selection** > **Type** to `Fixed-step`. Click **OK** to apply the changes and close the configuration parameters.

   Note SoC Blockset requires that the FPGA reference models specify the intended deployment hardware, in this case an FPGA.

3. In the new model, using Stream Connector block, SoC Bus Selector block, SoC Bus Creator block, and Subsystem blocks, create the following system.

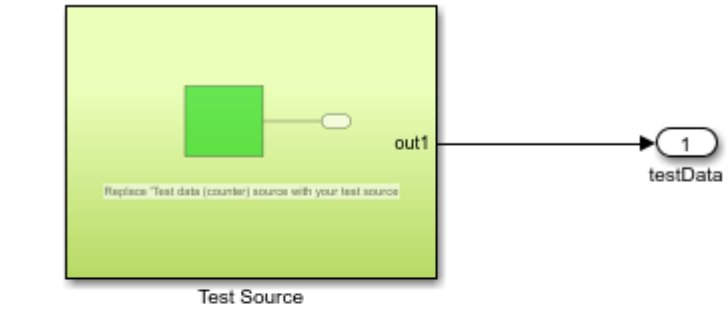


   Note The signals for `rdCtrlIn` and `rdCtrlOut` must use bus signal types set to `StreamS2MBusObj` and `StreamM2SBusObj`, respectively.

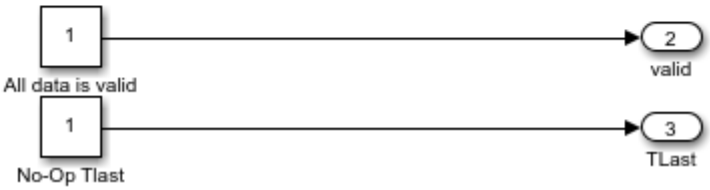
   Tip When your FPGA model includes more than one IP, you must define each IP as a subsystem and connect the subsystems using a Stream Connector or Video Stream Connector block. For additional information, see “Considerations for Multiple IPs in FPGA Model” on page 4-4.

   In the SoC Bus Creator block dialog mask, set **Control type** to `Valid`.

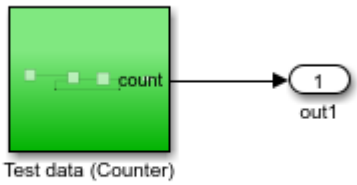
4. The `Test Source` subsystem simulates a free-running counter. Open the `Test Source` subsystem and create the following system.



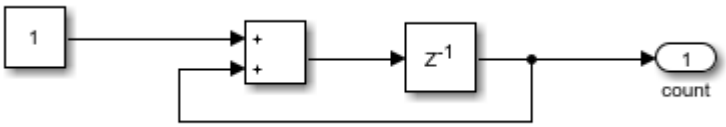
Add your 'Test Source' in the above subsystem



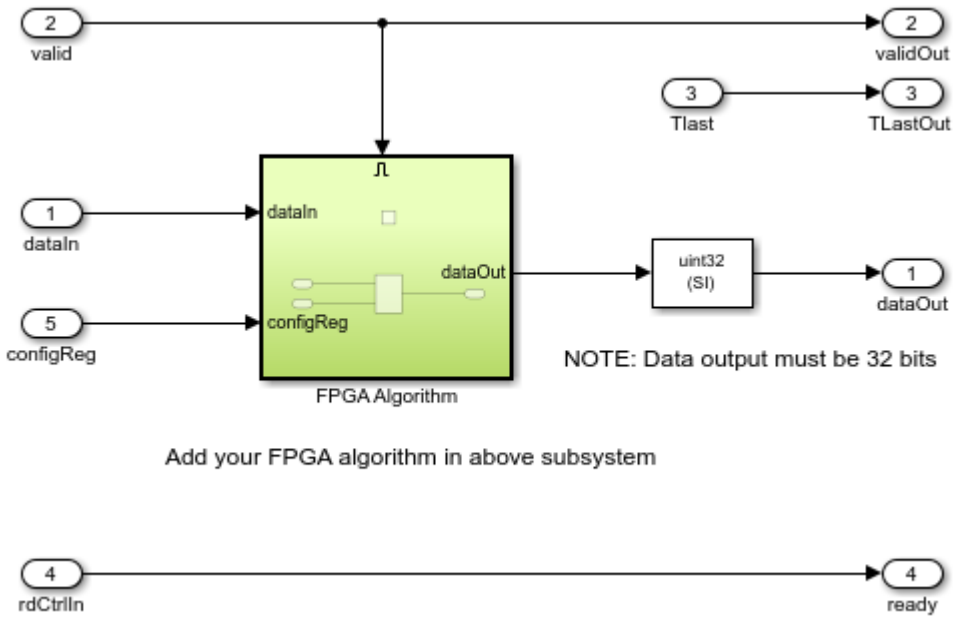
Note The sources, `All data is valid` and `No-Op Tlast`, must produce a signal with `boolean` data type.



Replace 'Test data (counter) source with your test source



5 The `FPGA Algorithm` subsystem simulates the multiplication of streamed data. Open the `FPGA Algorithm` subsystem and using an Enabled Subsystem, Logical Operator, and Data Type Conversion blocks, create the following system.



Top Model

1. In the project folder, open the model `soc_hwsw_top.slx`.
2. Add a Subsystem block into the FPGA area and label the block FPGA.
3. In the FPGA subsystem, using the Model block, create the following system.

4. Open the Model block dialog mask and set **Model name** to `soc_hwsw_fpga_sample.slx`.

The “Stream from FPGA to Processor Template” on page 2-14, the FPGA subsystem uses a model variant to select between the sample based model developed in this section and a frame based model. The frame based model allows faster simulations but does not support code generation.

See Also

Stream Connector | SoC Bus Selector | SoC Bus Creator

More About

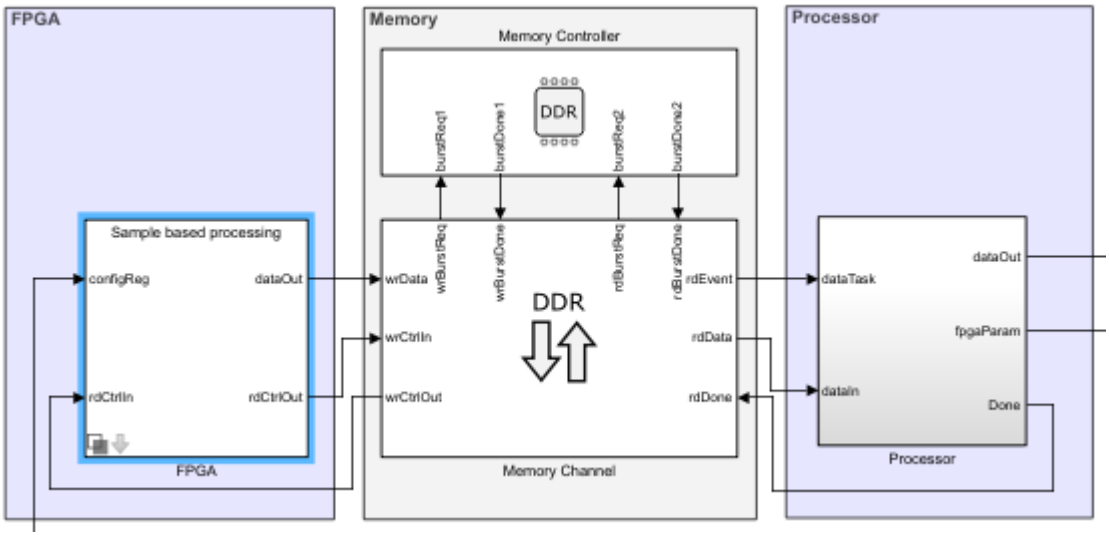
- “AXI4-Stream Interface” on page 5-7

Memory and Register Channel Connections

The memory channel models the data transfer from FPGA to processor using shared external memory. The register channel models the control of FPGA logic from processor. You can both configure the FPGA logic and read the status of FPGA logic from processor. The following sections show how to create these channel connections.

Memory Channel Connection

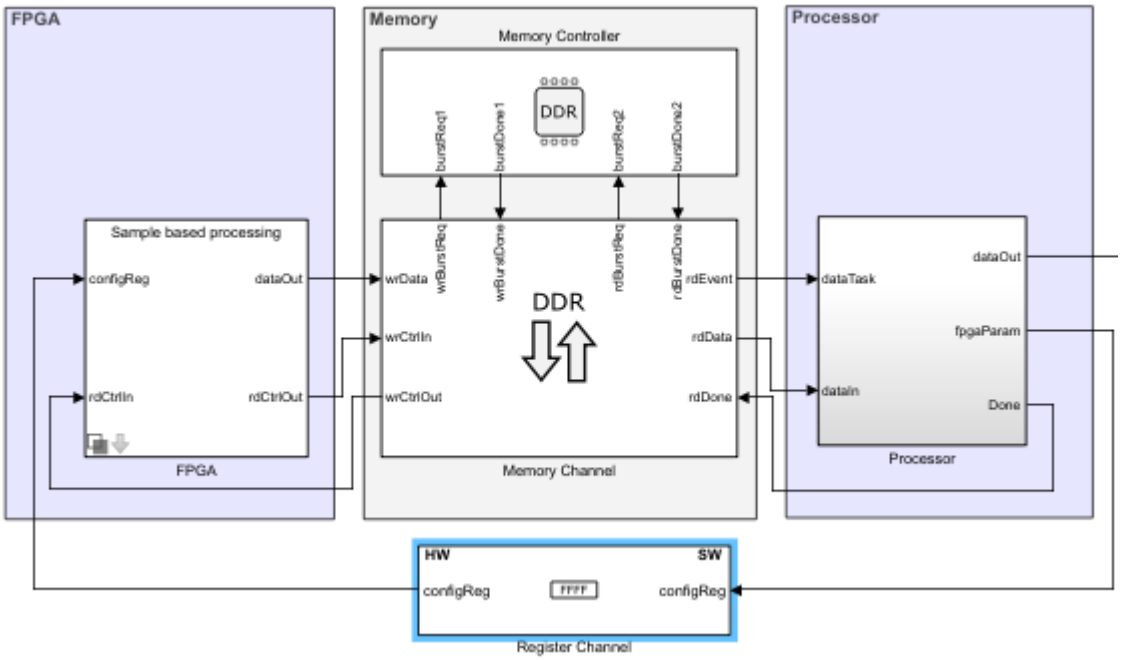
1. Open the `soc_hwsw_top.slx` model.
2. Add a Memory Channel block and a Memory Controller block to the `Memory` area. Together, these blocks model the memory connection through `DDR` between the processor and FPGA sides of your application.



3. Open the Memory Controller block dialog mask. Set **Number of masters** to 2. In the **Advanced** tab, the Memory Controller automatically inherits parameters from the **Hardware board** specified in the model configurations.
4. Connect the pair of Memory Controller burst ports, `burstReq` and `burstDone`, to the read and write burst request ports of the Memory Channel block.
5. In the model, open the Memory Channel block dialog mask. Set **Channel type** to `AXI4-Stream to Software via DMA`. Set **Buffersize (bytes)** to `FrameSize*4` and **Number of buffers** to 6. Click **OK**.

Register Channel Connection

1. Add a Register Channel block to the model and connect the block to the `Processor` and `FPGA` subsystems as shown in the following image.



2 Open the Register Channel block dialog mask. Add a new register with these properties.

Register	Direction	Data type	Dimension
configReg	Write	uint8	1

Set **Register write sample time** to `FPGASSTime`. Click **OK**. This sample time is set in the file `soc_hwsw_init.m`.

See Also

Memory Controller | Memory Channel | Register Channel

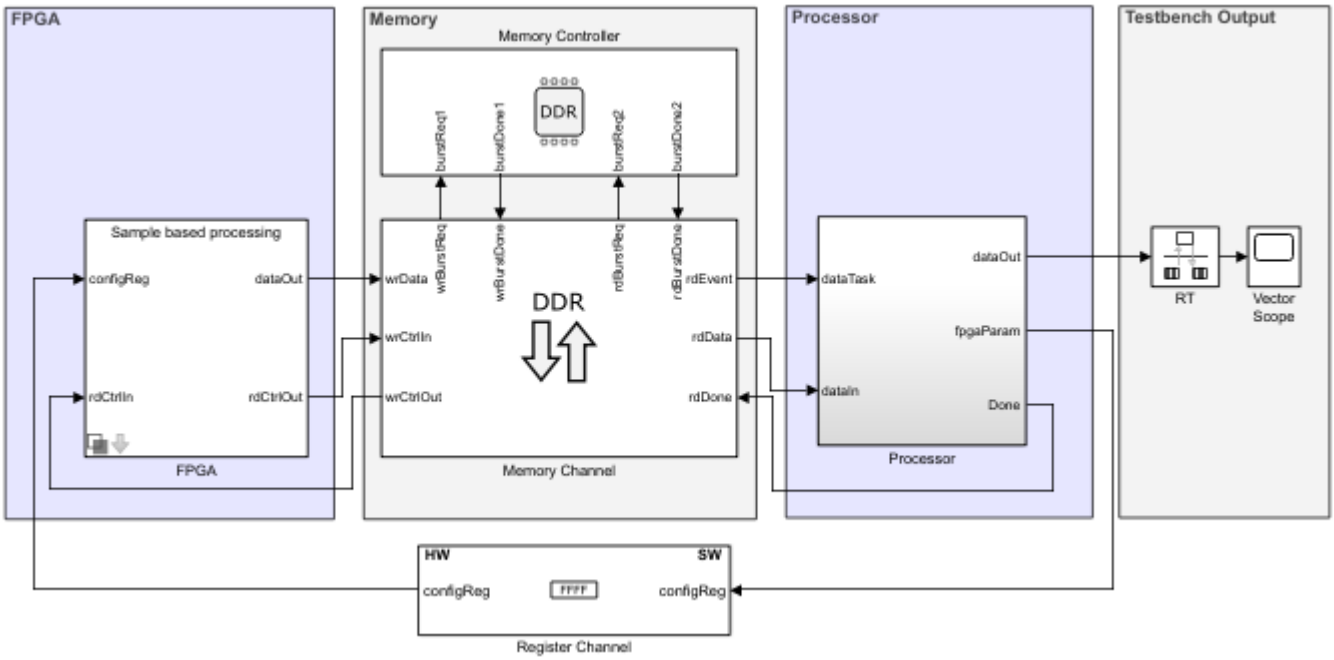
More About

- “Memory and Register Channel Connections” on page 2-41

Simulation and Analysis

This set of steps runs the `soc_hwsw_top.slx` model created in the previous steps. A visual of the processor output data shows the complete SoC application.

1. In the project folder, open the model `soc_hwsw_top.slx`.
2. Using a Scope block and Rate Transition block, update the model as shown in this diagram.



3. Run the model and open the `Vector Scope`.

The display in the `Vector Scope` shows the counter output.

See Also

“Use Template to Create SoC Model” on page 2-4 | “Stream from FPGA to Processor Template” on page 2-14

SoC Generation Workflows

You can deploy an SoC model on an SoC device by using one of these workflows.

- Use the **SoC Builder** tool to guide you through the steps required to build hardware and software executables, load them on an SoC device, and execute.
- Use the `socExportReferenceDesign` function to export a reference design from an SoC model, and then integrate your IP code to the reference design and deploy to an SoC device using the **HDL Workflow Advisor** tool.
- Use the **SoC Model Creator** tool to create an SoC model based on the selected reference design for the supported Xilinx RFSoC devices. Use the created model as a template to design and simulate an FPGA algorithm and processor algorithm. Then, use the **SoC Builder** tool to build and deploy the system on an RFSoC device.

All these workflows require the SoC Blockset and HDL Coder™ products.

Use SoC Builder Tool to Deploy SoC Model on SoC Device

If you are authoring an SoC model from scratch using SoC Blockset features, first simulate and refine the model as needed. Then, use the **SoC Builder** tool to guide you through the workflow of checking, building, loading, and executing your design on an SoC device. For an example of using the **SoC Builder** tool, see “Streaming Data from Hardware to Software” on page 7-31.

Use socExportReferenceDesign Function to Deploy SoC Model on SoC Device

If you are authoring an IP core using the HDL Coder custom IP core generation workflow, you can create a custom reference design and integrate the IP core into that design. Use the `socExportReferenceDesign` function to export a reference design from an SoC Blockset model. For an example of using the `socExportReferenceDesign` function, see “Export Custom Reference Design” on page 7-112.

Use SoC Model Creator and SoC Builder Tools to Create and Deploy SoC Model on RFSoC Device

This workflow enables algorithm and system designers to generate an HDL IP core and integrate it into a fixed reference design for rapid prototyping. Select a fixed reference design and configure it to create an SoC model using the **SoC Model Creator** tool. Edit the created model to include an FPGA algorithm and processor algorithm. Then, either simulate the system or use the **SoC Builder** tool to generate a bitstream and host I/O model, build a software application, and program the Xilinx RFSoC device. For more information on this workflow, see “RFSoC Support for Fixed Reference Design” (SoC Blockset Support Package for Xilinx Devices).

If you are designing an RFSoC application on an RFSoC device and building your application using a fixed reference design, launch the **SoC Model Creator** tool first. If you are building an application based on the content in the Simulink model, you can use the “RFSoC Template” on page 2-25 to get started quickly, or you can create the whole model from scratch.

See Also

SoC Builder | `socExportReferenceDesign`

More About

- “Generate SoC Design” on page 2-46
- “Custom Reference Design” (HDL Coder)
- “Custom IP Core Generation” (HDL Coder)

Generate SoC Design

In this section...

This tutorial outlines the steps to build hardware and software executables for your model and execute your application. Your SoC model can contain a processor model, an FPGA model, or both.

SoC Builder requires that you have a support package installed, based on the board selected in the configuration parameters. For more information, see “SoC Blockset Supported Hardware”.

Step 1: Set Up FPGA Design Software Tools

To generate SoC binaries, you must include the path to Vivado or Quartus executables in your system path. If the executables are not already in your system path, use `hdlsetuptoolpath` function to add them to your path.

Xilinx Software

Use the `hdlsetuptoolpath` function to set up your system environment for accessing Xilinx tools from MATLAB. This function adds the specified installation folder to the MATLAB search path. The following example assumes that Xilinx Vivado is installed at `C:\Xilinx\Vivado\2018.3\bin`.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado', ...
'ToolPath','C:\Xilinx\Vivado\2018.3\bin\vivado.bat')
```

Intel Software

Use the `hdlsetuptoolpath` function to set up your system environment for accessing Intel tools from MATLAB. This function adds the specified installation folder to the MATLAB search path. The following example assumes that Intel FPGA design software is installed at `C:\Intel\18.1\quartus\bin64`.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...
'ToolPath','C:\Intel\18.1\quartus\bin64')
```

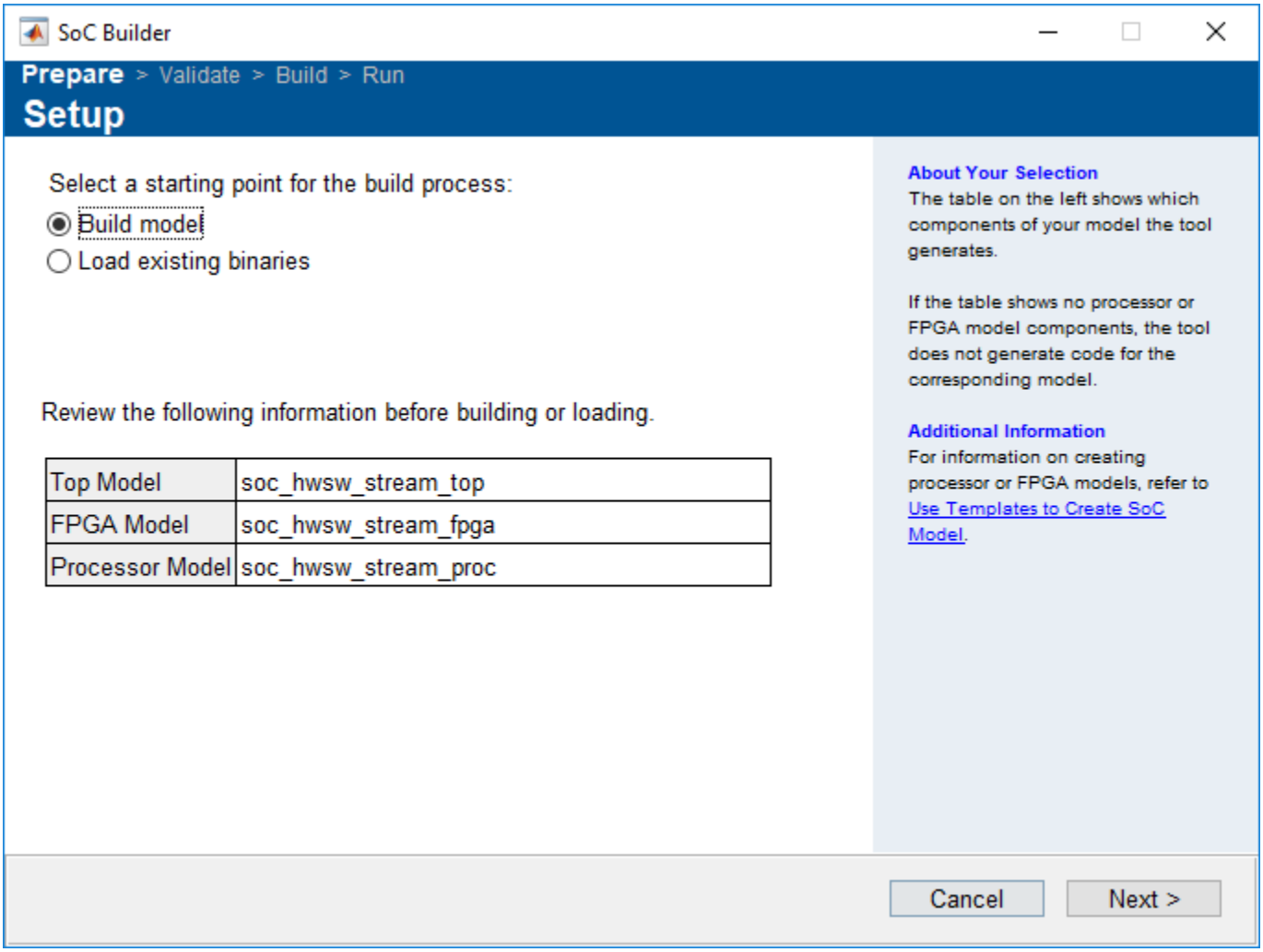
Step 2: Start SoC Builder

In the Simulink toolstrip, on the **System on Chip** tab click **Configure, Build & Deploy**.

Step 3: Prepare Model for Generation

Prepare your model by selecting a starting point for the build process, and then review the model information.

Note If no support package is detected, **SoC Builder** first prompts you to install the required support package.



Specify the starting point for the build process. If you are building a model that you have not built before, select **Build model**. If you previously completed the build process and saved the binaries in a folder, select **Load existing binaries**.

SoC Builder parses the model and displays the top model, the FPGA model (if one exists), and the ARM model (if one exists). Review this information for accuracy. If it seems incorrect, revise the model, save, and restart the **SoC Builder** tool.

Note If your FPGA model is set to a frame-based Simulink model variant, then the **SoC Builder** does not display the model in the table. To make it visible in the table, set the model variant to sample-based and recompile your design.

Click **Next**.

The next page of the **SoC Builder** provides information about the memory map of the model. To open the **Memory Mapper**, click **View/Edit**. Review the base addresses and offsets, and edit them if needed.

Note This memory map step of the **SoC Builder** is visible only if you have an FPGA model in your top model. If your FPGA model is set to frame-based modeling - then no FPGA model is visible, and therefore there is no access to the **Memory Mapper** tool.

Click **Next**.

Step 4: Select Project Folder

Specify a path to a project folder by entering the path in the **Project Folder** text box or by browsing to a folder location. The SoC Builder places all generated files, including reports, executables, and the bitstream, in this specified folder.

If you selected **Load existing binaries** as the starting point for the build process, specify the project folder location of the previous binaries and reports.

Click **Next**.

Step 5: Select Build Action

In the **Select Build Action** section, select one of these options:

- **Build, load and run** – Select this option to generate HDL and C code, build software executables and an FPGA programming file from your model. After building, **SoC Builder** loads the generated code to the FPGA board and executes the application.
- **Build only** – Select this option to generate HDL and C code, build software executables and an FPGA programming file from your model. **SoC Builder** saves the generated binaries in a folder, and you can continue execution later.
- **Build and load for external mode** – Select this option to build the design and run it in external mode. External mode enables you to tune parameters on the FPGA without having to rebuild the FPGA design. It also enables logging data from the FPGA and displaying it on the host. For more information about external mode, see “External Mode Simulations for Parameter Tuning and Signal Monitoring” (Simulink Coder).

Step 6: Validate Model

Check the model against the selected board and generate a report. Check the report to ensure that the design is generated as expected.

SoC Builder names the report `<project-folder>/html/`*`modelname`*`_system_report.html` and saves it in the project folder. The report contains an overview section with information about the

model, project folder, and generated files. The report also lists user IP cores and vendor-provided IP cores, with the address map of registers and memory blocks.

Step 7: Build Model

To generate a bitstream for your FPGA design and a compiled executable for your software, click **Build**.

Clicking **Build** opens an external shell and runs third-party tools for synthesis and implementation of the design. The generation time depends on the complexity of your model and your host computer. Once the generation is complete, the bitstream is generated with your model name. **SoC Builder** generates a JTAG testbench script if you selected the **Include MATLAB as AXI Master** option in the configuration parameters. The script shows how to set up MATLAB as an AXI Master and configure your FPGA design over JTAG. You can customize the script to create your own test bench. For more information about MATLAB as an AXI Master, see support package documentation: “SoC Blockset Supported Hardware”.

If `tee.exe` is not installed on your Windows machine, you may encounter a build error in the external shell while working with Intel boards. To resolve the error, follow these steps:

1. Download the `tee.exe` file for Windows from this link https://ss64.net/westlake/nt/tee.zip.
2. Unzip and copy the EXE file to the `C:\Windows` folder.
3. Add the folder path to the **System variables** pane as Windows environment variables.
    a. Right-click the **Computer** icon and choose **Properties**, or in Windows **Control Panel**, choose **System**.
    b. Choose **Advanced system settings**.
    c. On the **Advanced** tab, click **Environment Variables**.
    d. In the **System variables** pane, select the Path variable and click **Edit**.
    e. In the **Edit environment variable** pane, click **New** to add a new folder path.
4. Run the **SoC Builder** tool.

Step 8: Connect Hardware

Review the IPv4 address, SSH Port number, and login credentials. Edit any of these values if necessary. This step is critical if you have more than one board connected to the host computer, so that **SoC Builder** can identify the correct port connection. Verify that the displayed IP address matches the IP address for the board you intend to use.

Verify that the board is connected to the host with an Ethernet cable, and then click **Test Connection** to test the physical connection to the board.

Note This step in the **SoC Builder** is visible only if your top model includes a processor model.

Step 9: Load and Run

Note If your top model includes an FPGA model, but no processor model, the button shows as **Load**.

Verify that your board is connected to the host computer.

- If a processor model is present in your top model, connect to the board with an Ethernet cable.
- If the top model includes an FPGA model, but no processor model, connect to the board with a JTAG cable.

Click **Load and Run**. This action loads the generated bitstream to the FPGA, programs the processor, and runs the application.

If you selected **Tune parameters and monitor signals in external mode** in step 5, this action loads the bitstream to the FPGA and opens the model in external mode. You can now choose signals for logging and monitoring or change tunable parameters. In the **System on Chip** tab, in the **Run on Hardware** section, you can click **Monitor and Tune** to run the instrumented application on hardware. Click **Connect** if you previously built and loaded your design to an FPGA. This action connects your instrumented Simulink model to the FPGA model.

See Also

SoC Builder

Custom Hardware Board Configuration

A custom hardware board is a hardware board that not explicitly supported as a default selection in SoC Blockset. To create an SoC project to simulate a custom hardware board, configure a Simulink project as follows:

1. Create or open an existing SoC project. For more information on creating SoC projects, see “Use Template to Create SoC Model” on page 2-4.
2. In the top level model, open the Simulink configuration parameters dialog. In the **Hardware Implementation** panel, set **Hardware board** to `Custom Hardware Board`.
3. In the **Hardware Implementation** panel, open the **Target hardware resources > Processor** group. Set **Number of cores** to match the number of cores available on your SoC processor. The cores available in your processor can be found from the SoC manufacturer's data sheet.
4. Open the **Target hardware resources > FPGA design (mem controllers)** group and set the “FPGA design (mem controllers)” configuration parameters according to your SoC specifications. For information on deriving “FPGA design (mem controllers)” parameters, see the Memory Controller block which shares these parameters.
5. Open the **Target hardware resources > FPGA design (mem channel)** group and set the “FPGA design (mem channels)” configuration parameters according to your SoC specifications. For information on deriving “FPGA design (mem channels)” parameters, see the Memory Channel block which shares these parameters.

Note The `Custom hardware board` selection only supports simulation. For code generation, use one of the provided SoC Blockset hardware board selections.

See Also

“Hardware Implementation Pane”

Build Error for Rapid Accelerator Mode

SoC Blockset does not support “Rapid Accelerator Mode” simulation of models. Attempting to use SoC Blockset blocks and features in model running rapid accelerator mode results in undefined behavior.

In SoC Blockset models, set the simulation mode to normal mode, accelerator mode, or external mode.

See Also

More About

- “Rapid Accelerator Mode”

Build Error When FPGA or Processor Model Not Detected

When you use the **SoC Builder** tool to deploy your design on an SoC device, you must open **SoC Builder** from the top model of your design. The top model must include one referenced model, such as an FPGA model or a processor model. When you open **SoC Builder** from a location other than the top model, you might see this error message:

`FPGA or processor model not detected. You must launch SoC Builder from the top model`.

If you see this error message, follow these steps to run **SoC Builder** from the top model.

1. Close **SoC Builder**.
2. Navigate to the top model of your design.
3. Reopen **SoC Builder**, and follow the instructions to build your design. For more information about generating an SoC design, see “Generate SoC Design” on page 2-46.

For more information about SoC Blockset model structure, see “SoC Blockset Model Structure” on page 2-2.

If you still see the error message when you build your design from the top model, the **SoC Builder** cannot detect an FPGA reference model or a processor reference model in your design. The top model must reference an FPGA model or processor model (or both) that include a Model block.

- If your top model references an FPGA model, follow these steps to confirm correct FPGA model settings. For more information, see “User Logic on FPGA” on page 2-37.

  1. Open the FPGA model. On the **Modelling** tab, click **Model Settings**. In the **Configuration parameters** dialog box, in the **Hardware Implementation** pane, set **Hardware board** to `None` and **Device vendor** to `ASIC/FPGA`.
  2. In the **Solver** pane, set **Solver selection** > **Type** to `Fixed-step`.
  3. Click **OK** to apply the changes and close the dialog box.
- If your top model references a processor model, follow these steps to confirm correct processor model settings. For more information, see “Software and Task Management on Processor” on page 2-34.

  1. Check that the processor model is driven by a Task Manager block.
  2. Check that the **Hardware board** parameter of the processor model matches the **Hardware board** parameter of the top model. To set the **Hardware board** parameter, first open the processor model. On the **Modelling** tab, click **Model Settings**. In the configuration parameters dialog box, in the **Hardware Implementation** pane, set **Hardware board** to match the same parameter value for the top model.
  3. Click **OK** to apply the changes and close the dialog box.

See Also

SoC Builder

More About

- “SoC Blockset Model Structure” on page 2-2

- “Generate SoC Design” on page 2-46

Processor Software

What is Task Execution?

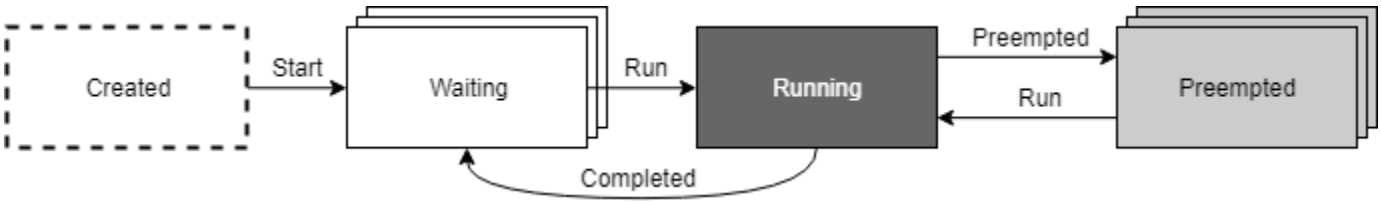
A task is a unit of execution or unit of work in a software application. Typically, task execution in an embedded processor is managed by the operating system (OS). When deployed to the embedded processor, a task corresponds to an OS thread. The SoC Blockset defines the execution life cycle and relation to OS threads as follows.

Task Execution Life Cycle

The life cycle of a task can be divided into five states:

- *Created* – The system creates all the tasks when the application starts and immediately moves them to the waiting state.
- *Waiting* – The task waits for the associated trigger signal, such as an OS timer or I/O device. After receiving the trigger signal, the task starts to run. If the task has the highest priority, it enters the running state. Otherwise, the task continues to wait until it becomes the highest priority, triggered task.
- *Running* – The task executes its code. When the code completes execution, the task immediately moves to the waiting state. If a trigger for a higher-priority task occurs, the running task moves to the preempted state.
- *Preempted* – The task is preempted and waiting to run. A task runs based on a combination of the task priority and the order the task entered the *Preempted* state. Assuming equal task priorities of all other tasks in *Ready to Resume* state, tasks run based on first-in-first-out (FIFO) ordering.
- *Terminated* – Tasks terminate when the application ends.

This figure shows the state diagram of a task execution life cycle for an application using an OS. For simplicity, the terminated state is not shown, but a task can reach the terminated state from any of the other states.



Task and Thread

A task is a conceptual unit of work in an algorithm. In an application executing on a device, a task is a section of code that executes in a thread within an operating system (OS). The OS thread determines the state of execution of the task. Within the SoC Blockset, a task specifically refers to the portion of the Simulink model contained within a rate or function-call subsystem. The trigger signal for that subsystem comes from a Task Manager block. When deployed to hardware, an OS thread uses the task properties. The thread executes the code generated from the subsystem. Conceptually, a *Task* in simulation is equivalent to a *thread* in generated code.

See Also

Task Manager

More About

- “Timer-Driven Task” on page 3-4
- “Event-Driven Tasks” on page 3-8

External Websites

- Task (computing)

Timer-Driven Task

Timer-driven tasks execute at a periodic rate equal to an integer multiple of the Simulink model fundamental sample time.

To create a timer-driven task, connect the task port of a Task Manager block to a periodic event port on a Model block. Each rate in a Model block generates a unique model periodic event port with the time step for the rate shown on the block icon. In the Model block dialog mask, use the **Schedule rates** parameter to enable model periodic event ports.

Note A timer-driven task requires a lower priority than an event-driven task.

Create a Simulink Model with an Timer Driven Task

This example shows how to create and configure a Simulink model to use the timer driven task feature of the SoC Blockset.

Create a Software Reference Model

This section shows how to create a reference model of the software for an SoC application model. The software contains one timer driven task subsystem that reacts to receiving UDP packets.

1. Create a new blank model.
2. In the Simulink editor, add a Subsystem block to the model. Add a Sine block and connect it to the Subsystem block. Connect the output of the Subsystem block to a Terminator block.
3. Open the Function-Call subsystem model.
4. Open the Block parameters dialog box of the Inport block, set the **Sample Time** to `0.1`.
5. In the Simulink editor, open the Configuration Parameters dialog box.
6. Select the **Hardware Implementation** pane, set **Hardware board** to `Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit`.
7. Save the model as `soc_task_createtimerdriventask_software.slx`.

The completed model should look similar to the following model.

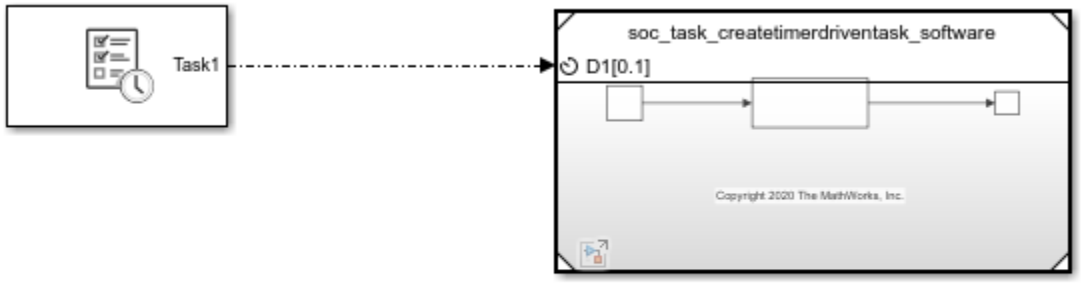
Create the SoC Application Model

This section shows how to create the top level SoC application model that contains the software reference model developed in the previous section.

1. Create a new blank model.

2. In the Simulink editor, add a Model block and open the Block Parameters dialog box.
3. Check **Main > Schedule Rates** and set **Main > Model name** to `soc_task_createtimerdriventask_software.slx`.
4. In the editor, add a Task Manager block to the model.
5. (Optional) Open the Block Parameters dialog box of the Task Manager block. By default, the task **Type** is `Timer-driven` with a **Period** of `0.1`. On the **Simulation** tab, you specify the task duration for that task. For more information on setting task duration, see Task Duration.
6. In the editor, connect the **Task1** port to the **D1[0.1]** port of the Model block.
7. Open the Configuration Parameters dialog box, select the **Hardware Implementation** pane, set **Hardware board** to `Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit`.
8. Update the diagram, press **Ctrl+D**.
9. Save the model as `soc_task_createtimerdriventask_application.slx`.

The completed model should look similar to the following model.



Run the Model with Timer Driven Task

In the Simulink editor, run the `soc_task_createtimerdriventask_application.slx` model. When the run completes, open the Simulation Data Inspector and select **Task1**. The Simulation Data Inspector shows that **Task1** triggers each `0.1` time steps.

See Also

Task Manager

More About

- “What is Task Execution?” on page 3-2

- “Event-Driven Tasks” on page 3-8

Event-Driven Tasks

Event-driven tasks start executing when triggered by an external event. Events can include internal events, such as memory stream or register writes, or external events, such as receiving a UDP data packet from a network connection. Assuming no other tasks are executing at the time of the event or the task has the highest priority, the event-driven task can respond immediately to the event. The task can then process the received data, and potentially generate other events in the model.

Note The triggered-subsystem of an event driven task can contain one and only one block that can generate new events.

Create a Simulink Model with an Event Driven Task

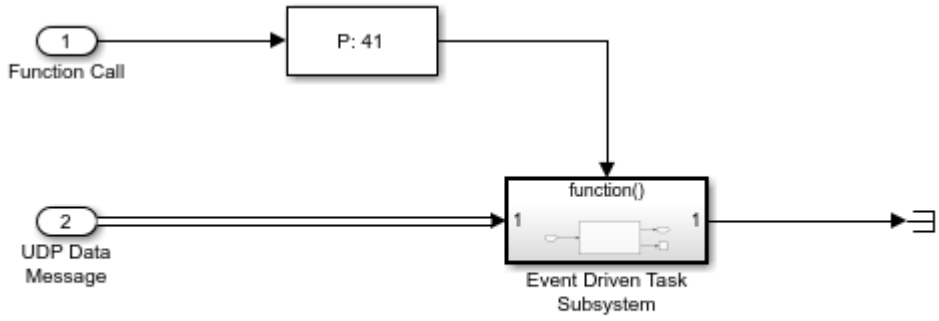
This example shows how to create and configure a Simulink® model to use the event driven task feature of the SoC Blockset.

Create a Software Reference Model

This section shows how to create a reference model of the software for an SoC application model. The software contains an event driven task subsystem that reacts to receiving UDP packets.

1. Create a new blank model.
2. In the Simulink editor, add a Function-Call Subsystem block to the model. Connect an Inport block to the input port of the Function-Call Subsystem block. Connect the output port to a Terminator block.
3. Add an Asynchronous Task Specification block to the model. On the Block Parameters dialog box, set the **Task priority** to `41`.
4. Connect the output port of Asynchronous Task Specification block to the function() input of the Function-Call Subsystem block.
5. Add an Inport block and open the Block parameters dialog box. On the **Signal Attributes** tab, check **Output function call**. Connect the Inport block to the input port of the Asynchronous Task Specification block.
6. Open the Function-Call subsystem model.
7. Add a UDP Read block to model. Open the Block Parameters dialog box, set **Maximum data length (elements)** to `1024` and check **Enable event-based execution**.
8. Connect the Inport block to the UDP Read block **UDP Data** port. Connect the **Data** port to the Outport block. Connect the **Length** port to a Terminator block.
9. Open the Configuration Parameters dialog box, select the **Solver** pane. Set **Solver selection > Type** to `Fixed-step` and check **Tasking and sample timer options > Higher priority value indicates higher task priority**.
10. Select the **Hardware Implementation** pane, set **Hardware board** to `Zedboard`.
11. Save the model as `soc_task_createeventdriventask_software.slx`.

The completed model should look similar to the following model.

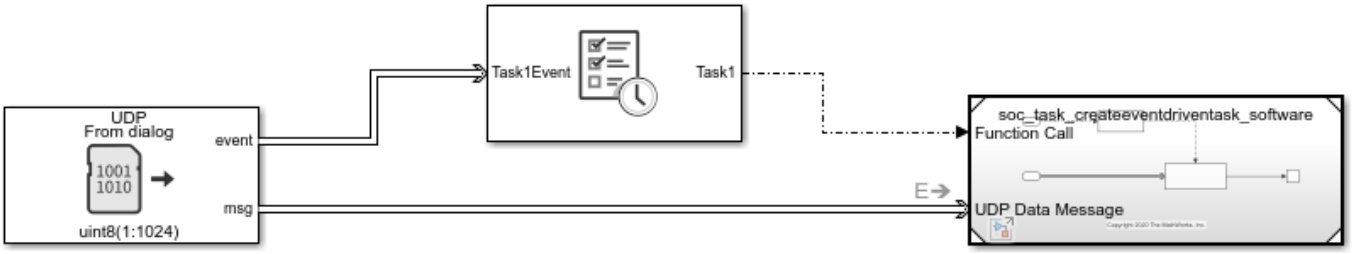


Create the SoC Application Model

This section shows how to create the top level SoC application model that contains the software reference model developed in the previous section.

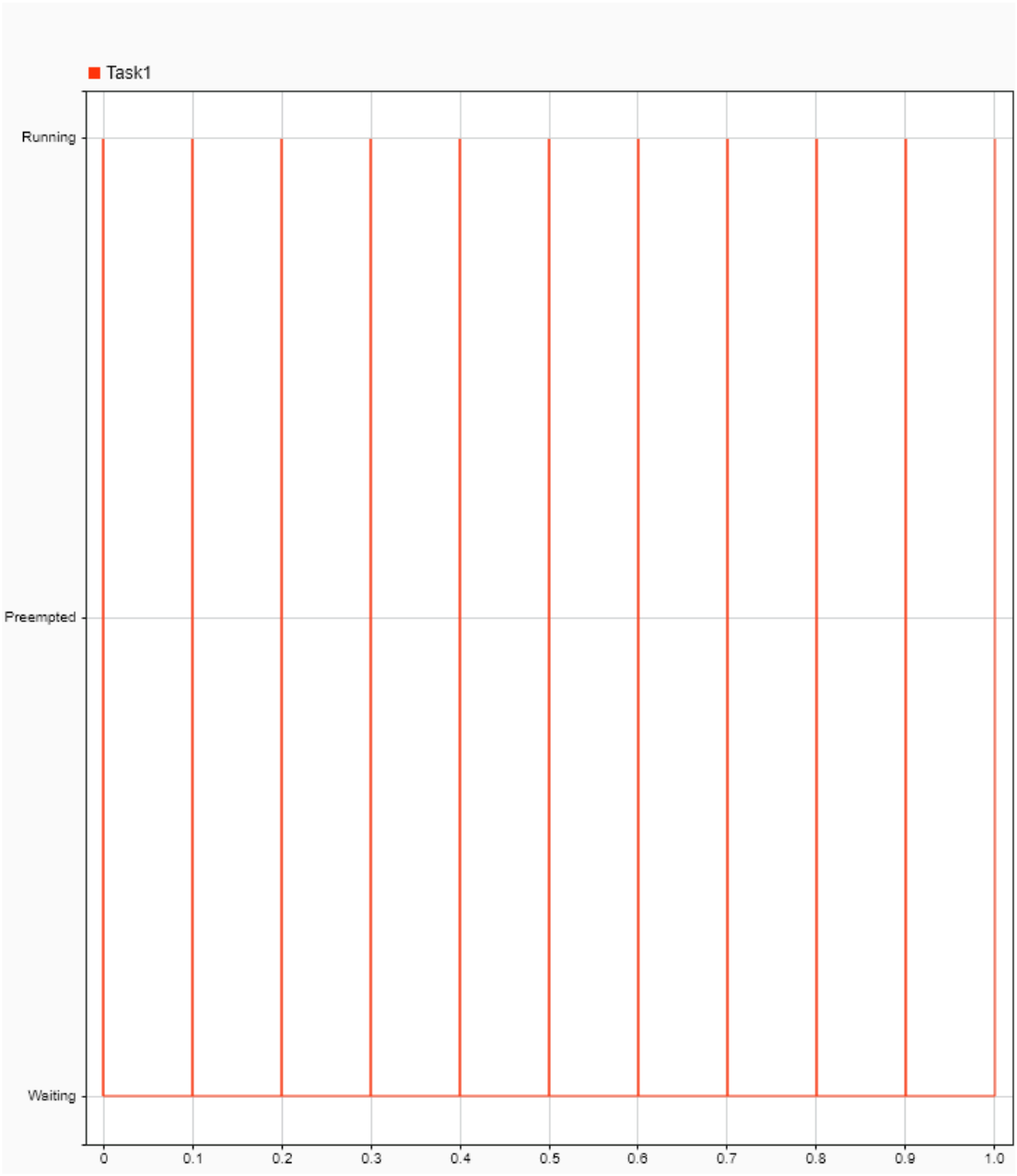
1. Create a new blank model.
2. In the Simulink editor, add a Model block. On the Block Parameters dialog box, set **Model name** to `soc_task_createeventdriventask_software.slx`.
3. Add a Task Manager block and open the Block Parameters dialog box. Set the **Main > Type** to `Event-driven` and **Main > Priority** to `41`. Each newly added event-driven task exposes an event message input port on the Task Manager block.
4. (Optional) On the **Simulation** tab, you specify the task duration for that task. For more information on setting task duration, see “Task Duration” on page 3-13.
5. In the editor, add an IO Data Source block to the model. Open the Block Parameters dialog box and enable **Show event port**.
6. Connect the IO Data Source block **Event** port to the Task Manager and the **UDP Data** port to the UDP Data Message port on the Model reference block.
7. Open the Configuration Parameters dialog box, select the **Solver** pane. Set **Solver selection > Type** to `Fixed-step` and check **Tasking and sample timer options > Higher priority value indicates higher task priority**.
8. Select the **Hardware Implementation** pane, set **Hardware board** to `Zedboard`.
9. Update the diagram, press **Ctrl+D**.
10. Save the model as `soc_task_createeventdriventask_application.slx`.

The completed model should look similar to the following model.



Run the Model with Event Driven Task

In the Simulink editor, run the `soc_task_createeventdriventask_application.slx` model. When the run completes, open the Simulation Data Inspector and select **Task1**. The Simulation Data Inspector shows that **Task1** triggers and executes each time a new UDP packet arrives. Although superficially the task execution appears periodic, this is only a byproduct of the current default settings of the IO Data Source block that generates the event with a time step of `0.1`.



See Also

Task Manager | IO Data Source

More About

- “What is Task Execution?” on page 3-2

- “Timer-Driven Task” on page 3-4

Task Duration

The total time an instance of a task spends in the running state defines the task duration. Task duration can vary due to multiple sources, in particular:

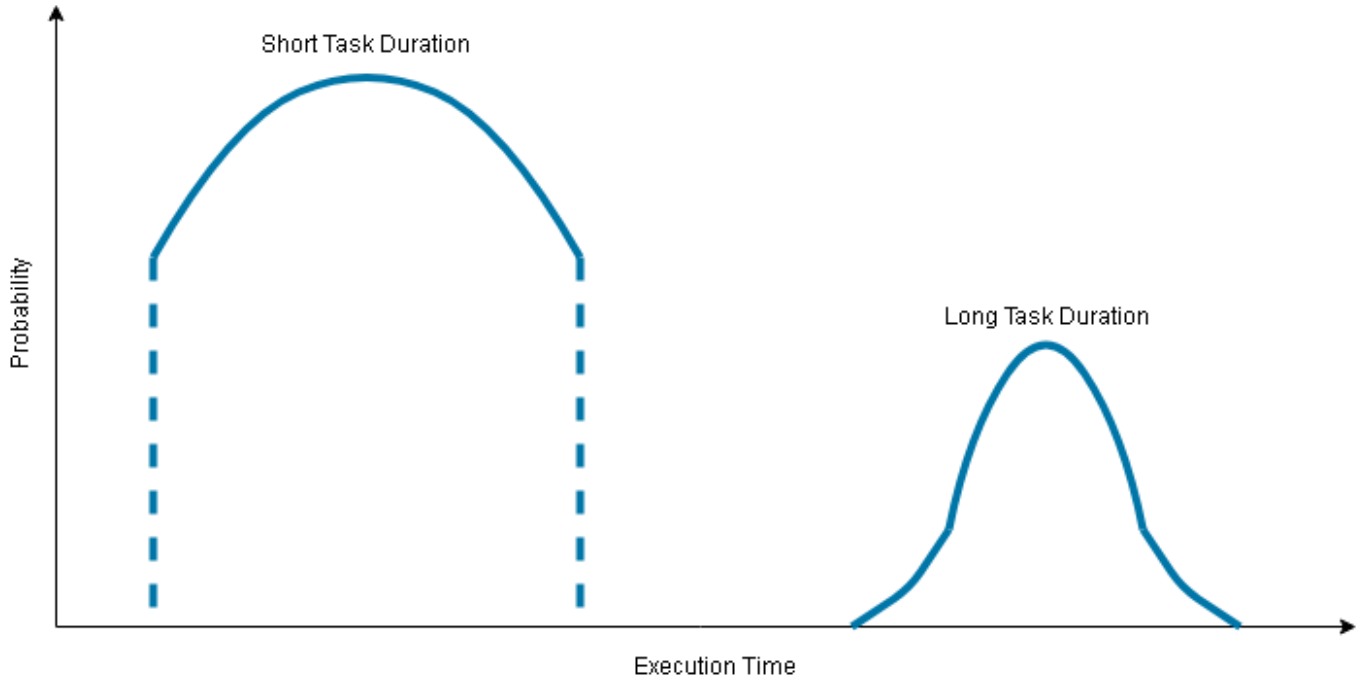
- Conditional branching in the task algorithm
- Dependence on signal values from other tasks
- Dependence on signals from external sources, such as I/O devices or hardware user logic
- Compiler settings and SoC device processor architecture

As a result, task duration for any given task instance can be nondeterministic.

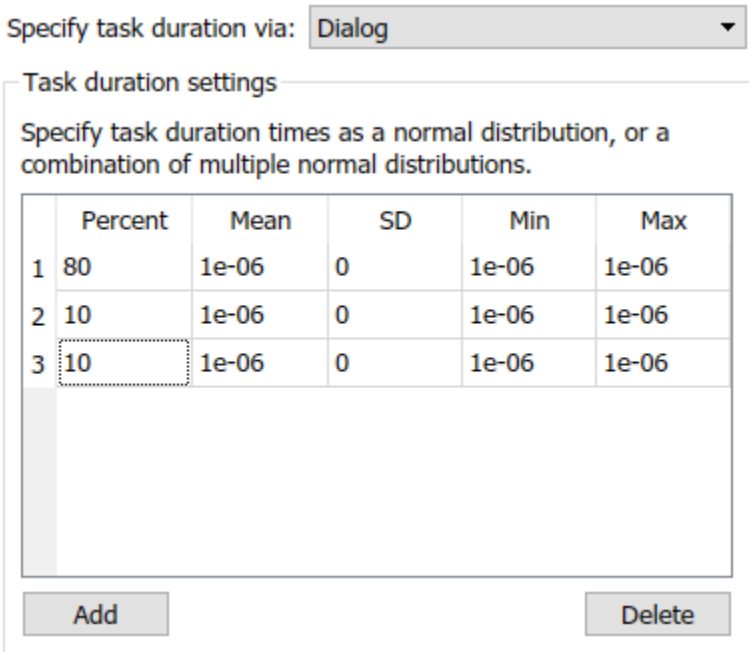
The Task Manager block provides four ways to simulate the nondeterministic task duration: approximation using a parameterized probability distribution, approximation using a calculated probability distribution, and replay of recorded task execution timing data.

Approximation Using Parameterized Probability Distribution

In simulation, the Task Manager block can define the task duration as random variable expressed as the weighted sum of truncated normal distributions. For example, this diagram shows the probability distribution of a task that executes with a short task duration, but can occasionally execute with a longer durations.



To create a probability distribution for a task duration, first open the Task Manager block dialog. Then, on the **Simulation** tab, set **Specify task duration via:** to `Dialog`. In the **Task duration settings** section, you can set the properties of each distribution by editing the text of that property. You can also add and delete probability distributions from the sum of distributions by clicking the **Add** and **Delete** buttons, respectively.



Note

- The sum of the `Percent` weights must equal 100.
- Each task can use a maximum of 5 distributions.

Approximation Using Calculated Probability Distribution

Each recording of task execution data, either from a previous simulation or from execution on an SoC device, generates several profiling files. The `metadata.csv` file contains the calculated mean and standard deviation for each task in that recording. To configure a task in the Task Manager block to use the derived statistical data for task duration, follow these steps:

1. Open the Task Manager block dialog mask.
2. On the **Simulation** tab, set **Specify task duration via** to `Recorded task diagnostics file`.
3. Specify the location and name of the `metadata.csv` file. The **Mean** and **Deviation** parameters are automatically updated with the data from the file.
4. Click **OK**.

Specification from Task Manager Input Port

An input port on the Task Manager block dynamically specifies the task duration. To expose this task duration input port, follow these steps:

1. Open the Task Manager block dialog mask.
2. On the **Simulation** tab, set **Specify task duration via** to `Input port`.

3 Click **OK** to expose a new input port, named **TaskNameDur**, on the block.

Replay of Recorded Task Execution Timing Data

A data file provides exact task duration for each task execution instance. A task execution data file can come from a previous or independent model simulation or directly from the task execution on a processor in an SoC device. For more information on replaying recorded task execution timing data, see “Task Execution Playback Using Recorded Data” on page 3-40.

See Also

Task Manager

More About

- “What is Task Execution?” on page 3-2
- “Task Execution Playback Using Recorded Data” on page 3-40

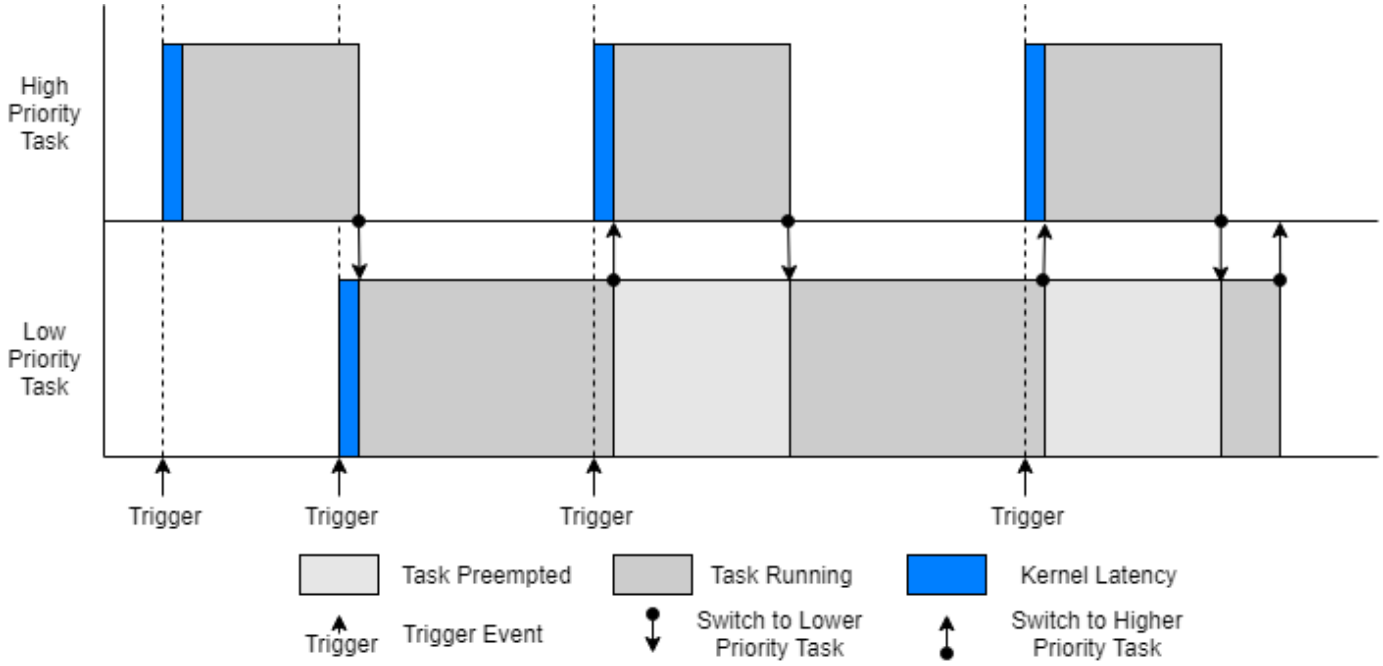
External Websites

- Truncated Normal Distribution

Kernel Latency

In a deployed application, switching between threads requires a finite amount of time depending on the current state of the thread, embedded processor, and OS. *Kernel latency* defines the time required for the operating system to respond to a trigger signal, stop execution of any running threads, and start the execution of the thread responsible for the trigger signal.

SoC Blockset models simulate *Kernel latency* as a delay at the start of execution of a task the first time the task moves from the waiting to running state. The following diagram shows the execution timing of a high-priority and low-priority task on a system that simulates a single processor core.



Other factors affecting kernel latency, such as context switch times, can be considered negligible compared to other effects and are not modeled in simulation.

Note Kernel latency requires advanced knowledge of the processor specifications and can be generally set to `0` without impact to the simulation.

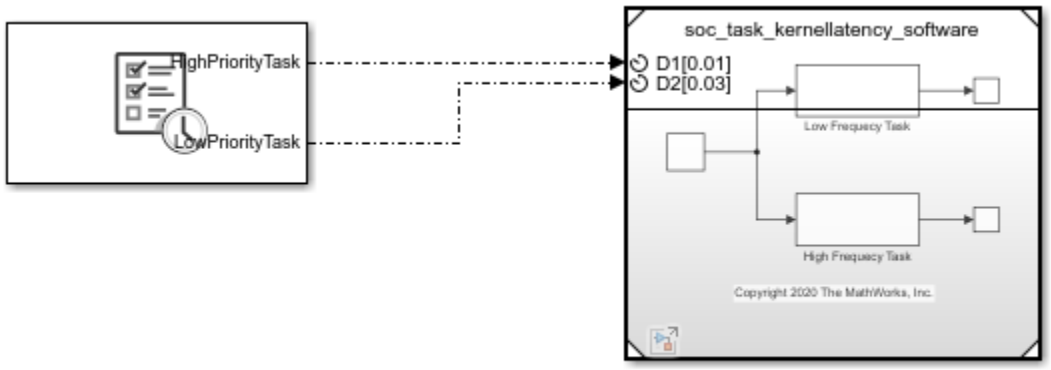
Effect Kernel Latency on Task Execution

This example shows the effect of kernel latency on the behavior and timing of two timer driven tasks in an SoC application.

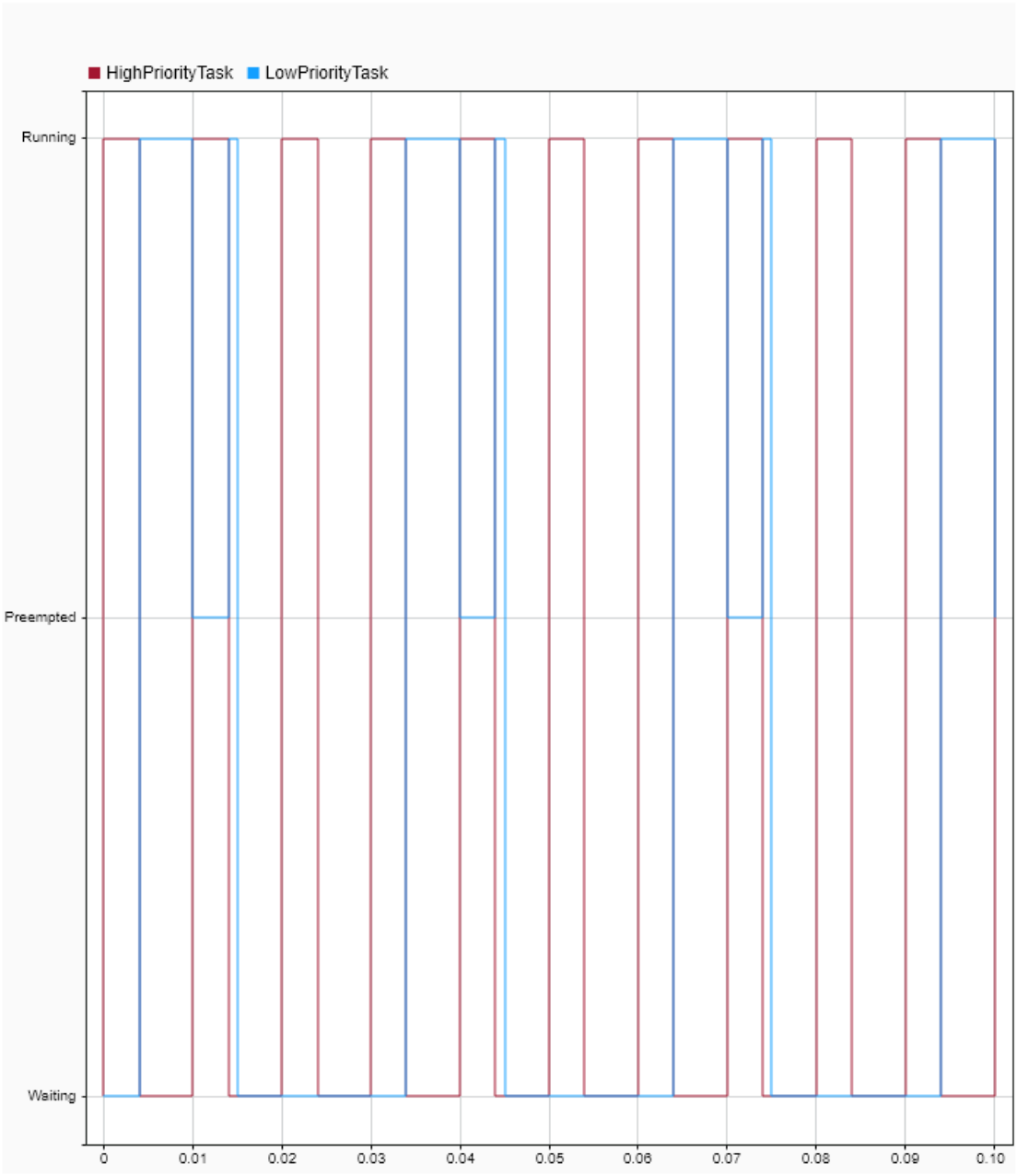
The following model simulates a software application with two timer driven tasks. The task characteristics, specified in the Task Manager block, are as follows:

Name	Period	Mean Duration
HighPriorityTask	0.01	0.004
LowPriorityTask	0.03	0.007

With these timing conditions, the high priority task preempts the low priority task. In the model Configuration Parameters dialog box, the **Hardware Implementation > Operating system/ scheduler > Kernel latency** is set to `0.002`.



Run the model and open the Simulation Data Inspector. Selecting the two task signal produces the following display.



Inspecting the Simulation Data Inspector, a change in task state from *Waiting* to *Running* shows a latency of `0.002` seconds. However, when the task changes from *Preempted* to *Running*, no latency

occurs. This timing matches with the expected behavior of task, experiencing a latency in startup of that task execution instance, but not when the task instance already exists.

See Also

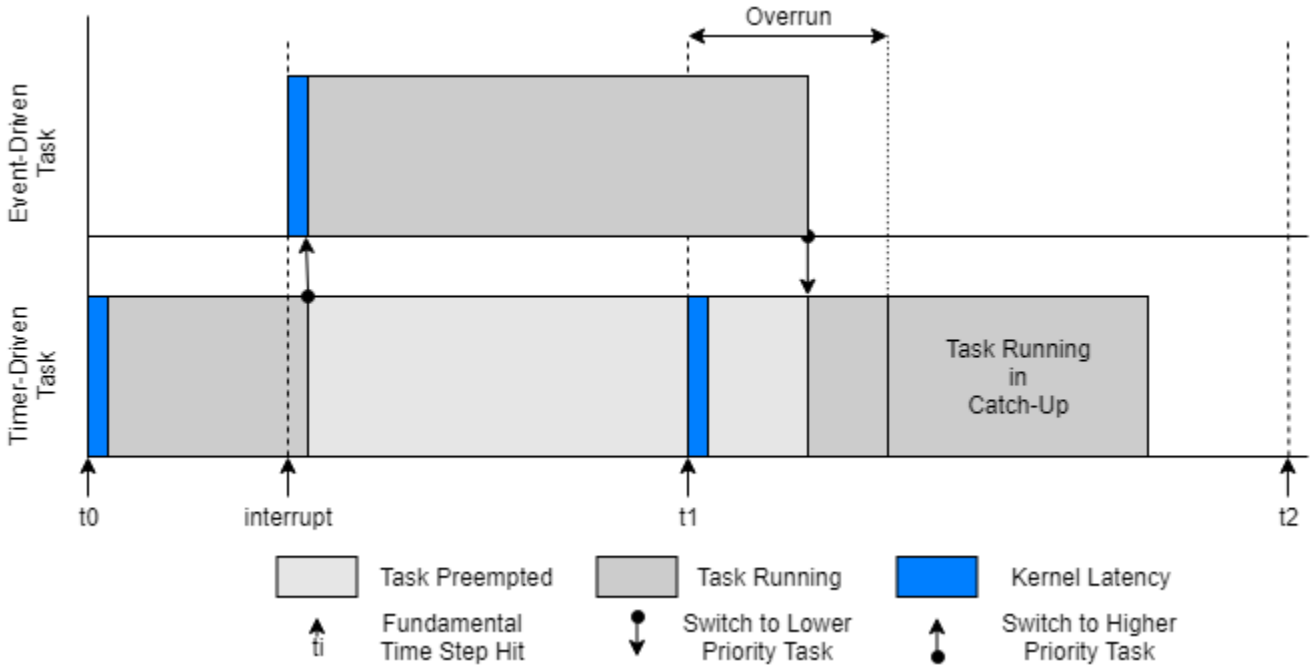
Task Manager

More About

- “What is Task Execution?” on page 3-2
- “Task Duration” on page 3-13

Task Overruns and Countermeasures

With finite processing resources available in a system, an execution instance of a task might not be able to complete before the start of the next task instance. This task overrun results in the start of the next instance of the task execution to be delayed. As a result, the next task must catch-up to avoid another overrun. This diagram shows a simplified execution of two tasks: a high-priority event-driven task and a low priority timer-driven task.



Due to the long execution time of the event-driven task, the first execution instance of the timer-driven task overruns into the start of the next execution instance. This overrun puts the second execution instance into catch-up mode.

When tasks overrun repeatedly, an execution backlog can develop in the application, potentially breaking the system. These sections discuss typical countermeasures to either reduce the chance of task overruns or handle situations when tasks overrun, preventing an execution backlog.

Increase of Task Execution Interval

For timer-driven tasks, reduce the chance of overruns by providing the task with more execution time. Increase available execution time by decreasing the task rate, which is equivalent to increasing the time between task execution instances. This extra time provides each task execution instance a better chance of running to completion, even in the presence of other tasks. The rate of a timer-driven task can be adjusted in the Task Manager block by setting the **Period** parameter.

Reduction of the task execution interval cannot be guaranteed in all cases. Some of these cases include:

- For event-driven tasks, multiple events can occur at the same time, depending on the priority of the event-driven task. This case forces other tasks to overrun due to lack of resources.

- Real-time requirements where a task, timer or event driven, must respond to the latest event trigger signal and new data regardless of whether previous task instances completed. This case fixes the task execution interval to a value determined by the design requirements.

In these cases, distributing tasks across multiple processor cores or allowing tasks to drop can be advantageous depending on the design requirements.

Distribution of Tasks Across Multiple Processor Cores

Most modern embedded processors provide multiple cores where tasks can be executed. By distributing tasks across these multiple processor cores, tasks can run simultaneously without directly competing for processing resources and reducing the chance of task overruns. In SoC Blockset, a task can be set to run on a specific processor core in the Task Manager block by setting the **Core** parameter to the core number. For more information on the selection, execution, and visualization of tasks on multiple cores, see “Multicore Execution and Core Visualization” on page 3-53.

Dropping Overrunning Tasks

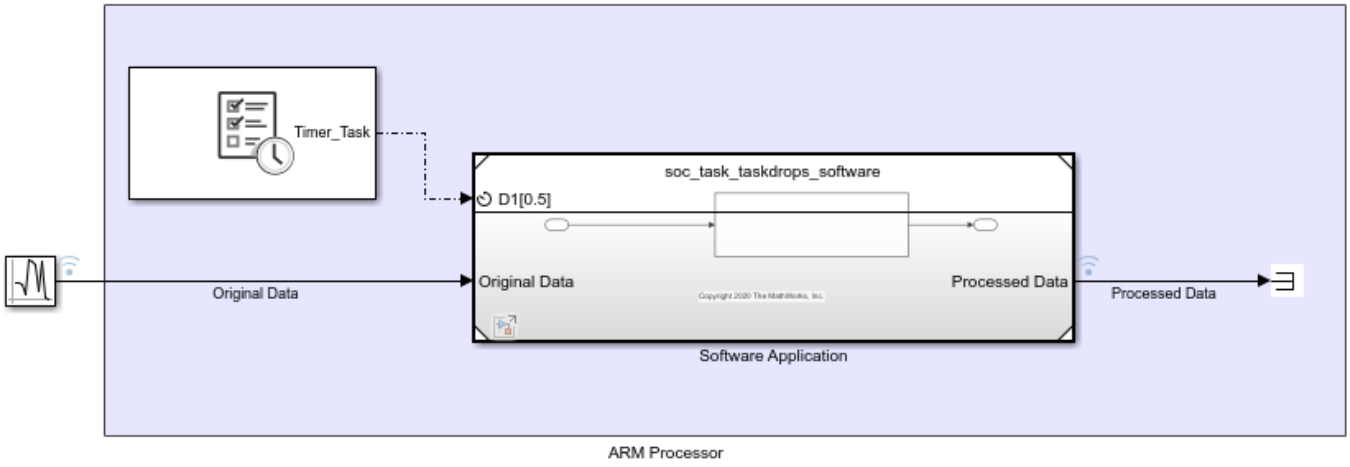
In some designs, a task must execute when the task trigger signal occurs or with the latest state of the system. If a task has been triggered and a new task trigger occurs, the new instance can be removed or *dropped*. After dropping the execution instance of the task that overran the next execution instance starts when the event trigger signal arrives. To drop tasks when an overrun occurs, in the Task Manager block, enable the **Drop task that overrun** parameter.

Task Drops in Simulation

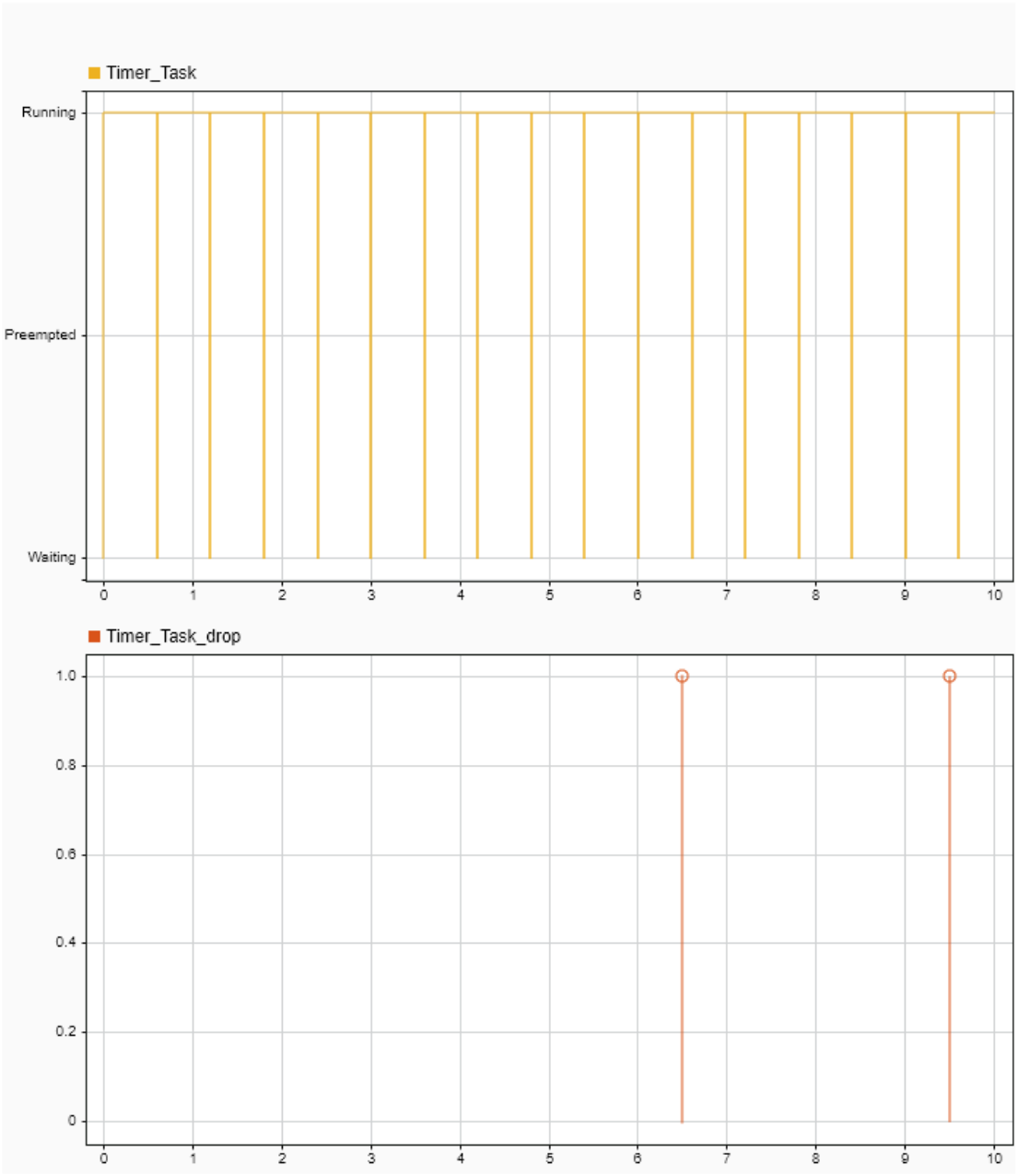
This example shows how to configure a task in the Task Manager block to drop when a task overrun occurs during simulation.

Task Overrun Without Task Drops

This model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the Timer Driven Subsystem, inside the Software Application Model Reference block. A Random Number block simulates a data source that the timer-driven task samples.



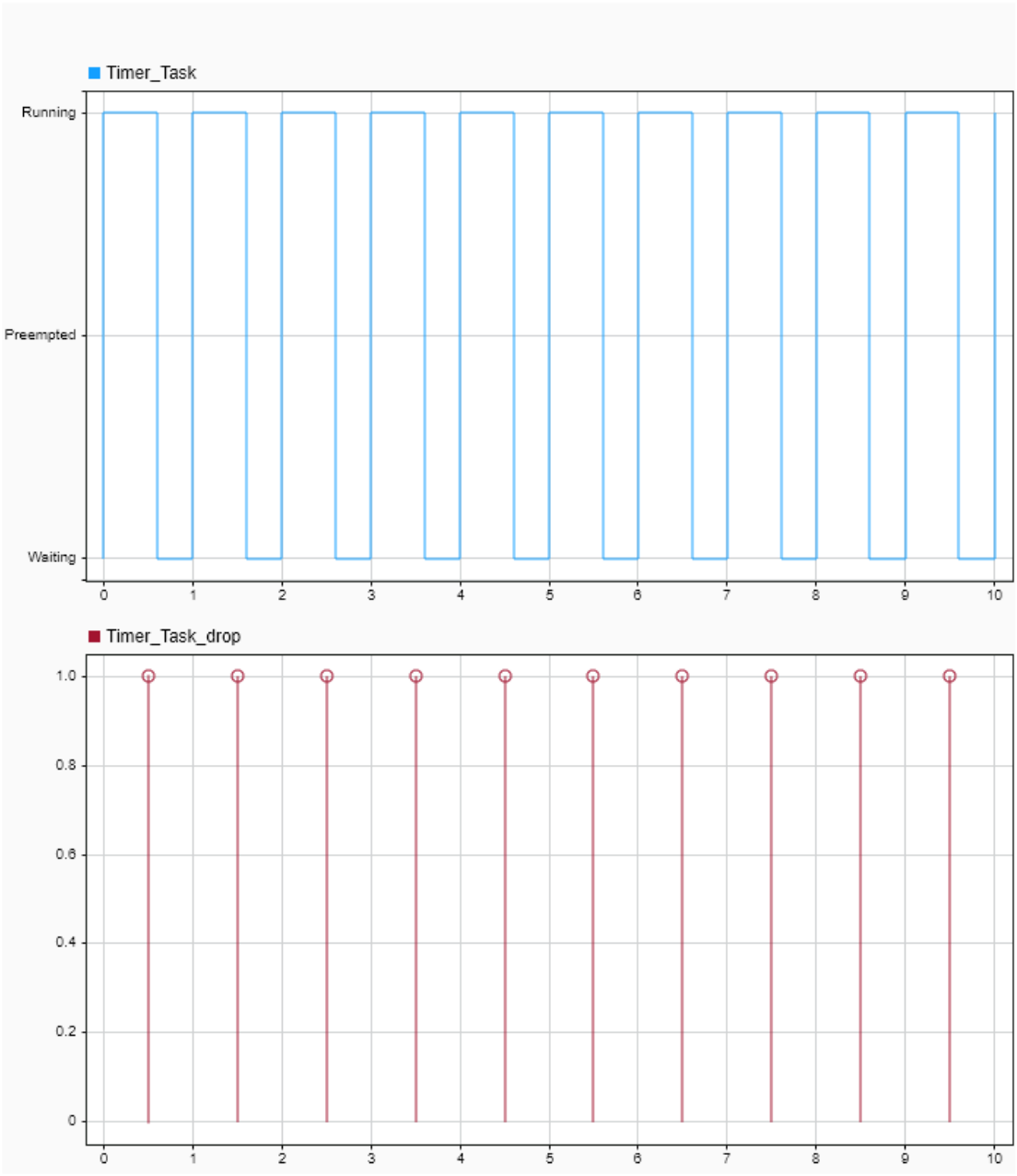
In this model, the task duration of 0.6 seconds exceeds the task period of `0.5` seconds causing the task to overrun. Click the Run button to build and run the model. When the model finishes running, the Simulation Data Inspector shows the task execution timing.



Inspecting the execution timing of the tasks shows that the start of each following task instance is delayed from the expected `0.5`-second interval by the overrun of the previous task. Even when **Drop tasks that overrun** is set to `off`, no more than 2 instances of a task can overrun execution. As shown in `Timer_Task_drop` signal, the additional task instances that overrun drop automatically.

Task Overrun With Task Drops

Using the same previously shown model, rather than overrunning the timer-driven task, the task drops so the next task instance starts at the `0.5`-second interval. Open the Task Manager block dialog mask, and select **Drop tasks that overrun**. Run the model again. Open the Simulation Data Inspector to view the task execution and dropped task instances.



See Also

Task Manager

More About

- “Multicore Execution and Core Visualization” on page 3-53

Value and Caching of Task Subsystem Signals

In SoC Blockset, a task subsystem can be treated as an independent model with the task duration simulating the expected execution time on an SoC device. When the Task Manager block executes a task, input signals connected to that task subsystem can either be sampled and cached at the start of the task execution or sampled at the end of the task execution instance. The task subsystem then executes using either the cached or latest value. The value of signals and buses output from the subsystem change at the end of the task execution instance.

To enable task subsystem input signal caching, first open the Simulink configuration parameters on the processor reference model. On the **Hardware Implementation** pane, select **Hardware board settings > Task and memory simulation > Cache input data at task start**.

See Also

Task Manager

More About

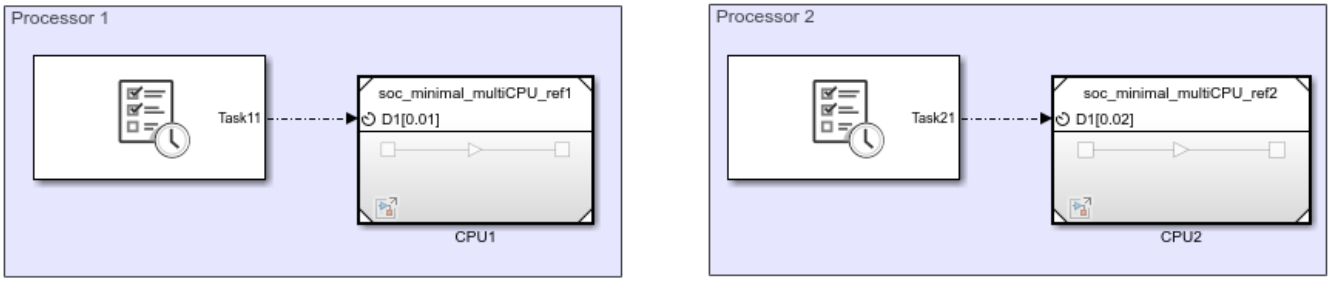
- “What is Task Execution?” on page 3-2
- “Task Duration” on page 3-13
- “Kernel Latency” on page 3-16

Multiprocessor Execution

SoC Blockset enables simulation of multiprocessor executions as they behave on a multiprocessor SoC. In multiprocessor simulations, each processor reference model executes simultaneously, where each processor execution is managed by an independent Task Manager block representing either the OS or bare-metal scheduler for that processor. Processors can interact with each other using interprocessor communication channels, through the Interprocess Data Channel block, enabling for synchronization of tasks and algorithms between the task manager of each processor.

Multiprocessor SoC Model

A multiprocessor SoC model contains at least two Task Manager blocks, each connected to a Model block representing the process to be run on a separate processor. This figure shows a minimal independent two-processor system.



In simulation, each Task Manager and Model block automatically acts as an independent processor. The tasks assigned to different Task Manager blocks run independently of the other processor while tasks within a single Task Manager block still behave dependently. For more information of task execution within a single processor, see “What is Task Execution?” on page 3-2 and “Multicore Execution and Core Visualization” on page 3-53.

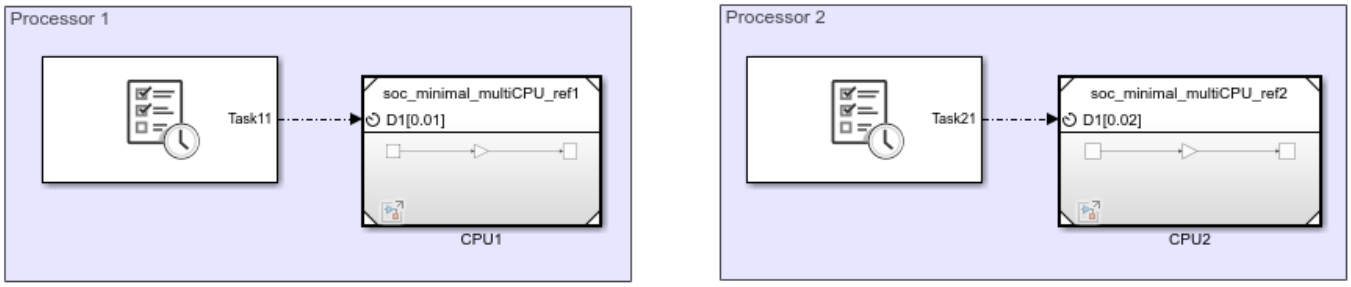
Note All tasks within the top-level model must use a unique identifier name.

Processors can communicate to each other asynchronously using a interprocess data channel. An interprocess data channel consists of the Interprocess Data Write, Interprocess Data Channel, and Interprocess Data Read blocks. For more information on processor to processor communication channels, see “Interprocess Data Communication via Dedicated Hardware Peripheral” on page 3-31.

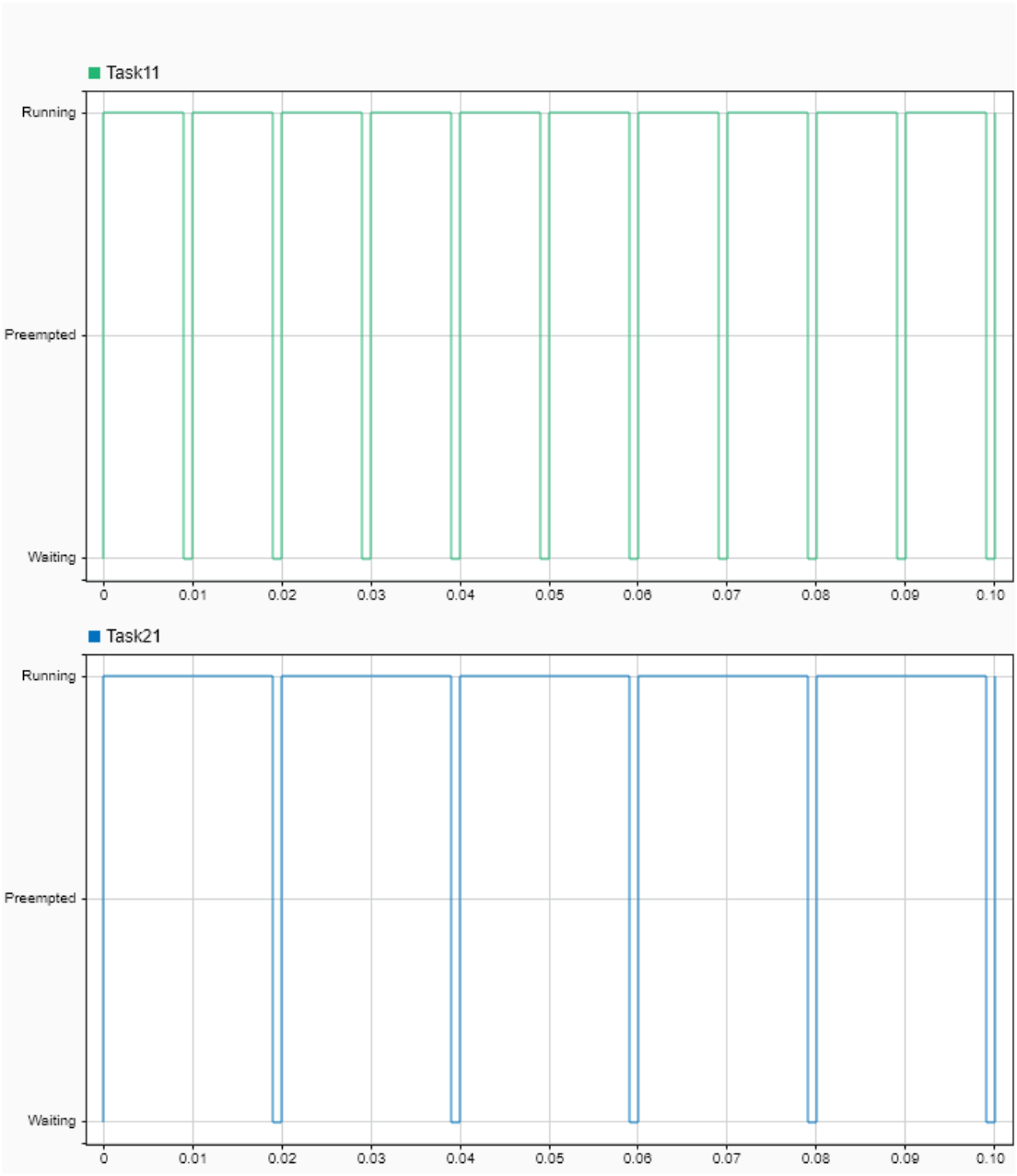
In code generation, the top-level Simulink model and each reference model must have their “Hardware board” parameter set to a supported multiprocessor hardware board, such as `TI Delfino F2837xD`. In the top-level model, you must set the **Processing Unit** parameter to `none` to indicate that the model does not build. In each reference model, you must set the **Processing Unit** parameter to a specific processor, such as `c28xCPU1`.

Multiprocessor Sample Model

This example shows a minimal multiprocessor model representing an TI Delfino F2837xD hardware board that contains a pair of C28x architectures processors in the same microcontroller die.



Each reference model, driven by the Task Manager, contains a free running counter and gain. The first model, `soc_minimal_multiCPU_ref1`, runs a timer task with a period of 0.01 and median task duration of 0.008. The second model, `soc_minimalCPU_ref2`, runs a timer driven task with a period of 0.02 and median task duration of 0.018. To run the simulation, on the Simulation tab, click Run.



Inspecting the execution timing of the two tasks, `Task11` and `Task21`, shows that each task executes independently of the other, simulating the expected behavior of the multiprocessor TI Delfino F2837xD device.

See Also

Task Manager | Interprocess Data Write | Interprocess Data Channel | Interprocess Data Read

More About

- “Multicore Execution and Core Visualization” on page 3-53

External Websites

- https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/c2000-real-time-control-mcus/overview.html

Interprocess Data Communication via Dedicated Hardware Peripheral

A variety of microcontroller units (MCUs) and SoCs provide dedicated hardware peripherals to enable processes executing on separate processors to communicate. The dedicated hardware connection eliminates the need to develop conventional channels through shared memory or through peripheral buses. Dedicated interprocess data communication in hardware is used in embedded MCUs that either support or do not support an operating system (OS). Without an OS, the process occupies the entirety of the processor resources. In this case, multiprocess systems require distribution across multiple processors within the single MCU. For example, the F2838xD family of processors from Texas Instruments™ contains a pair of interprocessor communication (IPC) peripherals that directly connect the C28 CPUs. For more information on the F2838xD processors and their IPC peripherals, see the Texas Instruments website TMS320F2838x Microcontrollers with Connectivity Manager.

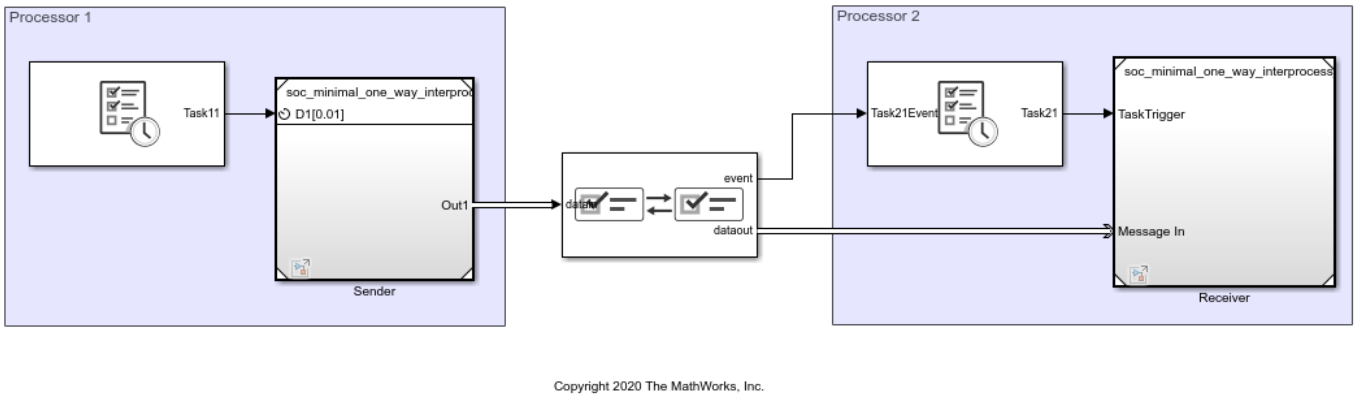
The SoC Blockset Interprocess Data Read, Interprocess Data Channel, and Interprocess Data Write blocks simulate communication between bare metal processes executing on separate processors. To create a monodirection data channel between two processors, add an Interprocess Data Write block into the processor reference model that sends data. Next, add an Interprocess Data Read block into the processor reference model that receives data. In each model, expose the event ports to the top-level model using the Outport and Inport blocks, respectively. Finally, connect the event ports in the top-level model using the Interprocess Data Channel block.

If the SoC models are built for a supported processor, such as those in the F2838xD family of processors, code is automatically generated for the hardware IPC peripherals.

One Way Interprocess Communication

This example shows one-way interprocess data communication between two bare metal processors.

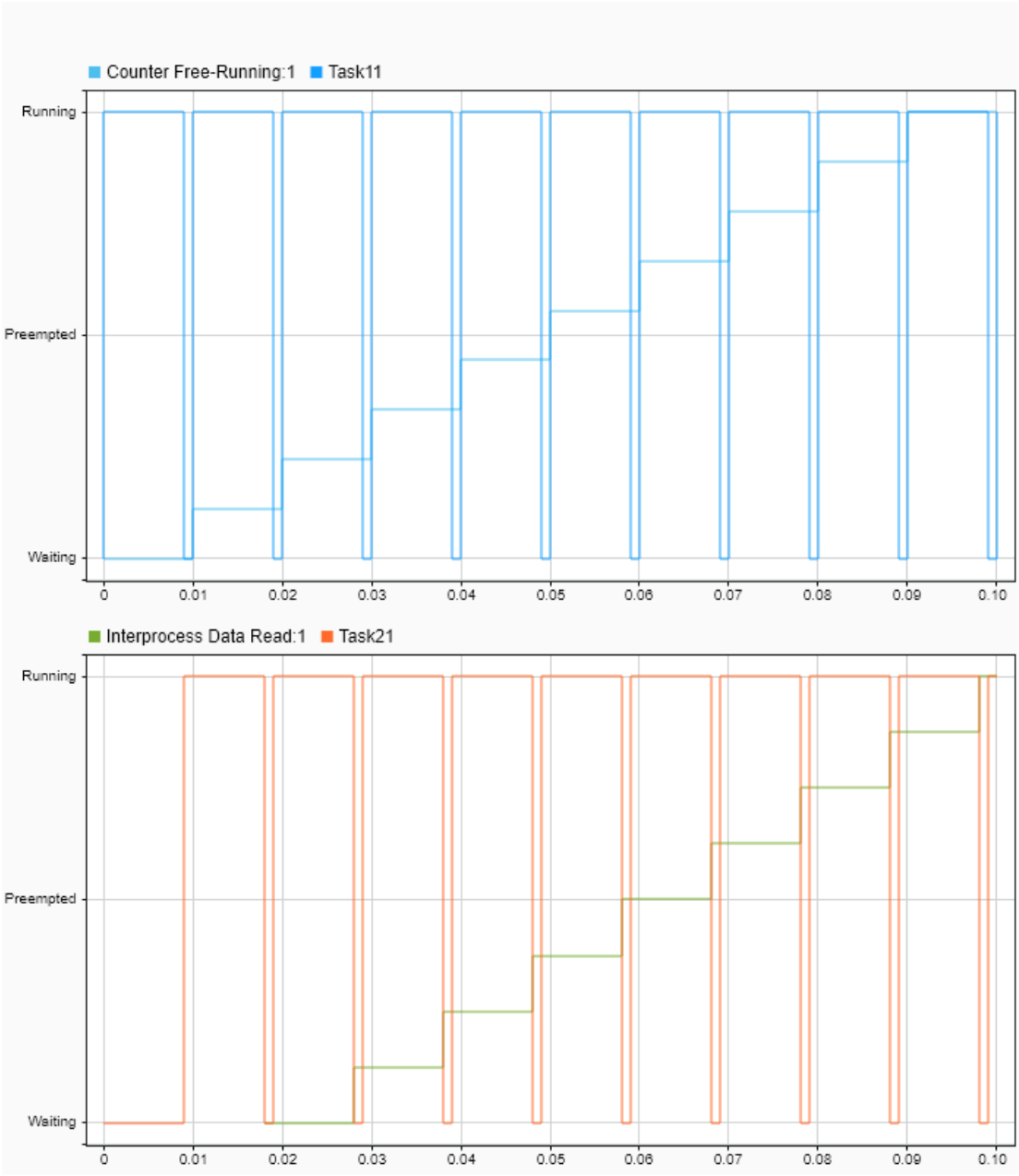
An algorithm in Processor1 sends a data message, using the Interprocess Data Write block, to the Interprocess Data Channel block at a 0.01 second interval. Processor2 two receives and processes the data messages asynchronously, using the Interprocess Data Read block.



Results

In the Simulation tab, click Run. When the simulation completes, open the Simulation Data Inspector to view the resulting signals and tasks. From the graphs, Processor1 sends the data value at the

completion of the first task, Task11, instance. The data then gets received by Processor2, triggering the event driven task, Task21. At the completion of Task21 instance, the final value gets emmitted in Processor2, potentially for additional processing by other tasks.



See Also

Interprocess Data Read | Interprocess Data Channel | Interprocess Data Write | Task Manager

Code Generation of Software Tasks

A Simulink model containing a Task Manager block simulates task execution. When a model gets deployed to an SoC hardware board, the SoC Blockset automatically creates and assigns the tasks to threads, links interrupts, messages, and system events to the generated code of the model.

Timer-Driven Tasks

An SoC Blockset model, when implemented onto hardware as generated and compiled code, uses an operating system (OS) timer to drive the base-rate time step of the model. All time based signals derive their time steps, known as sub-rates, from the base-rate time step of the model. A timer-driven task, created from the Task Manager block, uses a counter that increments at each base-rate timer step. When the counter reaches an integer multiple of the base-rate, the generated code posts to the semaphore associated with that task. Posting to the semaphore unblocks the thread and executes the task.

Event-Driven Task

Each event-driven task created from the Task Manager block gets a unique semaphore. A unique event elsewhere in the system posts to that semaphore and puts the task thread into the running state. OS kernel handles the management of the task thread until it returns to the waiting state.

See Also

Task Manager | SoC Builder

More About

- “Event-Driven Tasks” on page 3-8
- “Timer-Driven Task” on page 3-4

Recording Tasks for Use in Simulation

Each time a model containing a Task Manager block runs in simulation or on an embedded processor with external mode, Simulink records task execution data and statistics as a set of files. A diagnostics folder, with name *`modelname`*`_diagnostics`, contains two subfolders, `sim` and `hw`, for the data from simulations and recorded from hardware, respectively. Each run generates a unique folder, inside either the `sim` or `hw` folders, labelled by the date and time of the run. The folder name uses a time-date format, *`YYYY_MM_DD_hh_mm_ss`*, representing the year, month, day, hour, minute, and second, respectively.

Note To enable external mode in an SoC model, use the **SoC Builder** tool.

Each run generates a set of metadata, statistics, and execution recording files, including:

- `TaskInfo.mat` – This file contains task information, including the task names and types, used internally by the SoC Blockset.
- `metadata.csv` – This file contains the derived mean and standard deviation for all tasks recorded in the `profile.log` data file. The `metadata.csv` file can be used directly in the Task Manager block to set task duration statistics. For more information on setting task duration, see “Task Duration” on page 3-13.
- *`TaskName`*`.csv` – This file contains the recorded task execution data as a comma-separated variable list. The first column contains the start time of each task instance. The second column contains the task durations for each task instance. If a task is dropped, lost, or corrupted, the start time and duration of that task execution instance are both replaced by `-1`. For more information on using recorded task execution timing in simulation, see “Task Execution Playback Using Recorded Data” on page 3-40.

Note

- Tasks recorded from an embedded processor only start capturing task execution after successful connection of external mode. The lost start-up in task execution recordings from hardware should be considered when comparing timing results to recordings from simulation.
- When executing on an embedded processor, task execution recordings times will continue to run until the completion of all task instances scheduled in the Task Manager prior to the stop time of the model.

See Also

Task Manager

More About

- “Task Duration” on page 3-13
- “Task Execution Playback Using Recorded Data” on page 3-40

Task Priority and Preemption

Task priority informs the operating system of the importance of the task and the order in which a group of waiting tasks needs to execute. By setting the priorities of the tasks in the Task Manager block, tasks that need to react to critical or time-sensitive events can preempt lower priority and background tasks.

Tasks listed in the Task Manager block execute in a *rate monotonic* order. Rate-monotonic order requires the task with the highest static priority in the preempted state to immediately preempt all other tasks and enter the running state. Timer-driven tasks with shorter periods get higher static priorities. If two tasks with equal priority in the preempted state, when no other running task exists, then tasks execute in a first-in, first-out (FIFO) order.

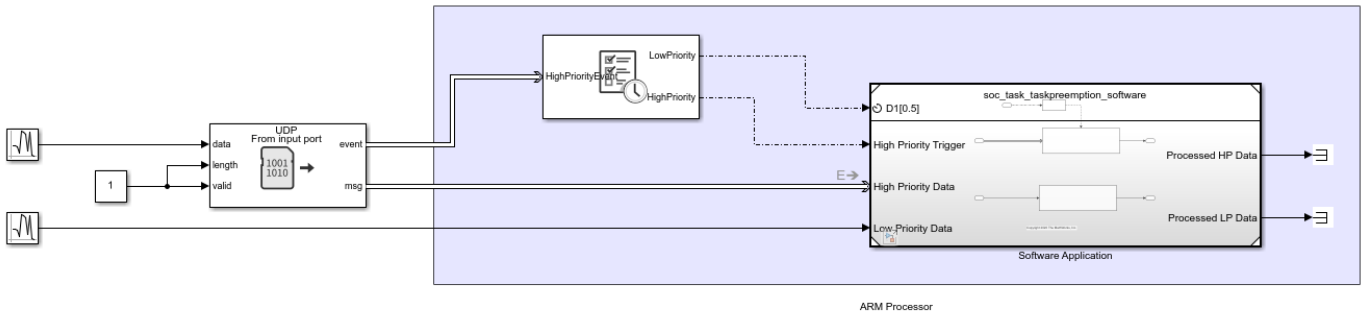
Each event-driven task listed in the Task Manager block can be set with an explicit execution priority. Timer-driven tasks inherit their priority from the base rate task priority of the model. In the configuration parameters, the base rate task priority is set by the **Hardware Implementation > Hardware board settings > Operating system/scheduler > Base rate task priority** parameter. The following example shows the interaction between a pair of competing tasks.

Preemption of Low Priority Task by High Priority Task

This example shows how the task manager changes the state of two tasks, preempting the lower priority task to allow the high priority task to run.

Task Manager with High and Low Priority Tasks

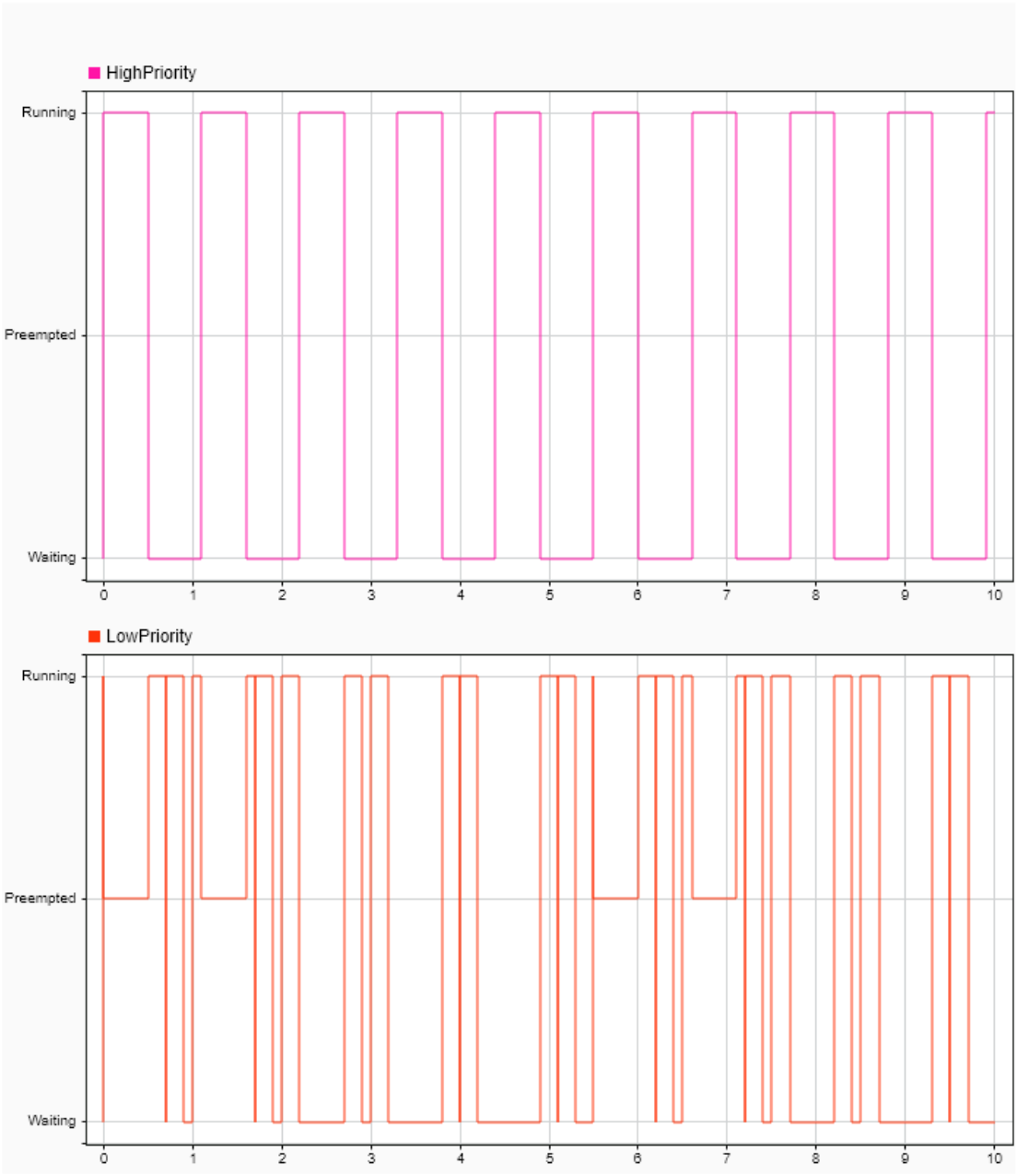
The following model simulates a software application with a high and low priority task. A Task Manager block schedules the execution of the task subsystems inside the Software Application Model Reference block.



The low priority, timer driven, task is scheduled to run every 0.5 seconds with a duration of 0.2 seconds. The high priority, event driven, task is scheduled to run when a new UDP data packet arrives, which occurs every 1.1 seconds and requires a task duration of 0.5 seconds. As a result of these timing conditions, the low priority task gets preempted to allow the high priority task to run.

Simulation Showing Task Preemption

Click the Run button to build and run the model. When the model finishes running, open the Simulation Data Inspector to see the results of the simulation. Select the HighPriority and LowPriority task waveforms to see the task preemption.



Inspecting the Simulation Data Inspector at time 1.0, the low priority task starts executing until time 1.1, getting preempted by high priority task. The low priority task then runs to completion at 1.7

seconds, overrunning the next instance of the low priority task that should have started at 1.5 seconds.

See Also

Task Manager

More About

- “What is Task Execution?” on page 3-2
- “Task Overruns and Countermeasures” on page 3-20

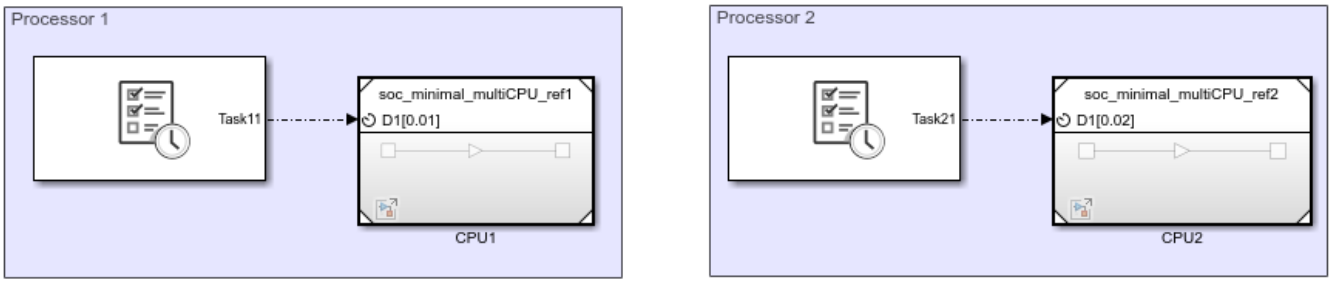
Run Multiprocessor Models in External Mode

In multiprocessor external mode simulation, each processor reference model can be deployed run simultaneously on the processors contained in the SoC or microcontroller. While model run on the separate processors, you can interact with each model to observe signals, tune model parameters, and evaluate the overall behavior of the multiprocessor system when running on the hardware. This section describes the typical workflow configuration used to setup an external mode simulation onto a supported multiprocessor hardware board, such as the `TI Delfino F2837xD`.

Note Hardware boards supporting multiprocessor deployment can be found in the SoC Blockset Support Package for Texas Instruments C2000™ Processors.

Process to Run Multiprocessor Model

1. Create or open a multiprocessor SoC model and configure the model for a supported hardware board, such as the `TI Delfino F2837xD`. This figure shows an example of a minimal multiprocessor model. For more information on creating and configuring a multiprocessor model, see “Multiprocessor Execution” on page 3-27.



2. Connect each CPU in the hardware board to your host computer. In the `TI Delfino F2837xD`, CPU1 can be connected using the SCIA native port which connects to the USB port on the `TI Delfino F28379D Launch Pad` hardware board. CPU2 can be connected to the host computer using an external FTDI, a serial to USB converter, connect to the SCIB native port on the hardware board. Both the SCIA and SCIB ports are now exposed as the COM ports on the host computer. Different hardware boards will require their own connection setup to expose their own connection ports, one for each processor in your system.

   Note The SCIB ports are mapped to the pins on the hardware boards as follows:

   - `TI Delfino F28379D LaunchPad` – Rx `GPIO19` & Tx `GPIO18b`
   - `TI Delfino F2837xD` – Rx `GPIO11` & Tx `GPIO9`

3. Open one of the processor reference model. In the Simulink toolstrip, on the **System on Chip** tab, click **Hardware Settings** to open the **Configuration Parameter** window.
4. On the **Hardware Implementation** > **Target Hardware Resources** > **External Mode** tab, set the **Communication interface** to `serial(using xcp)`. Set the **Serial port** to match `COM` port previously defined for that processor. Set the **Baudrate** to the maximum supported by the hardware board or else data drop may be observed.

5. Check the **Hardware Implementation > Task profiling on processor > Show in SDI** to enable **Simulation Data Inspector** logging. Close the **Configuration Parameter** window.
6. In the Simulink toolstrip, on the **System on Chip** tab, click **Configure, Build & Deploy** to launch the **SoC Builder**. For more information on **SoC Builder**, see SoC Builder.
7. In the **SoC Builder** tool, on the **Select Build Action** screen, select **Build and load for external mode**. Click **Next**.
8. In the following screen, select the CPUs that will run the external mode models.
9. Complete the remaining steps and on the **Run Application** screen, click **Load and Run** to launch the external mode models on the processors. The models automatically start running in external mode.
10. To stop the external mode execution, in the Simulink toolstrip, on the **System on Chip** tab, click **Stop**.

Note You must stop all models. Stopping only one model while leaving others running can produce undefined behavior.

View External Mode Simulation Data

During and after running an external mode simulation on multiple processors, the tasks and signals can be viewed in the **Simulation Data Inspector**. Each processor records an independent run in the Simulation Data Inspector and contains all the tasks and signals that executed on that processor. Since the external mode is launched by the **SoC Builder**, all the runs for the separate processors share a common time, allowing comparison of the processor runs to each other to see the overall behavior of the software portion of your system on the SoC hardware.

Note External mode, profiler, and data logging use the same communication channel. To prevent data drops and gaps, do not run external mode simulations with profiling or data logging enabled, and vice-versa.

See Also

SoC Builder | “Multiprocessor Execution” on page 3-27 | **Simulation Data Inspector**

More About

- “External Mode Simulations for Parameter Tuning and Signal Monitoring” (Simulink Coder)

Task Execution Playback Using Recorded Data

The Task Manager block can replay the execution timing of a task recorded from either a previous simulation of that task or from the execution of a task on a processor in an SoC device. To replay a task timing data file, use the following procedure:

1. In a Simulink model, open the Task Duration block dialog box.
2. Select a task from the list of available tasks.
3. In the **Simulation** tab, select **Play back recorded task diagnostics file**.
4. Click **Browse** to select a *`taskname`*`.csv` file from a previous task simulation.

While using the data file for the task timing information, the Task Manager still manages individual tasks according the scheduling of the system and can be preempted by other higher priority tasks in the model. For more information on task priority and preemption, see “Task Priority and Preemption” on page 3-35.

See Also

Task Manager

Related Examples

- “Task Execution” on page 7-78

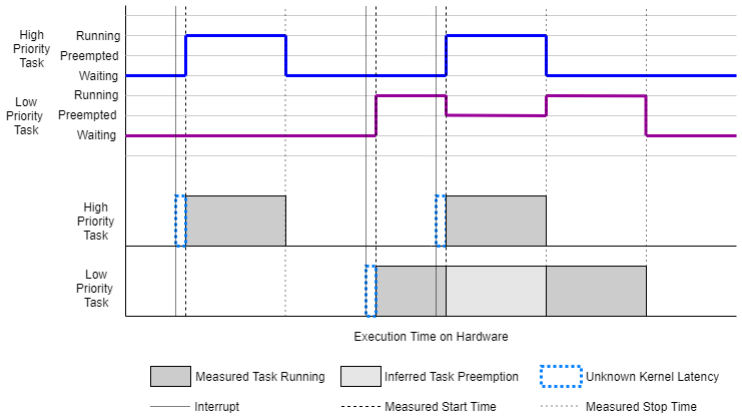
More About

- “Task Duration” on page 3-13

Code Instrumentation Profiler

In a code instrumentation profiler, code gets added into the generated code to record the start and stop times of each task executing on the processor. The recorded start and stop times of each task are sent to the development computer to be saved, processed, and displayed. The instantaneous state of each task gets inferred from the combined start and stop times and priorities of all the tasks within the process.

Consider a simple model with two tasks, one high- and low-priority executing on an embedded processor and measured by a code instrumentation profiler. This diagram shows the measurements made by the code instrumentation profiler and the inference on the individual task states resulting from these measurements.



Inspecting the diagram, it shows that the state of the low-priority task gets inferred from the higher-priority task's execution. Since only the start and end times of task execution get measured, some pertinent data can be lost, specifically kernel latency. As kernel latency precedes the start of the task, the actual time of the interrupt event is not directly observed and the start time of the task can assumed to be delayed from the actual time of the interrupt. Furthermore, when a task moves from the preempted to the running states, the kernel latency gets added into the interpreted execution time of the lower-priority task.

Code instrumentation profiling benefits from easy generation and deployment. On models deployed to processors with operating systems running a single process in a single tasking mode, task execution timing measurements be made with sufficient accuracy and precision. As only a minimal amount of code to record the start and stop times of the task get added to each task, the impact of the task execution timing by the code instrumentation profiler, inmost cases, can be considered negligible.

Limitations

Code instrumentation profiling provides lightweight measurement tooling of generated code. However, two limitations must be considered when measuring the task execution and duration times using the code instrumentation profiler. These limitations are as follows:

- Cannot measure kernel latency or components of kernel latency. Kernel latency can generally be treated as a constant. As kernel latency impacts all task start up time with approximately equal effect, an estimate of the kernel latency could be deduced with comparisons to the task timings in simulation. For more information on kernel latency, see “Kernel Latency” on page 3-16

- Cannot capture the effect of commands issued to the OS kernel from within the task using Custom Code blocks. The code instrumentation profiler records the start time, end time, and preemption of a task by other tasks. However, when the task makes a call to the OS kernel, the code instrumentation profiler does not record the change of control between the task and the kernel as a preemption. As kernel calls, without detailed knowledge of the timing, can be treated as non-deterministic, the measured task duration cannot be reliably measured using this type of profiler. For more information on task duration, see “Task Duration” on page 3-13.

See Also

Task Manager

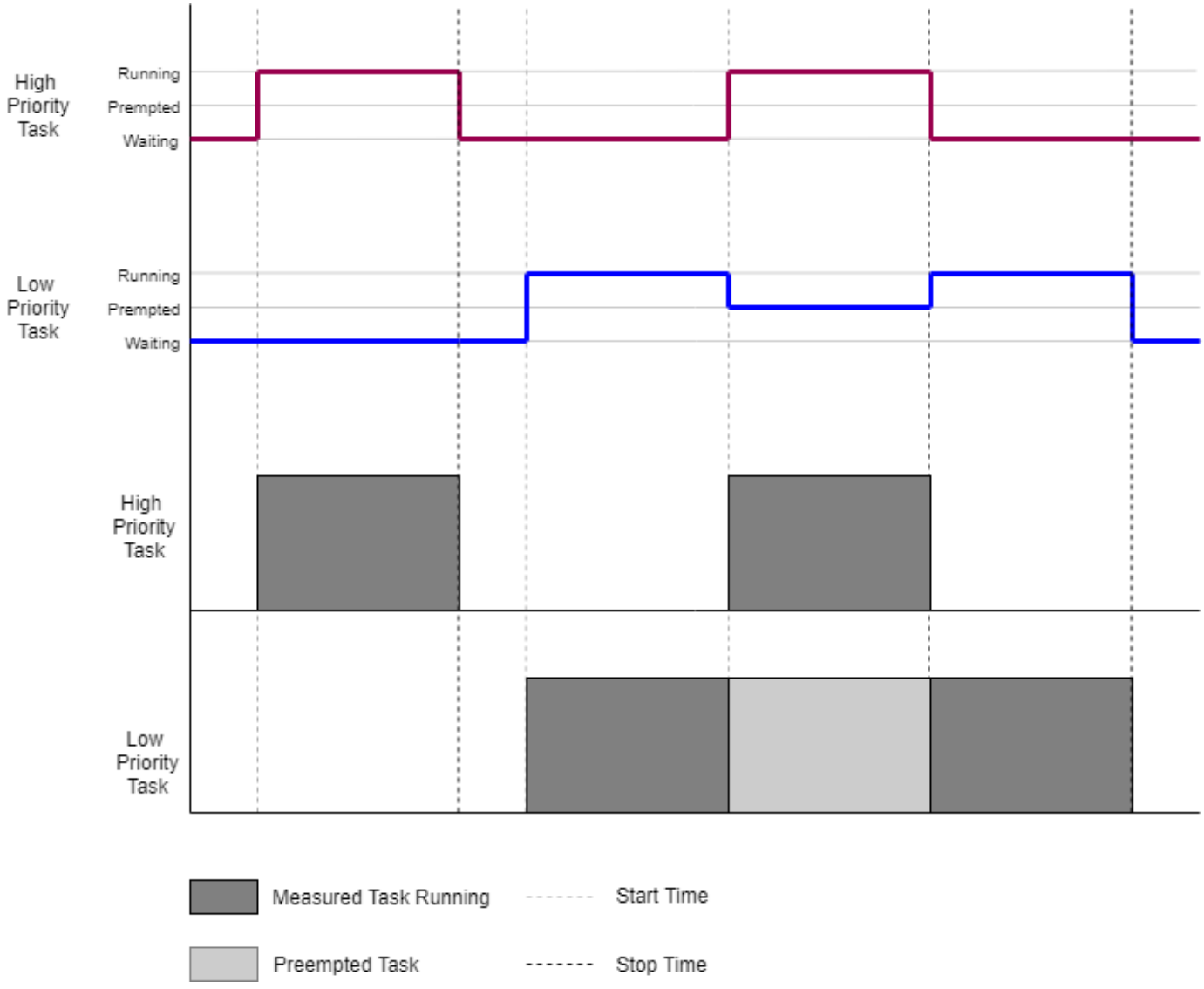
More About

- “Kernel Instrumentation Profiler” on page 3-43
- “Kernel Latency” on page 3-16
- “Task Duration” on page 3-13

Kernel Instrumentation Profiler

A Kernel instrumentation profiler uses a subset of the software tools and libraries included in the Linux kernel, for monitoring the actions made by the kernel to manage the execution of processes running on the SoC hardware. SoC Blockset features use LTTng, an open source tracing framework for Linux, as a Kernel instrumentation profiler to monitor the execution of tasks and events of the Simulink model deployed on the SoC hardware. For more information, see the LTTng website.

Unlike a code instrumentation profiler, a kernel instrumentation profiler directly measures the conditions and changes in state for all tasks by monitoring the Linux OS kernel. This diagram shows the measurements made in a multitasking process with high and low priority tasks.



When a high priority task preempts a low priority task, the low priority task enters into the *Preempted* state and the high priority task enters into the *Running* state. After the high priority task completes execution, the scheduler resumes the preempted low priority task.

When using a kernel instrumentation profiler, the LTTng tracing framework traces the task state transitions directly from the Linux kernel and gives accurate task execution time. In comparison, when you use a code instrumentation profiler, it can incorrectly include the kernel latency in the execution time of the task.

Kernel instrumentation profiling provides these advantages.

- High accuracy of timing measurements
- Knowledge of task execution and task state transition directly from the kernel
- CPU information of the processor core where the task executes

Limitations

You can perform kernel instrumentation profiling only on SoC hardware that runs using a Linux OS.

Kernel instrumentation profiling for an unlimited time duration on hardware with high task rate models could result in packet loss of profiling data streamed from hardware. For more information, see “Task Profiling on Processor”.

See Also

Task Manager

More About

- “Code Instrumentation Profiler” on page 3-41
- “Kernel Latency” on page 3-16
- “Task Duration” on page 3-13

Data Logging Techniques

Data logging enables real-time capture of signals from embedded hardware boards and platforms to be displayed and stored in the Simulation Data Inspector in Simulink. Depending on the demands application being developed, data logging can be achieved in these three general configurations.

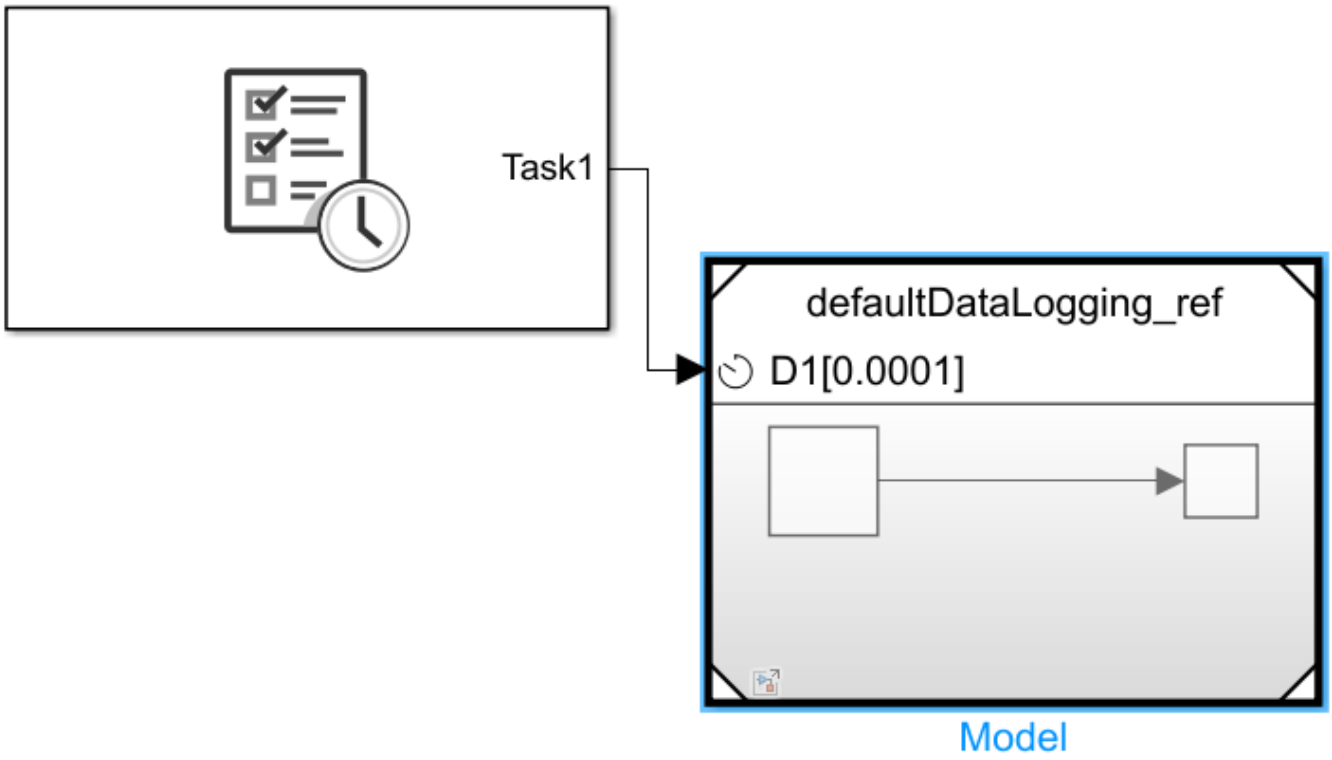
- Standard data logging
- Subsampled data logging
- Multiprocessor data logging

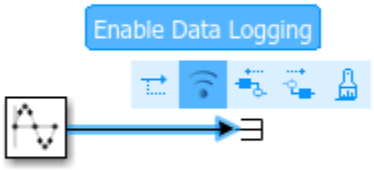
These examples show minimal models for each of these data logging configurations and the reasoning for each configuration type. All of these examples use the `TI Delfino F28379D` hardware board, however these techniques can be used with any supported SoC Blockset hardware board or platform.

Standard Data Logging

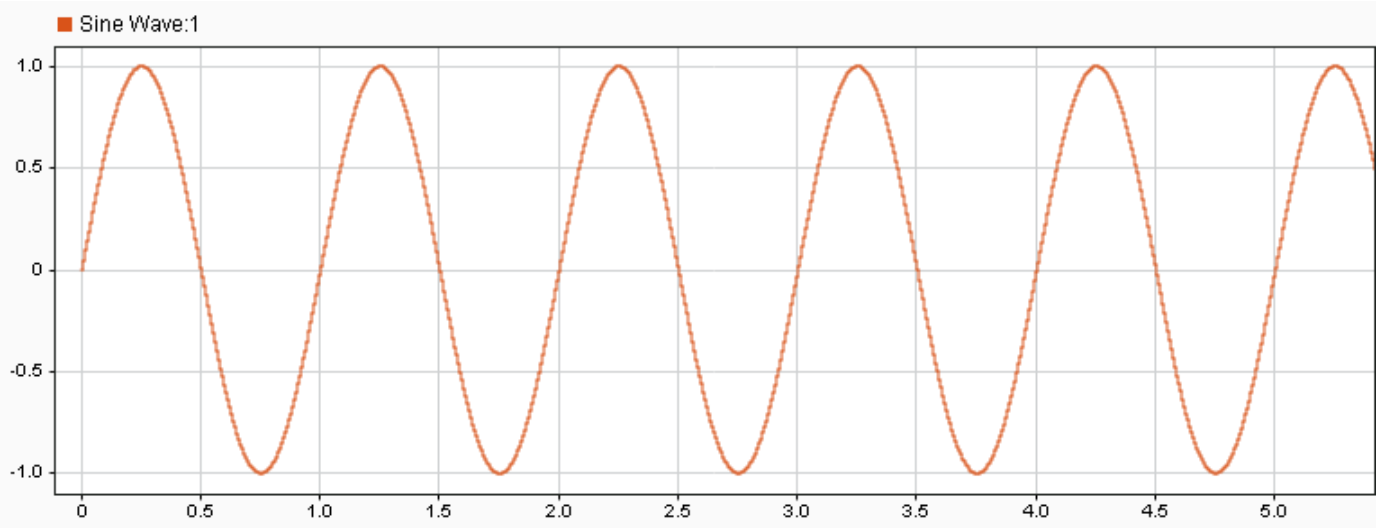
This example shows how to configure an SoC Blockset model to log data from hardware when the model is deployed to a TI Delfino F28379D LaunchPad. The system contains a single timer-driven task that consists of a Sine Wave block connected to a Terminator block. To log the output signal from the Sine Wave block, select the signal line, click the ellipsis, and select **Enable Data Logging**. This selection automatically registers this signal to be logged from the model during simulation and to be displayed to the Simulation Data Inspector. Open the model by executing this code.

```
open_system("defaultDataLogging_top.slx")
```





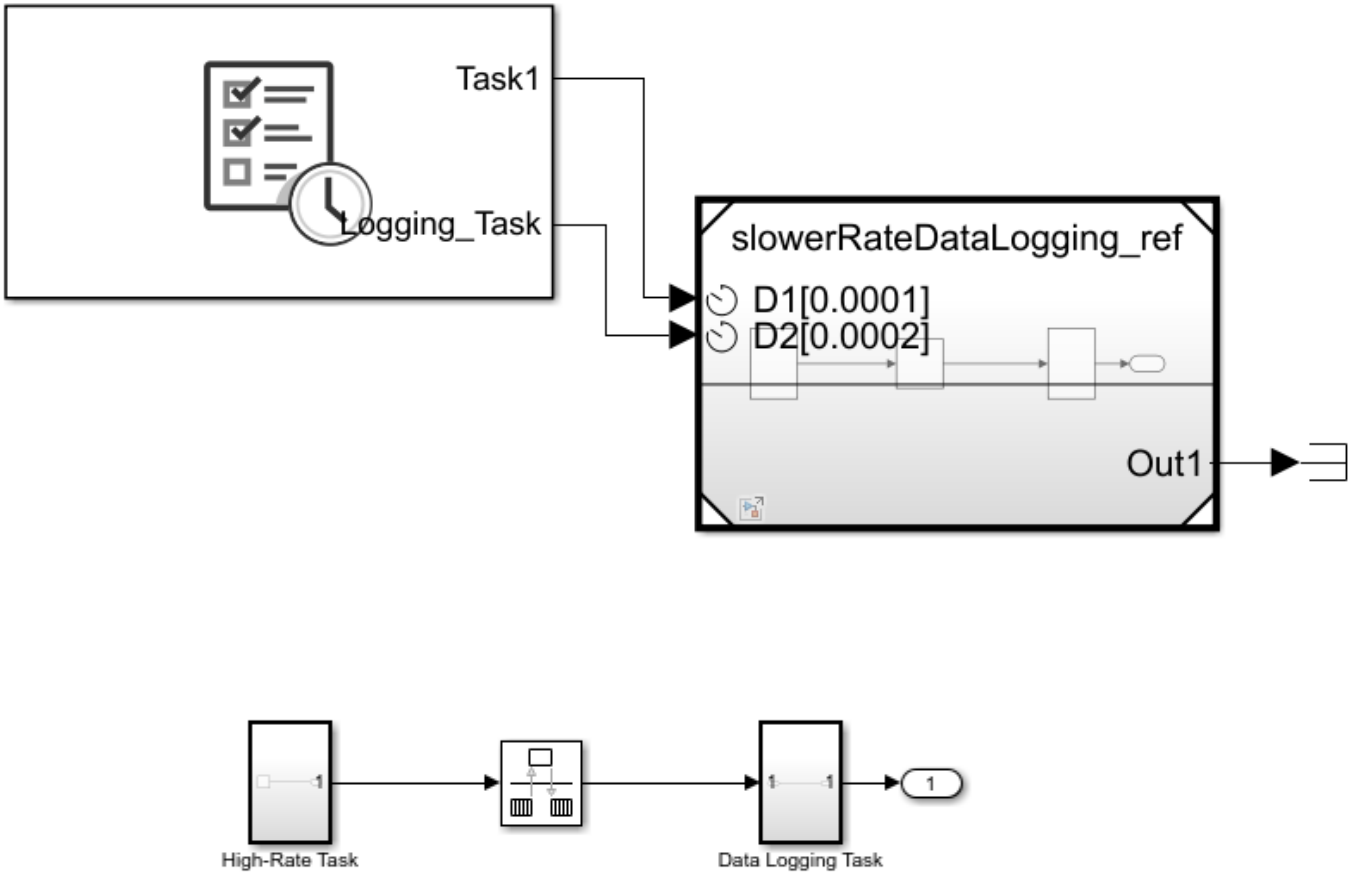
Use the SoC Builder tool to deploy the model to the TI Delfino F28379D LaunchPad. A host-target communication connection, set up by the **SoC Builder**, enables data to be automatically logged from the executable running on the hardware board to the Simulation Data Inspector in Simulink. This image shows the logged data signal from the model deployed to a TI Delfino F28379D LaunchPad.



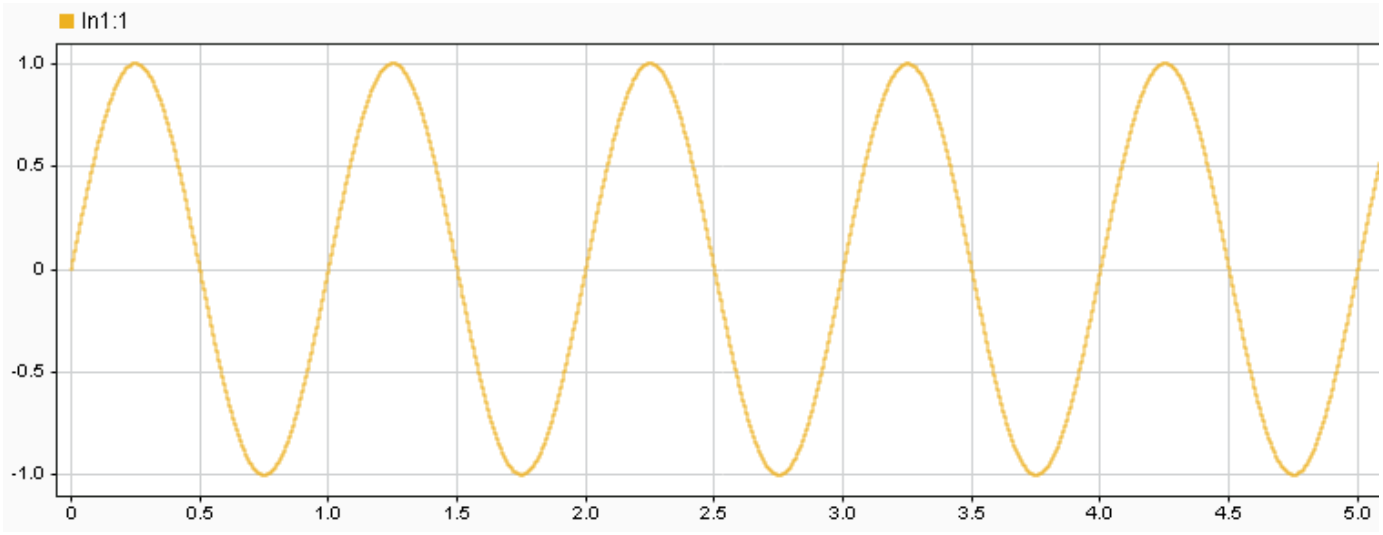
Subsampled Data Logging

This example shows how to configure a resource intensive SoC Blockset model to log data from hardware when the model is deployed to a TI Delfino F28379D LaunchPad. The system contains two timer-driven tasks. The first task consists of a Sine Wave block connected to a Terminator block that represents a task running at a high rate. The second task uses a Rate Transition block to subsample and log the signals from the high-rate task.

```
open_system("slowerRateDataLogging_top.slx")
```



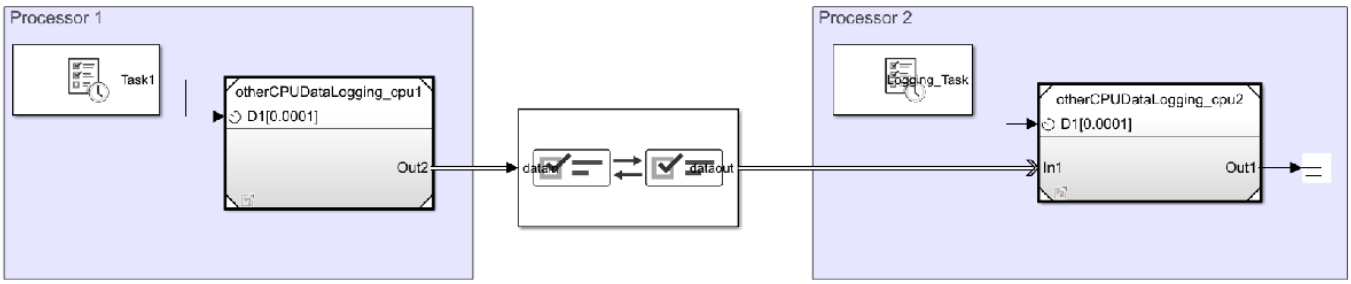
Use the SoC Builder tool to deploy the model to the TI Delfino F28379D LaunchPad. A host-target communication connection, set up by the **SoC Builder**, logs the subsampled data from the executable running on the hardware board and sends the data to the **Simulation Data Inspector** in Simulink. By enabling data logging in the slower, low-priority task, data can be captured on hardware from the resource intensive, high-priority task without interfering with its behavior or reaching the limits of the host-target communication system. This image shows the subsampled logged data signal from the model deployed to a TI Delfino F28379D LaunchPad.



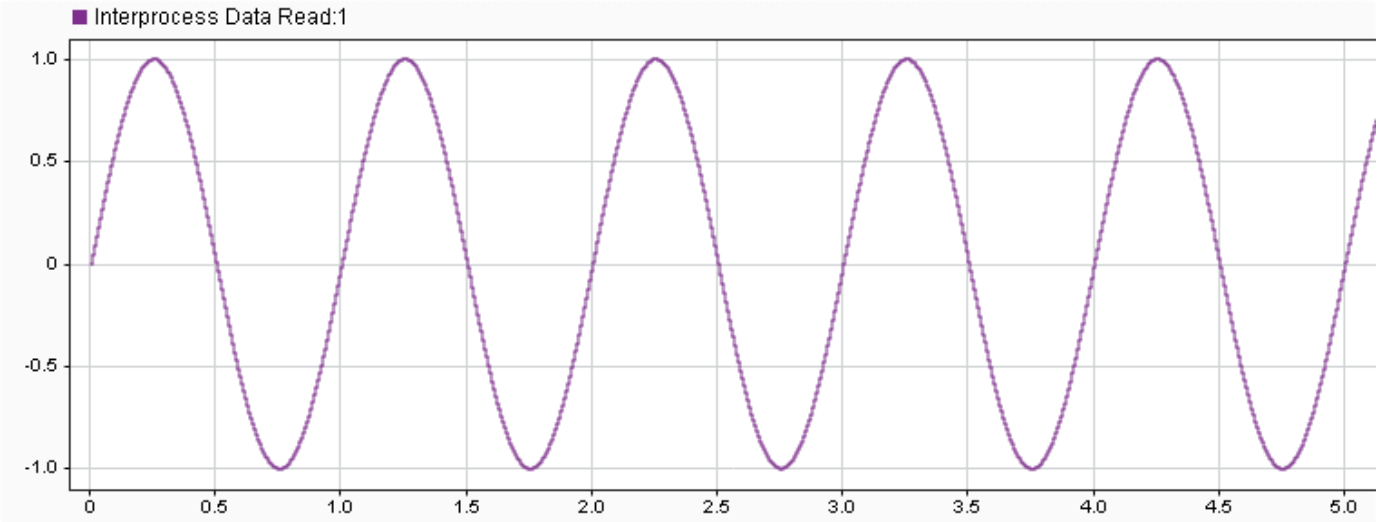
Multiprocessor Data Logging

This example shows how to configure a resource intensive SoC Blockset model to log data from hardware when the model is deployed to a TI Delfino F28379D LaunchPad. The system contains two timer-driven tasks divided across two processors. Task 1 (on processor 1) consists of a Sine Wave block connected to a Terminator block and represents a high-rate, resource intensive task. An Interprocess Data Channel block connects processor 1 and 2, providing data transfer between the processors. Task 2 (on processor 2) logs signals transeffed from task 1 back to Simulink.

```
open_system("otherCPUDataLogging_top.slx")
```



Use the SoC Builder tool to deploy the model to the TI Delfino F28379D LaunchPad. A host-target communication connection, set up by the **SoC Builder**, logs the signal data from the executable running on processor 2 of the hardware board and sends the data to the **Simulation Data Inspector** in Simulink. Using processor 2 to own and manage the host-target communication and data logging, data can be captured from the resource intensive, high-priority task on processor 1 without interfering with its behavior and enabling that task to consume most of the processor resources yet maintain quality of the data logging to Simulink. This image shows the logged data signal from task 1 on processor 1, captured on task 2 on processor 2, of the model deployed to a TI Delfino F28379D LaunchPad.



See Also

Simulation Data Inspector | SoC Builder | Interprocess Data Channel

See Also

Task Visualization in Simulation Data Inspector

The Simulation Data Inspector display provides a direct view into the execution timing, the task state, and the execution of tasks in simulation and profiled from generated code running on hardware. Each model run, in simulation or on hardware using external mode adds task execution timing and data to the current **Run**. This image shows the Simulation Data Inspector display with a **Run** captured from an SoC Blockset model.

Each **Run** contains these task related signal types:

- *`taskname`* – The execution instance state for the task, with name *`taskname`*, defined in the Task Manager block. For more information on task execution states, see “What is Task Execution?” on page 3-2.

  Note If a task instance does not run to completion during the simulation time, the final task execution instance does not render in the Simulation Data Inspector display.

- *`taskname`*`_drop` – An impulse indicating the scheduler dropped an execution instance of task, *`taskname`*`_drop`. For more information on task drops, see “Task Overruns and Countermeasures” on page 3-20.
- `Core: `*`n`* – Execution activity on core *n* of the simulated processor. For more information on multicore execution and visualization, see “Multicore Execution and Core Visualization” on page 3-53.

  Note If a task instance does not run to completion during the simulation time, the related core status over that instance does not render in the Simulation Data Inspector display.

See Also

Simulation Data Inspector | Task Manager

More About

- “What is Task Execution?” on page 3-2
- “Task Overruns and Countermeasures” on page 3-20
- “Multicore Execution and Core Visualization” on page 3-53

Multicore Execution and Core Visualization

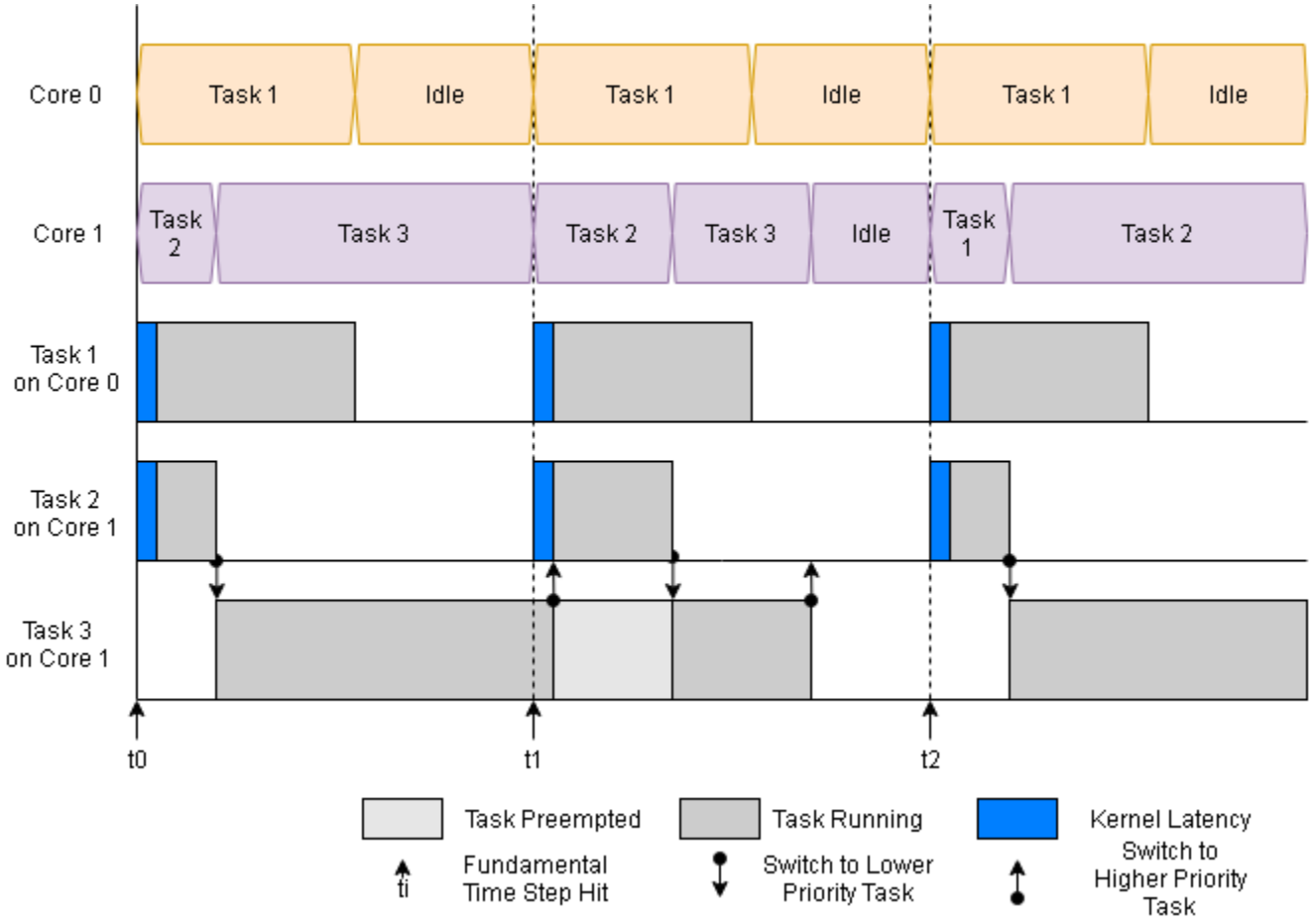
SoC Blockset enables simulation of task executions as they behave on a multicore processor. In multicore simulations, tasks can run simultaneously when assigned to different processor cores. Additionally, assigning lower-priority tasks to unique cores prevents these tasks from getting preempted, giving greater confidence to the final application.

Specify the Core for a Task

To set the processor core on which a task executes, open the Task Manager block dialog mask. Select a **Task** from the available tasks. In the task properties, set **Core** to a nonnegative integer value. During simulation, task instances execute on the specified core, subject to the preemption by other tasks executing on the same core. For more information on task preemption, see “Task Priority and Preemption” on page 3-35.

Core Visualization in Simulation Data Inspector

SoC Blockset provides a view of the processor cores on the Simulation Data Inspector. This diagram shows the visualization of the core activity relative to the task state.



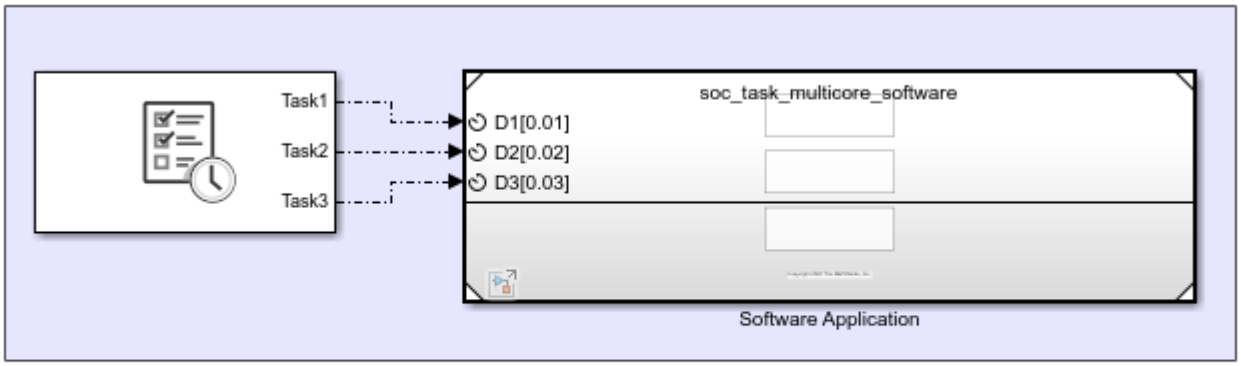
In the Simulation Data Inspector, the signal `corei` shows the current task executing on that core. When the core activity displays as idle, then that core has all tasks in the waiting state, and the kernel can use that core for background tasks that are not part of the main application.

Note If a task instance does not run to completion during the simulation time, the related core status over that instance appears empty in the Simulation Data Inspector display.

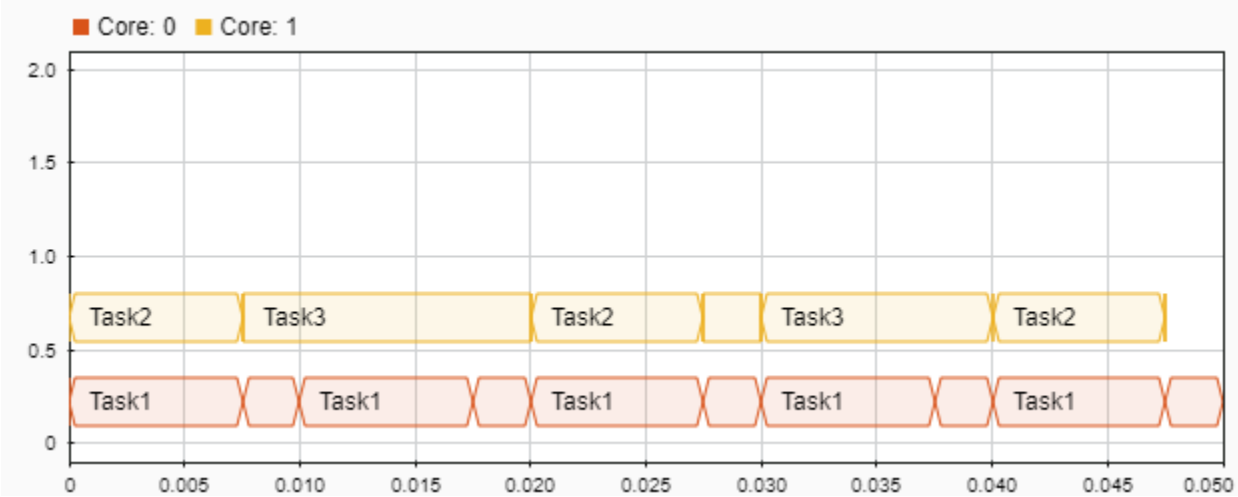
Multi-Core Task Execution

This example shows the simulation of multiple tasks, managed by the Task Manager block, execute on multiple cores with display the core activity shown in the Simulation Data Inspector.

This model simulates a software application, running on an ARM processor, with 3 timer-driven tasks. A Task Manager block schedules the execution of the tasks, inside the `Software Application` Model block. Tasks 1, with a period of `0.01` seconds, executes on `Core 0`. Tasks 2 and 3, with periods of `0.02` and `0.03` seconds, respectively, execute on `Core 1`.



Click the Run button to build and run the model. When the model finishes running, open the Simulation Data Inspector display to see the results of the simulation. Select the **Core 0** and **Core 1** to view the core execution status.



As shown in the Simulation Data Inspector, the core executes either the running task or moves to an idle state, to perform background kernel tasks. Additionally, as two cores are used in this application, high-priority, Task1 executes at the start of each trigger event. Similarly, Task2 and Task3 do not get preempted by Task1. As a result, the application makes better use of the available processor resources.

See Also

Task Manager | Simulation Data Inspector

More About

- “Task Priority and Preemption” on page 3-35

Programmable Logic

Using the Algorithm Analyzer Report

Executing the `socModelAnalyzer` function on a Simulink model or the `socFunctionAnalyzer` function on a MATLAB function results in a report that details the resources used by the model or function, respectively.

The report includes information for each mathematical or logical operator in the top model or function, with individual lines for each operator and data type. For example, multiplication with data type `double` and multiplication with data type `uint32` are listed separately. The report lists each instance of the operator as a separate line. The report includes these fields.

- `Path` – The path to the operator within the structural hierarchy of the top model or function
- `Count` – The number of times the operator is executed in the design
- `Operator` – The operator used
- `DataType` – The data type used for the output of the operator
- `Link` – A link to the location of the operator in the model or function

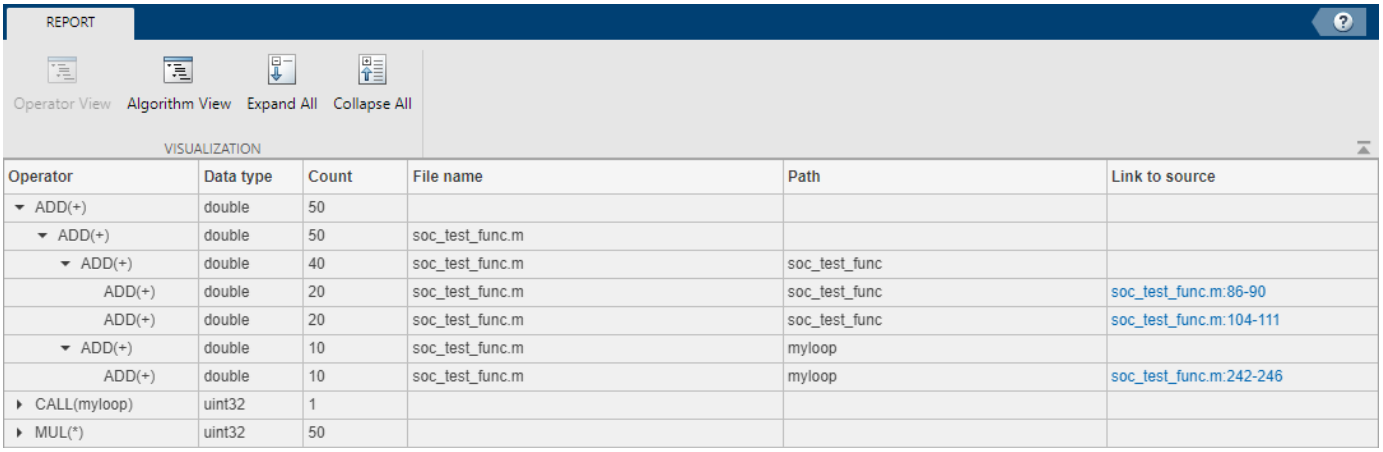
Open Report

Use one of these options to access the report.

- Execute the `socModelAnalyzer` function, and then click the `Open report viewer` link.
- Execute the `socFunctionAnalyzer` function, and the click the `Open report viewer` link.
- Execute the `socAlgorithmAnalyzerReport` function, specifying a MAT-file generated by the `socModelAnalyzer` or `socFunctionAnalyzer` function.

Operator View

View the generated report in the operator view. On the report toolstrip, click **Operator View**. Then, when clicking **Collapse All** each line represents the number of operator executions per data type. A line in the collapsed-view of the report represents one or more operators, with the same data-type. Expand a line to see the individual operators contributing to the count, their path in the model hierarchy, and a link to their location in the model.



Operator	Data type	Count	File name	Path	Link to source
ADD(+)	double	50			
ADD(+)	double	50	soc_test_func.m		
ADD(+)	double	40	soc_test_func.m	soc_test_func	
ADD(+)	double	20	soc_test_func.m	soc_test_func	soc_test_func.m:86-90
ADD(+)	double	20	soc_test_func.m	soc_test_func	soc_test_func.m:104-111
ADD(+)	double	10	soc_test_func.m	myloop	
ADD(+)	double	10	soc_test_func.m	myloop	soc_test_func.m:242-246
CALL(myloop)	uint32	1			
MUL(*)	uint32	50			

Algorithm View

View the generated report in the operator view. On the report toolstrip, click **Algorithm View**. Then, when clicking **Collapse All** each line represents a top node in the hierarchy. You can expand a line to navigate to the function or model that you are analyzing. Use this view when you are interested in analyzing the operators in a specific model or function. When using this view, you can collapse the view for other models or functions.

File name	Path	Count	Operator	Data type	Link to source
soc_test_func.m		101			
soc_test_func.m	myloop	20			
soc_test_func.m	soc_test_func	81			
soc_test_func.m	soc_test_func	40	ADD(+)	double	
soc_test_func.m	soc_test_func	40	MUL(*)	uint32	
soc_test_func.m	soc_test_func	20	MUL(*)	uint32	soc_test_func.m:104-107
soc_test_func.m	soc_test_func	20	MUL(*)	uint32	soc_test_func.m:103-116
soc_test_func.m	soc_test_func	1	CALL(myloop)	uint32	

See Also

`socModelAnalyzer` | `socFunctionAnalyzer` | `socAlgorithmAnalyzerReport`

Considerations for Multiple IPs in FPGA Model

When your FPGA model includes more than one block for which you'd like to generate HDL using HDL Coder, you must use a connector model to connect your blocks.

For additional information, see Stream Connector and Video Stream Connector blocks.

Export Custom Reference Design from SoC Model

You can use the `socExportReferenceDesign` function to generate a reference design from an SoC Blockset model and avoid the manual steps required to generate and register a custom reference design. The function generates these artifacts.

- Board registration files
- Reference design registration file
- IP repository
- Design files
- Constraint files

SoC models can be one of these types.

- An SoC Model with an FPGA, memory, and optional I/O (no processor)
- An SoC Model with a processor, FPGA, memory, and optional I/O

Create SoC Model of System

When exporting a custom reference design from an SoC model, the reference design does not include the design under test (DUT) and the interface to the DUT is exposed. After generating the reference design, you can integrate your custom IP by using the **HDL Workflow Advisor** tool. Your custom IP must have the same interface as the FPGA Algorithm block.

To export a custom reference design, first create an SoC model to model the system and the I/O available on your board. To create an SoC Blockset model, use one of these methods.

- Create a model by using an SoC Blockset template (recommended). For more information, see “Use Template to Create SoC Model” on page 2-4.
- Build an SoC model from scratch. For more information, see “Create an SoC Project Application” on page 2-31.

Include a DUT subsystem in the model. This subsystem must have the same interface as the IP core that you are developing. Because the generated reference design does not include the DUT subsystem, the DUT can be a simple model or just a pass-through block.

Prepare SoC Model for Reference Design Export

You can use the MATLAB as AXI master feature in the exported reference design to interact with the SoC device from the host. In Simulink, open the Configuration Parameters dialog box by clicking **Model Settings** on the **Modeling** tab, and on the left pane, select **Hardware Implementation**. Then, expand **Target hardware resources**, select **FPGA design (top-level)**, and then select **Include 'MATLAB AXI Master' IP for host-based interaction**.

In the **IP core clock frequency (MHz)** box, specify the IP core clock frequency in MHz.

To ensure that your SoC model supports code generation, use the **SoC Builder** tool to generate executables and deploy your model. For more information about the **SoC Builder** tool, see “Generate SoC Design” on page 2-46.

For an example showing this workflow on an FPGA-only case, see “Export Custom Reference Design” on page 7-112.

Additional Preparation When SoC Model Includes Processor

A device tree (DT) is a data structure that describes the hardware to the operating system. When you add an IP to the design, you should generate a new device tree so that the operating system can access the IP.

If your model contains both FPGA and processor subsystems, these additional steps are required before exporting the reference design.

1. In the configuration parameters , click **Hardware Implementation** on the left. Then, expand **Target hardware resources**, and select **Include processing system** in **FPGA design (top-level)**.
2. Run the **SoC Builder** tool, follow the guided steps for code generation, and then load the binaries to the FPGA. This step is required because **SoC Builder** automatically generates a device tree file (`.dtb`) on the SD card named `hdlcoder_rd/soc_prj.output.dtb` and a software model with matching device names.
3. Copy the device tree file from the folder `hdlcoder_rd` to the root folder of the SD card. In the generated `plugin_rd.m` file, the custom device tree file is specified as:

   ```
   hRD.DeviceTreeName = 'soc_prj.output.dtb';
   ```

Execute socExportReferenceDesign Function

Export the custom reference design for your model by using the `socExportReferenceDesign` function. For example, for a model named `soc_image rotation`, enter this code at the MATLAB command prompt.

```
socExportReferenceDesign('soc_image_rotation')
```

The function generates these artifacts in the current folder.

- Board registration files
- Reference design registration file
- IP repository
- Design files
- Constraint files

Integrate IP Core into Generated Reference Design

Add the generated folder to the MATLAB path. Use the **HDL Workflow Advisor** tool to guide you through the steps for integrating your IP and generating hardware and software executables for deployment on an SoC device.

For an example showing the full workflow on an FPGA-only case, see “Export Custom Reference Design” on page 7-112. If your model includes a processing system, these additional steps are required when using the **HDL Workflow Advisor** tool.

1. In Simulink, right-click the DUT block that you want to integrate into the reference design, and select **HDL Code > HDL Workflow Advisor** to open the **HDL Workflow Advisor** tool. Alternatively, use the `hdladvisor` function.
2. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to the platform generated by the `socExportReferenceDesign` function.
3. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.
4. In step 1.3, set the target interface by connecting each port in your IP to the corresponding port in the reference design.
5. Click **Run This Task** to run the **Set Target Interface** task.
6. Continue with the remaining steps of the **HDL Workflow Advisor** tool.
7. Optional: In step 4.2, you can choose to generate a software interface model with IP core driver blocks (requires an Embedded Coder® license). If you choose to generate this software interface model, clear **Skip this task** under **Generate a software interface model with IP core driver blocks for C code generation**.

   For more information, see the section titled "Generate a software interface model" in “Getting Started with Targeting Xilinx Zynq Platform” (HDL Coder).

   The generated software interface model contains AXI driver blocks that match the interface of the DUT subsystem. The device name is set to `'/dev/mwipcore'` by default. Change the device name in these AXI driver blocks to match the in the device tree file used by the SD card image.

   There are several ways to find the device name:

   - The device name is derived from the DUT name of the SoC model. If you export a reference design using an SoC model with the DUT name specified as `'soc_hwsw_stream_fpga/FPGA Algorithm Wrapper'`, the generated device name in the AXI driver blocks is `'/dev/mwfpga_algorithm_wrapper_ip0'`.
   - Find the device name in your operating system (OS) image after booting the SoC device. To do that, login to the board using UART or SSH protocols, and execute:

     ```
     ls/dev
     ```

     For example:

```
zynq> ls /dev
bus                             tty12
console                         tty13
cpu_dma_latency                 tty14
full                            tty15
gpiochip0                       tty16
iio:device0                     tty17
iio:device1                     tty18
input                           tty19
kmsg                            tty2
log                             tty20
loop-control                    tty21
loop0                           tty22
loop1                           tty23
loop2                           tty24
loop3                           tty25
loop4                           tty26
loop5                           tty27
loop6                           tty28
loop7                           tty29
mem                             tty3
memory_bandwidth                tty30
mmcblk0                         tty31
mmcblk0p1                       tty32
mtd0                            tty33
mtd0ro                          tty34
mtd1                            tty35
mtd1ro                          tty36
mtd2                            tty37
mtd2ro                          tty38
mtd3                            tty39
mtd3ro                          tty4
mtd4                            tty40
mtd4ro                          tty41
mtdblock0                       tty42
mtdblock1                       tty43
mtdblock2                       tty44
mtdblock3                       tty45
mtdblock4                       tty46
mwfpga_algorithm_wrapper_ip0    tty47
mwtest_source_wrapper_ip0       tty48
```

8 In step 4.4, set **Programming method** to **Download**.

9 When the **HDL Workflow Advisor** tool is done building, it returns a generated bitstream file. Program the FPGA with the generated bitstream file.

10 You can now deploy the software interface model in standalone mode, or use it in external mode to interact with the SoC device. For an example, see the section titled "Run the software interface model on Zynq ZC702 hardware" in “Getting Started with Targeting Xilinx Zynq Platform” (HDL Coder).

See Also

`socExportReferenceDesign`

More About

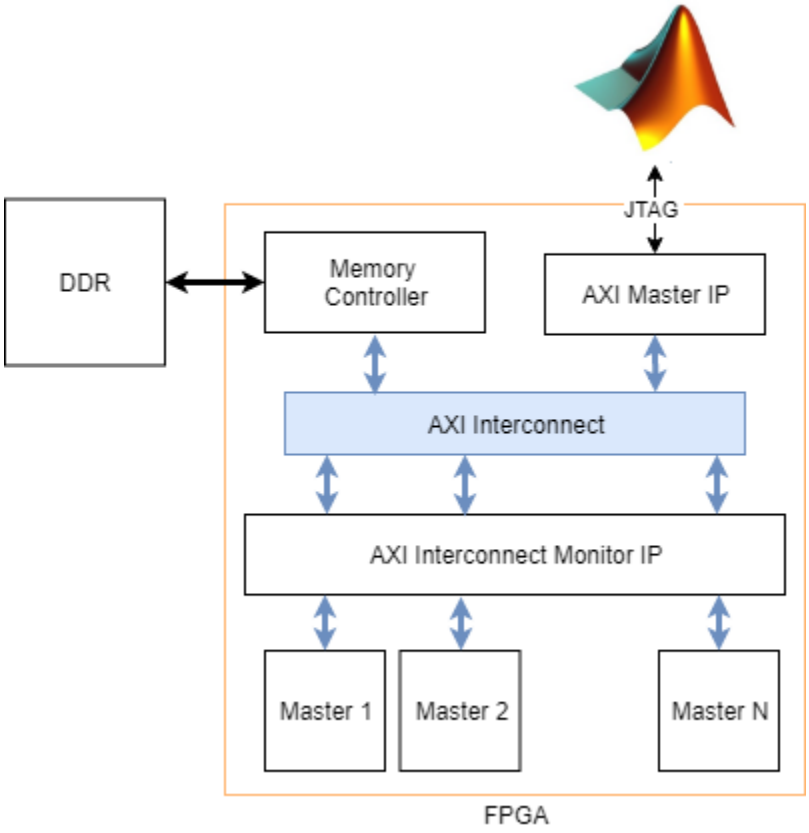
- “SoC Generation Workflows” on page 2-44
- “Custom Reference Design” (HDL Coder)
- “Export Custom Reference Design” on page 7-112
- “Getting Started with the HDL Workflow Advisor” (HDL Coder)

Memory Performance Information from FPGA Execution

In this section...

Similar to the memory performance plots generated in simulation, you can collect memory interconnect traffic information from a design running on the FPGA. You can then generate similar performance plots. You can also capture the memory transaction information to view in the **Logic Analyzer** tool similar to the burst transactions from the memory controller in simulation. Use these plots to monitor real memory performance, debug and improve the design, and compare them against the memory performance obtained in simulation.

To include an AXI interconnect monitor (AIM) IP in your design, in the configuration parameters of the model, select the **Include AXI interconnect monitor** option under **Hardware Implementation > Target hardware resources > FPGA design (debug)**. The AXI interconnect monitor IP collects information from the design while it is running on the FPGA. You can query this information from MATLAB by using the JTAG connection. All memory masters in your FPGA are connected to the AXI interconnect monitor IP. These masters can include Memory Channel and Memory Traffic Generator blocks that you generated HDL code for or any other masters in your design.



The **SoC Builder** tool generates a JTAG test bench script for your design. The script collects the performance metrics from the AXI interconnect monitor and launches the performance plot

application, which plots the memory performance plots for bandwidth, number of bursts, and transaction latencies. These plots are similar to the plots of memory performance in simulation. You can also modify the script to collect and display memory transaction waveforms similar to the burst waveforms of memory controller in simulation. For information on the simulation memory performance, see “Simulation Performance Plots” on page 5-19 and “Buffer and Burst Waveforms” on page 5-15.

For an example, see “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57, which shows how to monitor memory performance in both simulation and when running on the FPGA. The script generated by the **SoC Builder** tool uses the JTAG connection to enable any traffic generators in your design, and then samples the memory performance information from the AXI interconnect monitor IP as fast as it can. The sampling interval depends on the JTAG latency, which is typically from 10 ms to 20 ms. The script then displays plots similar to the performance plots from the Memory Controller block in your simulation. The plot displays the bandwidth, number of bursts, and transaction latency for each master.

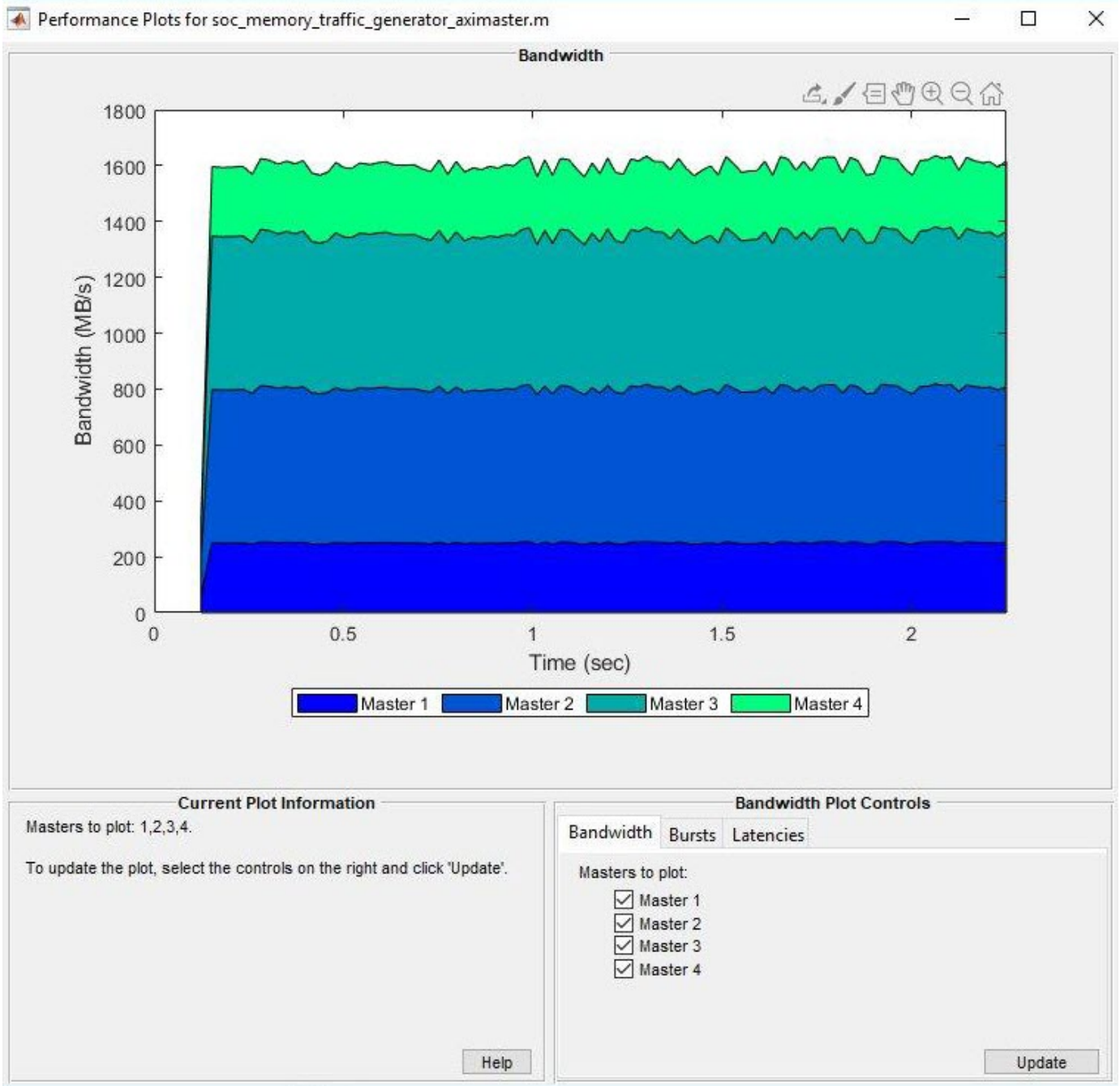
Note The AXI master itself is not connected to the AXI interconnect monitor. Therefore, the hardware diagnostics do not include the memory usage plots for test-bench-only masters that initialize the memory with predetermined data.

Memory Performance Plots

The script collects the performance metrics from the AXI interconnect monitor and launches the performance plot application.

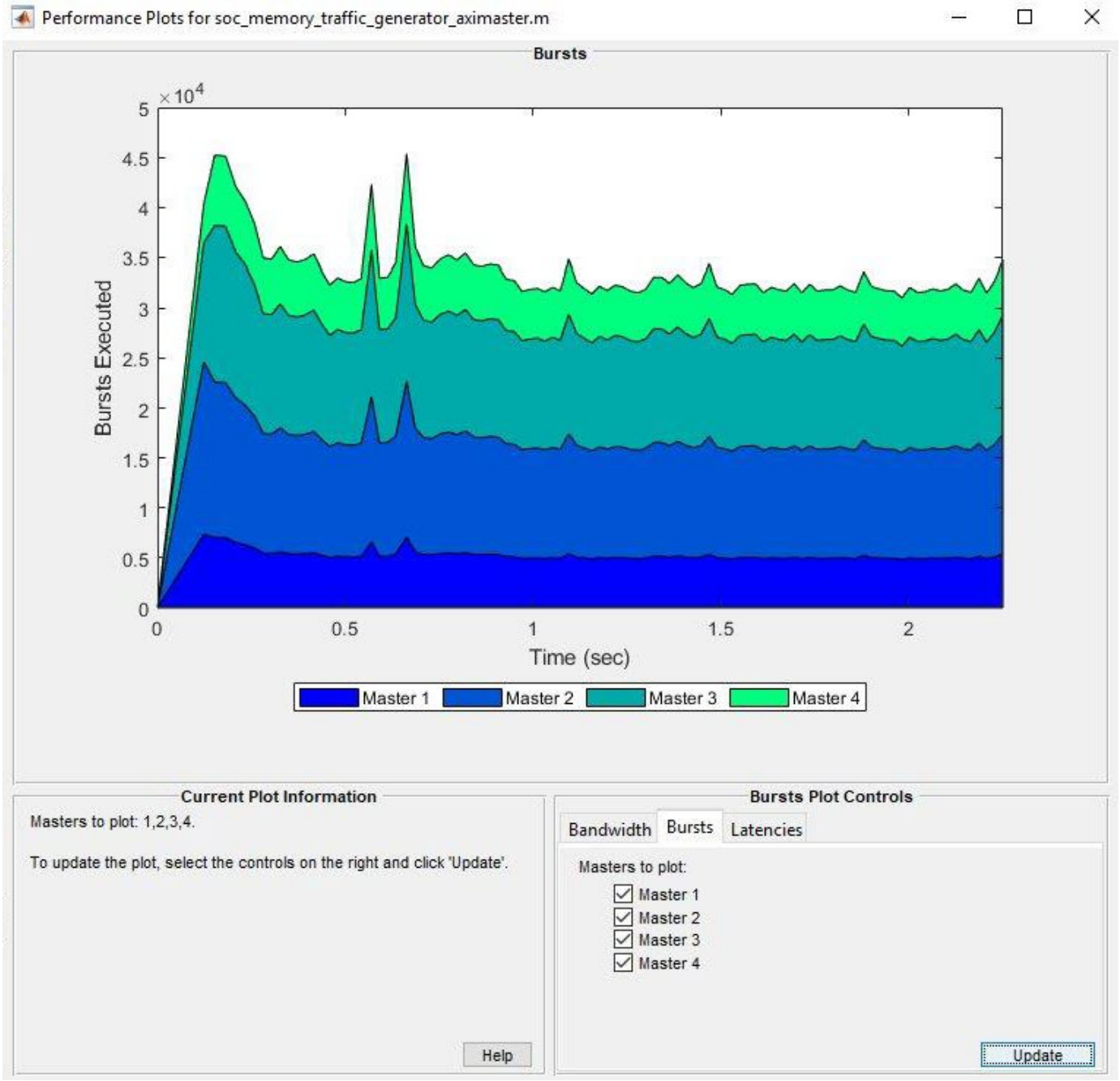
Memory Bandwidth Plots

In the **Bandwidth** tab, select the masters for which you want to graph bandwidth. Click **Create Plot** to see the bandwidth, in megabytes per second, for the selected masters over the duration of the run time. This figure shows the bandwidth for the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example.



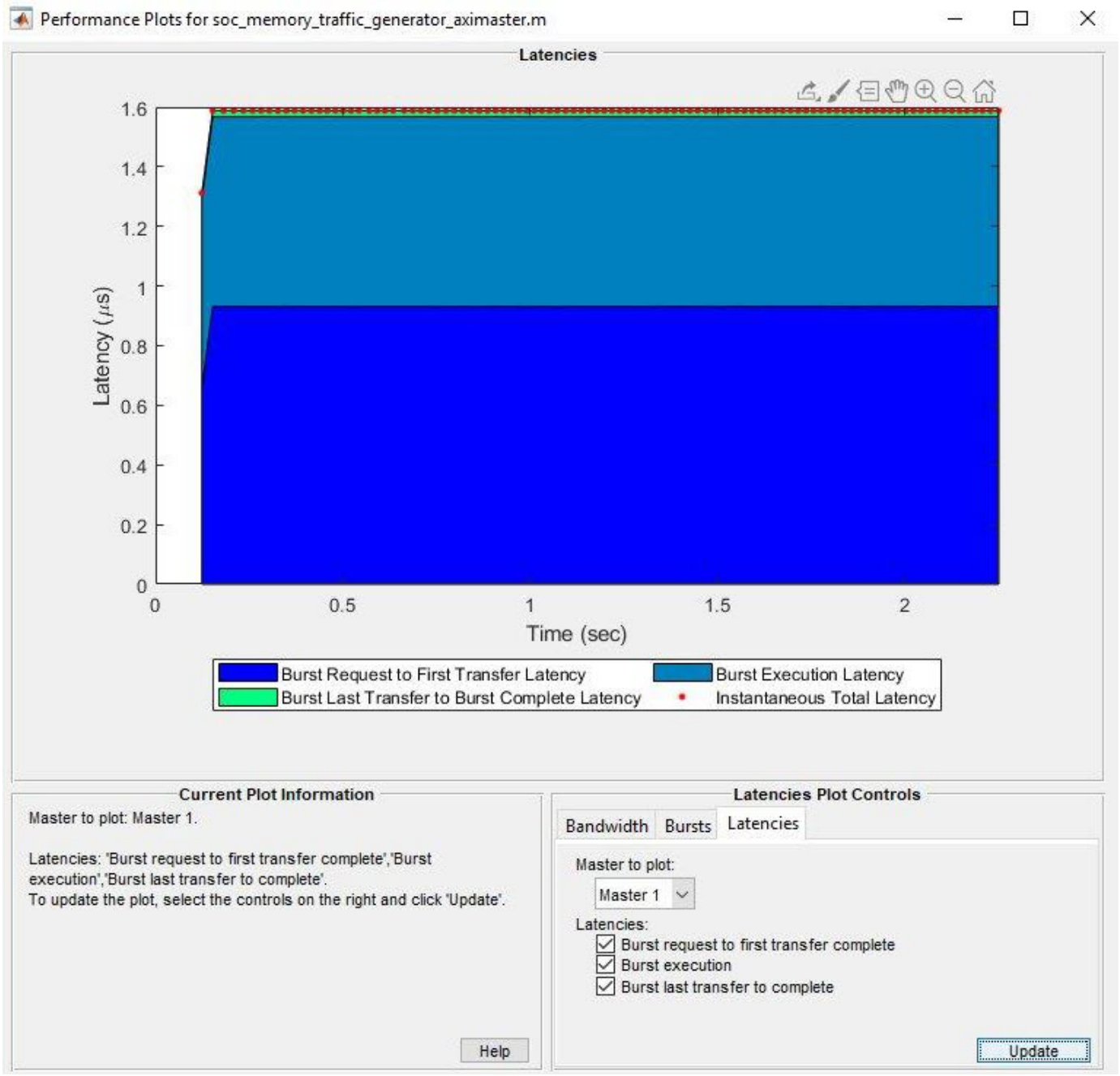
Memory Burst Plots

In the **Bursts** tab, select the masters for which you want to graph bursts. Click **Create Plot** to see the number of bursts executed for the selected master over the duration of the run time. This figure shows the burst count for the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example.



Memory Latency Plots

In the **Latencies** tab, select the master for which you want to graph latencies. Click **Create Plot** to see the latency, for the selected masters over the duration of the run time. This image shows the total latency for `Master 1` in the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example. You can then zoom in to analyze the peak instantaneous latency.



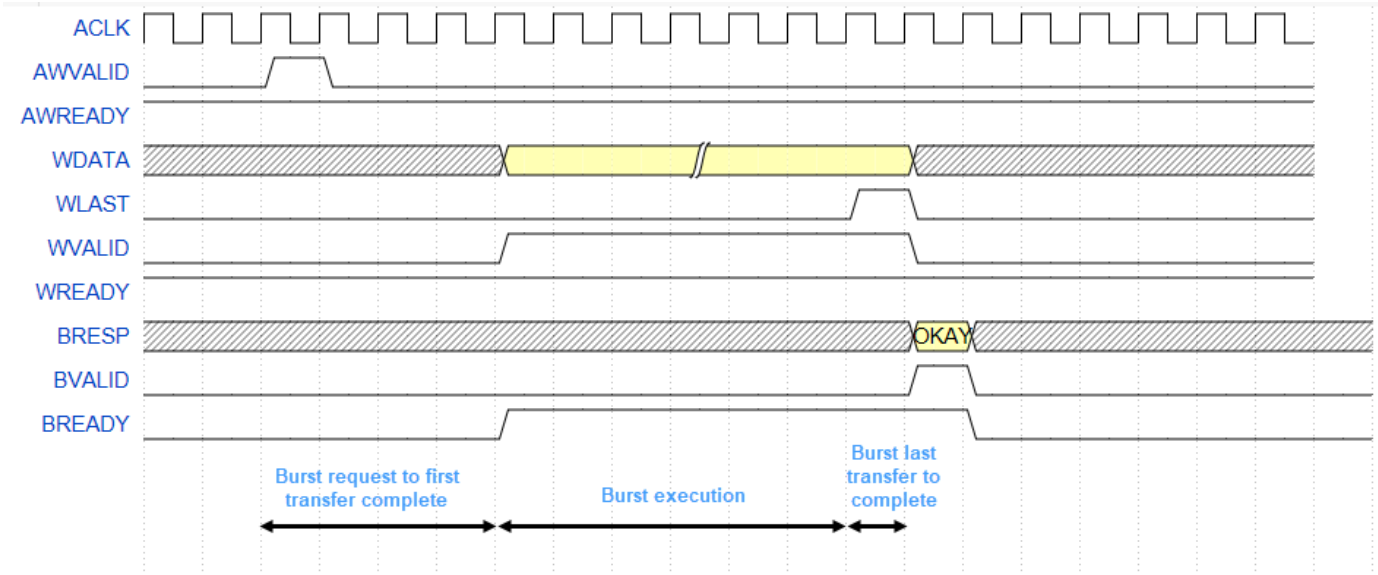
You can choose from any of these options:

- **Burst request to first transfer complete** — This option shows the time it takes from the moment the master issues a transaction request to the first transfer of data. This latency accounts for arbitration or interconnect delays.
- **Burst execution** — This option shows the time it takes from the first transfer of data to the burst last transfer.
- **Burst last transfer to complete** — This option shows the time it takes from last transfer to complete transaction. In case of read transaction, it is 0.

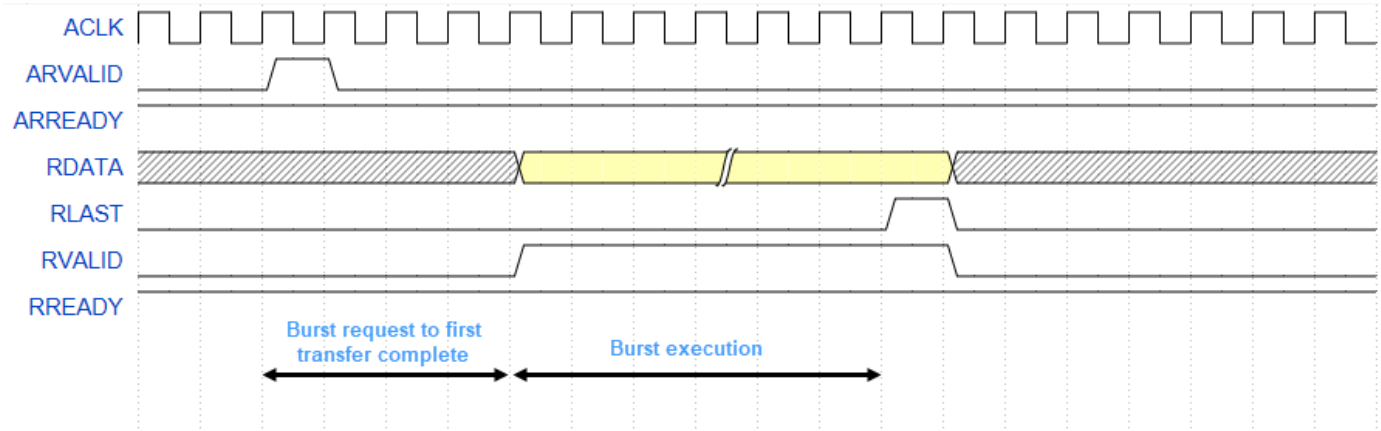
- **Instantaneous Total Latency** — This option shows discrete total latency measurements per burst.

Each latency value plotted is an average of the respective latency, measured from the memory transactions over a sampling interval. The following figure shows an AXI4 Master protocol write and read transaction on the hardware showing each of these latencies.

Write Transaction



Read Transaction



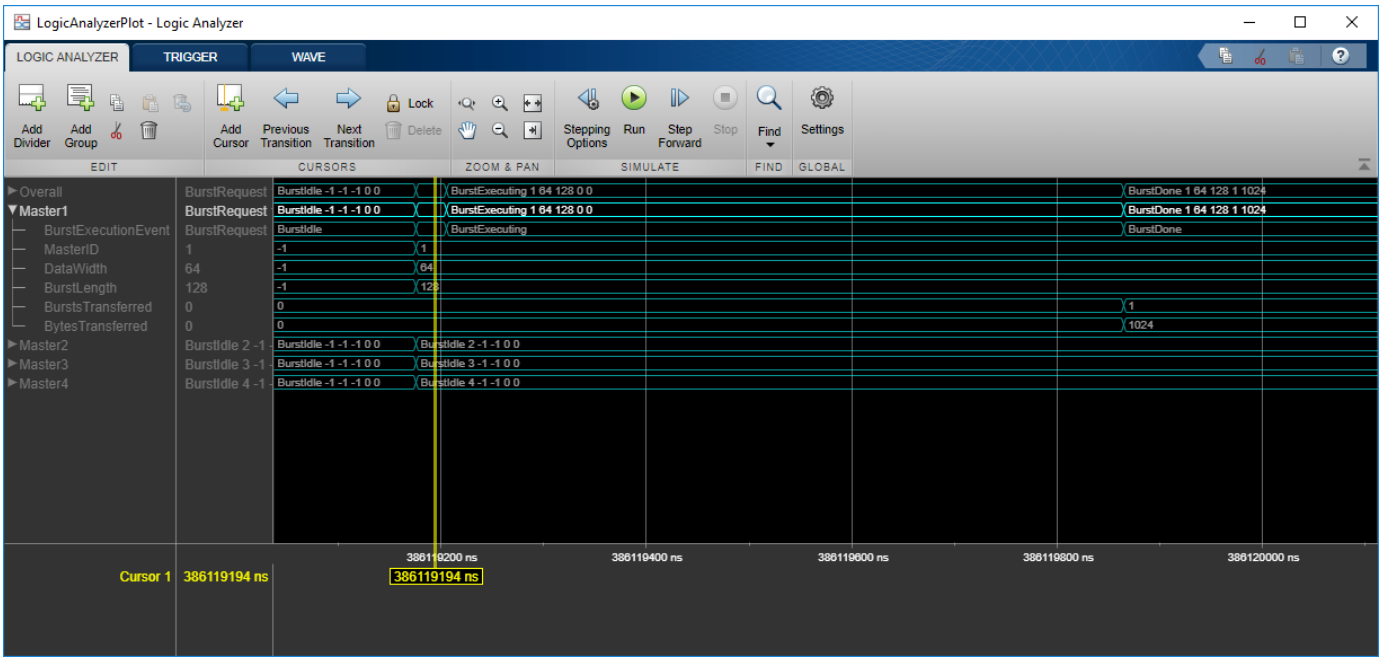
In read transaction, **Burst last transfer to complete latency** is zero.

Data Overflow

In `Profile` mode, the `collectMemoryStatistics` function samples memory metrics: bandwidth, burst, and latencies values from the hardware after every sample. After that, the function resets the metric counters and then starts the counters again for the next sample. If any of the metric counters exceeds the limit of $2^{32} - 1$ within the sampling interval, the counter is overflowed and the corresponding sample is indicated with * in the plot.

Burst Waveforms

You can also modify the generated script to configure the AXI interconnect monitor to collect event data for each burst transaction. You can view these events in the **Logic Analyzer** waveform viewer to examine arbitration behavior. Specify the number of transactions to capture, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.



The waveforms show the event type (`BurstIdle`, `BurstRequest`, `BurstExecuting`, or `BurstDone`) and these parameters of the burst transaction:

- `MasterID` –– ID number of the memory master that made the request
- `DataWidth` –– Data width in bits
- `BurstLength` –– Number of data words in the burst request
- `BurstsTransferred` –– Number of bursts in this request (valid only with `BurstDone` event)
- `BytesTransferred` –– Number of bytes in this request (valid only with `BurstDone` event)

You can compare these waveforms with the waveforms captured from your Memory Controller block in simulation.

Configuring and Querying the AXI Interconnect Monitor

The AXI interconnect monitor (AIM) is an IP core that collects performance metrics for an AXI-based FPGA design. Create an `socIPCore` object to set up and configure the AIM IP, and use the `socMemoryProfiler` object to retrieve and display the data.

For an example of how to configure and query the AIM IP in your design using MATLAB as AXI Master, see “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57. Specifically, review the `soc_memory_traffic_generator_axi_master.m` script that configures and monitors the design on the device.

Select Memory Monitor Mode

The AXI interconnect monitor can collect two types of data. Choose `Profile` mode to collect average transaction latency, and counts of bytes and bursts. In this mode, you can open a performance plot tool, and then configure the tool to plot bandwidth, burst count, and transaction latency. Choose `Trace` mode to collect detailed memory transaction event data and view the data as waveforms.

```
perfMonMode = 'Profile'; % or 'Trace'
```

Configure the AXI Interconnect Monitor

To obtain diagnostic performance metrics from your generated FPGA design, you must set up a JTAG connection to the device from MATLAB. Load a `.mat` file that contains structures derived from the board configuration parameters. This file was generated by the **SoC Builder** tool. These structures describe the memory interconnect and masters configuration such as buffer sizes and addresses. Use the `socHardwareBoard` object to set up the JTAG connection.

```
load('soc_memory_traffic_generator_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Configure the AIM. The `socIPCore` object provides a function that performs this initialization. Next, set up a `socMemoryProfiler` object to gather the metrics.

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',perfMonMode);
initialize(apmCoreObj);
profilerObj = socMemoryProfiler(hwObj,apmCoreObj);
```

Retrieve Diagnostic Data

To retrieve performance metrics or signal data from a design running on the FPGA, use the `socMemoryProfiler` object functions.

For `Profile` mode, call the `collectMemoryStatistics` function in a loop.

```
NumRuns = 100;
for n = 1:NumRuns
    collectMemoryStatistics(profilerObj);
end
```

JTAG design setup time is long relative to FPGA transaction times, and if you have a small number of transactions in your design, they can be completed by the time you query the monitor. In this case, the bandwidth plot shows only one sample, and the throughput calculation is not accurate. If this situation occurs, increase the total number of transactions the design executes.

For `Trace` mode, call the `collectMemoryStatistics` function once. This function stops the IP from writing transactions into the FIFO in the AXI interconnect monitor IP, although the transactions continue on the interconnect. Set the size of the transaction FIFO, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.

```
collectMemoryStatistics(profilerObj);
```

Visualizing Performance Metrics

Visualize the performance data using the `plotMemoryStatistics` function. In `Profile` mode, this function opens a performance plot tool, and you can configure the tool to plot bandwidth, burst count,

and average transaction latency. In `Trace` mode, this function opens the **Logic Analyzer** tool to view burst transaction event data.

```
plotMemoryStatistics(profilerObj);
```

See Also

Memory Controller | `socMemoryProfiler` | `collectMemoryStatistics` | `plotMemoryStatistics`

More About

- “Simulation Diagnostics” on page 5-15
- “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57

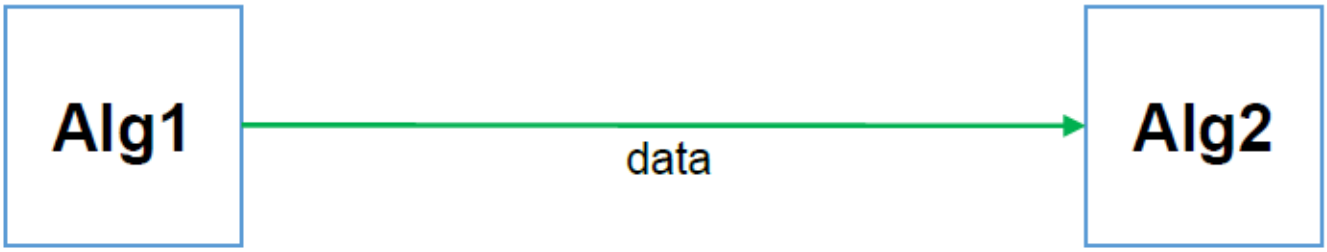
See Also

Memory

Memory and Register Data Transfers

An SoC application is composed of one or more algorithms. When an algorithm transfers data to another algorithm, the data is represented as a signal line in Simulink. For behavioral models, the data transfer is instantaneous.

This diagram shows a behavioral model of a datapath between two algorithms.



In the physical world, the algorithms can be on two separate devices, and data transfers do not happen instantaneously. Furthermore, the algorithms can run at different rates, and therefore require buffering and control logic for handshaking. For example, a simple handshake such as “data is valid” from the producer of the data and “ready to accept data” from the consumer serve as control logic.

If one processing element executes in an FPGA or ASIC, and the next processing element executes on an embedded processor, then a simple signal line represents more than just a complex hardware datapath. The data transfer also represents a processor interrupt handler, an operating system task scheduler, and a software driver stack.

In SoC Blockset, you model data transfers and handshake protocols through shared memory. Use a Memory Channel block for external memory or a Register Channel block for registers.

Modeling Datapath with Memory Channel Block

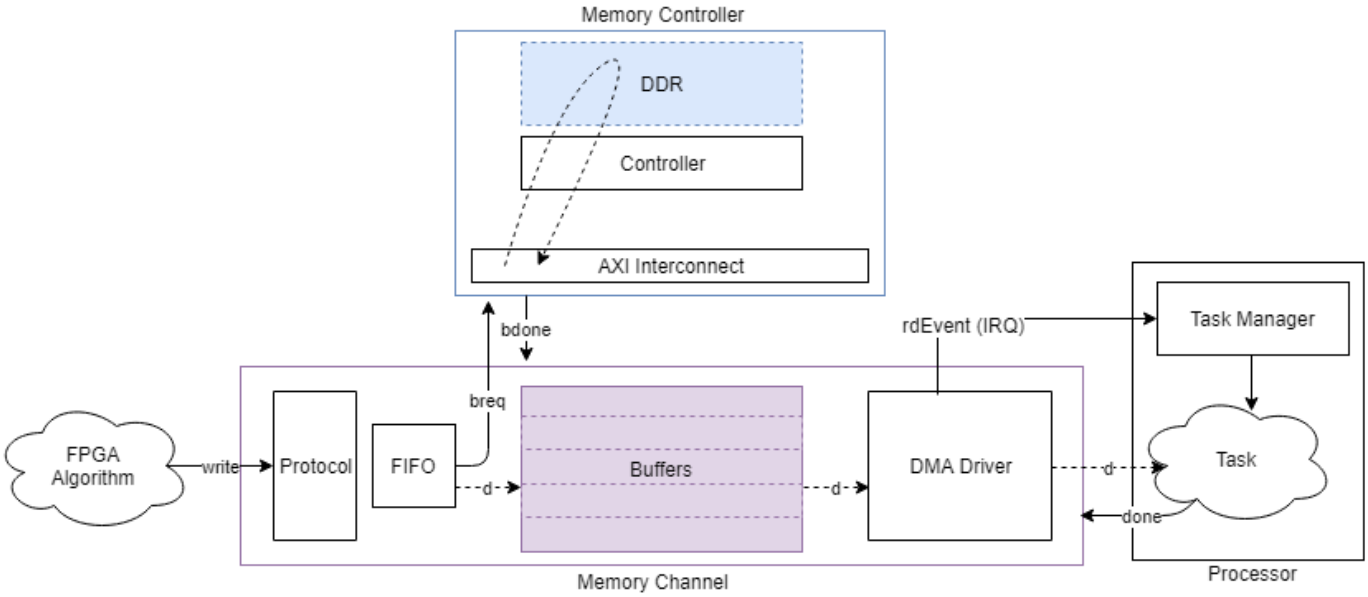
The Memory Channel block represents an abstraction to a complex datapath through external memory and supports different handshake protocols. It facilitates a refinement of the communication between processing elements from an instantaneous, protocol-less wire to a full direct memory access (DMA) connection between a processor and an FPGA.

By adding a Memory Channel block, you can model data movement from one part of the algorithm to another.

The block provides a model of the communication pipeline. The channel also provides a signaling interface.

The interface protocol depends on where the processing is executed. An FPGA or ASIC algorithm can perform data transfers by using standard protocols such as AXI4-Stream or AXI4. An embedded CPU algorithm can use a driver-interface exported to the user space.

This figure shows a model of the datapath from an FPGA algorithm streaming data to a processor algorithm.



Other Memory Channel type selections model additional common datapaths through external memory. For more information about Memory Channel configurations, see Memory Channel.

The writer and reader are connected to the memory and request access to the external memory from a memory controller. For more information about the Memory Controller block, see Memory Controller.

Modeling Datapath with Register Channel Block

The Register Channel block represents the serialization of the processor reads or writes through a common configuration bus such as AXI-Lite.

The Register Channel block provides a timing model for the transfer of register values between processor and hardware algorithms through a common configuration bus. Use this block when the processor writes a command or configuration register or when the processor reads a status register.

See Also

Memory Channel | Register Channel

More About

- “External Memory Channel Protocols” on page 5-5

External Memory Channel Protocols

The signal interfaces added to the channel model for the writer and reader are protocols that the algorithms use to communicate with the channel. Protocols do not change the core of the external memory channel model, which operates on burst transactions. They control only how the data gets in or out of those channels.

For FPGA or ASIC IPs, typical protocols include streaming data, streaming video data, and addressable data transfers. For software, typical protocols presented to an algorithm include simple data buffer, with details about interrupts, buffer management, and task scheduling left to the underlying OS.

Configure the Memory Channel block to support various protocols.

AXI4 Stream to Software via DMA

The AXI4-Stream Software configuration provides a software streaming protocol from hardware to software. Choose this configuration when a processor acts as a reader from the memory. This protocol includes a trigger configuration, which the Task Manager block receives. The trigger signals that a memory buffer is full and ready for reading. For more information about the AXI4-stream protocol, see “AXI4-Stream Interface” on page 5-7.

Software to AXI4-Stream via DMA

The Software to AXI4-Stream via DMA configuration provides a software streaming protocol from software to hardware. Choose this configuration when a processor acts as a writer to the memory. This protocol includes a trigger configuration, which the Task Manager block receives. The trigger signals that a memory buffer is empty and ready for writing. The processor then initiates a write transaction. Upon successful completion of the write transaction the processor receives a status signal from the Stream Write block. The processor reacts to that signal when the status is false. For more information about the AXI4-stream protocol, see “AXI4-Stream Interface” on page 5-7.

AXI4 Stream FIFO

The AXI4-Stream configuration provides a simple data valid and ready protocol for data streaming. You can generate a fully compliant AXI4-Stream interface from this protocol using HDL Coder.

For data stream channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The relationship of the stream to the managed buffers in the external memory is through an ‘end of buffer’ signal, known as `tlast` for AXI4-Stream. For more information about the AXI4-stream protocol, see “AXI4-Stream Interface” on page 5-7.

AXI4 Stream Video FIFO

The AXI4-Stream Video FIFO configuration provides a data valid and ready protocol similar to the AXI4 Stream FIFO. This protocol also has additional signaling to mark the start or the end of a video line and start or end of a video frame. This protocol is compatible with the HDMI Rx and HDMI Tx blocks, available with the SoC Blockset Support Package for Xilinx Devices. You can generate a fully compliant AXI-Stream video streaming interface from this protocol using HDL Coder. For information about the HDMI blocks, see documentation for SoC Blockset support packages.

For streaming video data channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The stream relates to the managed buffers in the external memory through the pixel control bus signals, which demarcate lines and frames. For more information, see “AXI4-Stream Video Interface” on page 5-12.

AXI4 Stream Video Frame Buffer

The AXI4-Stream Video Frame Buffer configuration provides The same signaling as the AXI4 Stream Video FIFO, with additional control signals for frame-buffer synchronization. This protocol is compatible with the HDMI Rx and HDMI Tx blocks, available with the SoC Blockset Support Package for Xilinx Devices. You can generate a fully compliant AXI-Stream video streaming interface from this protocol using HDL Coder. For information about the HDMI blocks, see documentation for SoC Blockset support packages.

For streaming video data channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The stream’s relationship to the managed buffers in the external memory is through the pixel control bus signals, which demarcate lines and frames.

AXI4 Random Access

The AXI4 configuration provides a simple, direct interface to the memory interconnect. Unlike the previous two streaming protocols, this protocol allows the algorithm to act as a memory master by providing the addresses and managing the burst transfer directly. This protocol represents a simplified master protocol. You can generate a fully compliant AXI-4 interface from this protocol using HDL Coder. For more information about the simplified AXI4 interface, see “Simplified AXI4 Master Interface” on page 5-9.

See Also

Memory Channel

More About

- “Simplified AXI4 Master Interface” on page 5-9
- “AXI4-Stream Interface” on page 5-7
- “AXI4-Stream Video Interface” on page 5-12

AXI4-Stream Interface

Using SoC Blockset, you can model a simplified, streaming protocol in your model. Use HDL Coder to generate AXI4-Stream interfaces in the IP core.

Simplified Streaming Protocol

When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement the following signals:

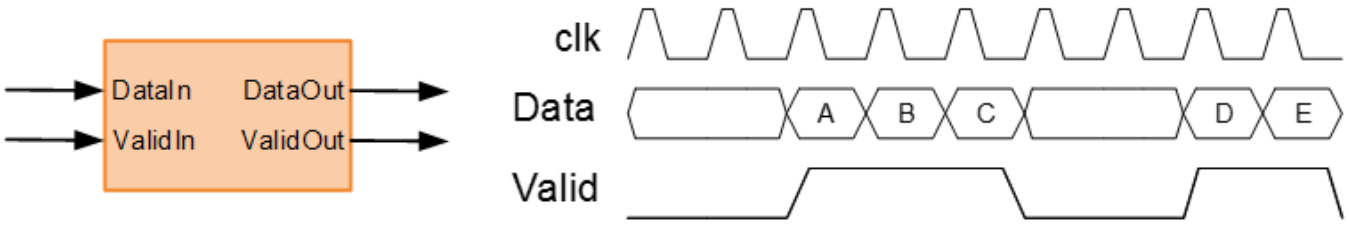
- Data
- Valid

When you map scalar DUT ports to an AXI4-Stream interface, you can optionally model the following signals and map them to the AXI4-Stream interface:

- Ready
- Other protocol signals, such as:
  - `TSTRB`
  - `TKEEP`
  - `TLAST`
  - `TID`
  - `TDEST`
  - `TUSER`

Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted.

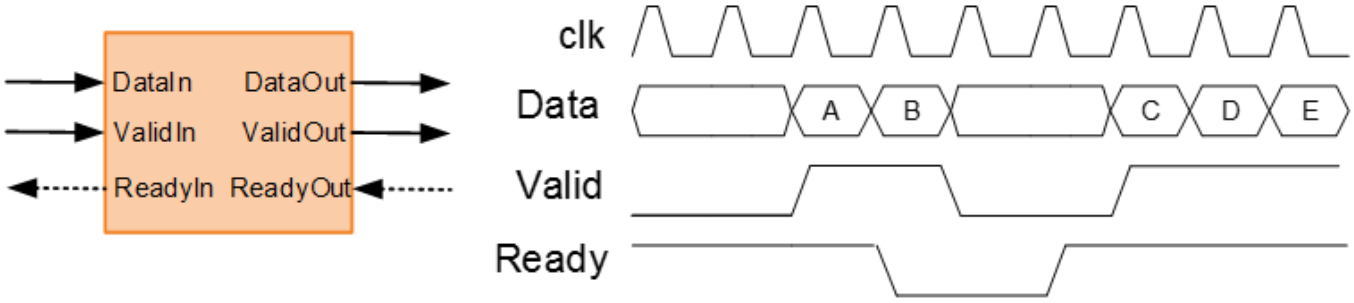


Ready Signal (Optional)

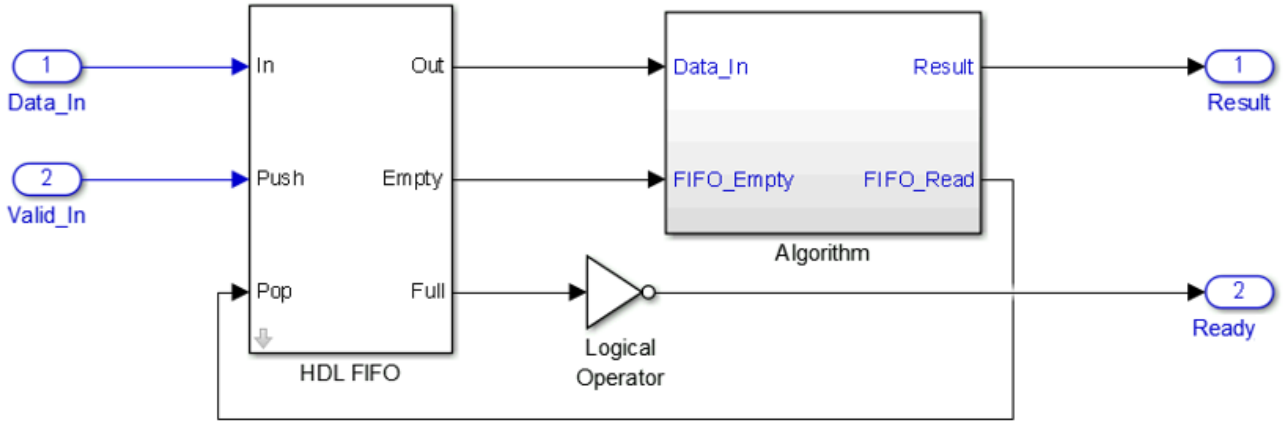
The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. In a Slave interface, the Ready signal enables you to apply back pressure. In a Master interface, the Ready signal enables you to respond to back pressure.

If you model the Ready signal in your AXI4-Stream interfaces, your Master interface ignores the Data and Valid signals one clock cycle after the Ready signal is deasserted. You can start sending Data and Valid signals once the Ready signal is asserted. You can send one more Data and Valid signal after the Ready signal is deasserted.

If you do not model the Ready signal, HDL Coder generates the signal and the associated back pressure logic.



For example, if you have a FIFO in your DUT to store a frame of data, to apply back pressure to the upstream component, you can model the Ready signal based on the FIFO Full signal.



See Also

Memory Channel | SoC Bus Creator

More About

- “External Memory Channel Protocols” on page 5-5
- “Simplified AXI4 Master Interface” on page 5-9
- “AXI4-Stream Video Interface” on page 5-12

Simplified AXI4 Master Interface

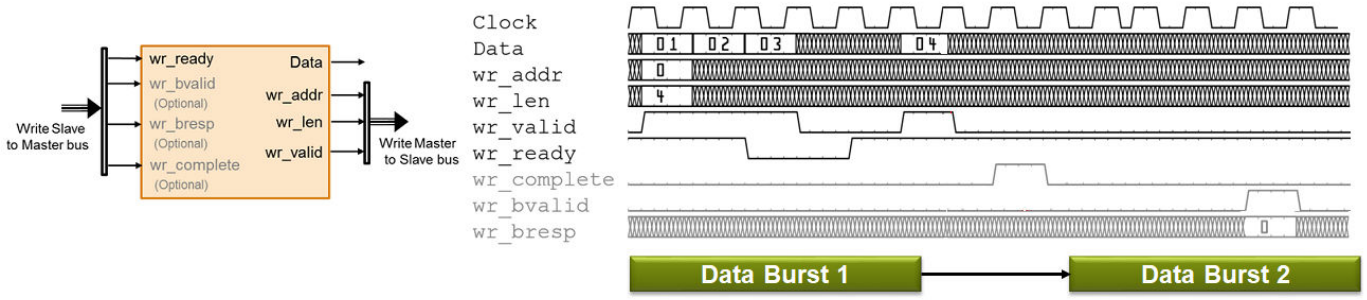
In this section...

For designs that require accessing large data sets from an external memory, model your algorithm with a simplified AXI4 Master protocol. When you run the `IP Core Generation` workflow, HDL Coder generates an IP core with AXI4 Master interfaces. The AXI4 Master interface can communicate between your design and the external memory controller IP by using the AXI4 Master protocol.

Simplified AXI4 Master Protocol - Write Channel

You can use the simplified AXI4 Master protocol to map to AXI4 Master interfaces. Use the simplified AXI4 Master write protocol for a write transaction and the simplified AXI4 Master read protocol for a read transaction.

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master write transaction.



The DUT waits for `wr_ready` to become high to initiate a write request. When `wr_ready` becomes high, the DUT can send out the write request. The write request consists of the `Data` and `Write Master to Slave bus` signals. This bus consists of `wr_len`, `wr_addr`, and `wr_valid`. `wr_addr` specifies the starting address that DUT wants to write to. The `wr_len` signal corresponds to the number of data elements in this write transaction. `Data` can be sent as long as `wr_valid` is high. When `wr_ready` becomes low, the DUT must stop sending data within one clock cycle, and the `Data` signal becomes invalid. If the DUT continues to send data after one clock cycle, the data is ignored.

The simplified AXI4 Master Protocol supports pipelined requests, so it is not required to wait for the `wr_complete` signal to be high before issuing a subsequent write request. The interface supports up to 16 transactions (or 16 data words) before the pipeline stalls and the `wr_ready` signal goes low.

Output Signals

Model the `Data` and `Write Master to Slave bus` signals at the DUT output interface.

- `Data`: The data that you want to transfer, valid each cycle of the transaction.
- `Write Master to Slave bus` that consists of:
  - `wr_addr`: Starting address of the write transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.

- `wr_len`: The number of data values that you want to transfer, sampled at the first cycle of the transaction. The `wr_len` signal is specified in words.
- `wr_valid`: When this control signal becomes high, it indicates that the `Data` signal sampled at the output is valid.

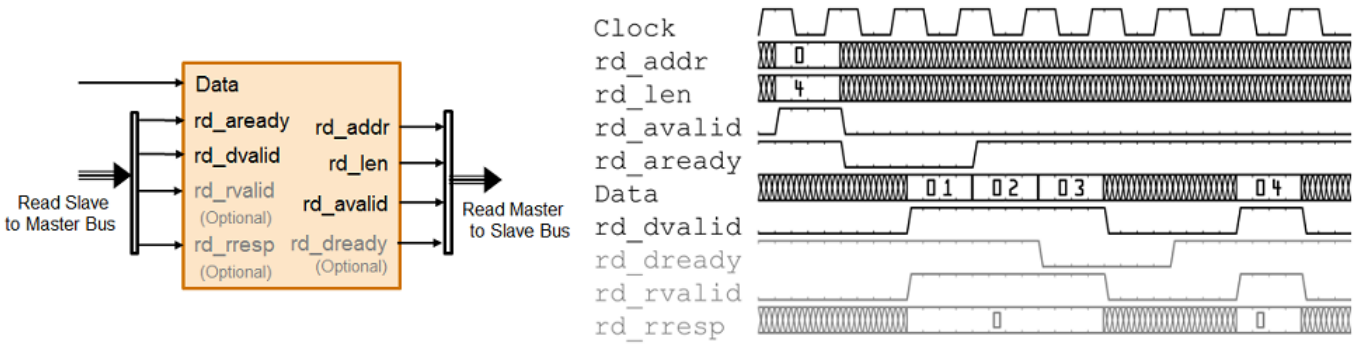
Input Signals

Model the `Write Slave to Master bus` that consists of:

- `wr_ready`: This signal corresponds to the backpressure from the slave IP core or external memory. When this control signal goes high, it indicates that data can be sent. When `wr_ready` is low, the DUT must stop sending data within one clock cycle. You can also use the `wr_ready` signal to determine whether the DUT can send a second burst signal immediately after the first burst signal has been sent. Multiple burst signals are supported, which means that the `wr_ready` signal remains high to accept the second burst immediately after the last element of the first burst has been accepted.
- `wr_bvalid` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. The `wr_bvalid` signal becomes high after the AXI4 interconnect accepts each burst transaction. If `wr_len` is greater than `256`, the AXI4 Master write module splits the large burst signal into 256-sized bursts. `wr_bvalid` becomes high for each 256-sized burst.
- `wr_bresp` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. Use this signal with the `wr_bvalid` signal.
- `wr_complete` (optional signal): Control signal that when remains high for one clock cycle indicates that the write transaction has completed. This signal asserts at the last `wr_bvalid` of the burst.

Simplified AXI4 Master Protocol - Read Channel

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master read transaction. These signals include the `Data`, `Read Master to Slave Bus`, and `Read Slave to Master Bus`.



The DUT waits for `rd_aready` to become high to initiate a read request. When `rd_aready` is high, the DUT can send out the read request. The read request consists of the `rd_addr`, `rd_len`, and `rd_avalid` signals of the `Read Master to Slave bus`. The slave IP or the external memory responds to the read request by sending the `Data` at each clock cycle. The `rd_len` signal corresponds to the number of data values to read. The DUT can receive `Data` as long as `rd_dvalid` is high.

Read Request

To model a read request, at the DUT output interface, model the `Read Master to Slave bus` that consists of:

- `rd_addr`: Starting address for the read transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.
- `rd_len`: The number of data values that you want to read, sampled at the first cycle of the transaction. The `rd_len` signal is specified in words.
- `rd_avalid`: Control signal that specifies whether the read request is valid.

At the DUT input interface, implement the `rd_aready` signal. This signal is part of the `Read Slave to Master bus` and indicates when to accept read requests. You can monitor the `rd_aready` signal to determine whether the DUT can send consecutive burst requests. When `rd_aready` becomes high, it indicates that the DUT can send a read request in the next clock cycle.

The simplified AXI4 Master Protocol supports pipelined requests, so it is not required to wait for the read response to complete before issuing a subsequent read request. The interface supports up to 4 read transactions before the pipeline stalls and the `rd_aready` signal goes low.

Read Response

At the DUT input interface, model the `Data` and `Read Slave to Master bus` signals.

- `Data`: The data that is returned from the read request.
- `Read Slave to Master bus` that consists of:
  - `rd_dvalid`: Control signal which indicates that the `Data` returned from the read request is valid.
  - `rd_rvalid` (optional signal): response signal from the slave IP core that you can use for diagnosis purposes.
  - `rd_rresp` (optional signal): Response signal from the slave IP core that indicates the status of the read transaction.

At the DUT output interface, you can optionally implement the `rd_dready` signal. This signal is part of the `Read Master to Slave bus` and indicates when the DUT can start accepting data. By default, if you do not map this signal to the AXI4 Master read interface, the generated HDL IP core ties `rd_dready` to logic high.

See Also

Memory Channel | SoC Bus Creator

More About

- “External Memory Channel Protocols” on page 5-5
- “AXI4-Stream Interface” on page 5-7
- “AXI4-Stream Video Interface” on page 5-12

AXI4-Stream Video Interface

In this section...

Using SoC Blockset, you can implement a simplified, streaming pixel protocol in your model. Use HDL Coder to generate an HDL IP core with AXI4-Stream Video interfaces.

Streaming Pixel Protocol

You can use the streaming pixel protocol for AXI4-Stream Video interface mapping. Video algorithms process data serially and generate video data as a serial stream of pixel data and control signals.

To generate an IP core with AXI4-Stream Video interfaces, in your DUT interface, implement these signals:

- Pixel Data
- Pixel Control Bus

The **Pixel Control Bus** is a bus that has these signals:

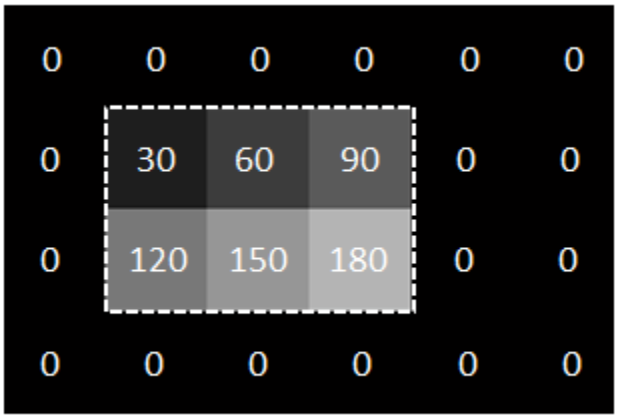
- `hStart`
- `hEnd`
- `vStart`
- `vEnd`
- `valid`

The signals **hStart** and **hEnd** represent the start of an active line and the end of an active line respectively. The signals **vStart** and **vEnd** represent the start of a frame and the end of a frame.

You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

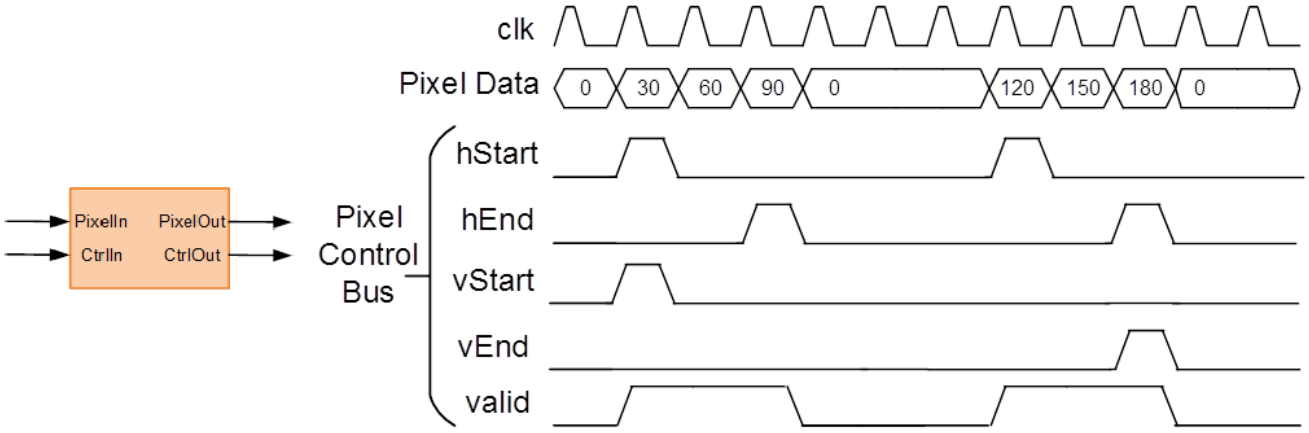
Protocol Signals and Timing Diagrams

This figure is a 2–by–3 pixel image. The active image area is the rectangle with a dashed line around it and the inactive pixels that surround it. The pixels are labeled with their grayscale values.



Pixel Data and Pixel Control Bus

This figure shows the timing diagram for the **Pixel Data** and **Pixel Control Bus** signals that you model at the DUT interface.



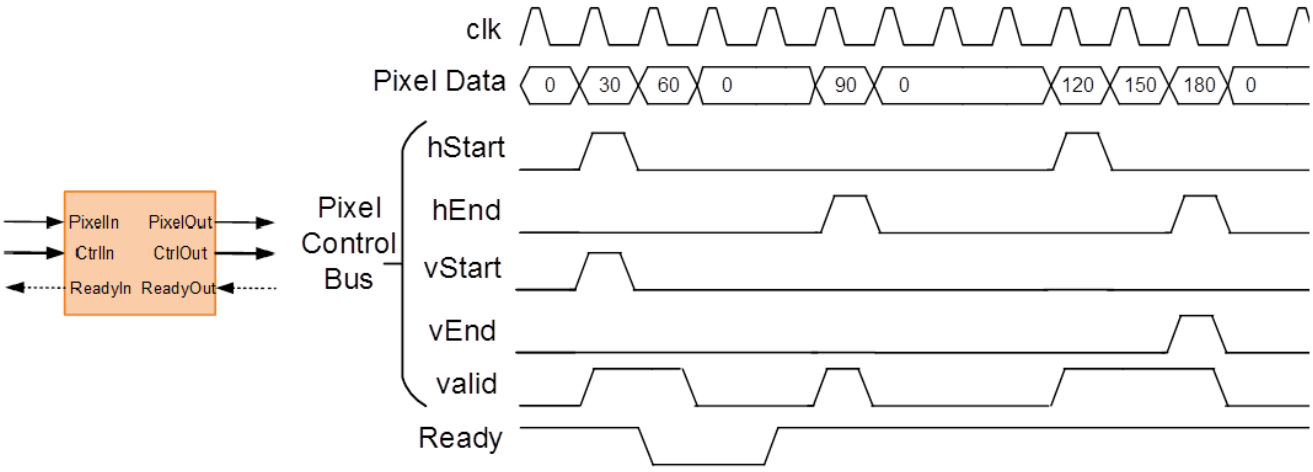
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

The **hStart** signal becomes high at the start of the active lines. The **hEnd** signal becomes high at the end of the active lines.

The **vStart** signal becomes high at the start of the active frame in the second line. The **vEnd** signal becomes high at the end of the active frame in the third line.

Optional Ready Signal

This figure shows the timing diagram for the **Pixel Data**, the **Pixel Control Bus**, and the **Ready** signal that you model at the DUT interface.



When you map the DUT ports to an AXI4-Stream Video interface, you can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

If you model the **Ready** signal in your AXI4-Stream Video interfaces, your Master interface must deassert its **valid** signal one cycle after the **Ready** signal is deasserted.

If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

See Also

Memory Channel | SoC Bus Creator

More About

- “External Memory Channel Protocols” on page 5-5
- “Simplified AXI4 Master Interface” on page 5-9
- “AXI4-Stream Interface” on page 5-7

Simulation Diagnostics

SoC Blockset enables simulation and evaluation of memory transactions in Simulink without the need to deploy a model to an SoC device. Use this diagnostic information to analyze the performance of your models, and adjust as needed to meet the desired system performance requirements. The simulation generates two types of visualization of the memory traffic:

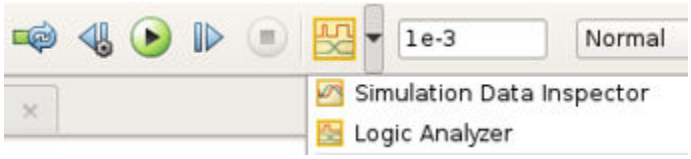
- "Simulation Performance Plots" on page 5-19 – Provides high level performance diagnostics of the model's memory system. Memory bandwidth, burst counts, and transaction latencies are calculated from a simulation of your model. You can view this information for each memory master in your model, or an overall view from the memory controller.
- "Buffer and Burst Waveforms" on page 5-15 – Provides burst transaction debug information from simulation, including the use of buffer regions.

You can also capture actual bandwidth, number of bursts, and latency measurements from the design running on the FPGA, and view information about individual burst transactions. This information is captured by including an AXI interconnect monitor IP in the FPGA design, and querying the data over a JTAG AXI master connection from the host. See "Memory Performance Information from FPGA Execution" on page 4-10.

Buffer and Burst Waveforms

SoC Blockset enables logging simulation signals, and visualizing the logged signals using the *Logic Analyzer*. To enable signal logging, Set **Memory diagnostics level** to `Basic diagnostic signals` in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.

After simulating your model, locate the **Logic Analyzer** at the top of your Simulink window.



The **Logic Analyzer** tool provides visualization of signal waveforms to show timing of various events of the memory model.

The **Logic Analyzer** displays signals from the Memory Controller and from the Memory Channel blocks.

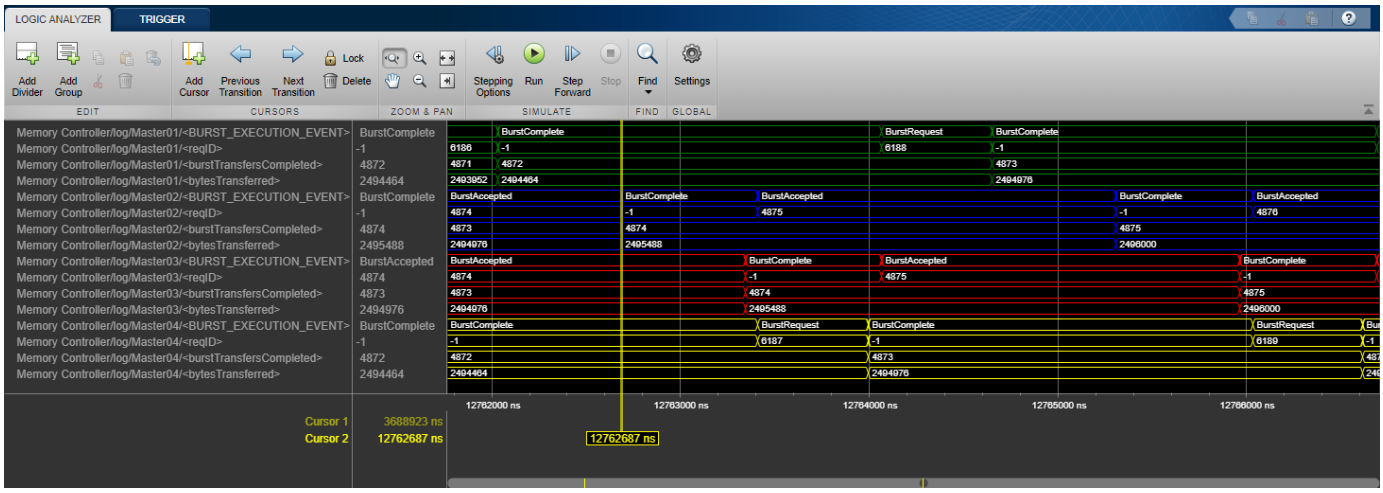
- **Burst Waveforms**

  Waveforms from the memory controller include information for bursts from the masters in the system. The waveforms are color coded to differentiate the different masters. These waveforms give insight into the sequencing of each of the masters through the shared memory. For each master, view the following signals:

  - `BURST_EXECUTION_EVENT`: State of the current burst request. Valid states are: none (idle), request, executing, done. For more information about the memory controller state, see Memory Controller.
  - `ReqID`: Identifier of the current burst request. An incrementing number that is unique throughout simulation.

- `burstTransfersCompleted`: A running count of transferred bursts. If no bursts are dropped within the memory channel, the count of transferred bursts matches `ReqID`. If bursts are dropped, `ReqID` becomes larger than this count.
- `BytesTransferred`: A running count of transferred bytes.

The following figure shows the signals after simulating “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57.



The waveforms include burst information for the four masters, displayed in different colors. This information correlates to the “Memory Controller Latency Plots” on page 5-22.

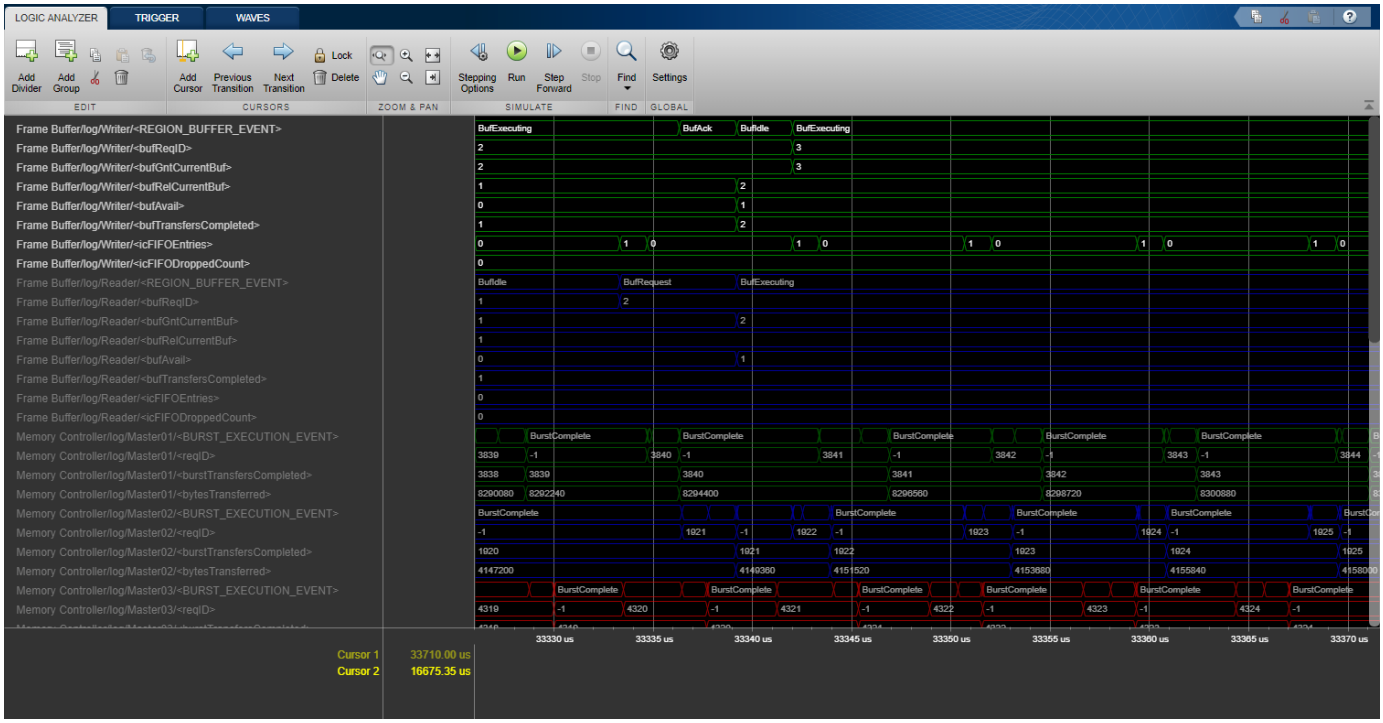
- **Buffer Waveforms**

  Waveforms from the memory channel include information for buffer read and write transactions in the channel. Each memory region is divided into several buffers specified by the **Number of buffers** parameter of the Memory Channel block. The writer fills the buffers, and the reader empties them. These waveforms give insight into the sequencing of the writer and reader for a given region. The buffer waveforms include the following signals:

  - `REGION_BUFFER_EVENT`: State of the current buffer request. Valid states are: none (idle), request, executing, done. For more information about the state of the memory channel, see Memory Channel.
  - `BufReqID`: Identifier of the current buffer request. An incrementing number that is unique throughout simulation.
  - `BufferAddress`: Starting address offset of the current buffer. The buffer address repeats as the simulation cycles through the buffers, reflecting the address boundaries of the buffers.
  - `BufGntCurrentBuf`: The currently active buffer specified from 1 to the number of buffers in the channel. `BufGntCurrentBuf` points to the buffer being written to (on the writer side), or the buffer being read from (on the reader side).
  - `BufRelCurrentBuf`: The buffer currently released by the reader or writer specified from 1 to the number of buffers in the channel. On the reader side, when a buffer is released it is available to the writer for writing. On the writer side, when a buffer is released it is available to the reader for reading.
  - `BufAvail`: The number of buffers currently available to the reader for reading. This value is identical on the reader and the writer side.

- `BufTransfersCompleted`: A running count of transferred buffers. If no buffers are dropped within the memory region, the count of transferred buffers matches `BufReqID`. If buffers are dropped, `BufReqID` is larger than this count.
- `icFIFOEntries`: Number of bursts written to the interconnect FIFO.
- `icFIFODroppedCount`: Number of bursts dropped from the interconnect FIFO.

- The following figure shows the buffer signals after simulating “Histogram Equalization Using Video Frame Buffer” on page 7-21.



You can relate the memory model operation with the protocol interface to understand the performance of your model. The following figure shows how to relate the memory model operation with the protocol interface.

1. Backpressure from rdReady
2. Reader cannot finish buffer 2.
3. Writer's FIFO begins filling up.
4. Writer is blocked and must assert back-pressure upstream.
5. Reader gets to finish buffer. Everyone starts moving again.

See Also

Logic Analyzer | Memory Channel | Memory Controller

More About

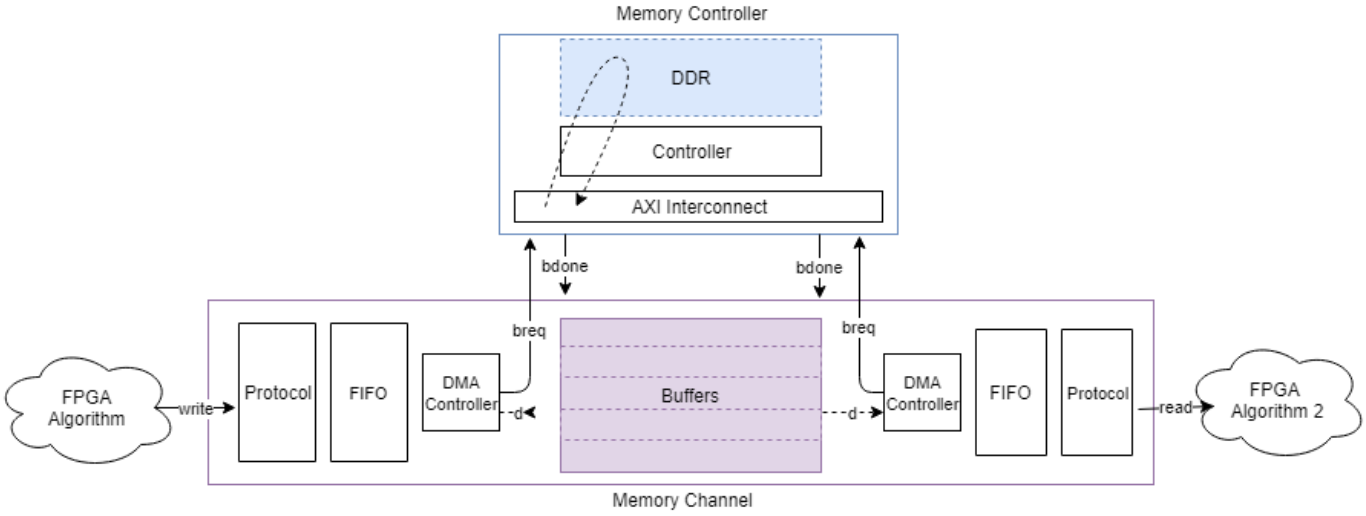
- "Simulation Performance Plots" on page 5-19

Simulation Performance Plots

SoC Blockset enables post-simulation analysis of memory diagnostic data. These plots provide high-level performance diagnostics of the memory system of the model. These plots are calculated measurements from a simulation of your model. It considers the data type, sample time, and clock frequency to calculate the bandwidth of your memory model and considers the number of bursts executed per memory port.

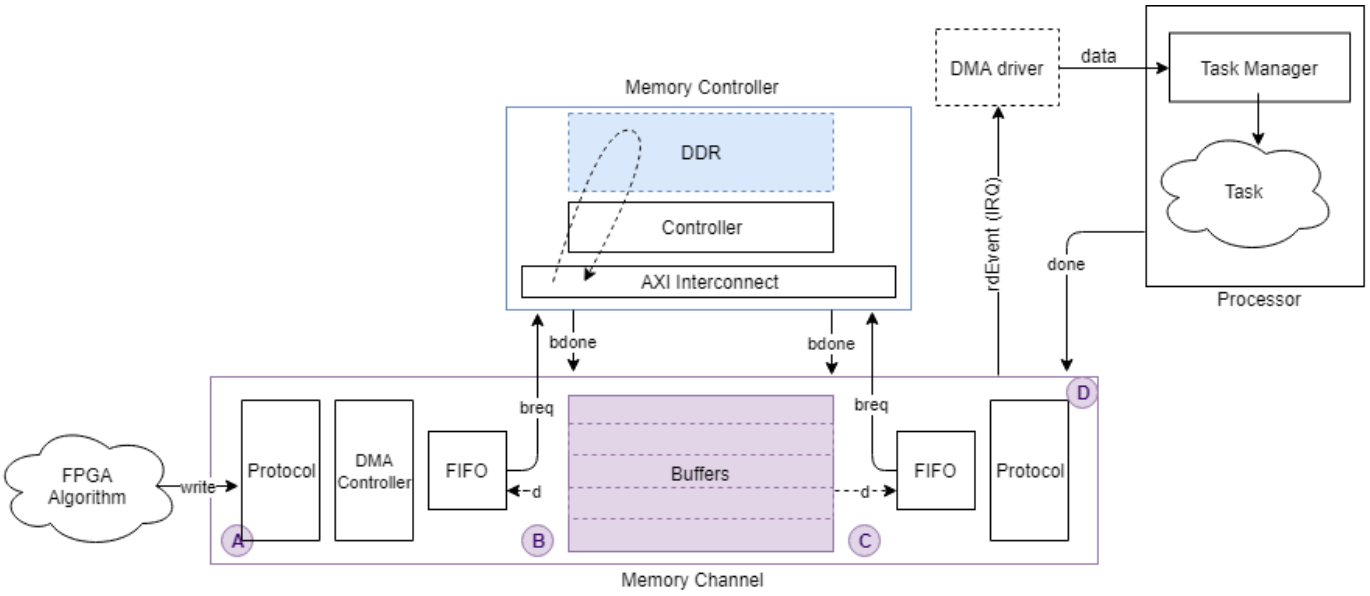
To enable signal logging in simulation, select **Hardware Implementation** on the Configuration Parameters dialog box. Under **Hardware Board Settings > Target Hardware Resources > FPGA design (debug)**, select the desired **Memory channel diagnostic level**.

This figure shows the datapath from one FPGA algorithm to another FPGA algorithm through a memory channel.



You can view channel latency plots for the datapath (represented by A, B, C, and D in the image) from the Memory Channel block mask. You can view memory bandwidth, burst count, and control-latency measurements (represented by 1, 2, 3, and 4 in the image) from the Memory Controller block mask.

The datapath from an FPGA algorithm to a processor is served through a DMA driver and a task processor and is illustrated in this image.



Memory Channel Latency Plots

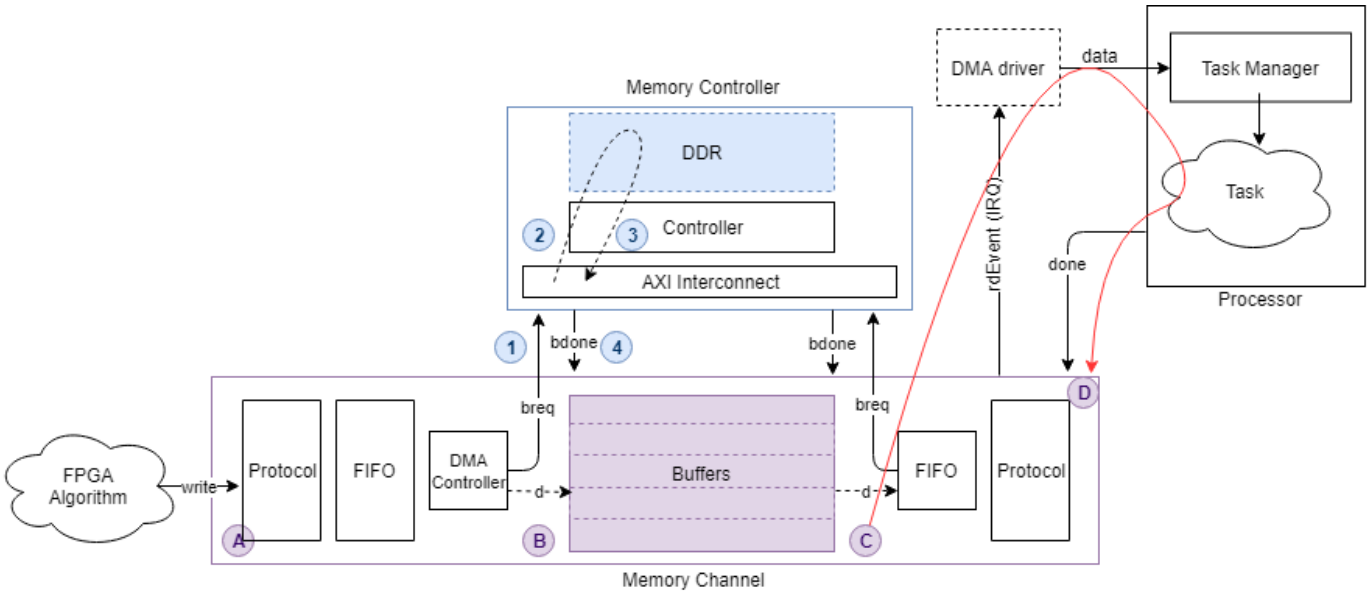
Memory Channel latency information is available post simulation per channel. After simulating your model, open the Memory Channel block mask. On the **Performance** tab, click **Launch performance plots**. This action opens a new window with several control options to display these different latencies:

- **Buffer write complete** – This option shows the time it takes between issuing a write request to when the buffer is fully written. It is the path between A and B in the figure.
- **Buffer read complete** – This option shows the time it takes between issuing a read request to when the buffer is read and is available again for writing. It is the path between C and D in the figure. This option is only available if the reader is an FPGA algorithm (not a processor algorithm). If the reader is a processor algorithm, this time shows as zero.
- **Buffer task execution complete** – This option shows the time it takes between issuing a read request to when the buffer is read and is available again for writing. It is the path between C and D in the figure. This option is only available if the reader is a processor algorithm (not an FPGA algorithm). If the reader is an FPGA algorithm, this time shows as zero.

  The **Buffer task execution complete** shows the time it takes for these events to occur:

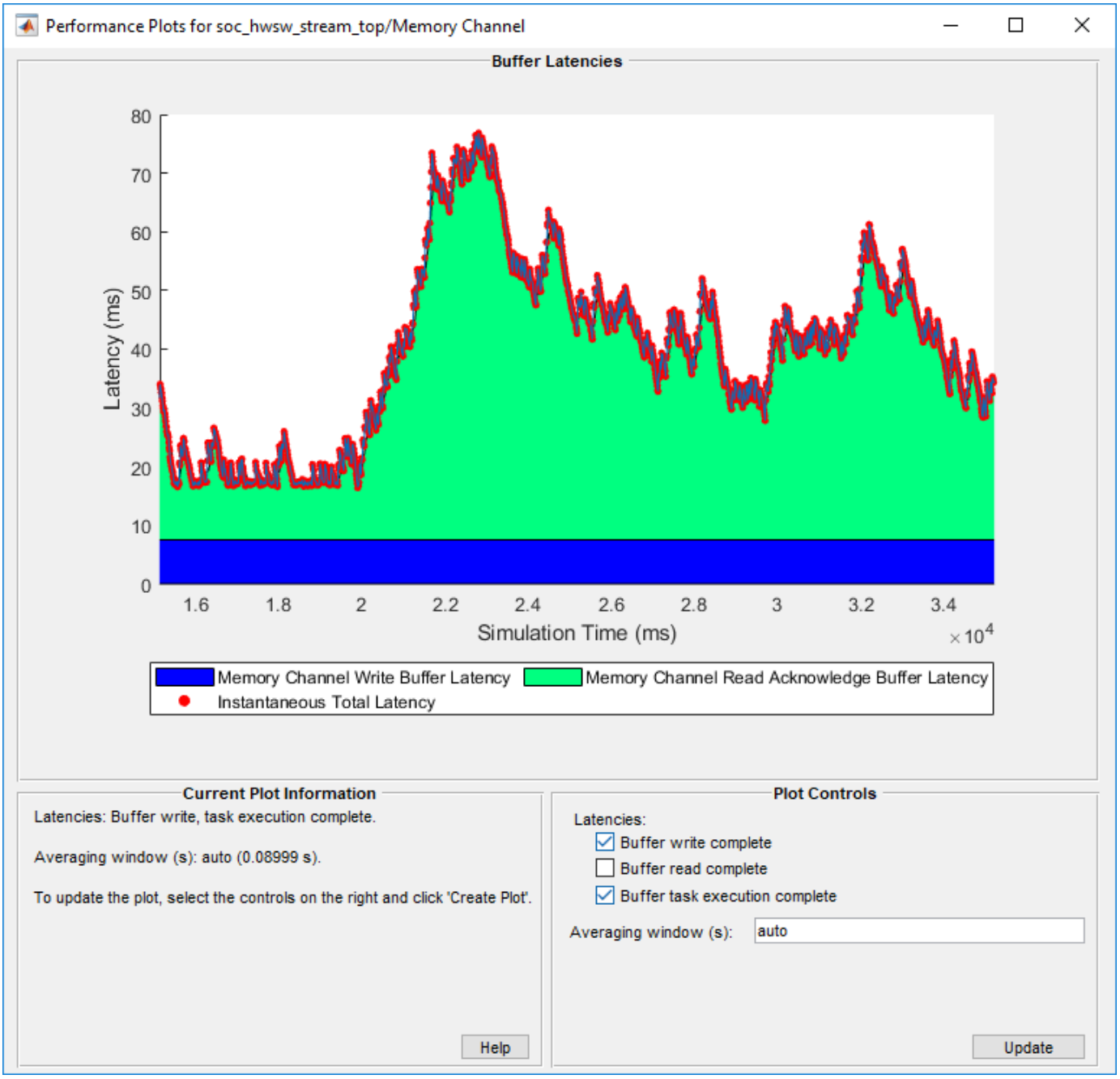
  1. The write buffer is full.
  2. The channel issued an interrupt request (IRQ) to the processor.
  3. An interrupt service routine (ISR) is executed.
  4. A task is scheduled.
  5. The task started executing.
  6. The task read data.
  7. The task optionally processed the data.
  8. The task sends a `done` signal back to the channel.

This following figure shows the latency path for a task execution to complete, as a red arrow from C to D.



- **Averaging Window (s)** – Specify a time, in seconds, for the averaging window width. The plot is graphed as a moving average, using a time window with the width specified. You can also specify `min`, `max`, or `auto`.
  - `min` – Use this value to see data without any averaging. The total latency graph is aligned with the **Instantaneous Total Latency** marks.
  - `max` – Use this value to see the overall average for the entire simulation.
  - `auto` – Use this value to see averaging over the number of buffers in your channel.
- **Instantaneous Total Latency** – This shows discrete total latency measurements per buffer.

If you add **Buffer write complete** to **Buffer read complete** or **Buffer task execution complete**, the plot displays the full latency from writer to reader. This image shows the total latency plot for the “Streaming Data from Hardware to Software” on page 7-31 example.

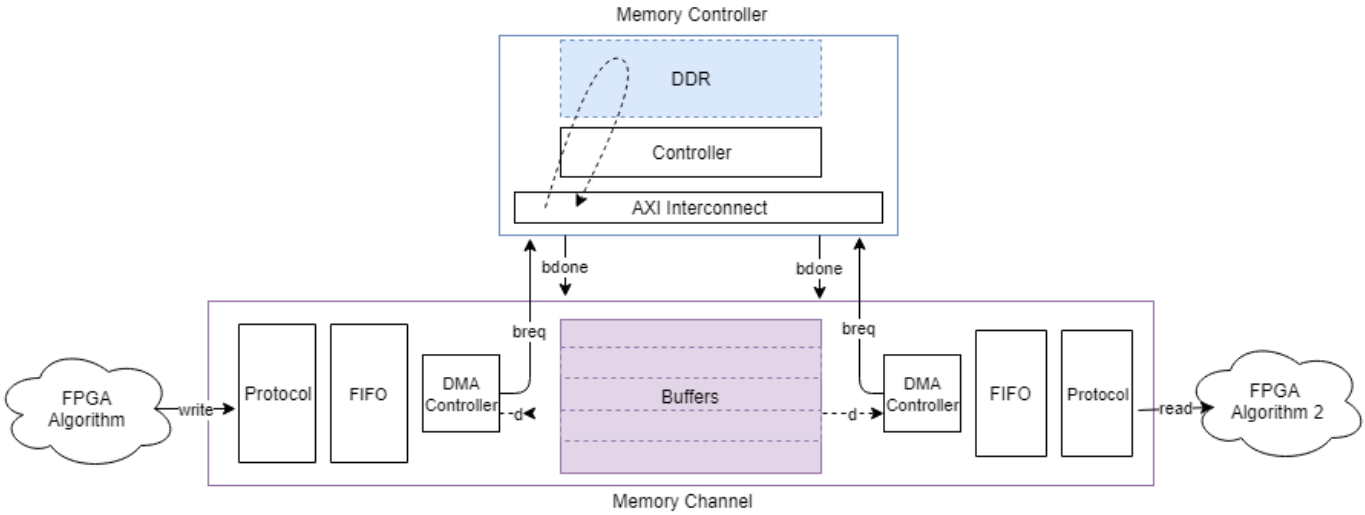


Note that the latencies are showing over an averaging window of one second. The instantaneous total latency shows a peak in latency as 76.8267 ms. Use this information to verify the model against the requirements.

Memory Controller Latency Plots

Memory Controller latency information is available post simulation. After simulating your model, open the Memory Controller block mask. On the **Performance** tab, click **Launch performance plots**. This action opens a new window with several control options to display performance metrics.

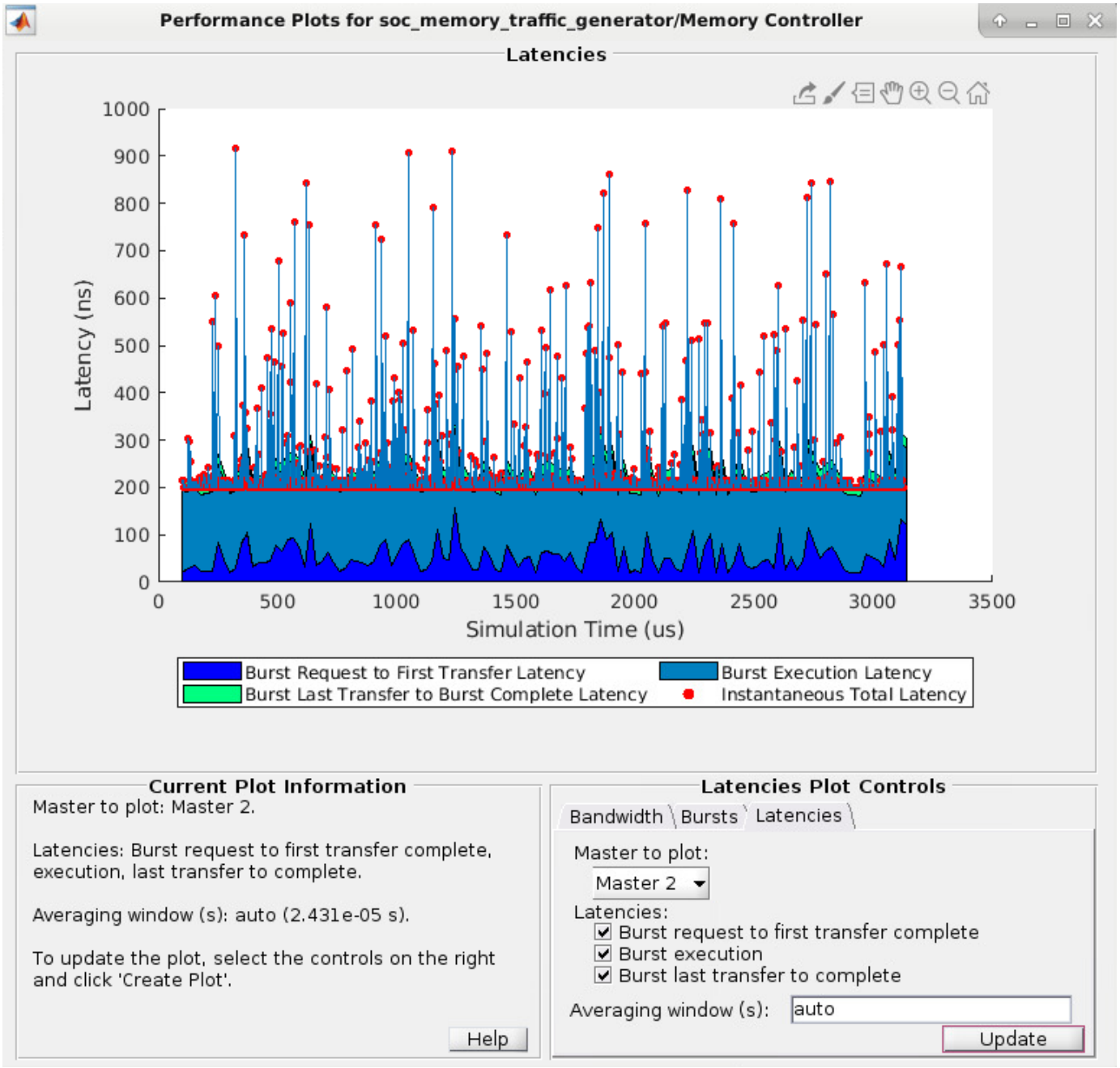
This figure shows the datapath from one FPGA algorithm to another FPGA algorithm through a memory channel.



In the **Latencies** tab, select the master for which you want to graph latencies. Choose from any of these options:

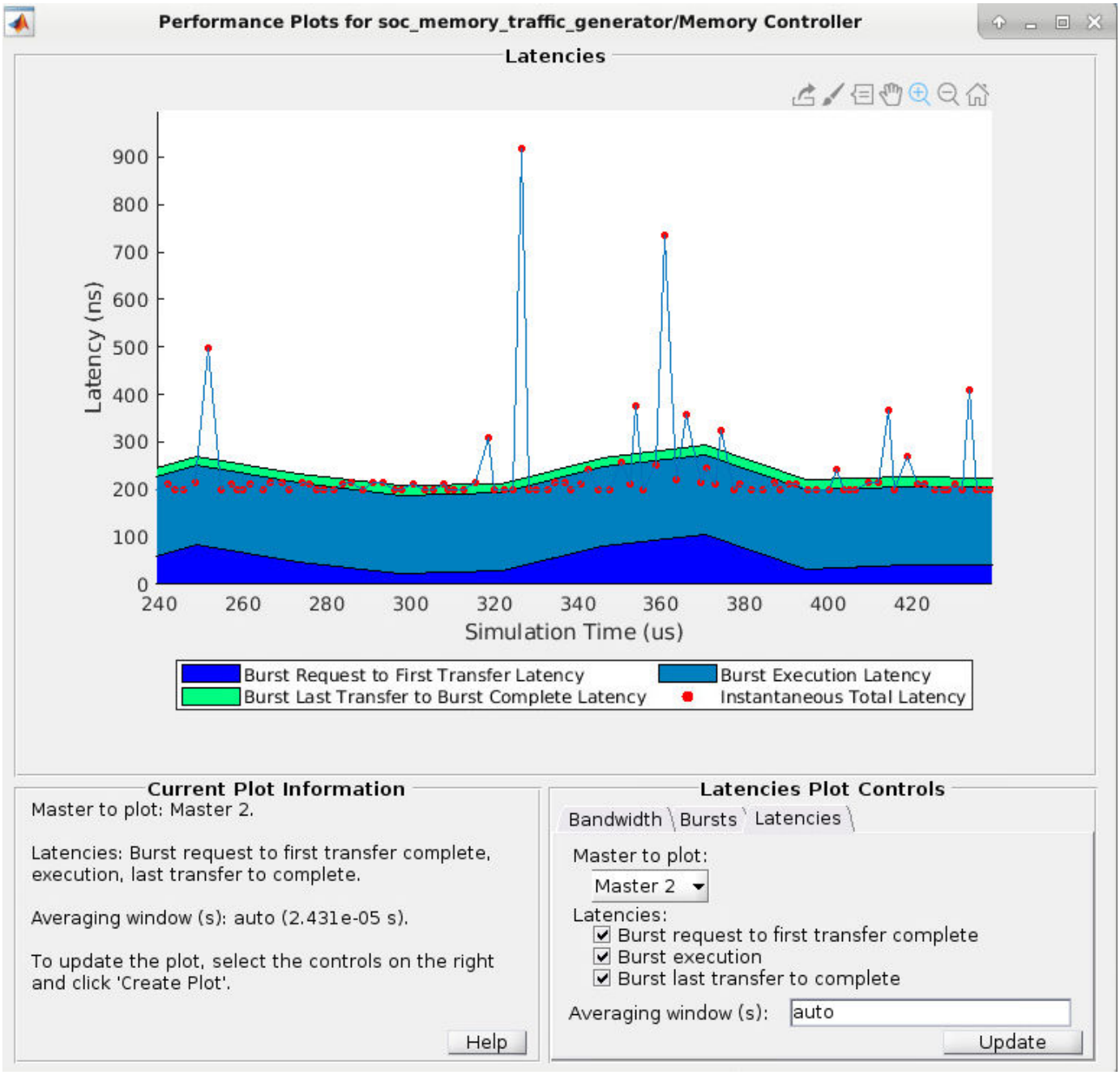
- **Burst request to first transfer complete** – This option shows the time it takes from the moment the Memory Channel block issues a burst-write request to the first transfer of data. This latency accounts for arbitration or interconnect delays. It is the path between 1 and 2 in the figure.
- **Burst execution latency** – This option shows the time it takes from the first transfer of data to when a burst is written to memory. It is the path between 2 and 3 in the figure.
- **Burst last transfer to complete latency** – This option shows the time it takes from the moment a burst completes to when the Memory Controller block issues a `burst-done` signal to the Memory Channel block. It is the path between 3 and 4 in the figure.
- **Averaging Window (s)** – Specify a time, in seconds, for the averaging window width. The plot is graphed as a moving average, using a time window with the width specified. You can also specify `min`, `max`, or `auto`.
  - `min` – Use this value to see data without any averaging. The total latency graph is aligned with the **Instantaneous Total Latency** marks.
  - `max` – Use this value to see the overall average for the entire simulation.
  - `auto` – Use this value to see averaging over 1% of the bursts during the simulation.
- **Instantaneous Total Latency** – This option shows discrete total latency measurements per burst.

Click **Create Plot** to see the latency, for the selected masters over the duration of the simulation time. This image shows the total latency for `Master 2` in the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example.



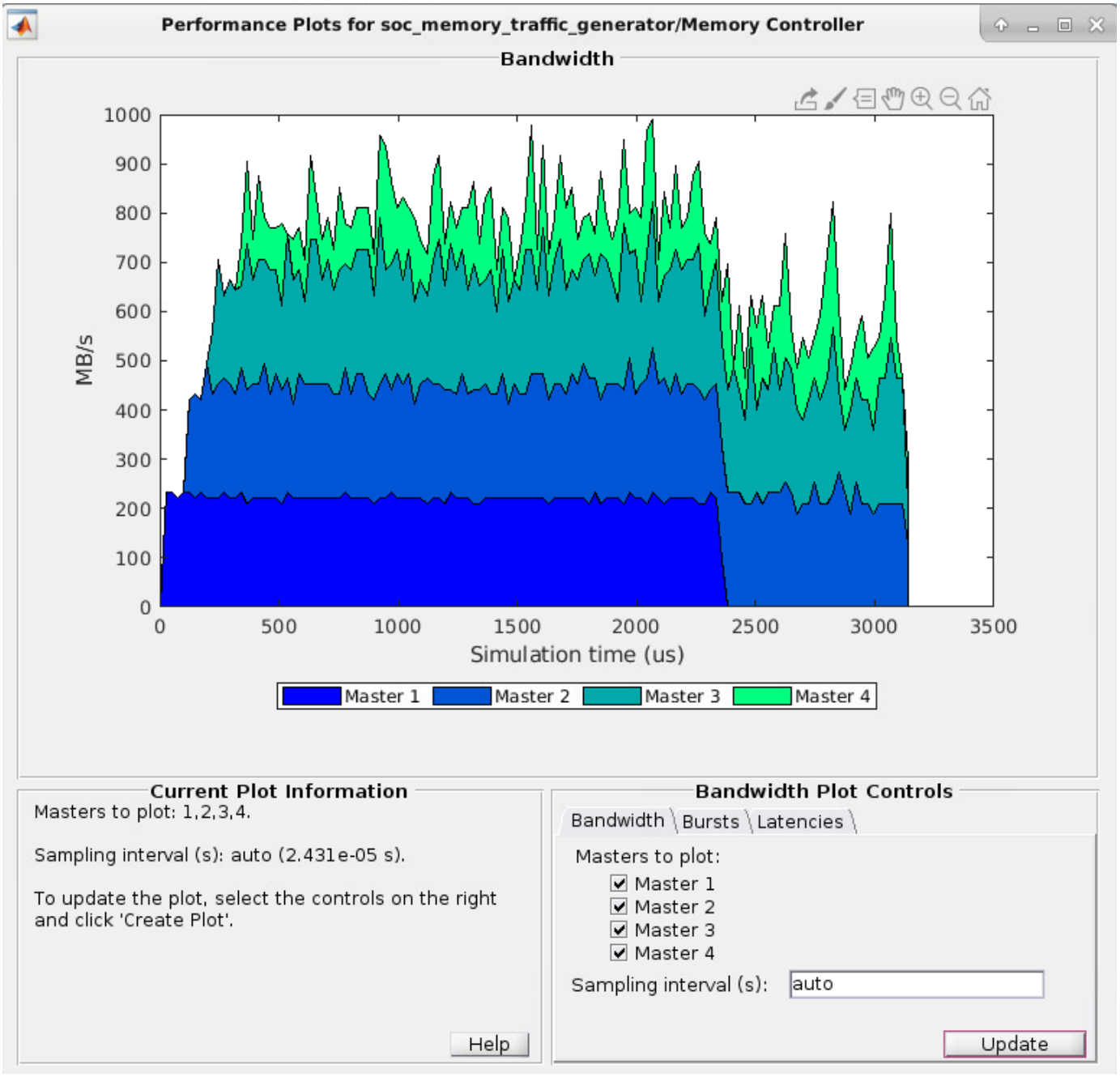
Note Memory controller latency plots are not available when the master is a processor.

You can then zoom in to analyze the peak instantaneous latency:



Memory Bandwidth Plots

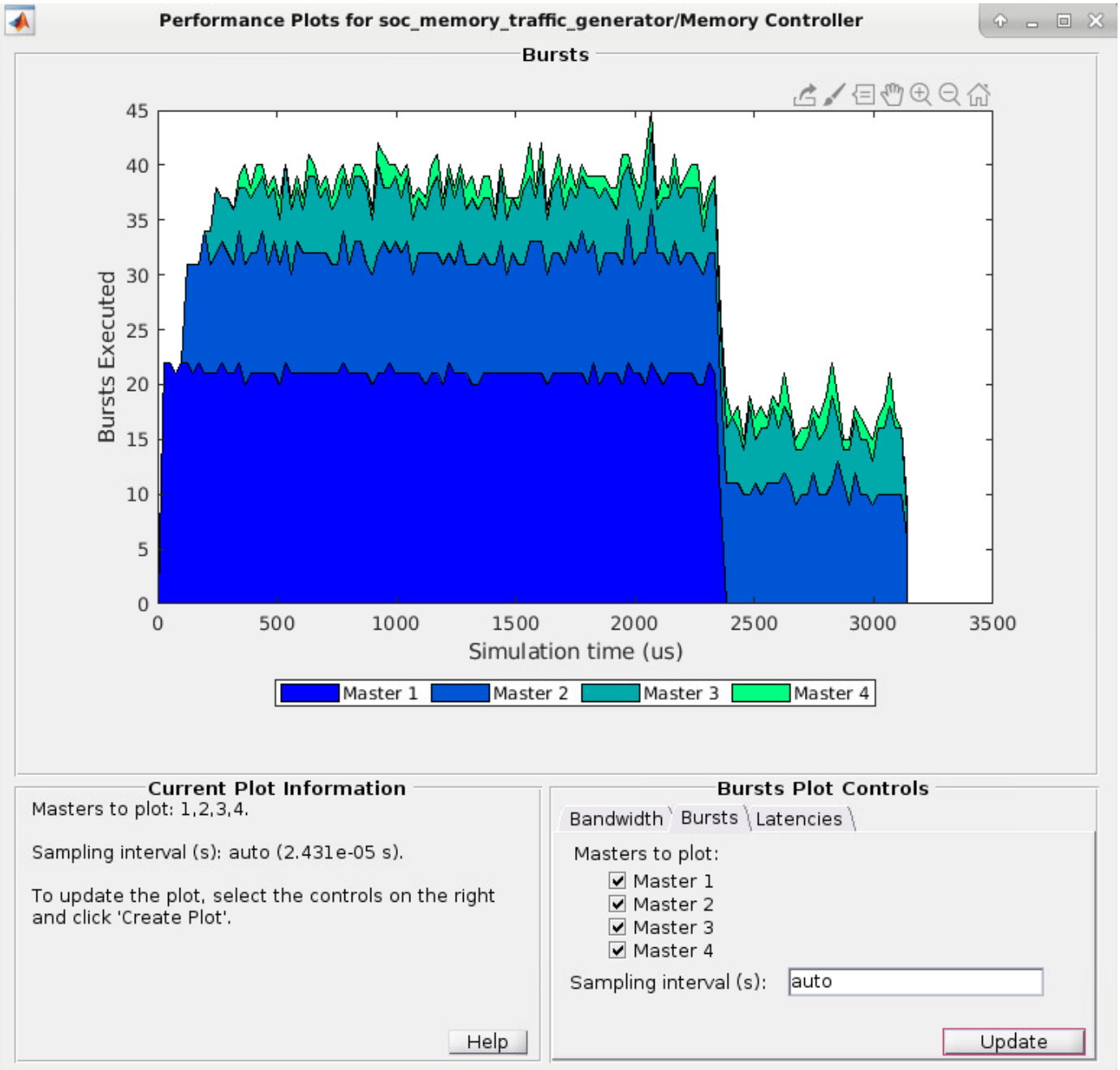
In the **Bandwidth** tab, select the masters for which you want to graph bandwidth. Click **Create Plot** to see the bandwidth, in megabytes per second, for the selected masters over the duration of the simulation time. This image shows the bandwidth for the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example.



Note Bandwidth information is not displayed when a master is a processor.

Memory Burst Plots

In the **Bursts** tab, select the masters for which you want to graph bursts. Click **Create Plot** to see the number of bursts executed for the selected master over the duration of the simulation time. This image shows the burst count for the “Analyze Memory Bandwidth Using Traffic Generators” on page 7-57 example.



Note Bandwidth information is not displayed when a master is a processor.

See Also

Memory Controller | Memory Channel

More About

- “Memory Performance Information from FPGA Execution” on page 4-10

Simulation Performance Tips

To enhance the simulation performance of your SoC Blockset model, apply these settings to the top model and the referenced models in your SoC design (such as the processor model or FPGA model).

- Turn on compiler optimization for accelerator mode. Click **Hardware Settings** on the Simulink Toolstrip to open the configuration parameters dialog box. Then, select **Simulation Target** in the left pane, and set **Undefined function handling** to `Filter out`.
- Expand the `...` at the bottom. Under **Advanced parameters**, perform these actions:
  - Set **Compiler optimization level** to `Optimizations on (faster runs)`. This action turns on compiler optimization for accelerator mode, but it also increases the time of the first `ctrl-D` to build the library.
  - Select **Block reduction**.
  - Select **Conditional input branch execution**.
- In the configuration parameters, select **Code Generation** in the left pane. Under **Toolchain settings**, set **Build configuration** to `Faster Runs`.
- In the left pane, under **Code Generation**, select **Optimization**. Set **Default parameter behavior** to `Inlined`.
- To remove signal logging, enter this code at the MATLAB command prompt.

```
mdlsignals = find_system(gcs,'FindAll','on','LookUnderMasks','all', ...
              'FollowLinks','on','type','line','SegmentType','trunk');
ph = get_param(mdlsignals,'SrcPortHandle');
for i=1: length(ph)
set_param(ph{i},'datalogging','off')
end
```

- To disable model animation, right-click the model canvas, and set **Animation Speed** to `None`.
- Set the FPGA and processor model references to accelerator mode. Navigate the model hierarchy to the model reference, right-click the model reference block, and select **Block Parameters**. Then, in the Block Parameters dialog box, set **Simulation mode** to `Accelerator`.

See Also

“What Is Acceleration?”

Peripherals

Simulate PWM Waveforms and Events

The PWM Write and PWM Interface blocks together enable a variety of pulse width modulation (PWM) waveforms and events to be simulated in an SoC model.

Internal Counter and Comparator Triggers

A PWM peripheral, at minimum, contains an internal timer with a counter and one or more comparators. The timer drives the counter on a continuous loop. The counter can operate in one of three modes:

- Up – The counter increments up to the maximum value of the counter and then overflows, resetting to zero to start the count again. The counter forms a discrete sawtooth waveform.
- Down – The counter decrements from the maximum value of the counter to zero and then underflows, resetting to the maximum value to start the count again. The counter forms a discrete sawtooth waveform.
- Up-Down – The counter increments from zero to the maximum value of the counter and then the count decrements until the count reaches zero again. This cycle repeats to create a discrete triangular waveform.

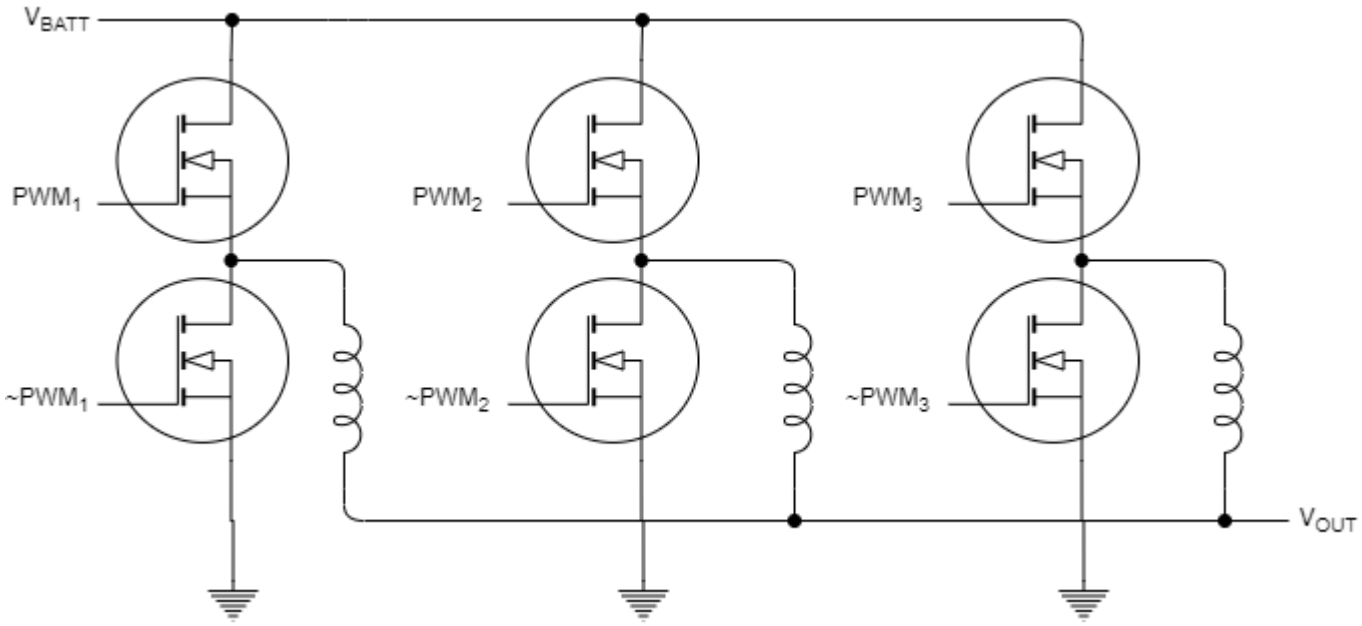
The discrete waveforms generated by the timer and counter define the period and phase of the final PWM waveform. The counter is used as the reference for the comparators to modify the state of the output signal that becomes the PWM waveform.

The PWM contains a bank of comparators. The count value of each comparator gets compared to the value of the counter. When the counter crosses that count value, the comparator triggers. When a trigger occurs, the comparator can change the current output state of the PWM waveform (for example, setting the output to `0`). Additionally, the trigger can generate an event that can be used by the Task Manager block or other peripherals, such as the ADC Interface block, to coordinate the input and output signals in the microcontroller unit (MCU).

With the combination of the period and phase control of the internal timer and multiple comparators, you can create a variety of PWM waveforms to support your specific application requirements.

Phase-Offset Waveforms

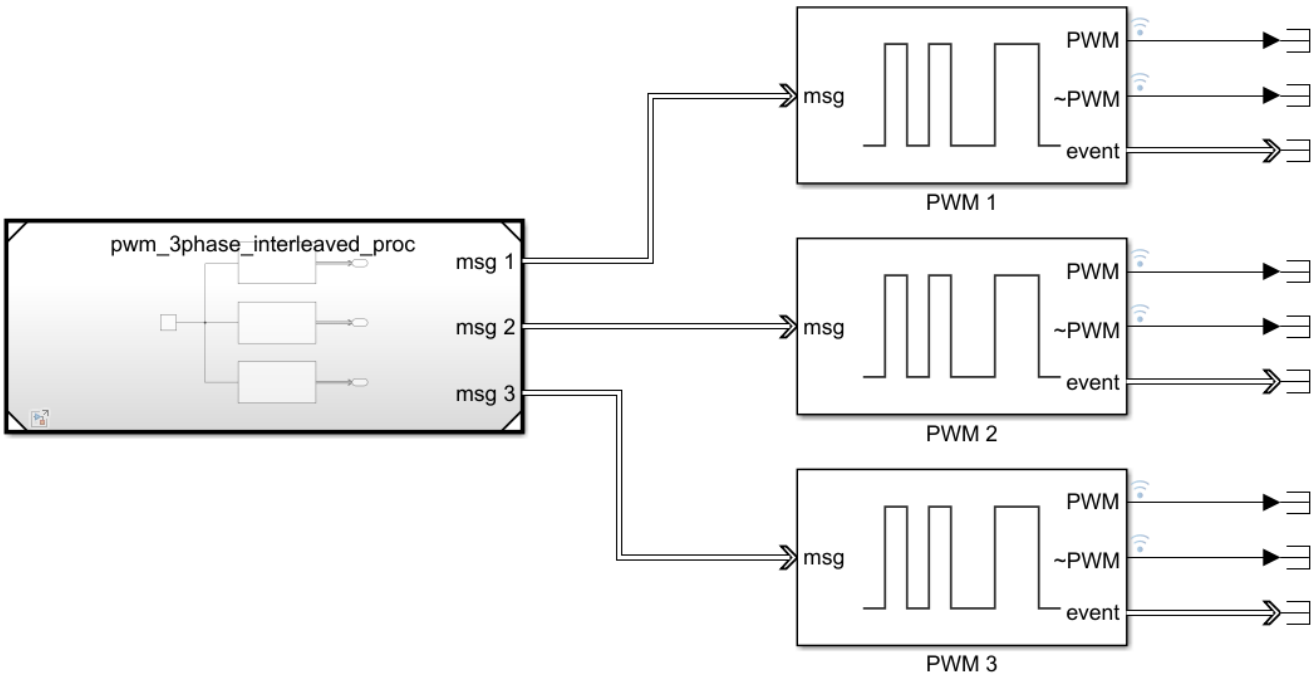
This example shows how to generate phase-offset PWM waveforms. You can use the phase-offset PWM waveform to drive MOSFETs in 3-phase switching power circuits, such as the interleaved DC/DC converter circuit shown in theis figure. Each MOSFET pair gets driven by a PWM of the same frrequency, where each branch phase is offset by 120°.



Model

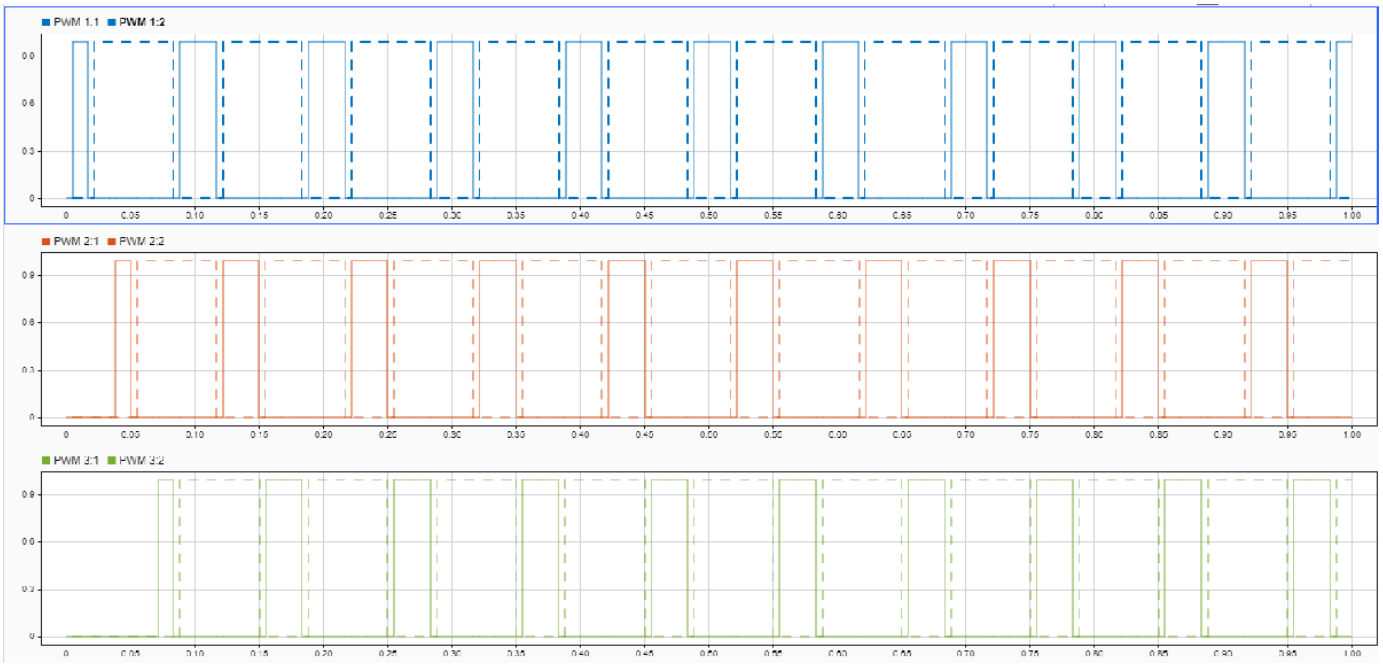
This model contains the three PWM Interface blocks that each drive a separate PWM output. `PWM 1` has a phase of 0°. You can open the `PWM 2` and `PWM 3` blocks and inspect the **Phase > Phase offset in degree (0-360)** parameters, which are 120° and 240°, repsectively. All PWM Interface blocks in a model share an underlying synchonization allowing the PWM block output to be synchonized with offset phases.

```
open_system("pwm_3phase_interleaved_top.slx");
```



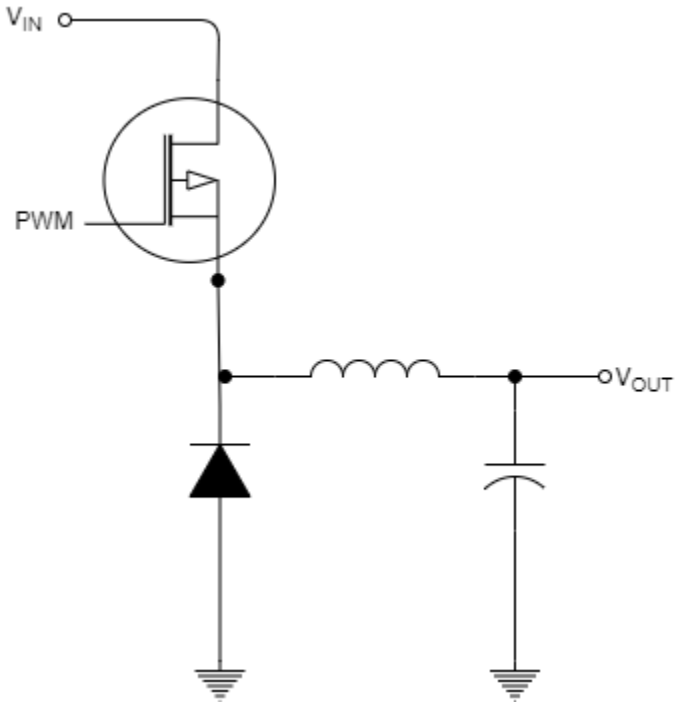
Results

In the **Simulation** tab, click **Run**. When the simulation completes, open the **Simulation Data Inspector** to view the resulting signals from the PWM outputs. The signals show the PWM and PWM complement waveforms from each PWM Interface with each block offset by 120°.



Pulse Center Measurement Event from Waveform

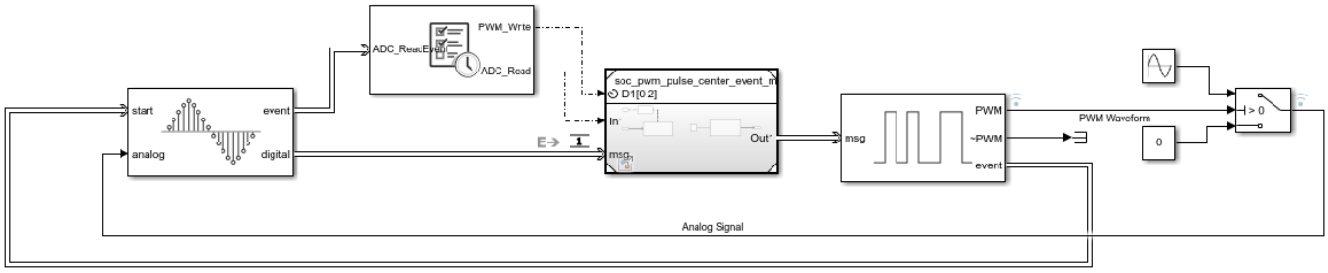
This example shows how to generate an event for a task in the pulse center of a PWM waveform. You can use the triggering of an event in the pulse center of a PWM waveform to get correct current measurements from ADCs in switching power circuits, such as the buck converter circuit shown in this figure. The model in this example shows a basic ADC sampling from a PWM-driven switch.



Model

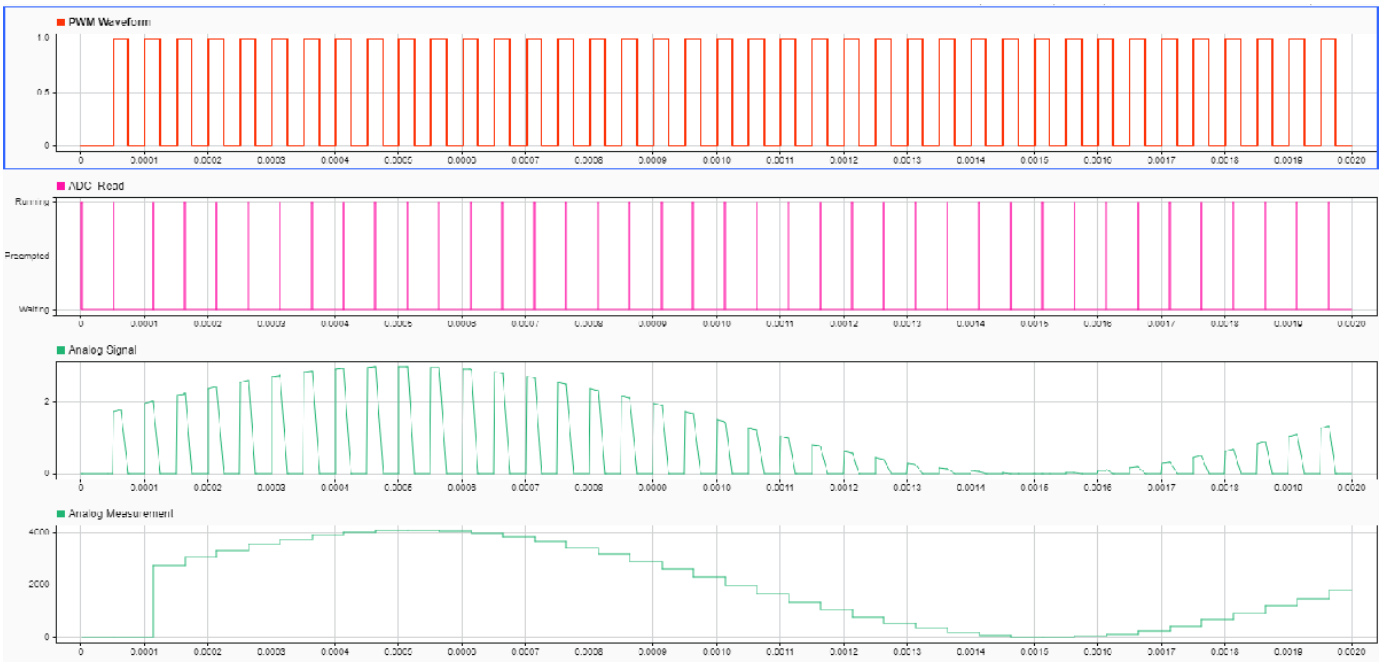
This model uses two tasks. A timer-driven task sets the comparator values for the PWM Write block. The first comparator value, `0.5`, sets the duty-cycle of the waveform produced by the PWM Interface block. The second comparator value, `0.25`, sets the value of the comparator that triggers an event. In the PWM Interface block, the **Counter mode** parameter is set to *Up*, and the **Event trigger mode** parameter is set to *Compare 2*. These settings result in an event generated each time the internal PWM counter exceeds 25% of the total counter value. The output of the PWM Interface block drives a switch that samples from a Sine block. The event signal connects and triggeres the ADC Interface block to sample the output of the switch at the center of the PWM pulse center. A event-driven task triggers on each event and uses the ADC Read block to sample the measured value.

```
open_system("soc_pwm_pulse_center_event_top")
```



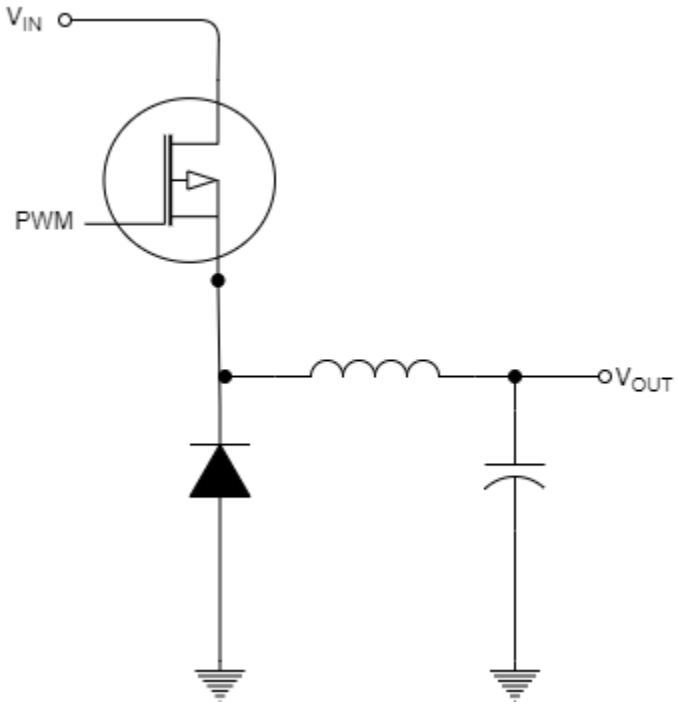
Results

In the **Simulation** tab, click **Run**. When the simulation completes, open the Simulation Data Inspector to view the resulting signals from the PWM, ADC, and task event signals. From inspection, the `ADC_Read` event occurs in the pulse center of the PWM waveform. As a result, the `Analog Measurement` signal captures samples from the sine wave while ignoring the zero-valued gaps when the switch is off.



Symmetric PWM Waveform

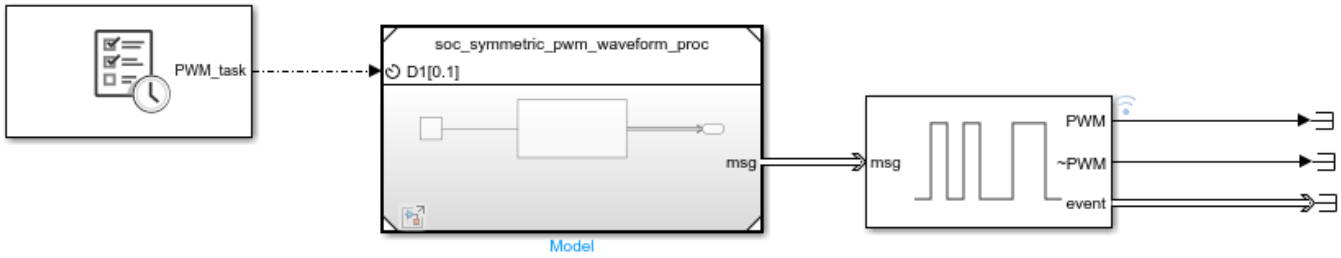
This example shows how to generate a symmetric PWM waveform. In power switching circuits, such as the buck converter shown in this figure, symmetric PWM waveform signals can generate fewer harmonics in the output currents and voltages.



Model

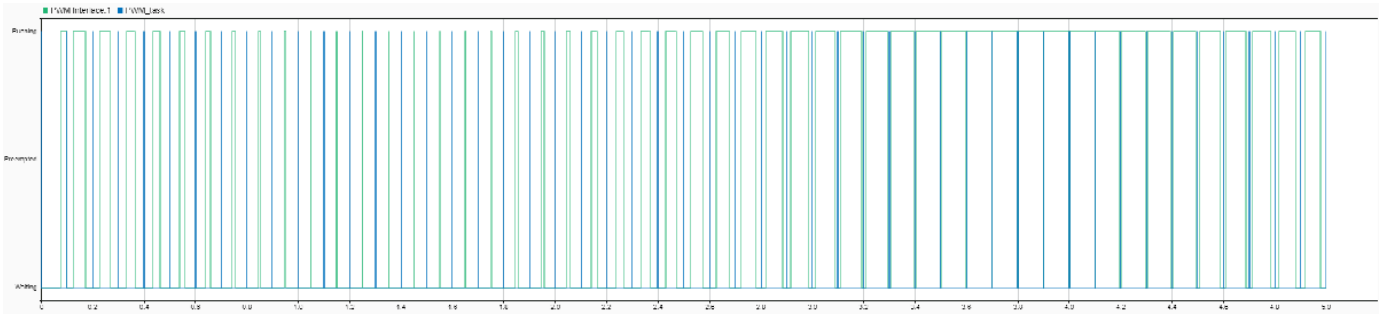
This model contains a single task with a Sine block that sets the pulse width of a PWM waveform. Inspecting the connected PWM Interface block, the **Main > Counter mode** parameter is set to `Up-Down`, resulting in the internal counter forming a triangular wave. The **PWM output > At start of period** parameter is set to `Low`, and the **PWM output > At compare 1 up count** and **PWM output > At compare 1 down count** parameters are both set to `Change`. These settings result in a symmetrical waveform with the pulse center at the center of the PWM waveform.

```
open_system("soc_symmetric_pwm_waveform_top.slx");
```



Results

In the **Simulation** tab, click **Run**. When the simulation completes, open the Simulation Data Inspector to view the resulting signals from the PWM Interface block and `PWM_task` outputs. From inspection, the PWM Interface block output signal is symmetrical and centered in the PWM pulse.



See Also

PWM Interface | PWM Write | ADC Read | ADC Interface | Task Manager | **Peripheral Configuration**

Related Examples

- “Get Started with SoC Blocks on MCUs” on page 7-132
- “Integrate MCU Scheduling and Peripherals in Motor Control Application” on page 7-140
- “Partition Motor Control for Multiprocessor MCUs” on page 7-135

External Websites

- Symmetric PWM Outputs Generation with the TMS320C14 DSP

Record Data from Hardware I/O Devices

Models using recorded data in simulation can reproduce the behavior of the application when implemented onto a physical hardware or device. SoC Blockset provides a set of functions that can connect and record I/O device data directly from a hardware board. The recorded data file can then be used in an SoC Blockset model simulation.

Process to Record Data

To record I/O data from a hardware board, you can follow the general sequence of steps below.

1. *Configure Hardware* – Connect and configure your hardware board. You may need to install the hardware support package for your hardware board.
2. *Create Data Recorder* – A data recorder object manages the I/O hardware peripherals and stores the data during the data collection process.
3. *Choose I/O Devices* – Choose from the available I/O devices on the hardware board and add them to the data recorder object.
4. *Setup Recorder* – Prepare the hardware board for the data recording process. This setup includes any initialization and configuration of the hardware I/O devices to be recorded.
5. *Start Recording* – Start the data recorder on the hardware. The data recorder executes and collects data from the hardware I/O devices for the specified period.
6. *Execute Hardware Operations* – Run hardware operations on the hardware board that exercise the peripherals being recorded. Operations can include sending signals to an analog-to-digital converter or reading data received on a UDP channel.
7. *Save Data* – Save the data stored in the data recorder to a file on your development computer.

The resulting data file can now be used in the simulation of the hardware blocks.

Use Memory and I/O Device Data in Processor Simulation

The **Processor I/O** sub-library in SoC Blockset contains blocks that simulate the data transfer between the processor system and memory or I/O devices in the SoC application. **Processor I/O** blocks including the following:

Memory

- Register Write
- Register Write
- Stream Read

External

- TCP Read
- TCP Write
- UDP Read
- UDP Write

In simulation, an IO Data Source block sends data messages to the **Processor I/O** block. The IO Data Source block can generate data from one of three sources:

- Replay recorded data from file
- From input port
- Zeros

These modes allow tasks to simulate using either previously recorded or data generated in simulation.

For event-driven task, the IO Data Source sends a message to the connected **Processor I/O** block and event signal to the Task Manager block to start the task containing the **Processor I/O** block. This configuration allows simulation of asynchronous tasks with data recorded from actual

See Also

Task Manager | Stream Read | TCP Read | UDP Read | IO Data Source

Examples

Random Access of External Memory

This example shows how to model external memory accesses from FPGA for rotating an ASCII art image. Many applications require FPGA to access memory in random fashion as per the requirements of algorithm. You will learn how to design memory address generation along with other AXI4 master signals to read and write specific regions of memory using SoC Blockset. You will simulate, implement and verify your design on hardware.

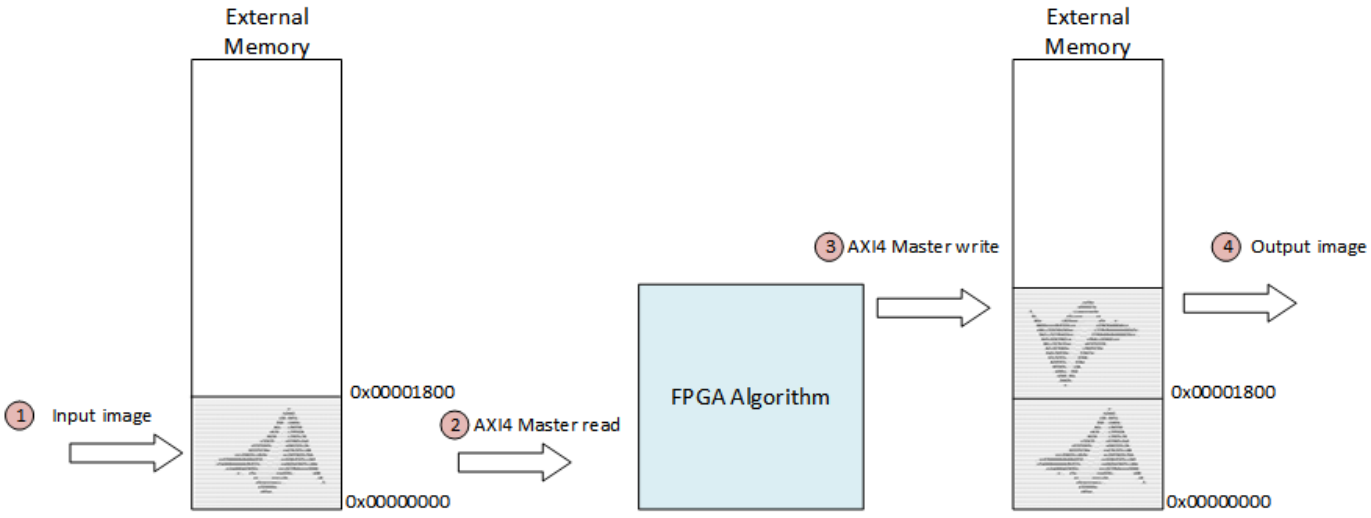
Supported hardware platforms:

- Artix® 7 35T Arty FPGA evaluation kit
- Xilinx® Kintex® 7 KC705 development board
- Xilinx Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Xilinx Zynq UltraScale™+ RFSoC ZCU111 Evaluation Kit
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

Design Task

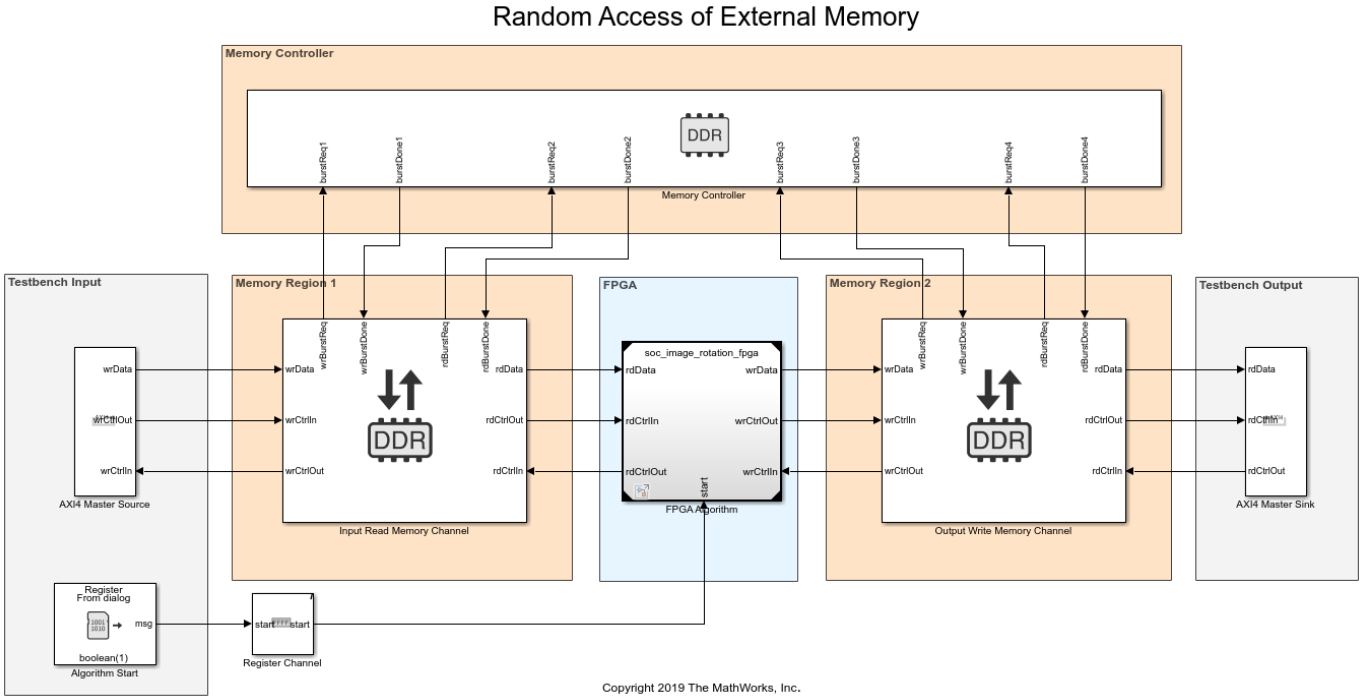
The ASCII art image is encoded as 24-by-64 matrix of uint8 characters. The design task is to rotate the image by modeling AXI4 Master interfaces in FPGA logic for external memory access. By simulating the design with external memory model and the AXI4 protocol, you verify the behavior at application design time. This saves time otherwise spent in debugging the design on hardware during the implementation phase.

The overall dataflow is as described in figure below. The image is stored in the external memory at the memory region from address 0x00000000 to 0x000017FF. FPGA algorithm reads the image from this region and rotates it by writing it in the reverse order into the memory region from 0x00001800. Finally, the data is read back from the memory.



Model Structure

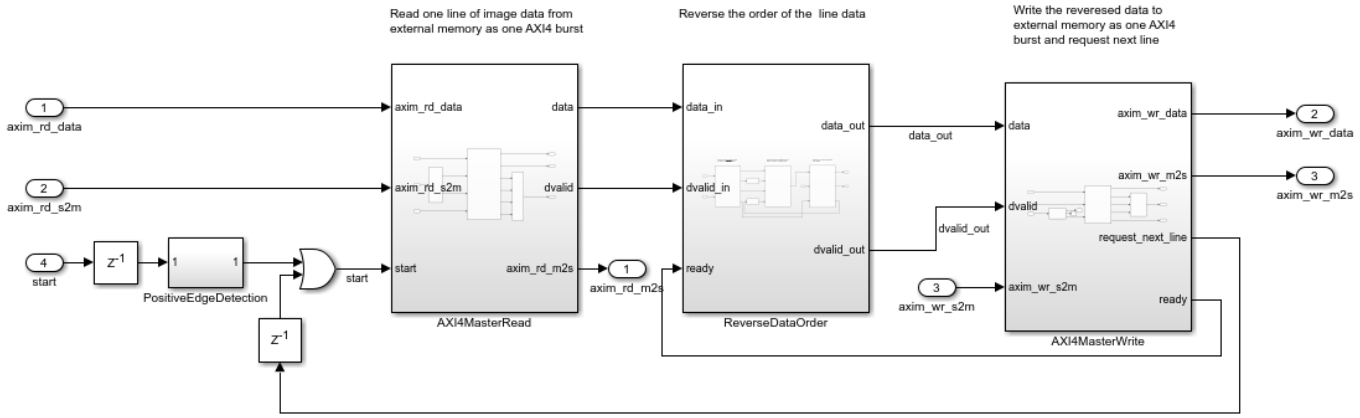
The models are structured using Model references. Top model ' **soc_image_rotation** ' includes the FPGA model ' **soc_image_rotation_fpga** ' using Model block as *model reference*.



The top model covers the following areas:

- **Testbench Input**: It models the stimuli to set up the design for simulation. The `AXI4 Master Source` block initializes the input image data to the external memory. The `Algorithm Start` block sends a Start signal to the FPGA algorithm via Register Channel block. Open preload function soc_image_rotation_init.m to see how model parameters and input data are initialized.
- **Testbench Output**: The `AXI4 Master Sink` block models the reading of the output image data from the external memory. The output data is saved in the variable AXI4MasterSinkContent in the workspace. Open stop function soc_image_rotation_post.m to see how input data and output data are plotted.
- **Memory**: Memory system is modeled using one Memory Controller and two Memory Channel blocks. `Input Read Memory Channel` block models memory region 1 where input image is stored and `Output Write Memory Channel` block models memory region 2 where the rotated image is stored.
- **FPGA**: This area instantiates the FPGA model reference which models the logic for AXI4 Master interfaces and data rotation.

FPGA model implements the algorithm in three subsystems, `AXI4MasterRead`, `ReverseDataOrder` and `AXI4MasterWrite`. Open FPGA subsystem for image rotation:



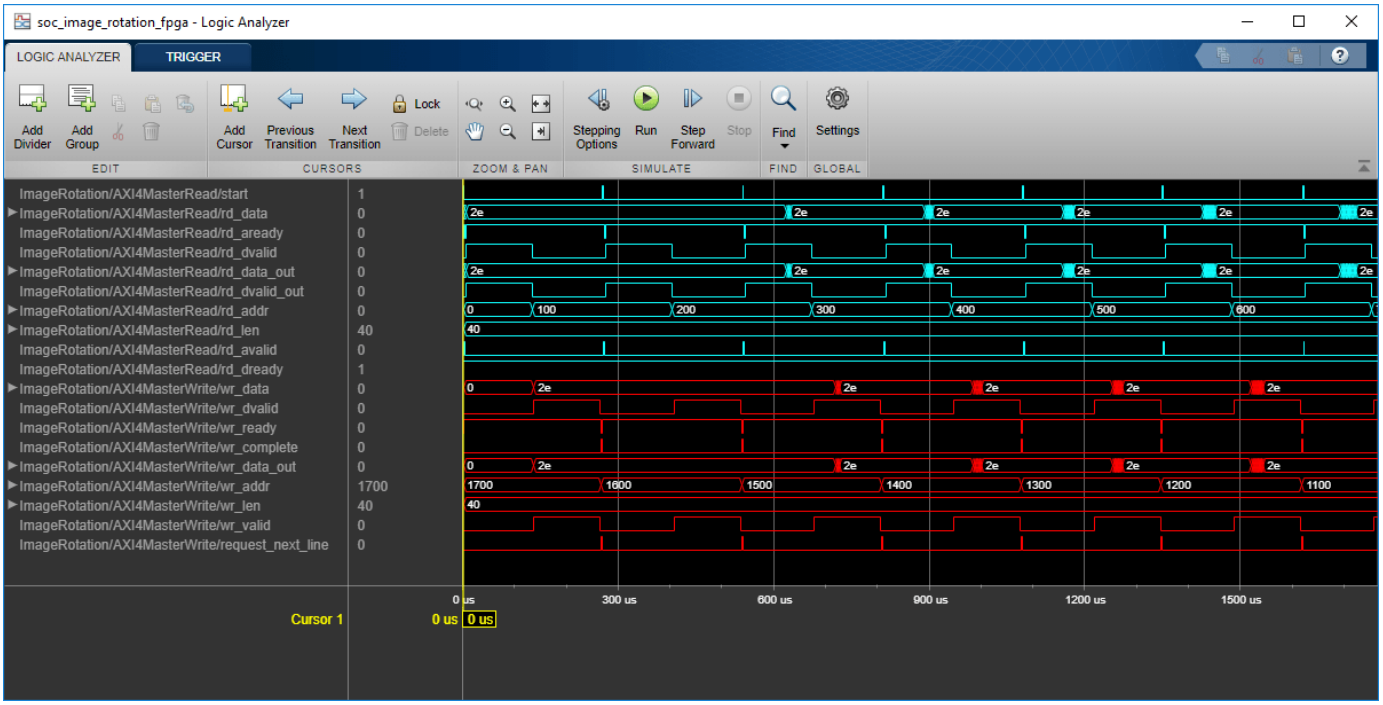
As the positive edge of `start` signal is detected, `AXIMasterRead` reads one line of image data and delivers it to `ReverseDataOrder` to reverse the order of data. The reversed data is then written to external memory by `AXIMasterWrite` subsystem. Once the data for one line is written, it sends a signal `request_next_line` to trigger reading of next line by `AXIMasterRead`. This cycle continues until all lines of the image are processed.

Open AXI4MasterReadController and AXI4MasterWriteController blocks to inspect the MATLAB® code for AXI4 Master interfaces. These blocks design the addressing logic for read and write operations as per AXI4 protocol. SoC Blockset supports AXI4 Master protocol and for timing diagrams of AXI4 signals, please refer to Model Design for AXI4 Master Interface Generation.

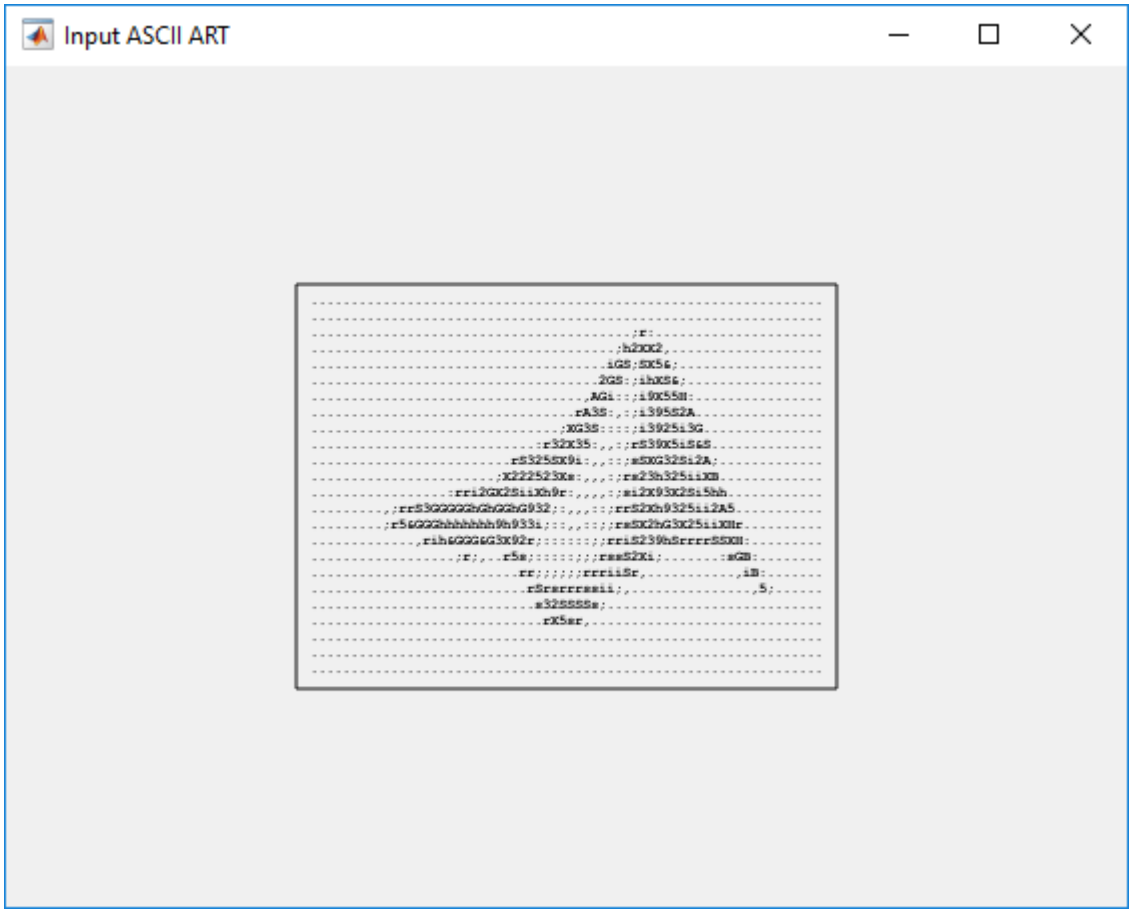
Simulation

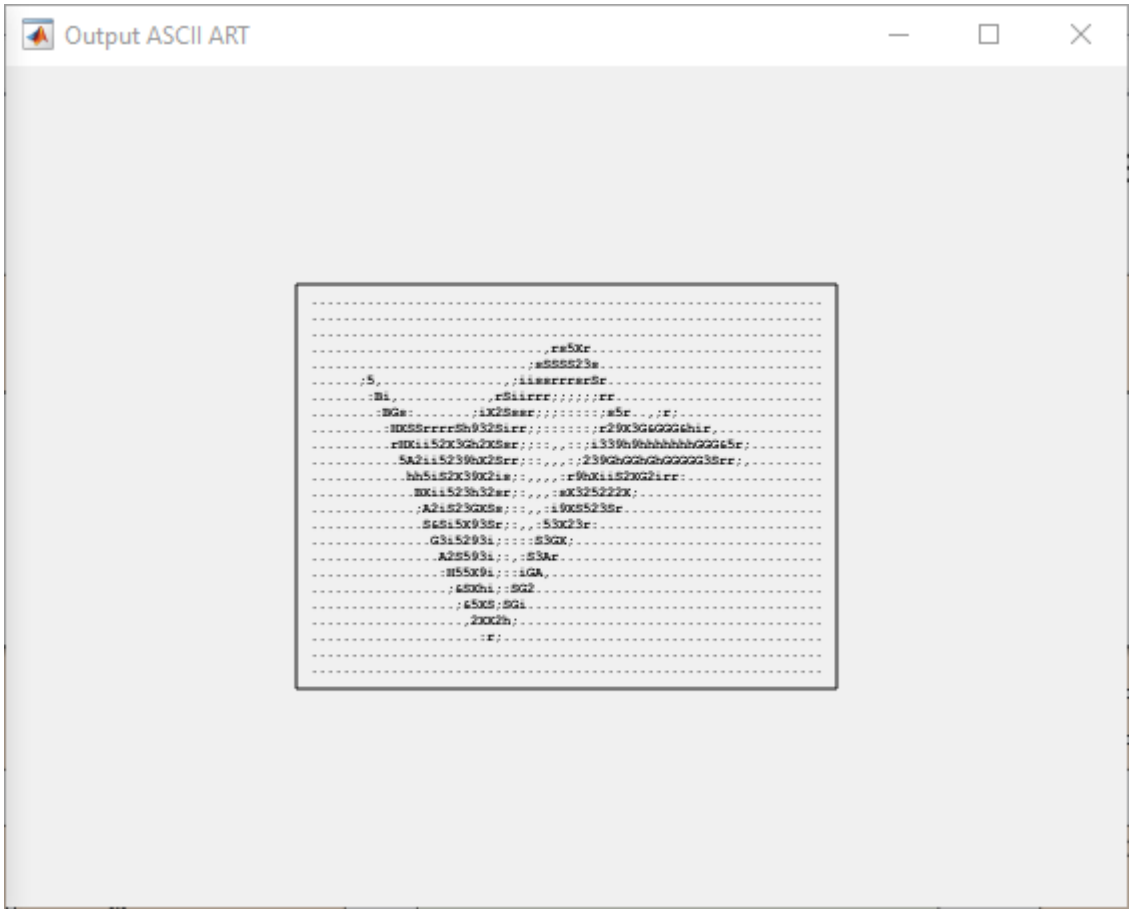
Run the model and open the Logic analyzer from the FPGA model. Notice the following key points:

- One line of data is written/read by masters in one burst. Since each line is 64 characters long; the burst length is 64 (0x40). Note this value on signals `rd_len` and `wr_len`.
- Each character has 4 bytes as it is extended to uint32 data type, which makes the length of line 64x4 = 256 (0x100) bytes. Therefore, addresses increment/decrement by 0x100. Note this on `rd_addr` and `wr_addr` signals.
- One read burst is followed by one write burst. Observe how `rd_dvalid` and `wr_dvalid` toggle alternatively.
- `request_next_line` asserts after each write burst, which trigger the next read burst.



The input and output images are plotted at the end of simulation:


Input ASCII ART



Implementation

Following products are required for this section:

- HDL Coder™
- SoC Blockset Support Package for Xilinx Devices, or
- SoC Blockset Support Package for Intel® Devices

To implement the model on a supported FPGA board, use the SoC Builder application. Make sure you have installed required products and FPGA vendor software before implementation.

Open SoC Builder by clicking 'Configure, Build, & Deploy' button in the toolstrip and follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Notice that the base address 0x00000000 is assigned to `Input Read Memory Channel` block, and base address 0x00001800 is assigned to `Output Write Memory Channel` block. The AXI4 address is the sum of base address and address from FPGA algorithm. For example, `wr_addr` from FPGA algorithm starts with 0x1700. The output data will be written to the external memory from address 0x00001800 + 0x1700 = 0x00002F00. Refer to Model Design for AXI4 Master Interface Generation for more information about base address register calculation. Click 'Next'.

- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next' to 'Load Bitstream' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load' button to load pre-generated bitstream.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox',...
         'soc','supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
         'soc_image_rotation-zc706.bit'), './soc_prj');
```

To run this example, copy the example test bench to your project folder.

```
copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','soc_image_rotation_aximaster.m'),...
         './soc_prj','f');
```

Enter the following command to run the test bench:

```
soc_image_rotation_aximaster
```

The test bench performs the following operations:

- Initializes image rotation IP
- Writes input image data to external memory
- Starts the image rotation operation
- Reads back and display output image data from external memory

If your FPGA board is not Xilinx Zynq ZC706 evaluation kit you need to do the following settings in the configuration parameters of the top model before launching the SoC Builder.

- Select the 'Hardware board' under 'Hardware Implementation' panel to match your board.
- Uncheck 'Include processing system' under 'Hardware Implementation -> Target hardware resources -> FPGA design (top-level)' panel.
- Set 'Interconnect data width (bits)' to '32' under 'Hardware Implementation -> Target hardware resources -> FPGA design (mem channels)' panel.

Available pre-generated bitstreams are:

- 'soc_image_rotation-zc706.bit'
- 'soc_image_rotation-arty.bit'
- 'soc_image_rotation-zcu102.bit'
- 'soc_image_rotation-XilinxZynqUltraScale_RFSoCZCU111EvaluationKit.bit'
- 'soc_image_rotation-kc705.bit'

- 'soc_image_rotation-a10soc.sof'

Modify the *copyfile* command and example test bench to match your board and selected project folder as appropriate.In case of Altera Arria® 10 SoC development kit and Altera Cyclone® V SoC development kit use below *copyfile* command corresponding to your board.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
         'supportpackages','intelsoc','intelsocexamples','bitstreams',...
         'soc_image_rotation-a10soc.sof'), './soc_prj');
```

Note that pre-generated bitstream may not work if you customized the memory map.

Conclusion

This example shows modeling of AXI4 Master interfaces for accessing external memory in random fashion using SoC Blockset by rotating an ASCII art image. You can use this as a guide to design your own algorithm to access memory directly using AXI4 Master protocol.

Packet-Based ADS-B Transceiver

Packet-based systems are common in wireless communications. Data is received over the air and is decoded as discrete packet data on a compute device. For given system requirements, it is difficult to design a system and implement directly on SoC as it often involves long iterations of debugging and integration on hardware since hardware effects are difficult to account for at design time. In this example, you will design packet-based airplane tracking application based on Automatic Dependent Surveillance Broadcast (ADS-B) standard, partitioned between FPGA and embedded processor. Unlike traditional methods, you will simulate the application design with memory interface before implementation on hardware using SoC Blockset to shorten development time. You will then validate the design on hardware by automatically generated code from the model.

Supported Hardware Platforms:

- Xilinx® Zynq® ZC706 evaluation kit + Analog Devices® FMCOMMS2/3/4 card.
- ZedBoard™ + Analog Devices FMCOMMS2/3/4 card.

Design Task and System Requirements

As per ADS-B standard a message packet contains a total of 120 bits which has an 8 bit preamble and 112 bits of information about the aircraft including its position and velocity. For an introduction to the Mode-S signaling scheme and ADS-B technology for tracking aircraft, refer to the 'Airplane Tracking Using MATLAB®' example in Communications Toolbox.

Our task is to design a system to receive ADS-B messages off the air and decode with following performance requirements:

- Latency: 0.5 seconds
- Drop sample rate: < 1 in 105 messages
- Throughput: 0.125 MBps (for capacity of maximum 300 aircrafts)

Design Using SoC Blockset

Design Parameters: Data is transferred from FPGA to processor across shared memory as a frame of samples. There are two key design parameters, **Frame Size** and **Number of Buffers** which affect the above performance requirements.

- **Frame Size**: Frame Size is the number of samples in a frame. It will be used for determining the buffer size in memory channel.
- **Number of Buffers**: Number of frame buffers in memory channel. Data is continuously written into memory by FPGA algorithm as frame buffers which are then read by processor to execute its identification algorithm task.

Select the design parameters to satisfy the system requirements as follows:

Design to Meet Latency Requirement: Latency is the time period between when the data is received by the FPGA logic and the data is ready to be processed by the processor. It comprises of two parts, latency through the FPGA logic and the latency for the processor to be available to process data.

Latency through the FPGA logic is the time required for data processing through the FPGA. This is typically on the order of a number of clock cycles with the clock running in MHz range. Latency for the processor to be available to process data, is determined by the time it takes for samples to

transfer from FPGA to processor through FIFO and memory frame buffers. If we size FPGA FIFO equivalent to one frame buffer, then the maximum latency can be written as follows:

$$MaxLatency = (NumberOfBuffers + 1) * (TimeToGatherAFrame)$$

As the Time to gather a frame is directly proportional to Frame Size, therefore, the maximum latency in the data transfer is directly proportional to Frame Size and Number of buffers.

Time to gather a frame is a constant for continuously streaming applications and is equal to Frame Size times the FPGA output sample time. However, for asynchronous packet-based systems, this time also depends on the frequency of arrival of packets. If you choose a Frame size larger than the packet size, then you may have to wait for an indeterminate time for arrival of all the packets required to make a frame. If you choose the packet size smaller than packet size, then it will adversely affect the throughput. Therefore, for asynchronous packet based systems, Frame Size equal to packet size is a reasonable choice. This allows each packet to transfer to processor as soon as the FPGA processing is completed, thereby reducing the latency.

For this example, the decoded packet length is 112 bits, packed into four 32-bit samples. So, the frame size is 4.

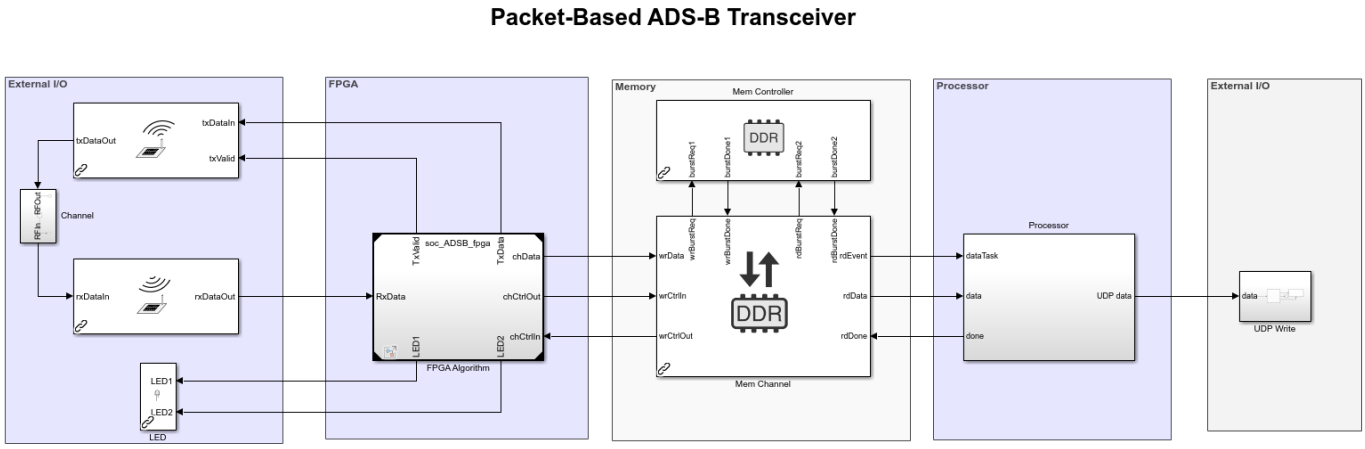
Design to Meet Throughput Requirement: Throughput is the amount of data produced as output per unit of time. This is a function of the data processing in FPGA and the data transfer & processing by processor. For FPGA logic, the data is processed at clock frequencies of the order of MHz and an output is produced every few clock cycles. For data transfer and processing by processor, it depends on Frame Size. A typical tradeoff is larger Frame size results in higher throughput but it increases the latency. Conversely, a smaller frame size results in lower latency but it decreases the throughput.

Design to Meet Drop Samples Requirement: An application may tolerate occasional drop data caused by the variations in task execution durations. Frame buffers in a memory channel hold data when it can't be immediately processed by the processor. Therefore, increasing the number of Frame buffers reduces the sample drop-outs but it adversely affects the latency as explained earlier.

Choose the Number of Buffers value such that you are able to meet the Drop samples requirement without affecting the maximum latency requirement.

For this example, the mean task duration, as measured on ZC706 is 114us. Each packet duration is 120us. Even if the packets arrive back to back, they can be processed with minimal number of frame buffers since on average the task is processed before the new packet arrives. So, set the number of frame buffers to the minimum possible, 3.

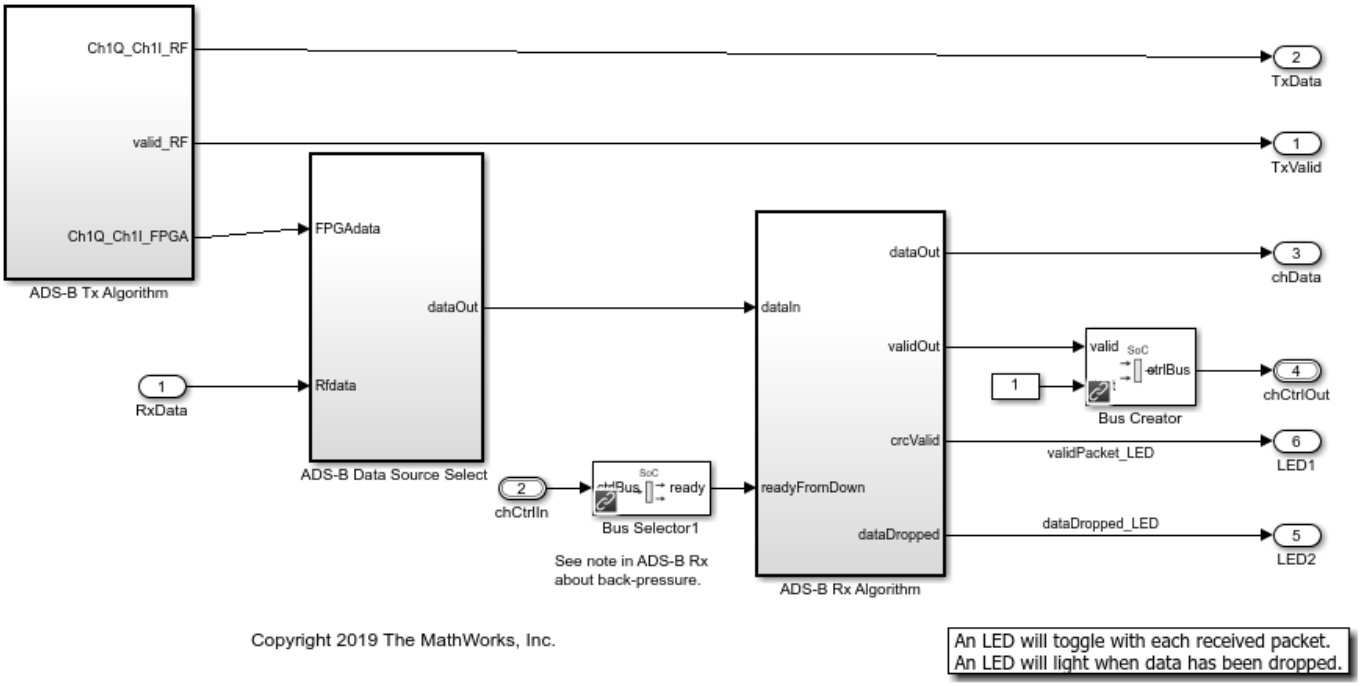
Create an SoC Model: Use the “SDR Template” on page 2-22 for creating an SoC model for wireless communications applications.



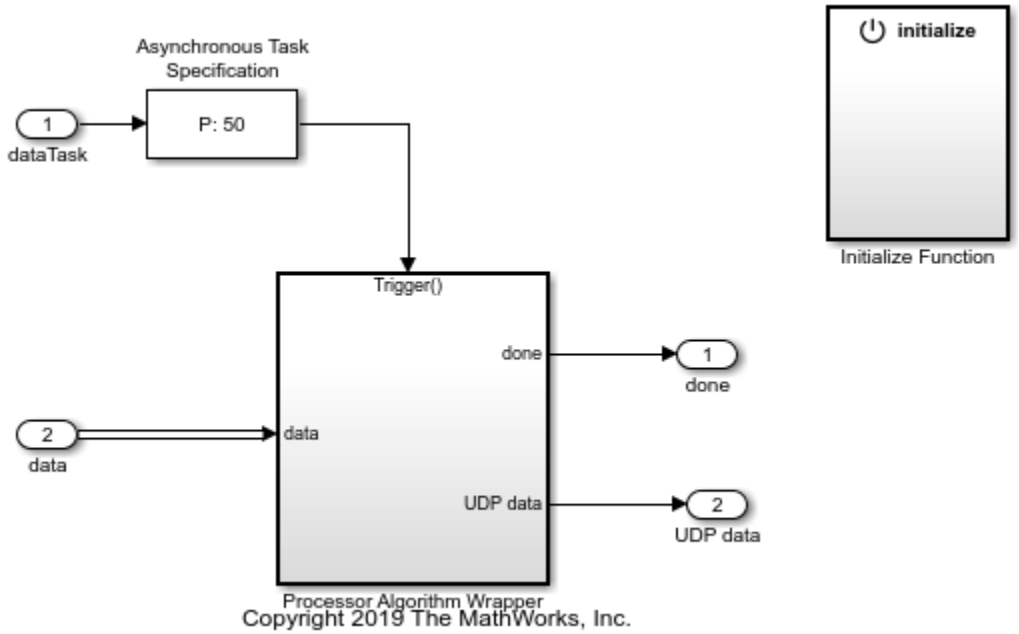
The top model is depicted with bounding boxes that segment the model as follows:

- **External I/O**: This part of the model contains the AD9361 RF Input and Output blocks which are connected to each other using a simplified channel model. In addition this region has LED blocks that connect the FPGA logic.
- **FPGA**: The FPGA section of the model contains the FPGA algorithms which are designed in a separate model and instantiated here using model reference.
- **Memory**: This section models the memory channel between FPGA and processor. It simulates the latencies in the HW/SW connection.
- **Register Channel**: This section models three FPGA registers that are configured by the processor.
- **Processor**: This section contains the Task Manager that is connected to processor model. The Task Manager controls the scheduling of processor tasks. The processor algorithm and initialization tasks are modeled in a separate model and is instantiated here using model references.

FPGA model contains *the ADS-B Transmitter Algorithm* that transmits test ADS-B packets at a variable rate and the *ADS-B Receiver Algorithm* that decodes received ADS-B messages.

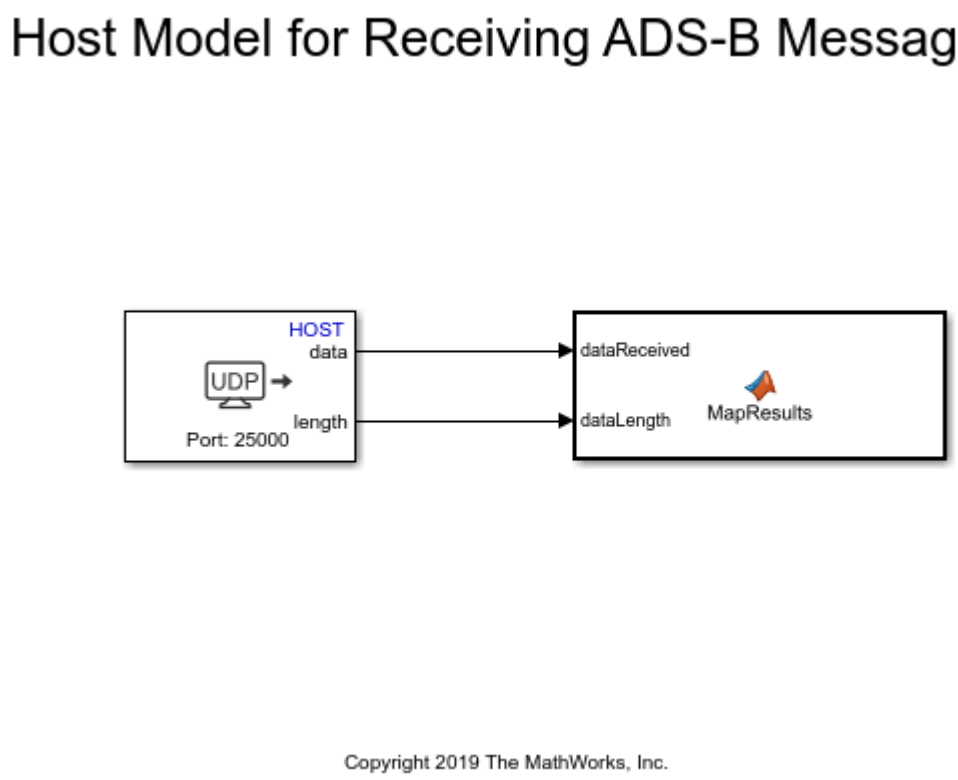


The processor model contains *Processor Algorithm* that unpacks the received ADS-B packets into information bits and sends them via UDP Send block to another system for reporting the aircraft information. The processor algorithm task is denoted as *dataTask* in the Task Manager block and is specified as event-driven. The Task Manager schedules data asynchronously by means of a buffer ready event *rdEvent* in the memory channel.



The *Initialize Function* subsystem initializes appropriate hardware configuration registers. The AD9361 blocks set the center frequency, gain mode, and baseband sample rate of the attached FMC RF board. The other blocks model three memory mapped configurations of the ADS-B packet detector datapath. These include selection of input to receiver algorithm, transmit period of test packets from FPGA and threshold value for detection algorithm.

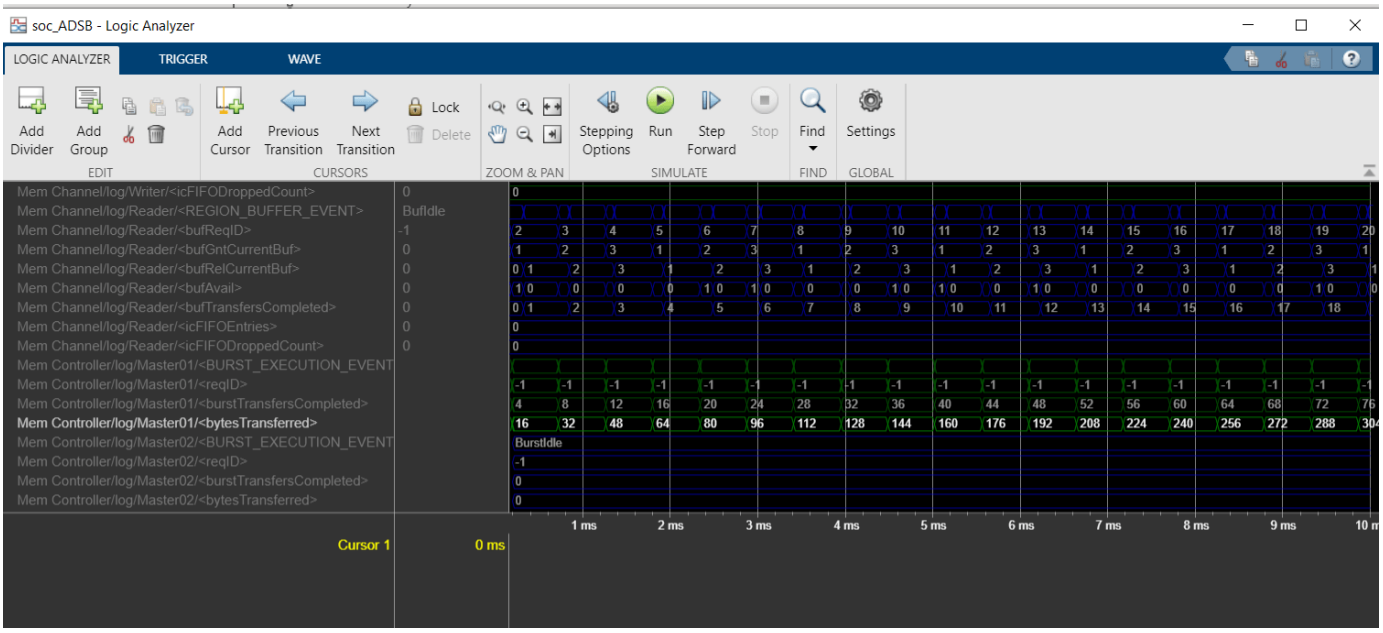
The model soc_ADSB_UDP_HostPrintout is a host UDP-based receive model that decodes ADS-B messages. Run this model in parallel to the ADSB simulation or deployment model to display the decoded ADS-B messages and also optionally map the aircraft location.



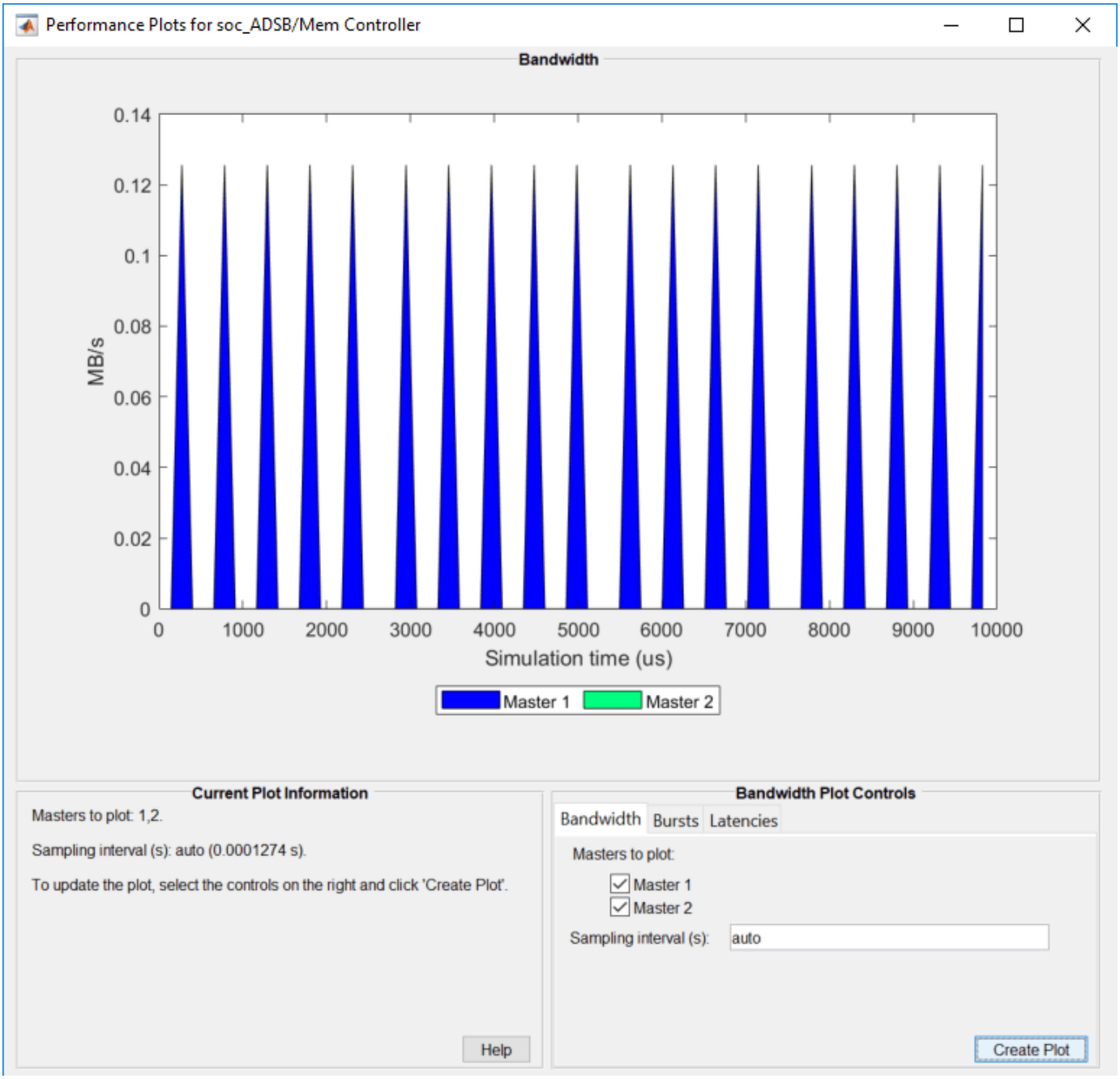
Simulate

Run the model to visualize data transfer between the FPGA and the processor. The time period between the arrival of packets is a function of number of aircrafts. Given system requirement of detecting 300 aircrafts, there will be on average 300*6.2 = 1860 messages per second (or a message every 1/1860 = 0.54 ms). You can set the number of aircrafts using the variable *NumAircraft* which in turn sets the period in the *Initialize Function* subsystem. The default setting is 300 to match the allowable system capacity.

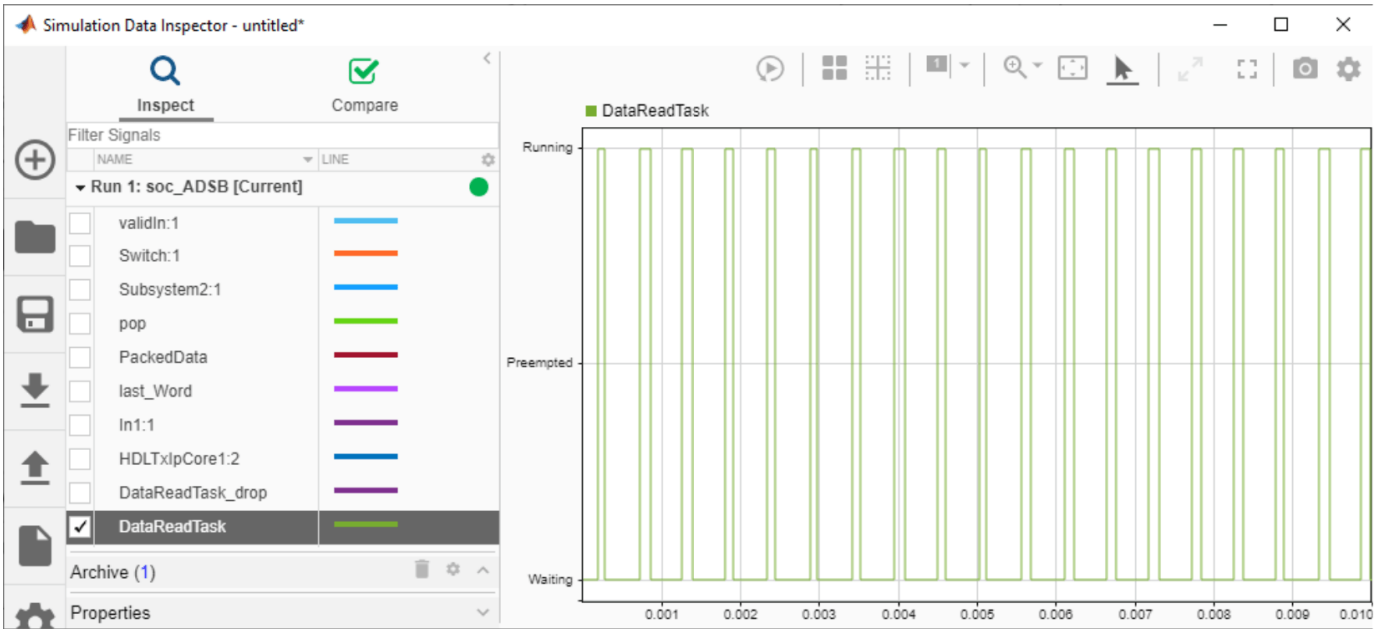
Open the Logic Analyzer window to see the waveforms, and notice that the memory transfers are taking place in buffers of 4 samples, or 16 bytes.



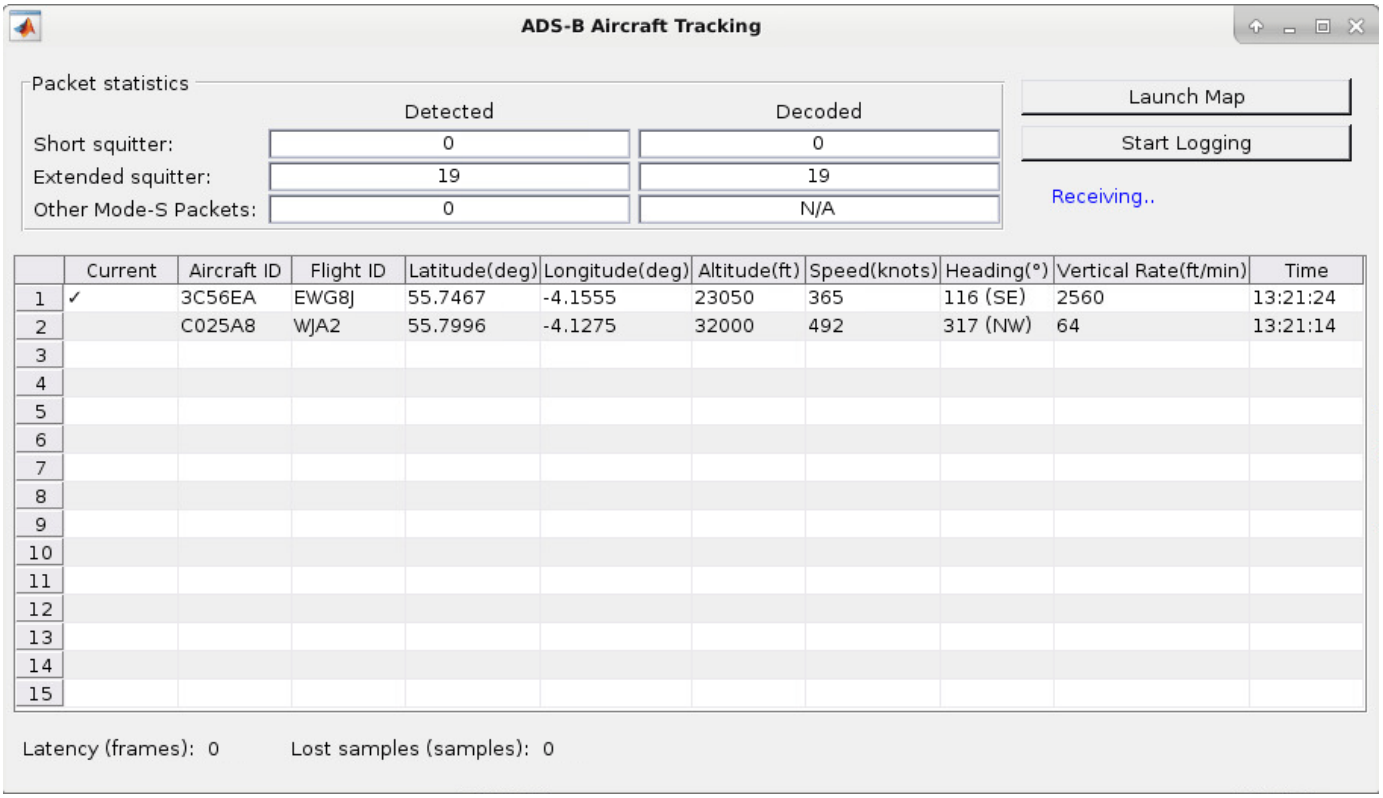
To view the external memory bandwidth usage, open the *Mem Controller* block, select the *Performance* tab and click *View performance plots* . Select all the masters and click *Create Plot*. The plot shows the bandwidth of 0.125 MBps. Since 4 bytes of data is transferred every 32us, the expected bandwidth is 4/32e-6 = 0.125 MBps.



Using the Simulation Data Inspector, you can visualize the task execution schedule. The data task is driven by the event from FPGA notifying the processor that a packet has been decoded by the FPGA, written to external memory, and read by the DMA driver.

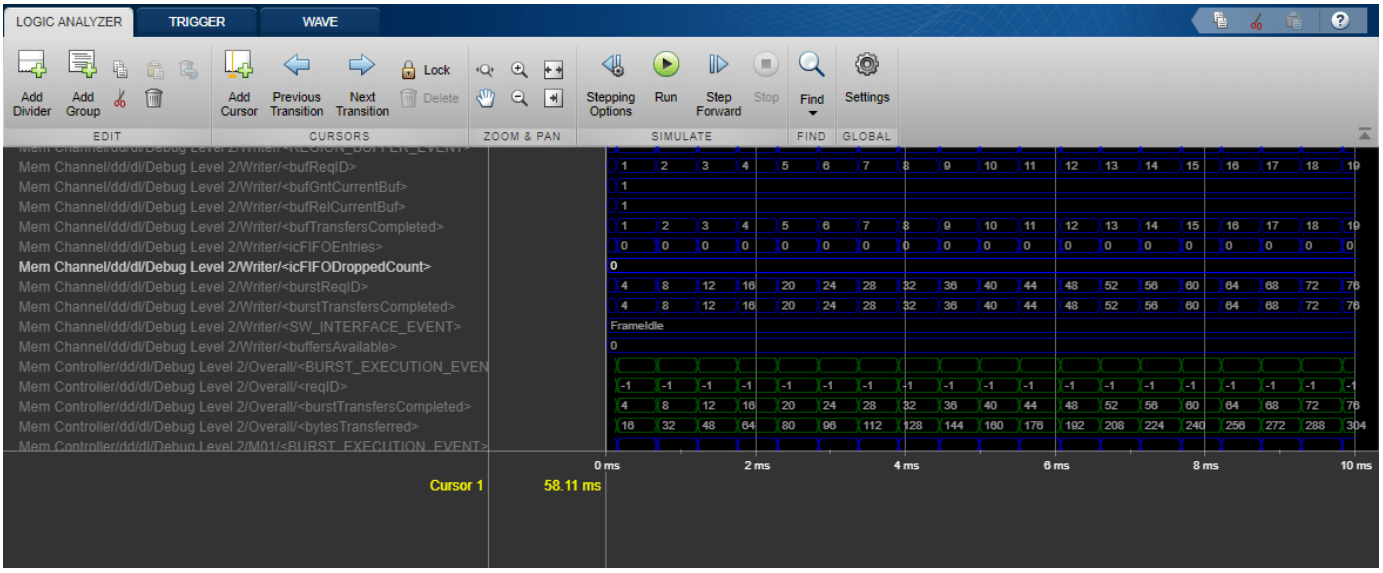


To see the decoded messages, run the companion UDP receive model. This model will display the aircraft tracking information on a GUI.



Hardware Requirements Analysis

As discussed earlier, since mean task duration of 114us is less than the packet duration of 120us, the messages are not dropped on average, during the transfer to the processor. This is confirmed by looking at the number of dropped samples at FIFO using signal *icFIFODroppedCount* in the Simulation Data Inspector.



The SoC model can be used to explore the design space. Consider the worst-case scenario when the plane messages are received densely and there is more computation load on the processor. You can modify the model settings and simulate and determine whether packets are dropped in this more aggressive scenario.

Set the *NumAircraft* to 990 (a new message every 163us) to simulate back to back arrival of plane messages. Modify the task specification on the Task Manager block to simulate more computation load on processor. On the Simulation tab, choose the second distribution by setting the Percent value to 100% on second row and 0% on the first row. This assigns a mean task duration of 163us, which will result in some task executions taking longer than allowed. Set the simulation time to 0.1ms and simulate. For 990 planes, the messages arrival rate is 990*6.2 = 6138 messages per second. The drop packet requirement is therefore, 6138/105 = 58 messages per second or 5.8 messages in 0.1 sec. Upon simulation notice in the Logic Analyzer that this requirement is violated as 18 messages have been dropped.

Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- Embedded Coder®
- “SoC Blockset Support Package for Xilinx Devices”

To implement the model on a supported SoC board use the SoC Builder tool. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board. To open SoC Builder, select the 'System on Chip' tab in the Simulink toolstrip, and click the 'Configure, Build, & Deploy' button. Once SoC Builder opens, follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
- Click 'Test Connection' on 'Connect Hardware' screen to test the connectivity of host computer with SoC board. Click 'Next' to go to 'Run Application' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load and Run' button to load pre-generated bitstream and run the model on SoC board

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
         'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
         'soc_ADSB-zc706.bit'),'./soc_prj');
```

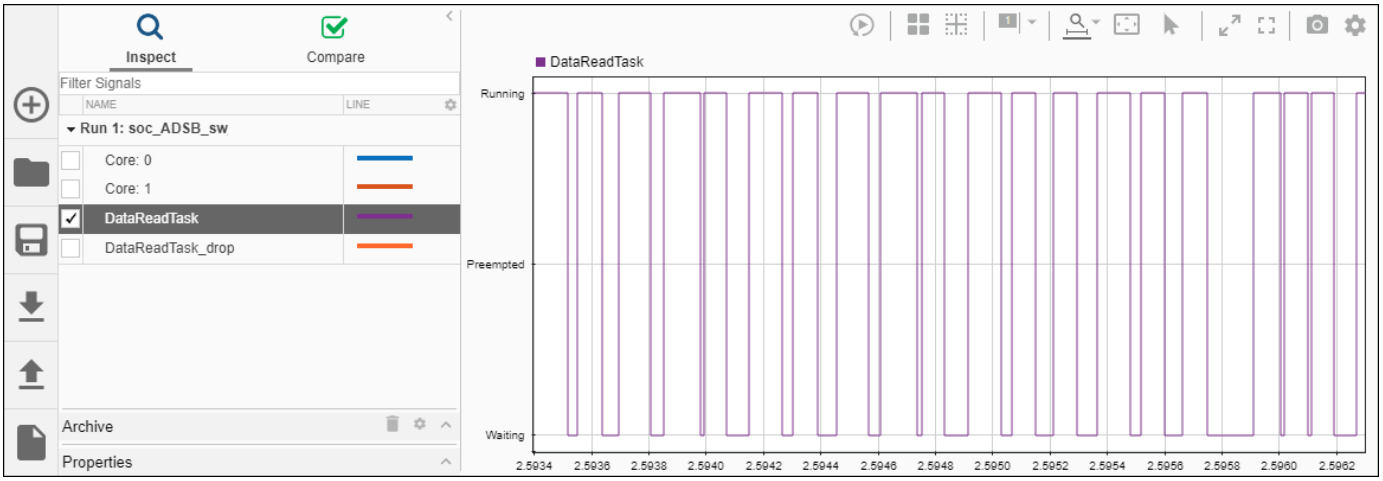
Implementation on ZedBoard: To implement the model on ZedBoard, you must first configure the model to ZedBoard and set the following example parameters. Open **Model Configuration Parameters**, navigate to **Hardware Implementation** tab and perform the following:

- Select **ZedBoard** from the drop-down list under 'Hardware board' on both top and processor model.
- Navigate to **Target hardware resources > FPGA design (top level)** tab, enable **Include MATLAB as AXI Master IP for host-based interaction** and set **IP core clock frequency (MHz)** to 4 MHz.
- Navigate to **Target hardware resources > FPGA design (debug)** tab and enable **Include AXI Interconnect monitor**.
- Navigate to **Device details** and select **Support long long** on both top and processor model.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the *copyfile* command to match Zedboard bitstream 'soc_ADSB-zedboard.bit'.

Profiling Results

To enable processor task profiling, open configuration parameters and navigate to **Hardware Implementation > Hardware Board settings > Task Profiling on processor** and select 'Show on SDI' and 'Save to file'. Set the Simulation stop time to 10 seconds and run the model in external mode. After simulation is completed, open Simulation Data Inspector (SDI) and navigate to the latest run and add signal *DataReadTask* to the plot. Observe that the simulation model accurately predicted how the application would perform on hardware.



Summary

This example showed how SoC Blockset is used to design packet-based ADS-B standard to meet system requirements. By simulating the design with memory channel as interface between the FPGA and the Processor you validated that the system requirements of throughput and drop packets are met at the design time. You implemented the design on SoC device from the model and verified the results on hardware. Although ADS-B is not a computationally intensive standard, it is useful to demonstrate the design process for packet-based systems intended for implementation on a SoC device. You can follow the same design procedure for even more computationally intensive requirements for this application or another packet-based application.

Histogram Equalization Using Video Frame Buffer

Video processing applications often store a full frame of video data to process the frame and modify the next frame. In such designs video frames are stored in external memory while FPGA resources are used to process same data. This example shows how to design a video application with HDMI input and output performing histogram equalization using external memory for video frame buffering.

Supported hardware platform

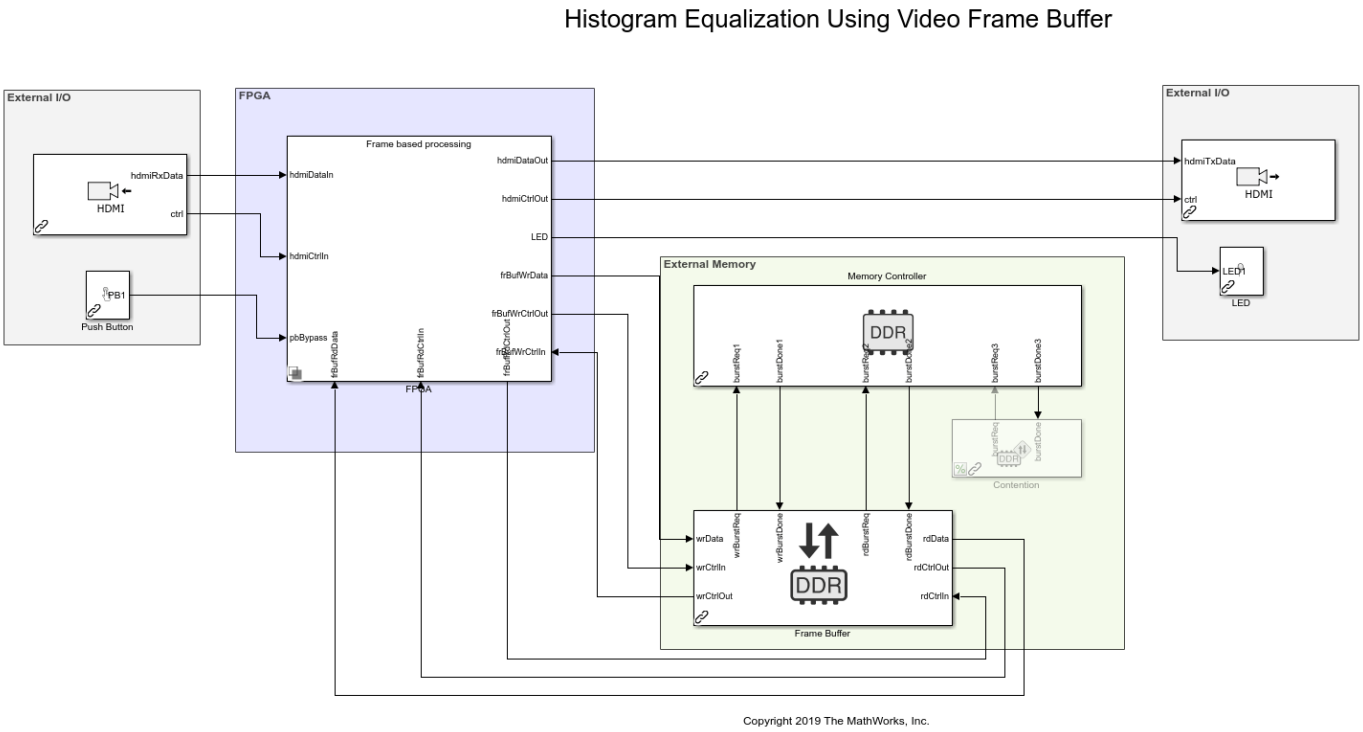
- Xilinx® Zynq® ZC706 evaluation kit + FMC-HDMI-CAM mezzanine card

Design Task and System Requirements

Consider an application involving continuous streaming of video data through the FPGA. In the top model soc_histogram_equalization_top the FPGA calculates the histogram of the incoming video stream, in the 'FPGA' subsystem, while streaming the same video stream to external memory for storage. Once the histogram has been calculated and accumulated across the entire video frame, a synchronization signal is toggled to trigger the read back of the stored frame from external memory. The accumulated histogram vector is then applied to the video stream read back from external memory to perform the equalization algorithm. The external memory frame buffer is modeled using the 'Memory Channel' block in `AXI4-Stream Video Frame Buffer` mode.

The 'HDMI Input' block reads a video file and provides video data and control signals to downstream FPGA processing blocks. Video data is in YCbCr 4:2:2 format, and the control signals are in the `pixel control bus` format. The 'HDMI Output' block reads video data and control signals, in the same format as output by the 'HDMI Input' block, and provides a visual output using the Video Display block.

The Push Button block enables bypassing of the histogram equalization algorithm, routing the unprocessed output from the external memory frame buffer to the output.



There are a number of requirements to consider when designing an application that interfaces with external memory:

- **Throughput**: What is the rate that you need to transfer data to/from memory to satisfy the requirements of your algorithm? Specifically for vision applications, what is the frame-size and frame-rate that you must be able to maintain?
- **Latency**: What is the maximum amount of time that your algorithm can tolerate between requesting and receiving data? For vision applications, do you need a continuous stream of data, without gaps? Are you able to buffer samples internal to your algorithm in order to prevent data loss when access to the memory is blocked?

For this histogram equalization example, we have defined the following requirements:

- Throughput must be sufficient to maintain a 1920x1080p video stream at 60 frames-per-second.
- Latency must be sufficiently low so as not to drop frames.

With the above throughput requirement, we can calculate the value that is required for the frame buffer:

$$1920 \times 1080 \times 60 = 124.416\,\mathrm{Msps}$$

As the video format is YCbCr 4:2:2, we require 2 bytes-per-pixel (BPP), this equates to a throughput requirement of

$$2 \times 124.416 = 248.832\,\mathrm{MB/s}$$

Because the algorithm must both write and read the video data to/from the external memory, this throughput requirement must be doubled, for a total throughput requirement of

$$2 \times 248.832 = 497.664\,\mathrm{MB/s}$$

Design Using SoC Blockset

In general, your algorithm will be a part of a larger SoC application. In such applications, it is likely that there will be other algorithms also requiring access to external memory. In this scenario, you must consider the impact of other algorithm's memory accesses on the performance and requirements of your algorithm. Assuming that your algorithm shares the memory channel with other components, you should consider the following:

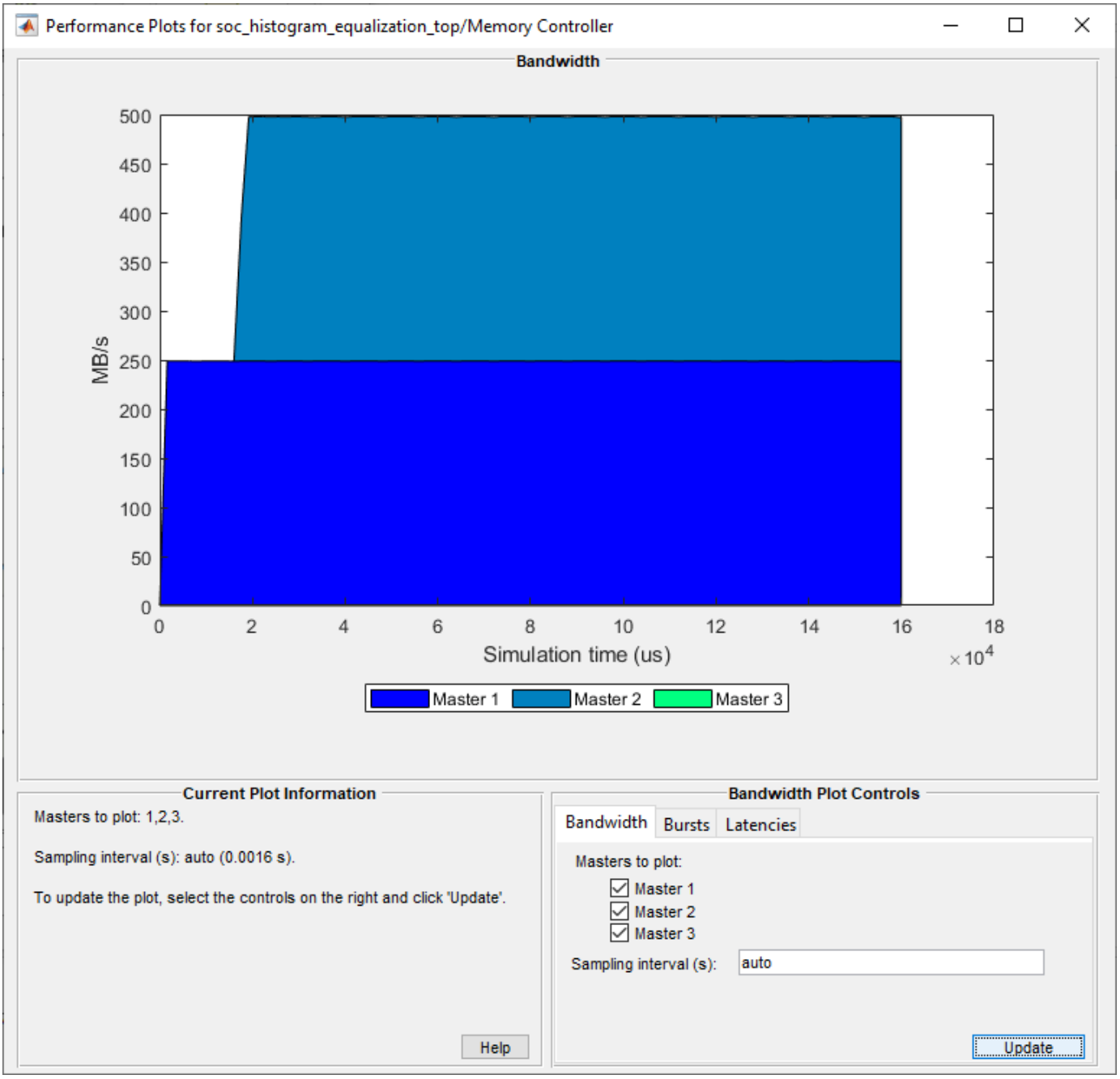
- What is the total available memory bandwidth in the SoC system?
- How will your algorithm adapt to shared memory bandwidth?
- Can your algorithm tolerate an increased read/write latency?

By appropriate modeling of additional memory consumers in the overall application, you can systematically design your algorithm to meet your requirements in situations where access to the memory is not exclusive to your algorithm.

To avoid modeling of all memory readers and writers in the overall system, you can use 'Memory Traffic Generator' blocks to consume read/write bandwidth in your system by creating access requests. In this way, you can simulate additional memory accesses within your system without explicit modeling.

Modeling Additional Memory Consumers

Simulate the system without additional memory consumers and view the memory performance plot from the 'Memory Controller' block.



Here, the memory masters are as follows:

1. Master 1: Frame Buffer write
2. Master 2: Frame Buffer read
3. Master 3: Contention (Memory Traffic Generator) (commented out)

Note that both active masters are consuming 248.8 MB/s of memory bandwidth.

More Memory Consumers: Consider that your algorithm is part of a larger system, and a secondary algorithm is being developed by a colleague or third-party. In this scenario, the secondary algorithm

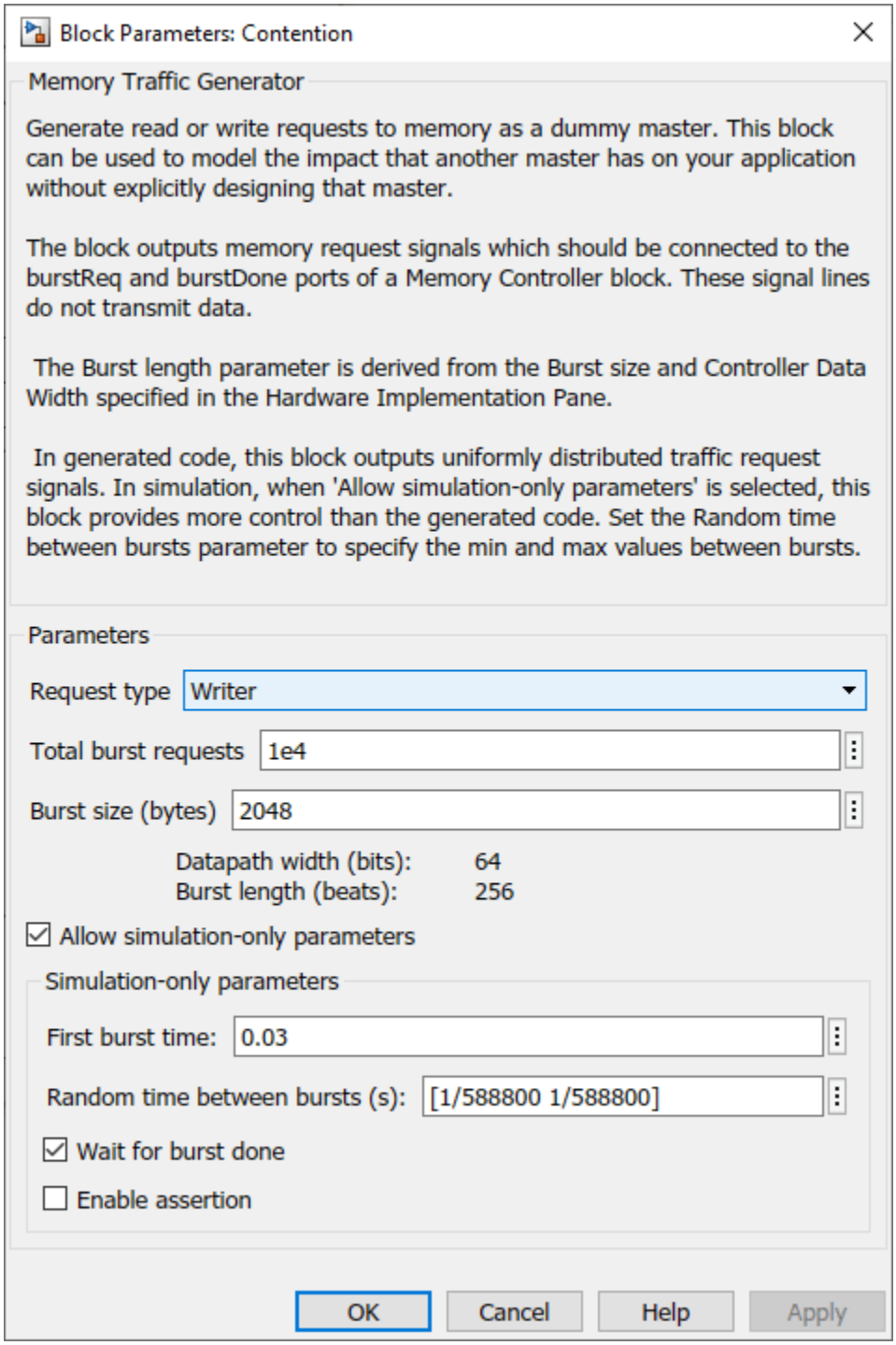
will be developed separately for the interest of time and division of work. Rather than combine the two algorithms into a single simulation, you can model the memory access of the secondary algorithm using a Memory Traffic Generator, and simulate the impact, if any, that it will have on your algorithm.

For example, assume that you are provided with the following memory requirements for the secondary algorithm:

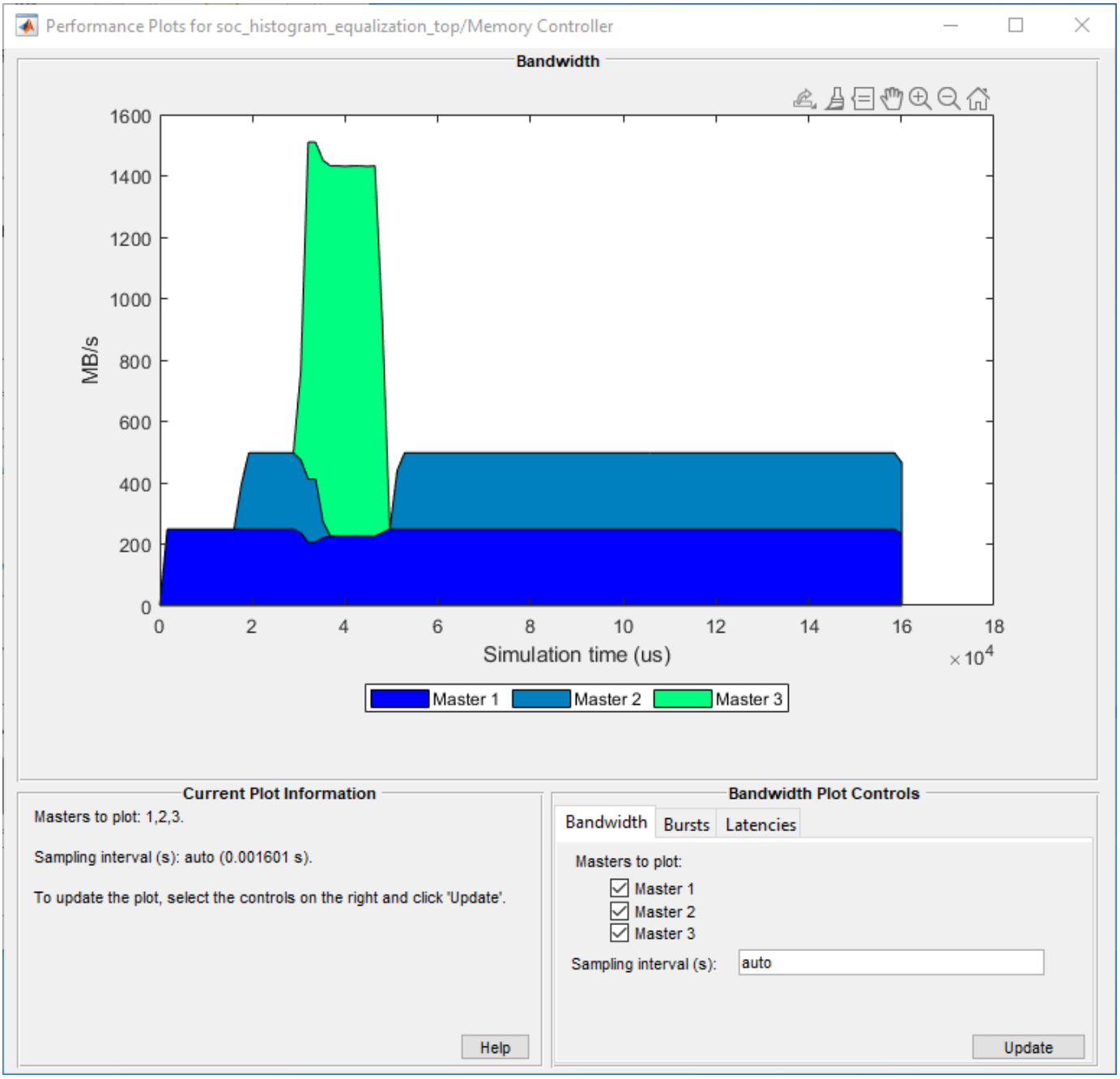
- Throughput: 1150 MB/s

Given that the primary algorithm consumes ~500 MB/s of the memory bandwidth, and the total available memory bandwidth is 1600 MB/s, we know that the total bandwidth requirement for our system exceeds the total available bandwidth by ~50 MB/s.

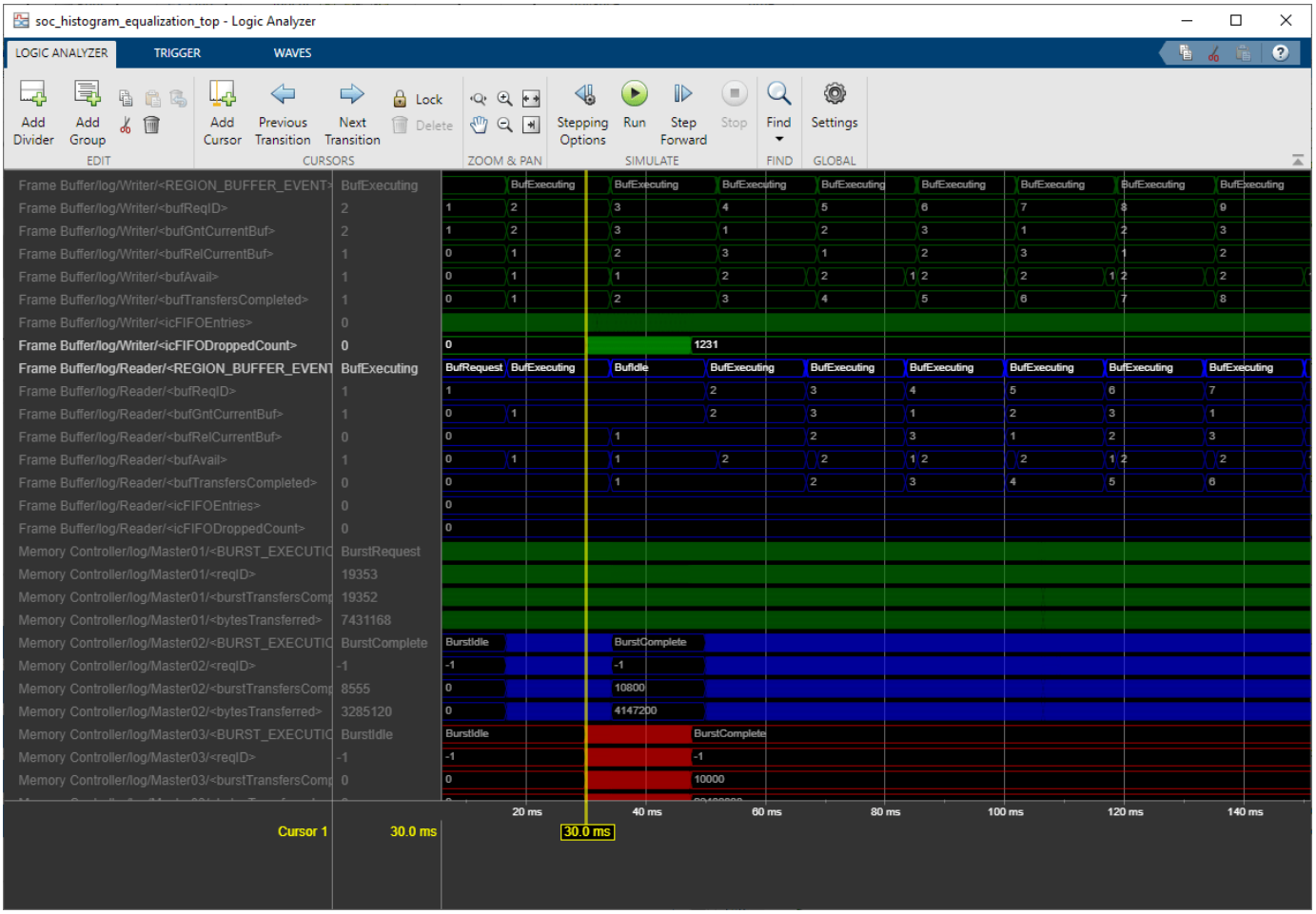
To enable the modeling of the secondary algorithm memory access, uncomment the `Contention` Memory Traffic Generator block. The block mask settings are shown below.



Simulating the system with the secondary algorithm's memory accesses, results in the following Memory Bandwidth Usage plot.



As you can see, at around 0.03s - when the secondary algorithm begins memory access requests, the other masters do not achieve their required throughput. Looking at the logic analyzer waveform, we can see this manifested as dropped buffers for the Frame Buffer write master and the idle state for the Frame Buffer read master.



Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- SoC Blockset Support Package for Xilinx Devices. For more information about the support package, see “SoC Blockset Supported Hardware”

To implement the model on a supported SoC board use the SoC Builder application. Open the mask of 'FPGA' subsystem and set model variant to 'Pixel based processing'.

Comment out the 'Contention' block.

Click, 'Configure, Build, & Deploy' button in the toolstrip to open SoC Builder

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.

- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
- Click 'Next' to 'Load Bitstream' screen.

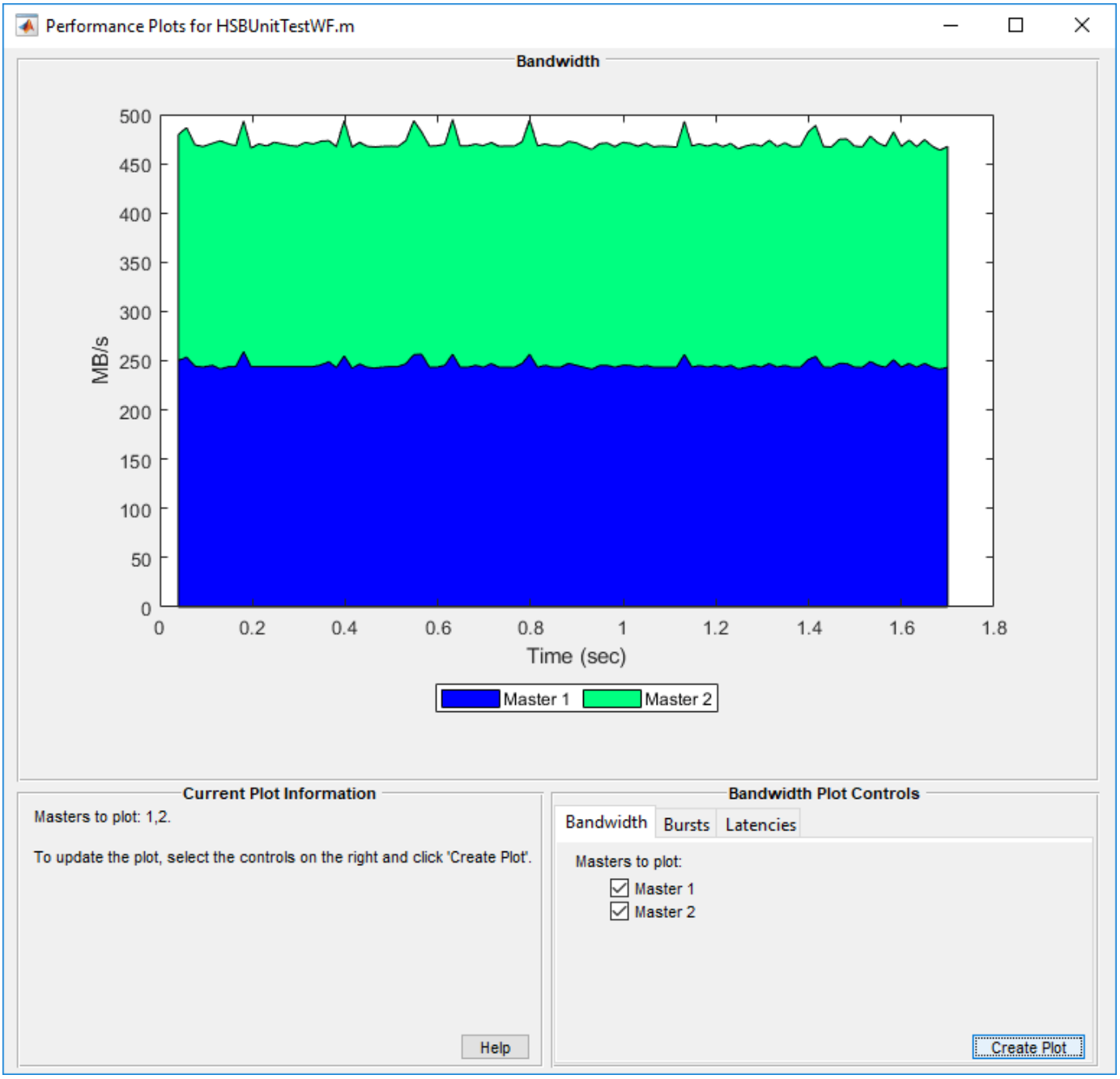
The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load and Run' button to load pre-generated bitstream and run the model on SoC board

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
         'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
         'soc_histogram_equalization_top-zc706.bit'), './soc_prj');
```

Now the model is running on hardware. To get the memory bandwidth usage in hardware, execute the following aximaster test bench for soc_histogram_equalization_top_aximaster.

The following figure shows the Memory Bandwidth usage when the application is deployed on hardware.



Summary

You designed a video application with real time HDMI I/O and frame buffering in external memory. You explored effects of other consumers of memory on overall bandwidth. You used SoC Builder to implement the model on hardware and verify the design.

Streaming Data from Hardware to Software

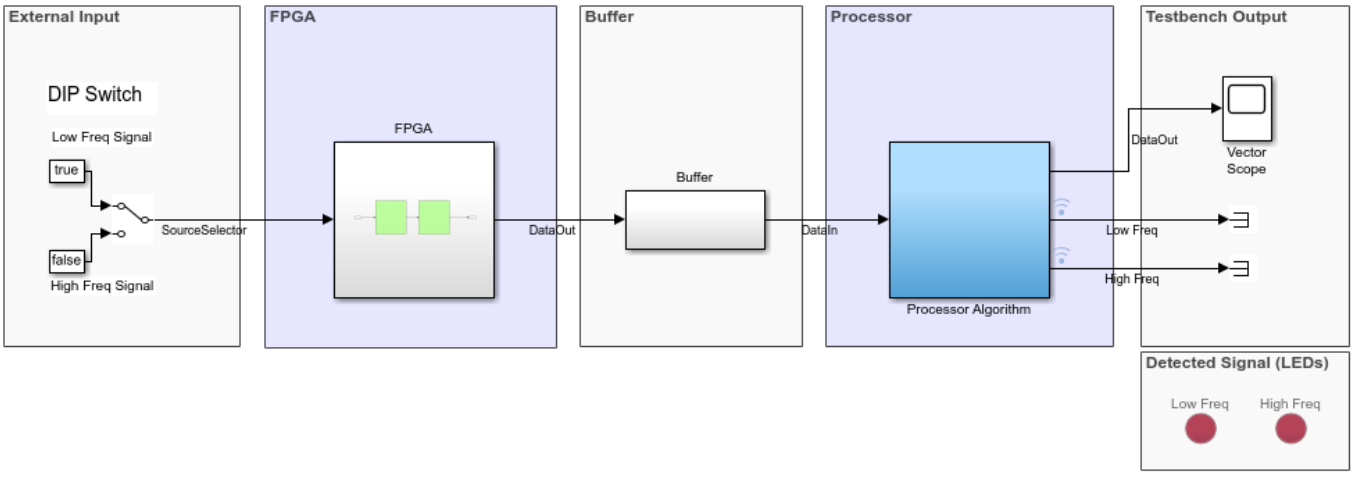
This example presents a systematic approach to design the data-path between hardware logic (FPGA) and embedded processor using SoC Blockset. Applications are often partitioned between hardware logic and embedded processor on a system-on-chip (SoC) device to meet throughput, latency and processing requirements. You will design and simulate the entire application comprising of FPGA & processor algorithms, memory interface and task scheduling to meet the system requirements. You will then validate the design on hardware by generating code from the model and implementing on a SoC device.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Xilinx Zynq UltraScale™+ RFSoC ZCU111 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

Design Task and System Requirements

Consider an application that continuously process data on the FPGA and the embedded processor. In this example, the FPGA algorithm filters the input signal and streams the resulting data to the processor. In the implementation model soc_hwsw_stream_implementation, the *Buffer* block represents the transfer of data from FPGA to processor. The processor operates on the buffered data and classifies the data as either high or low frequency in the *Processor Algorithm* subsystem. FPGA generates a test data of either low or high frequency sinusoid based on the DIP switch setting in *Test Data Source* subsystem.



The application has following performance requirements:

- Throughput: 10e6 samples per second

- Maximum latency: 100ms
- Samples dropped: < 1 in 10000

Challenges in Designing Datapath

The FPGA processes data sample by sample while the processor operates on a frame of data at a time. The data is transferred asynchronously between FPGA and processor, and the duration of software task can vary for each execution. Therefore, a queue is needed to hold the data between FPGA and processor to prevent data loss. This queue is implemented in two stages, one as a FIFO of bursts of data samples in FPGA memory and other as a series of frame buffers in external memory. You will need to set three parameters related to the queue: frame size (number of samples in a frame of data), number of frame buffers and FIFO size (number of bursts of samples in FIFO).

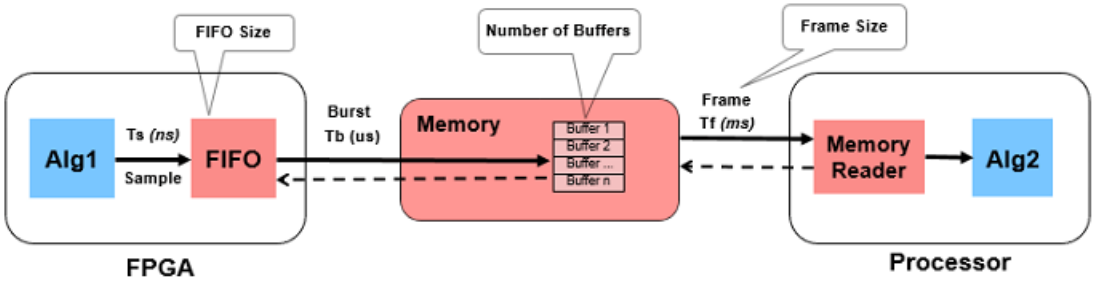


Figure 1

These design parameters affect performance and resource utilization. Increasing the frame size allows more time for software task execution and to meet throughput requirements at the cost of increasing latency. Typically, you set these parameters only when you are ready to implement on hardware, which presents the following challenges:

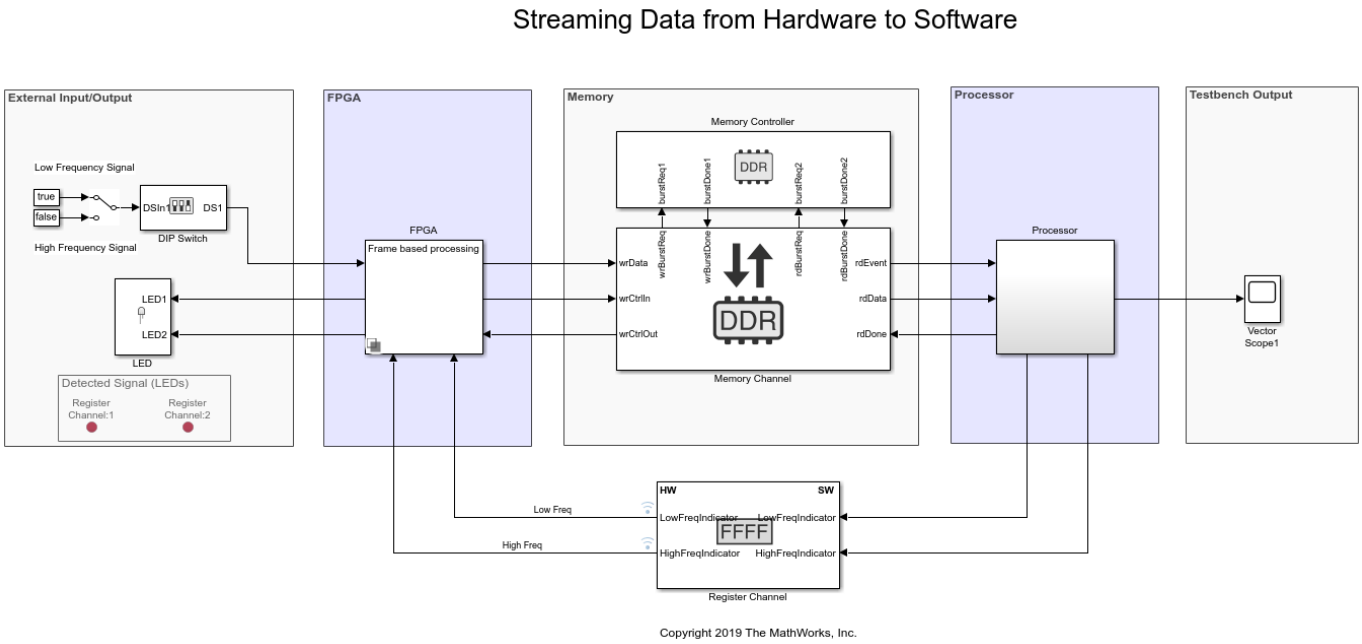
- It is difficult to debug issues like dropping of samples in hardware due to lack of visibility.
- It is difficult to design your application efficiently without first evaluating the effects of hardware interfaces. It can take many design-implementation iterations as you can assess performance only via implementation on hardware.
- It is difficult to optimize design since performance and cause-effect relationships are difficult to determine through implementation.

Ideally you want to account for these hardware effects while you are developing the application at design time, before implementing and running on hardware. One way to satisfy these requirements is to simulate the hardware effects, at design time. If you can simulate the variation in task durations, utilization of memory buffers/FIFOs and external memory transfer latencies, you can evaluate their effects on application design and implement the proven design on hardware. SoC Blockset allows you to simulate these effects so you can evaluate the performance of the deployed application before running on hardware.

Design Using SoC Blockset

Create an SoC model soc_hwsw_stream_top from the implementation model soc_hwsw_stream_implementation using the “Stream from FPGA to Processor Template” on page 2-14. The top model includes FPGA model soc_hwsw_stream_fpga and processor model soc_hwsw_stream_proc instantiated as model references. The top model also includes Memory

Channel and Memory Controller blocks which model shared external memory between FPGA and processor. These were earlier modeled using buffer block in the implementation model. To improve simulation performance, FPGA algorithm is also modeled for Frame-based processing soc_hwsw_stream_fpga_frame and is included as model variant subsystem at the top level. You can select the model to run in Frame-based or Sample-based processing by selecting from the mask of FPGA subsystem.



Design to Meet Latency Requirement : Latency in the datapath from FPGA to processor comprises of the latency through the FPGA logic and the time for data transfer from FPGA to processor through memory channel. In this example, the FPGA clock is 10MHz and the latency is on the order of nanoseconds. This is negligible in comparison with latency within the memory channel, which is on the order of milliseconds. Therefore, let us focus on designing for latency for data transfer in the following manner.

Begin with a few potential frame sizes and calculate Frame period for each frame size in Table -1. Frame period is the time between two consecutive frames from FPGA to processor. For this example, FPGA output sample time is 10e-6 as a valid data is output every 100 clock cycles from the FPGA.

$$FramePeriod = Framesize * FPGAOutputSampleTime$$

Latency of the memory channel is due to time elapsed by samples in the queue of frame buffers and FPGA FIFO. Let us size FPGA FIFO equivalent to one frame buffer. To stay within the maximum latency requirement, calculate the number of frame buffers for each frame size as per:

$$(NumFrameBuffers + 1) * FramePeriod <= MaxLatency$$

Maximum latency allowed for this example is 100 ms. Since the number of buffers account for maximum latency requirement, for all the cases in Table-1, latency requirement is met. The maximum number of frame buffers allowed by the software DMA driver is 64. A minimum of 3 frame buffers is needed in external memory for data transfer. While one of the frame buffers is written by FPGA, the other frame buffer is read by processor. Therefore, Case #8-10 from the table below are rejected as they violate the minimum buffer requirement.

#	Frame Size	Frame period (ms)	Number of buffers	Meets or Violates requirements
1	5	0.05	1999	
2	100	1	99	
3	800	8	11	
4	1000	10	9	
5	1600	16	5	
6	2000	20	4	
7	2400	24	3	
8	8000	80	<1	Violates min buffers req
9	18000	180	<1	Violates min buffers req
10	30000	300	<1	Violates min buffers req

Table -1

To visualize the latency, simulate the model and open *Memory Channel* block, go to *Performance* tab and click on *View performance plots* . Select all the latency options under *Plot Controls* and click *Create Plot* . As captured in Figure - 2, you will notice that the composite latency meets the < 100 ms requirement.

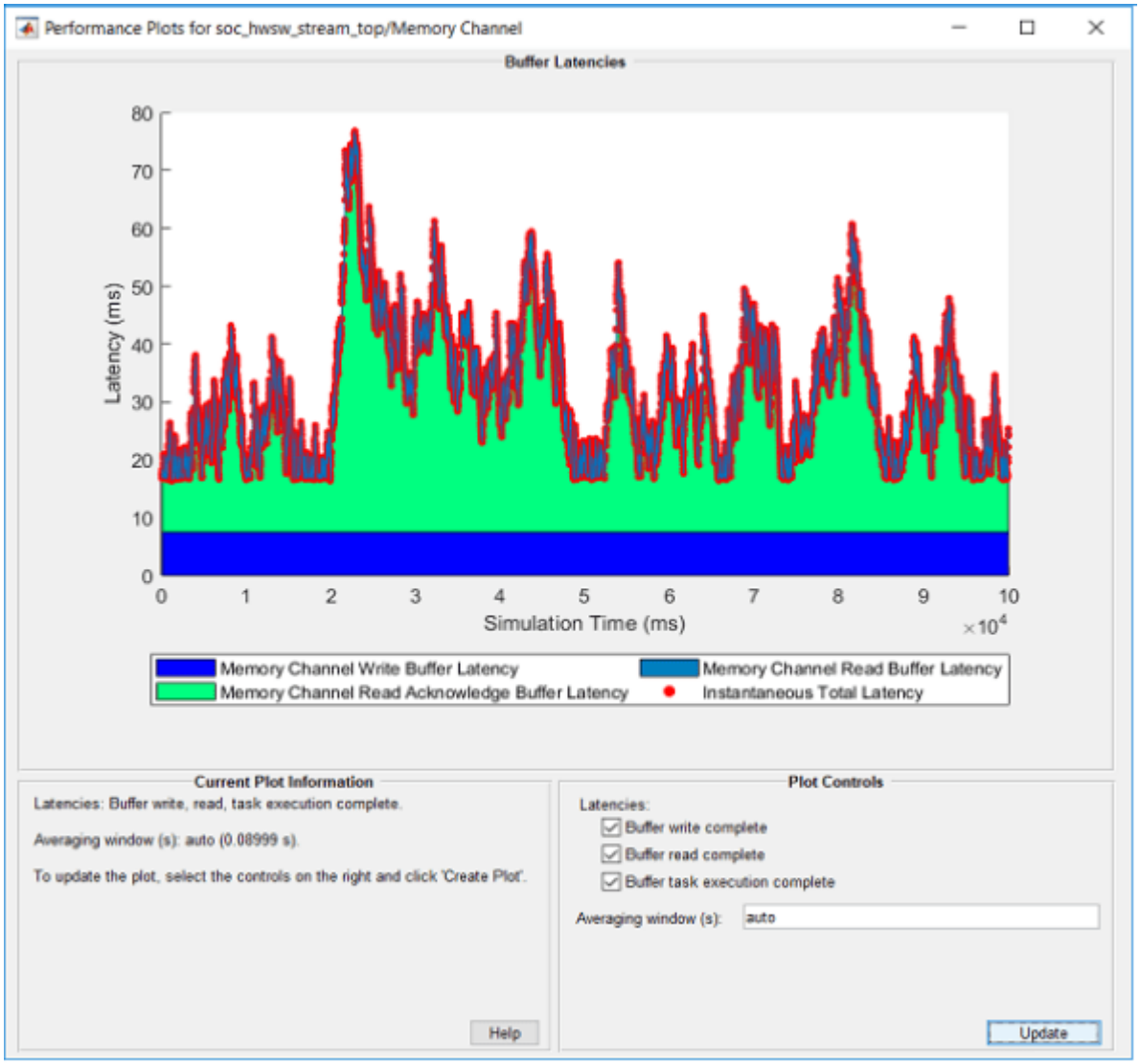


Figure - 2

Design to Meet Throughput Requirement : On average, the software tasks processing must complete within a frame period, as otherwise, task will overrun leading to dropping of data and violate the throughput requirement. i.e.

$$FramePeriod > MeanTaskDuration$$

There are various ways of obtaining mean tasks durations corresponding to frame sizes for your algorithm, which are covered in “Task Execution” on page 7-78 Example. Mean task durations for various frame sizes are captured in Table 2.

#	Frame Size	Frame period (ms)	Number of buffers	Mean Task Duration (ms)	Meets or Violates requirements
1	5	0.05	1999	0.059	Violates throughput
2	100	1	99	1.06	Violates throughput
3	800	8	11	7.858	
4	1000	10	9	9.61	
5	1600	16	5	15.3	
6	2000	20	4	19.067	
7	2400	24	3	22.812	
8	8000	80	<1	76.56	Violates min buffers req
9	18000	180	<1	175.23	Violates min buffers req
10	30000	300	<1	289.52	Violates min buffers req

Table -2

To simulate the model with the parameters corresponding to rows (#2-#7) in the table use the function `set_hwsw_stream_set_parameters` function with row # as an argument. Set the model parameters for row # 2 as below:

```
soc_hwsw_stream_set_parameters(2); % row # 2
```

Since the Mean Task Duration of 1.06 ms is more than the Frame Period of 1.0 ms, the data is dropped in the memory channel. Open Logic Analyzer and notice that signal *icFIFODroppedCount* is increasing throughout the simulation as captured in Figure 3, indicating accumulating amount of dropped data.

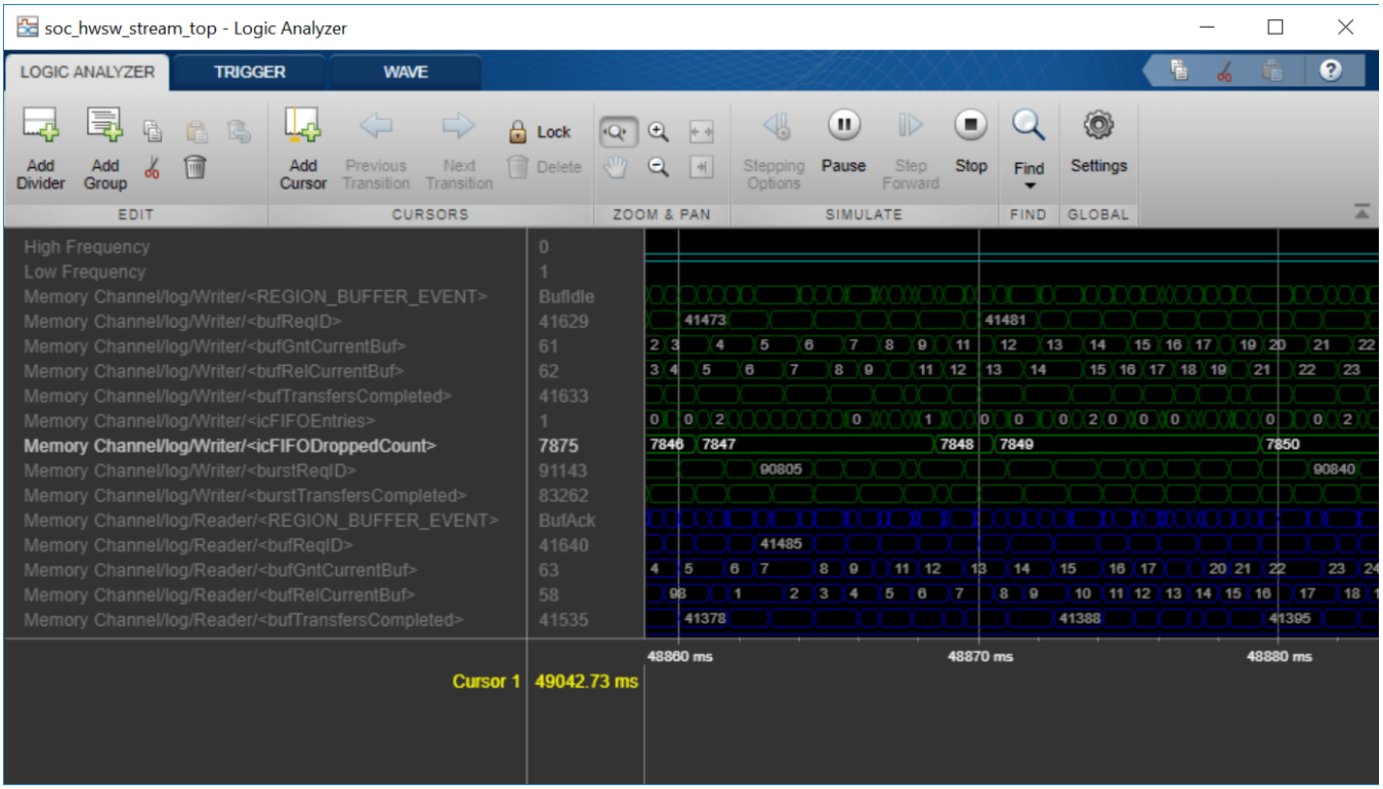


Figure - 3

Since data is dropped during transfer from FPGA to processor through memory, this is reflected as a drop in throughput. Open *Memory controller* block, go to *Performance* tab and click on *Plot data throughput* button under Performance tab to see the memory throughput plot as in figure 4. Note that the throughput is less than the required 0.4 MBps. Since the FPGA output data sample time is 10e-6 and each sample is 4 bytes wide, the required streaming throughput for the system is 4 bytes/10e-6 = 400 KBps.

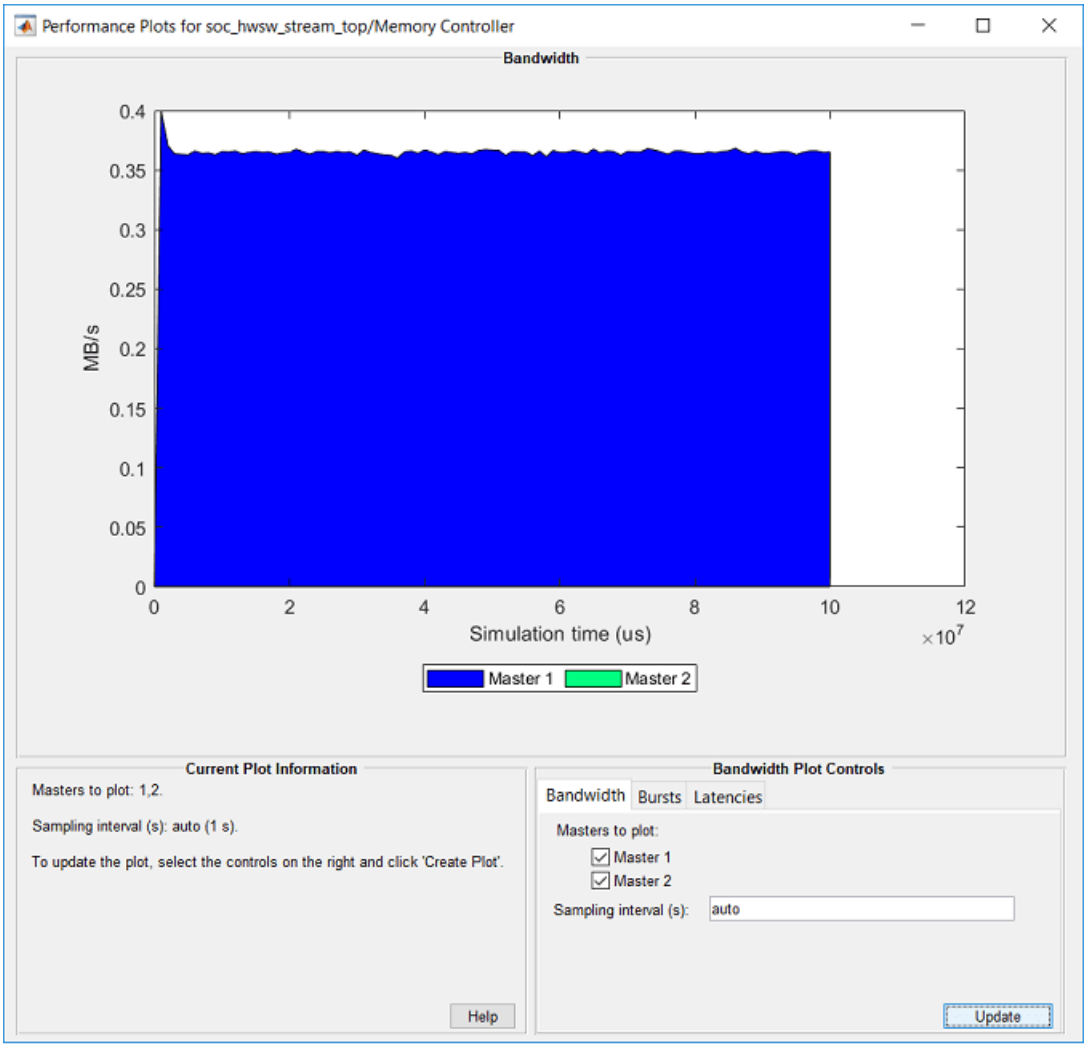


Figure - 4

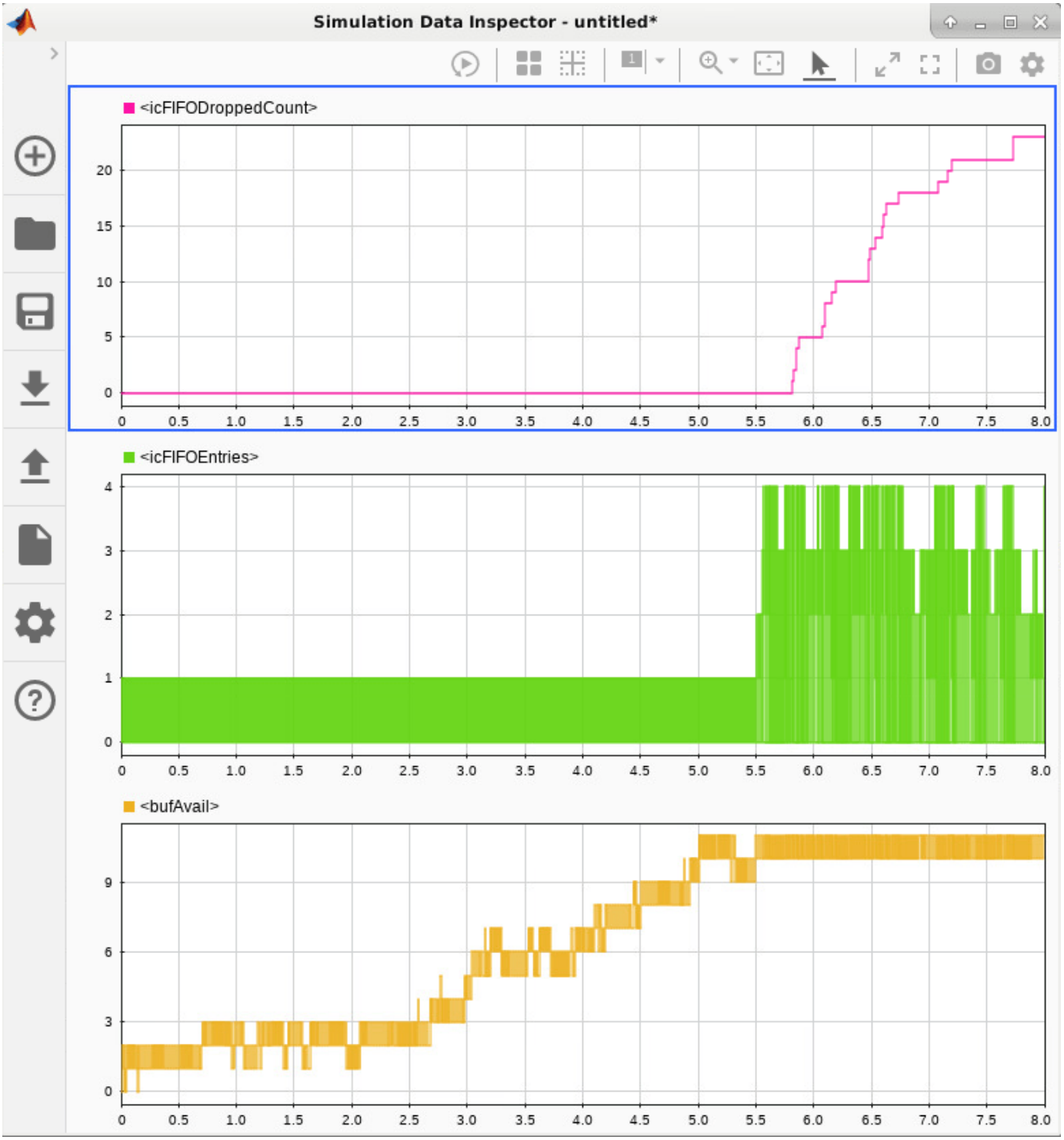
Design to Meet Drop Samples Requirement : Since the task durations can vary for many reasons like different code execution paths and variation in OS switching time, it is possible that data is dropped in the memory channel. Specify the mean task execution duration and statistical distribution for task durations in the mask of Task Manager block. Size the FIFO equivalent to one frame buffer. Set the FIFO burst size to 16 Bytes and calculate the FIFO depth:

$$FIFO_depth = FrameSize/FIFOBurstSize$$

Now, simulate the model for 100 sec (10e6 samples at 10e-6 samples per second) for cases # 3-7. Open the Logic analyzer and note the number of samples dropped on signal *icFIFODroppedCount*.

```
soc_hwsw_stream_set_parameters(3); % set the model parameters for #3
```

Open Simulation Data Inspector and add signals from memory channel as shown in Figure 5 below. Note that as buffers usage (signal *buffAvail*) increase to the maximum 11, the FIFO usage (signal *isFIFOEntries*) begin to increase. When FIFO is completely used, the samples get dropped (signal *isFIFODroppedCount*)



The results of simulation for all the cases #3-7 and resultant sample dropped per 10000 are tabulated in Table 3.

#	Frame Size	Frame period (ms)	Number of buffers	Mean Task Duration (ms)	Avg Samples dropped per 10000	Meets or Violates requirements
1	5	0.05	1999	0.059		Violates throughput
2	100	1	99	1.06		Violates throughput
3	800	8	11	7.858	172.6	Violates drop samples
4	1000	10	9	9.61	0	Meets all requirements
5	1600	16	5	15.3	1	Meets all requirements
6	2000	20	4	19.067	2.25	Violates drop samples
7	2400	24	3	22.812	3.9	Violates drop samples
8	8000	80	<1	76.56		Violates min buffers req
9	18000	180	<1	175.23		Violates min buffers req
10	30000	300	<1	289.52		Violates min buffers req

Table - 3

The highlighted entries (rows #4 and #5) are valid design choices since they meet throughput, latency and drop samples requirement.

Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- Embedded Coder®
- SoC Blockset Support Package for Xilinx Devices, or
- SoC Blockset Support Package for Intel Devices

For more information about support packages, see “SoC Blockset Supported Hardware”

To implement the model on a supported SoC board use the SoC Builder tool. Open the mask of 'FPGA' subsystem and select model variant to 'Sample based processing'. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board. To open SoC Builder click, 'Configure, Build, & Deploy' button in the toolstrip and follow these steps:

- Select **Build Model** on **Setup** screen. Click **Next**.
- Click **Next** on **Review Task Map** screen.
- Click **View/Edit Memory Map** to view the memory map on **Review Memory Map** screen. Click **Next**.
- Specify project folder on **Select Project Folder** screen. Click **Next**.
- Select **Build, load and run** on **Select Build Action** screen. Click **Next**.
- Click **Validate** to check the compatibility of model for implementation on **Validate Model** screen. Click **Next**.
- Click **Build** to begin building of the model on **Build Model** screen. An external shell will open when FPGA synthesis begins. Click **Next**.
- Click **Test Connection** on **Connect Hardware** screen to test the connectivity of host computer with SoC board. Click **Next** to go to **Run Application** screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click **Load and Run** button to load pre-generated bitstream and run the model on SoC board

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
        'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
        'soc_hwsw_stream_top-zc706.bit'), './soc_prj');
```

While the application is running on hardware, toggle the DIP switch on your board to change the test data from 'low' to 'high' frequency and notice the blinking of corresponding LED on the board. You can also read the samples dropped count in the model running on external mode. Thus, you verify that your implementation from SoC Blockset model matches the simulation and meets the requirements.

Implementation on other boards: To implement the model on a supported board other than Xilinx® Zynq® ZC706 evaluation kit board, you must first configure the model to the supported board and set the example parameters as below.

- On the **Hardware** tab, click **Hardware Settings** to open the **Configuration Parameters** window.
- In the **Hardware Implementation** tab, select your board from **Hardware board** drop-down list on both top and processor model.
- Navigate to **Target hardware resources > FPGA design (top level)** tab and enable **Include MATLAB as AXI Master IP for host-based interaction** and set **IP core clock frequency (MHz)** to 10 MHz.
- Navigate to **Target hardware resources > FPGA design (debug)** tab and enable **Include AXI Interconnect monitor**.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the *copyfile* command to match the bitstream corresponding to your board. In case of Altera Arria® 10 SoC development kit and Altera Cyclone® V SoC development kit use below *copyfile* command corresponding to your board. In case of Altera Arria® 10 SoC development kit, copy '.periph.rbf' and '.core.rbf' files.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
         'supportpackages','intelsoc','intelsocexamples','bitstreams',...
         'soc_hwsw_stream_top-c5soc.rbf'), './soc_prj');
```

The following are the available pre-generated bitstream files:

- 'soc_hwsw_stream_top-zc706.bit'
- 'soc_hwsw_stream_top-zedboard.bit'
- 'soc_hwsw_stream_top-zcu102.bit'
- 'soc_hwsw_stream_top-XilinxZynqUltraScale_RFSoCZCU111EvaluationKit.bit'
- 'soc_hwsw_stream_top-c5soc.rbf'
- 'soc_hwsw_stream_top-a10soc.periph.rbf'
- 'soc_hwsw_stream_top-a10soc.core.rbf'

In summary, this example showed you a systematic approach to design the datapath between hardware logic and embedded processor using SoC Blockset. You chose design parameters of frame

size, number of frame buffers and FIFO size to meet the system performance requirements of throughput, latency and drop samples. By simulating and visualizing the effects of these parameters on the complete model containing hardware logic, processor algorithms, external memory and processor task durations, you uncovered issues like loss of throughput, latency and dropping of samples before implementing on hardware. This workflow ensures that the design works on hardware before implementation and avoids long design-implementation iterations.

Streaming Data from Software to Hardware

This example shows how to design the data-path from an embedded processor to hardware logic (FPGA) using SoC Blockset™. Design and simulate the entire application comprising of FPGA and processor algorithms, memory interface, and task scheduling to meet the system requirements. Then, validate the design on hardware by generating code from the model and implementing it on a System-on-Chip (SoC) device.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Xilinx Zynq UltraScale™+ RFSoC ZCU111 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

Design Task and System Requirements

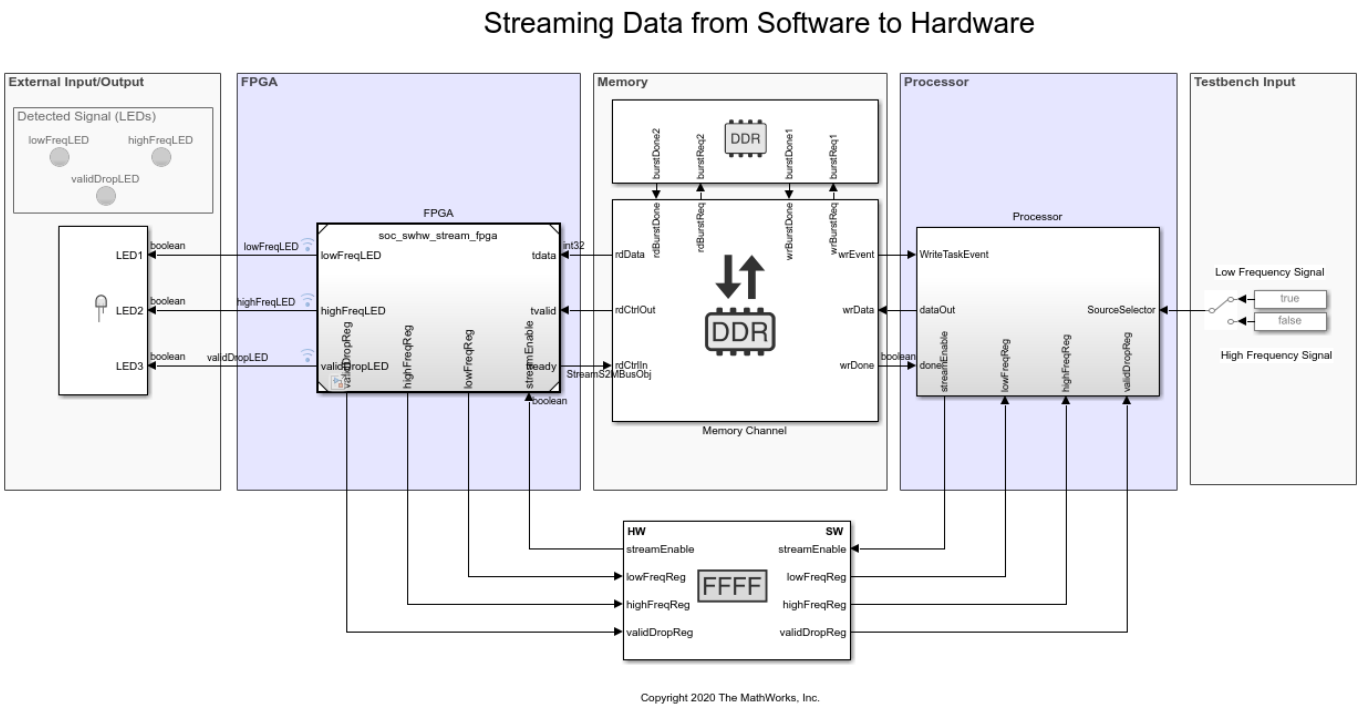
In this example, the embedded processor sends test data of either a low or high frequency sinusoid to the FPGA. The FPGA algorithm detects the frequency of the signal by filtering and lights up a light-emitting diode (LED) to indicate the detection. This example models the data-path similar to the “Streaming Data from Hardware to Software” on page 7-31 example. In this example, the data-flow is reversed as compared to the Streaming Data from Hardware to Software example.

The application has these performance requirements.

- Throughput: 10e6 samples per second
- Maximum latency: 10 ms
- Data streaming: Continuous

Design Using SoC Blockset

Create SoC model soc_swhw_stream_top using the template “Stream from Processor to FPGA Template” on page 2-18. The top model includes FPGA model soc_swhw_stream_fpga and processor model soc_swhw_stream_proc instantiated as model references. The top model also includes Memory Channel and Memory Controller blocks that model shared external memory between the FPGA and processor.



Design to Meet Latency Requirement: Begin with a few potential frame sizes and calculate the frame period for each frame size in Table-1. The frame period is the time between two consecutive frames from the FPGA to processor. For this example, the FPGA output sample time is 1/10e6, or 1e-7, as the FPGA algorithm runs at 10 MHz. The frame period is calculated as

$$FramePeriod = Framesize * FPGAOutputSampleTime$$

The latency of the memory channel is due to the time elapsed by samples in the queue of frame buffers and the FPGA FIFO. Select the FPGA FIFO size such that it is equivalent to the size of one frame buffer. To stay within the maximum latency requirement, calculate the number of frame buffers for each frame size such that:

$$(NumFrameBuffers + 1) * FramePeriod <= MaxLatency$$

The maximum latency allowed for this example is 10 ms. Calculate the maximum frame buffers for all of the cases in this table. Because the number of buffers accounts for the maximum latency requirement, all of the cases meet the latency requirement.

#	FrameSize	FramePeriod (ms)	Max NumFrameBuffers	Meets or Violates Requirements
1	100	0.01	64	
2	1000	0.1	64	
3	10000	1	9	
4	20000	2	4	
5	100000	10	<1	Violates min buffers req
6	1000000	100	<1	Violates min buffers req

Table -1

The range for number of buffers is dictated by memory architecture constraints. The maximum number of frame buffers allowed by the software Direct Memory Access (DMA) driver is 64. The minimum number of frame buffers is 3. While the processor writes one frame buffer, the FPGA reads from another frame buffer. Therefore, the range for the number of frame buffers is:

$$3 <= NumFrameBuffers <= 64$$

Case #5 and #6 violate the minimum buffer requirements.

Design to Meet Throughput Requirement: On average, the software processing must complete within a frame period. If it does not, the software task does not generate data fast enough for consumption by the FPGA, violating the throughput requirement. i.e.

$$FramePeriod > MeanTaskDuration$$

Various ways exists for obtaining mean task durations corresponding to frame sizes for your algorithm. These concepts are covered in the “Task Execution” on page 7-78 example. Mean task durations for various frame sizes are captured in the following Table-2. Because the mean task duration is greater than the calculated frame period, case #1 and #2 violate the throughput requirement.

#	FrameSize	FramePeriod (ms)	Max NumFrameBuffers	Mean Task Duration (ms)	Meets or Violates Requirements
1	100	0.01	64	0.1	Violates throughput
2	1000	0.1	64	0.3	Violates throughput
3	10000	1	9	0.6	
4	20000	2	4	1	
5	100000	10	<1	3	Violates min buffers req
6	1000000	100	<1	10	Violates min buffers req

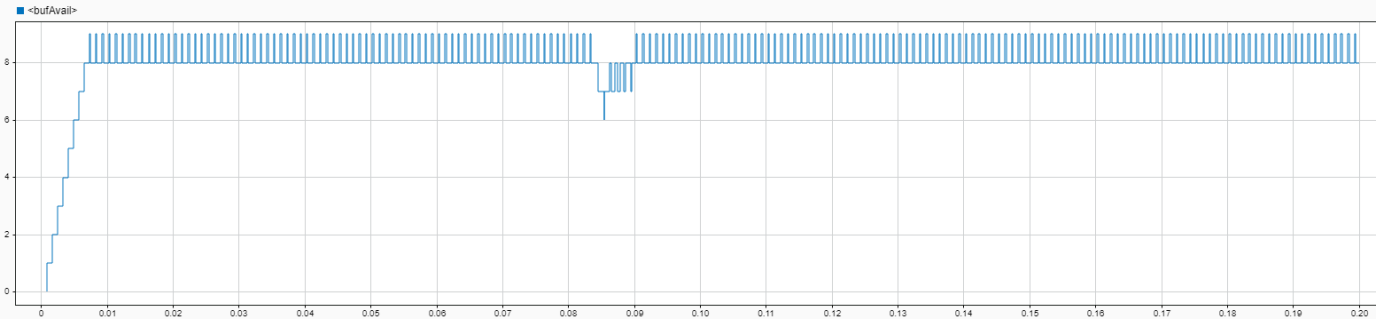
Table -2

Design to Meet Data Continuity Requirement: To meet the data continuity requirement, fill in the frame buffers in the memory (priming) before starting to stream the data. When temporary disruptions occur due to processor execution, the data is available from the preciously filled frame buffers filled earlier. Priming is accomplished by designing software logic under the

`soc_swhw_stream_proc/Writer/Priming` subsystem, which generates a `streamEnable` command for the FPGA to start streaming data after the memory is almost full.

Because the task durations can vary for many reasons such as different code execution paths and variation in OS switching time, the software task might not deliver data to the FPGA through shared memory on time. This can result in loss of data continuity. Specify the mean task execution duration and its statistical distribution in the mask of the Task Manager block, and then simulate to verify if this requirement is met.

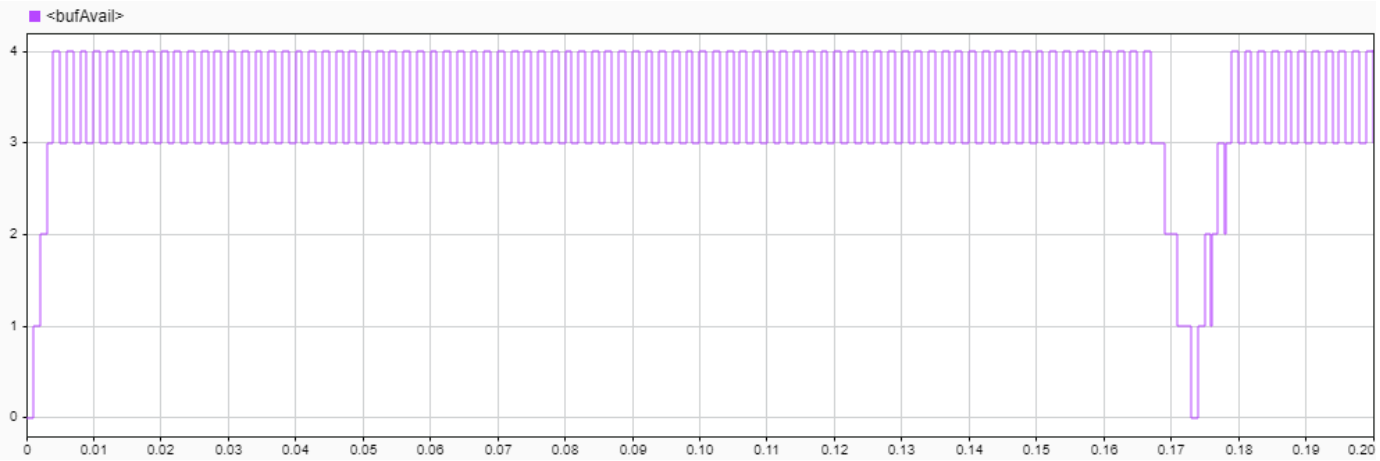
By default, the model is configured with case #3 parameters by default. Simulate the top model, and Click **Data Inspector** from the **Simulation** tab. Add `bufAvail` signals on the top view. In this case, the available software buffer signal does not drop to zero, and `validDropLED` in the top model does not light up, indicating that the data is streamed continuously.



Set the model for case # 4 as in this code and simulate the model again.

```
soc_swhw_stream_set_parameters(4); % row # 4
```

In this case, the available software buffers drop to zero, and the `validDropLED` in the top model lights up.



Case #4 violates the data continuity requirement. Case #3 is proven to be the optimal case that meet all of the design requirements. This Table-3 shows the updated results.

#	FrameSize	FramePeriod (ms)	Max NumFrameBuffers	Mean Task Duration (ms)	Meets or Violates Requirements
1	100	0.01	64	0.1	Violates throughput
2	1000	0.1	64	0.3	Violates throughput
3	10000	1	9	0.6	Meets all requirements
4	20000	2	4	1	Violate data continuity
5	100000	10	<1	3	Violates min buffers req
6	1000000	100	<1	10	Violates min buffers req

Table -3

Run `soc_swhw_stream_set_parameters(3)` command to restore the model with case #3 parameters before deployment of the model.

Implement and Run Model on Hardware

These products are required for this section:

- HDL Coder™
- Embedded Coder®
- SoC Blockset Support Package for Xilinx Devices, or SoC Blockset Support Package for Intel Devices

For more information about support packages, see “SoC Blockset Supported Hardware”.

To implement the model on a supported SoC board use the SoC Builder tool. By default, the model is implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board. To open SoC Builder click, **Configure, Build, & Deploy** button in the toolstrip and follow these steps:

1. Select **Build Model** on the **Setup** screen. Click **Next**.
2. Click **Next** on the **Review Task Map** screen.
3. On **Review Memory Map** screen, click **View/Edit Memory Map** to view the memory map. Click **Next**.
4. Specify the project folder on the **Select Project Folder** screen. Click **Next**.
5. Select **Build, load for external mode** on the **Select Build Action** screen. Click **Next**.
6. On **Validate Model** screen, click **Validate** to check the compatibility of model for implementation. Click **Next**.
7. On **Build Model** screen, click **Build** to begin building of the model. An external shell opens when FPGA synthesis begins. Click **Next**.
8. Click **Test Connection** on the **Connect Hardware** screen to test the connectivity of the host computer with SoC board. Click **Next** to go to the **Run Application** screen.

The FPGA synthesis can take more than 30 minutes to complete. To save time, you can use the provided pregenerated bitstream by following these steps.

1. Close the external shell to terminate synthesis.
2. Copy pregenerated bitstream to your project folder by running this `copyfile` command below.

3 Click **Load and Run** to load the pregenerated bitstream and open the generated software model **soc_swhw_stream_top_sw**.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc','supportpackages
```

After loading the bitstream, run the generated software model **soc_swhw_stream_top_sw** in external mode by clicking **Monitor and Tune** on the toolstrip. This will light up LED2 on the board, indicating the detection of high frequency signal by the FPGA. To change the frequency of the sinusoid signal dynamically at run-time, replace the SourceSelector terminator block with a Constant block, and then run the model again in external mode. Modify the constant value from 0 to 1 to change the frequency of signal from a high to low respectively.

Implementation on other boards: To implement the model on a supported board other than ZC706, first configure the model to the supported board, and then set the example parameters as below.

- On the **Hardware** tab, click **Hardware Settings** to open the **Configuration Parameters** window.
- In the **Hardware Implementation** tab, select your board from **Hardware board** drop-down list on both top and processor model.
- Navigate to **Target hardware resources > FPGA design (top level)** tab and set **IP core clock frequency (MHz)** to 10 MHz.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the `copyfile` command to match the bitstream corresponding to your board. In case of Altera Arria® 10 SoC development kit and Altera Cyclone® V SoC development kit use below `copyfile` command corresponding to your board. In case of Altera Arria® 10 SoC development kit, copy '.periph.rbf' and '.core.rbf' files.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc','supportpackages
```

The following are the available pre-generated bitstream files:

- 'soc_swhw_stream_top-zc706.bit'
- 'soc_swhw_stream_top-zedboard.bit'
- 'soc_swhw_stream_top-zcu102.bit'
- 'soc_swhw_stream_top-XilinxZynqUltraScale_RFSoCZCU111EvaluationKit.bit'
- 'soc_swhw_stream_top-c5soc.rbf'
- 'soc_swhw_stream_top-a10soc.periph.rbf'
- 'soc_swhw_stream_top-a10soc.core.rbf'

In summary, this example showed how to design the data-path from processor to FPGA for continuous streaming. You designed and modeled the behavior using SoC Blockset and went through the workflow required to implement it on an SoC device.

Triggering Software Tasks by FPGA Interrupts

This example shows how to model an algorithm partitioned between hardware and software. The hardware IP is implemented in FPGA fabric, and triggers a software task implemented in the embedded processor. Design, simulate, and implement a complete design on SoC hardware.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Xilinx Zynq UltraScale™+ RFSoC ZCU111 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

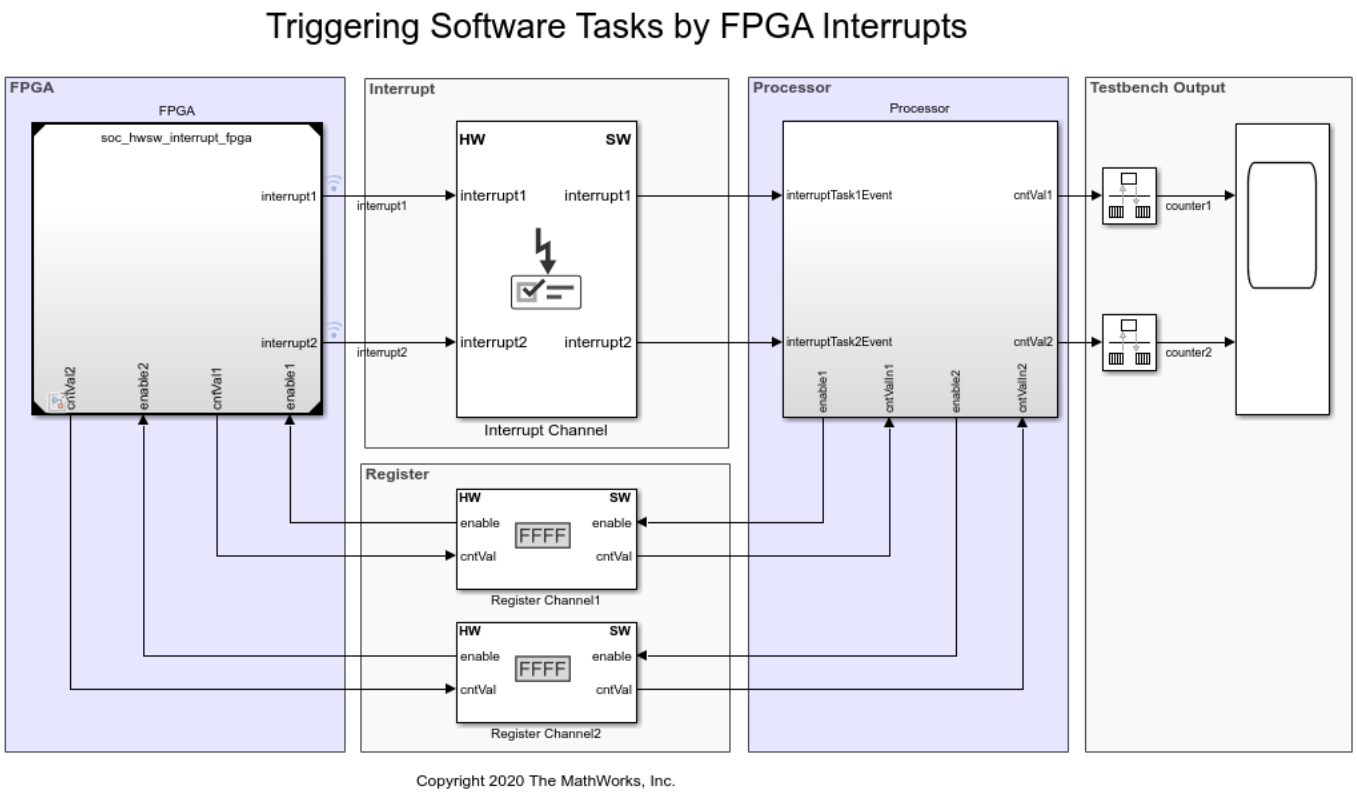
Introduction

Many System-on-Chip (SoC) applications require the hardware device to trigger an asynchronous task on the processor. This functionality can be useful when implementing low-latency control loops that span between the hardware and software. It is also useful in handling urgent hardware requests by the software. For such designs, the hardware device raises an interrupt request to the interrupt controller to signify that data is ready for action by the software task on the processor.

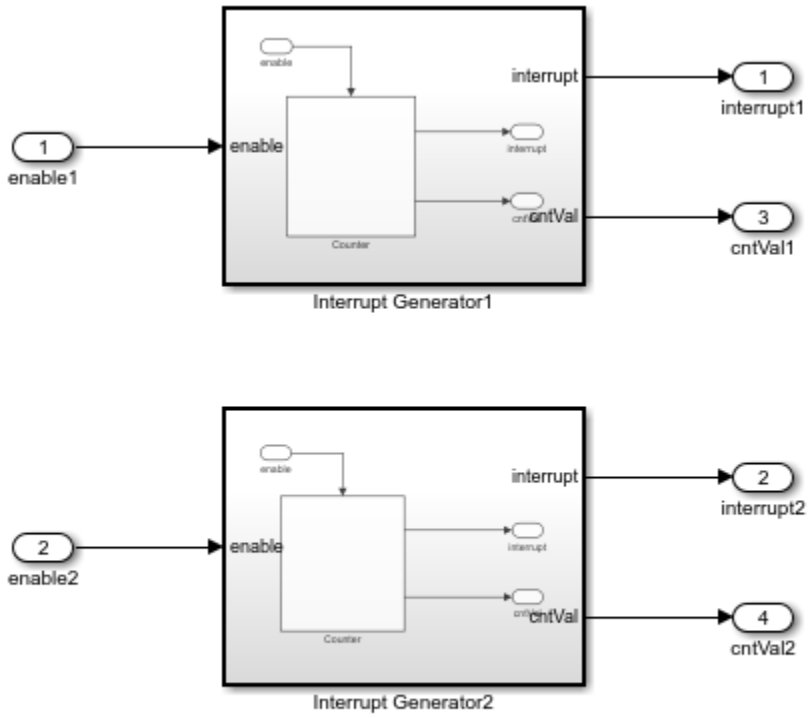
Design Task

This example includes two hardware IP cores in the FPGA. These IP cores generate the interrupt signals asynchronously. The interrupts trigger two separate tasks on the processor, one for each hardware IP core. The processor tasks access the hardware devices by reading or writing registers.

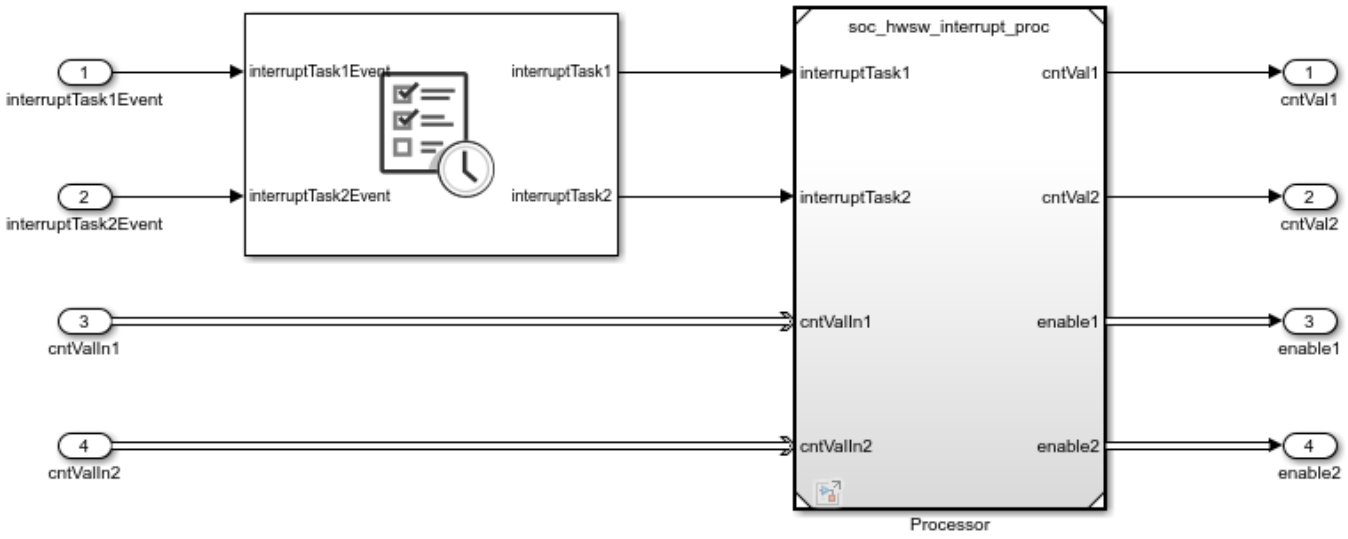
Model Structure



The top model sw_hw_interrupt_top includes the FPGA model sw_hw_interrupt_fpga and processor model sw_hw_interrupt_proc as model references. The top model also includes the Interrupt Channel and Register Channel blocks to model data transfers from the FPGA to the Processor.

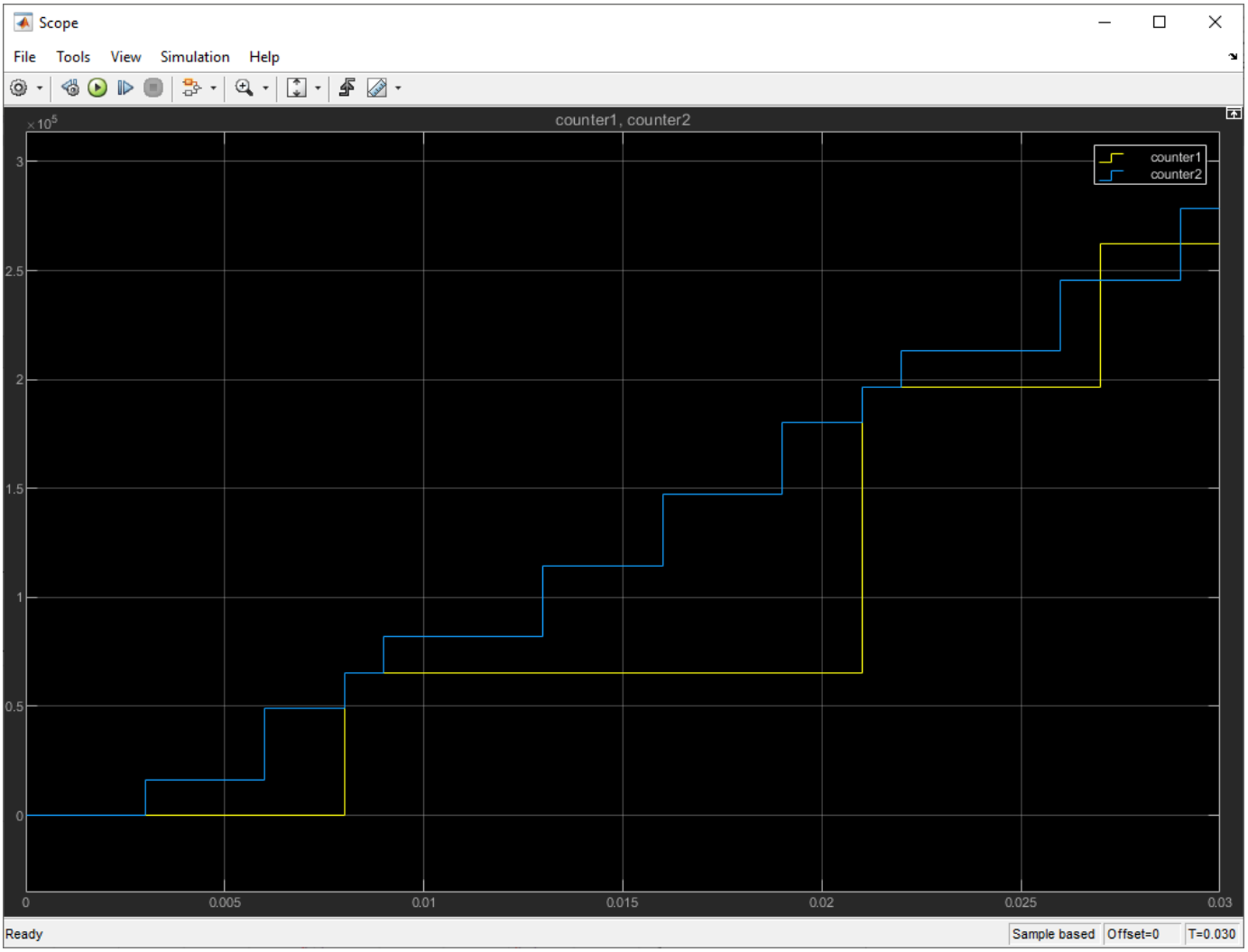


The FPGA model implements two IP cores that generate interrupts based on incrementing 32 bit-counter values. The first IP core generates an interrupt when either the 16th or the 18th bit of the counter value changes from 0 to 1. The second IP core generates an interrupt when either the 14th or 16th bit of the counter value changes from 0 to 1. The counter value at the time of generating the interrupt is registered and transferred to the processor using the Register Channel block. The processor model implements two software tasks, one for each IP core, that read the counter value, previously registered by the FPGA.

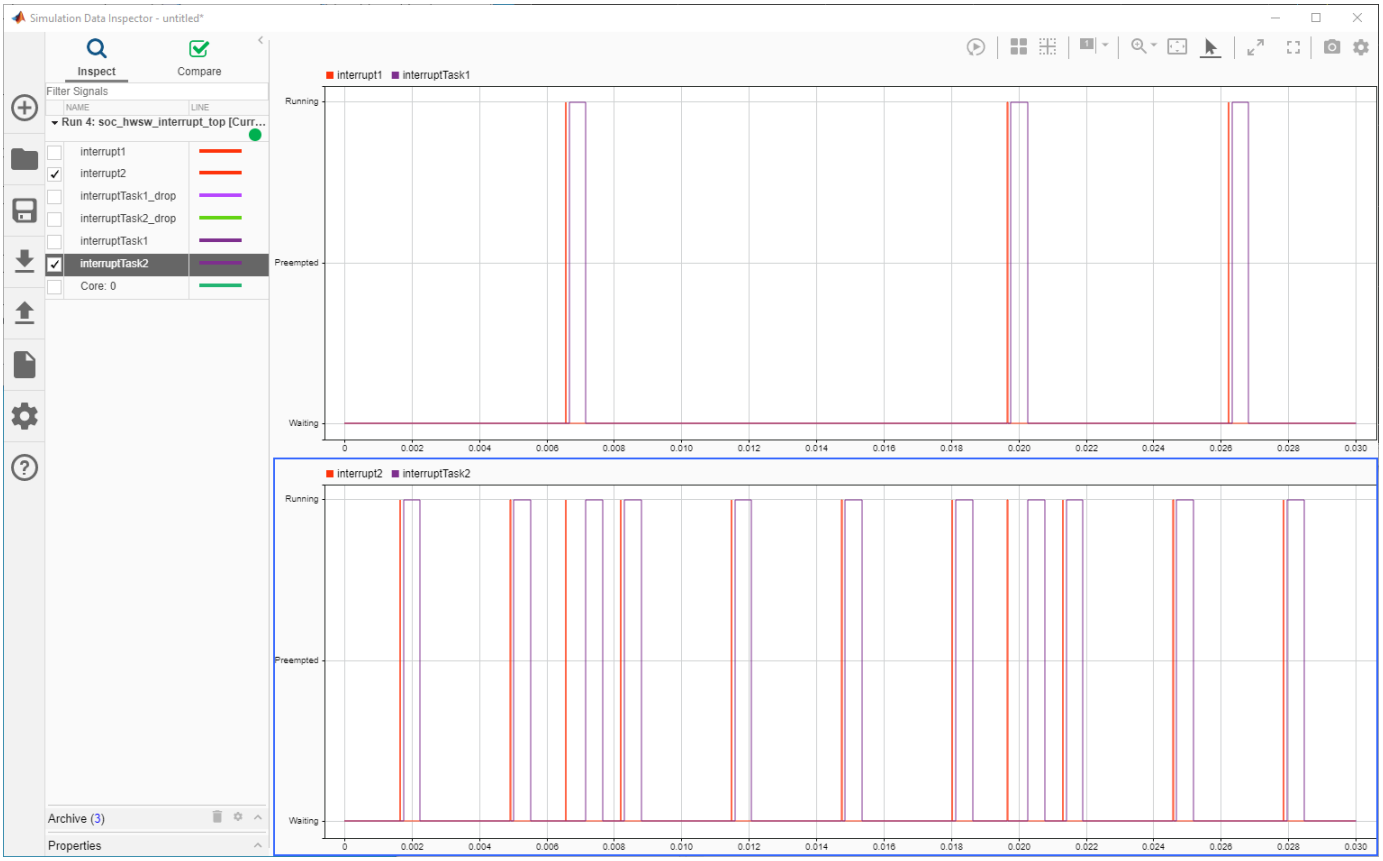


Simulation

Simulate the system for 0.03 seconds and open the Scope block on the top model to visualize the counter values as they are read by the processor.



Next, on the **Simulation** tab, click **Data Inspector** to view the timing of various events triggered by interrupts. Add interrupt1 and interruptTask1 signals in the top view, and add interrupt2 and interruptTask2 signals in the bottom view. The simulation plot shows that everytime an interrupt is triggered, the corresponding task is executed. At time 0.01802 seconds, interrupt2 is raised and takes 100 microseconds to process by the Interrupt Channel block and to trigger the corresponding interruptTask2, which takes 500 microseconds to run.



At time 0.01966 seconds, interrupt1 & interrupt2 are raised at the same time, and they are serviced by the processor based on their relative priority. Since interrupt1 is connected first at the input ports of the Interrupt Channel block, it has a higher priority than interrupt2. interruptTask2 waits to be executed while interruptTask1 is still executing due to the higher priority of interrupt1.

Implement and Run on Hardware

These products are required for this section:

- HDL Coder™
- Embedded Coder®
- SoC Blockset Support Package for Xilinx Devices, or SoC Blockset Support Package for Intel Devices

For more information about support packages, see “SoC Blockset Supported Hardware”.

To implement the model on a supported SoC board use the SoC Builder tool. By default, the model will be implemented on **ZedBoard** as it is configured with that board. To open SoC Builder click, 'Configure, Build, & Deploy' button in the toolstrip and follow these steps:

1. Select **Build Model** on the **Setup** screen. Click **Next**.
2. Click **Next** on the **Review Task Map** screen.
3. On **Review Memory Map** screen, click **View/Edit Memory Map** to view the memory map. Click **Next**.

4. Specify the project folder on the **Select Project Folder** screen. Click **Next**.
5. Select **Build, load for external mode** on the **Select Build Action** screen. Click **Next**.
6. On **Validate Model** screen, click **Validate** to check the compatibility of model for implementation. Click **Next**.
7. On **Build Model** screen, click **Build** to begin building of the model. An external shell opens when FPGA synthesis begins. Click **Next**.
8. Click **Test Connection** on the **Connect Hardware** screen to test the connectivity of the host computer with SoC board. Click **Next** to go to the **Run Application** screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

1. Close the external shell to terminate synthesis.
2. Copy pregenerated bitstream to your project folder by running this `copyfile` command below
3. Click **Load and Run** to load the pregenerated bitstream and run the model on SoC board

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
        'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
        'soc_hwsw_interrupt_top-zedboard.bit'), './soc_prj');
```

After loading the bitstream, run the generated software model **soc_hwsw_interrupt_sw** in external mode. From the model toolbar, open the **Data Inspector** and add interruptTask1 in the top view and interruptTask2 into the bottom view. Observe that everytime an interrupt is triggered the corresponding task is executed.

Implementation on other boards: To implement the model on a supported board other than ZedBoard, first configure the model to the supported board, and then set the example parameters as below.

- On the **Hardware** tab, click **Hardware Settings** to open the Configuration Parameters window.
- On the **Hardware Implementation** tab, select your board from **Hardware board** on the top and processor model.
- On the **Target hardware resources > FPGA design (top level)** tab enable **Include MATLAB as AXI Master IP for host-based interaction** and set **IP core clock frequency (MHz)** to 10 MHz.

Next, click **Configure, Build, & Deploy** on the toolstrip to open **SoC Builder** and follow the steps as previously stated for ZedBoard above. Modify the `copyfile` command to match the bitstream corresponding to your board. In case of Altera Arria® 10 SoC development kit and Altera Cyclone® V SoC development kit use below `copyfile` command corresponding to your board. In case of Altera Arria® 10 SoC development kit, copy '.periph.rbf' and '.core.rbf' files.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
        'supportpackages','intelsoc','intelsocexamples','bitstreams',...
        'soc_hwsw_interrupt_top-c5soc.rbf'), './soc_prj');
```

The following are the available pre-generated bitstream files:

- 'soc_hwsw_interrupt_top-zc706.bit'
- 'soc_hwsw_interrupt_top-zedboard.bit'
- 'soc_hwsw_interrupt_top-zcu102.bit'

- 'soc_hwsw_interrupt_top-XilinxZynqUltraScale_RFSoCZCU111EvaluationKit.bit'
- 'soc_hwsw_interrupt_top-c5soc.rbf'
- 'soc_hwsw_interrupt_top-a10soc.periph.rbf'
- 'soc_hwsw_interrupt_top-a10soc.core.rbf'

In summary, this example showed how interrupts from the FPGA trigger actions in the processor. You modeled the behavior using SoC Blockset, and went through the workflow required to implement it on an SoC device.

Analyze Memory Bandwidth Using Traffic Generators

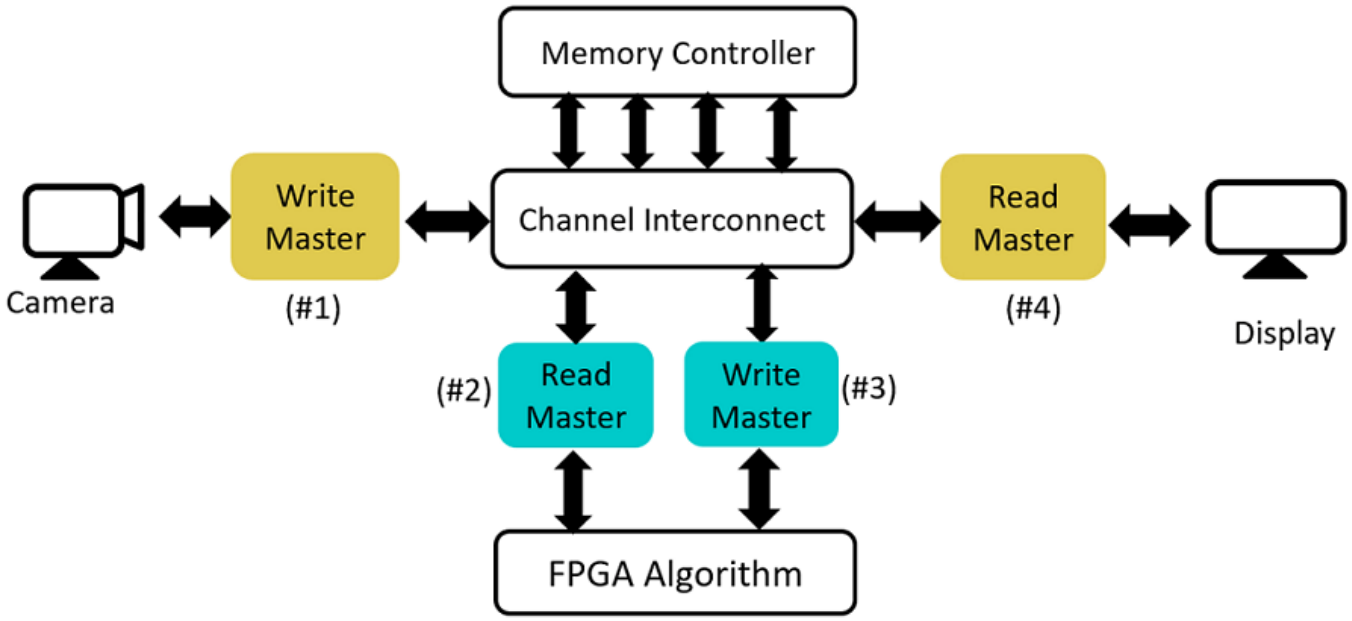
This example demonstrates how to analyze memory bandwidth for an SoC application. In memory-intensive hardware designs, you may have multiple masters accessing a common DDR memory. In such cases, it is important to analyze the dynamic requirement of all memory masters to guide algorithm design and hardware board requirement for deployment. You can simulate the memory traffic using Memory traffic generators, analyze the bandwidth usage and verify it on the hardware.

Supported hardware platforms

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx® Kintex® 7 KC705 development board

Design Task

Consider an application performing HD video processing in FPGA on real-time input and output. This application requires four memory consumers vying for DDR access simultaneously. Memory master 1 writes incoming video frames to memory and Memory master 4 reads video frames out of memory and connect to output display. Memory master 2 reads the data from memory for processing in FPGA and Memory master 3 writes the data back to memory.



Each master operates on HD video with following characteristics:

- Frame size: 1920x1080p
- Pixel size: 2 Bytes (YCbCr format)
- Frame period: 1/60 = 16.67ms (for 60 FPS)
- Frame data: 1920x1080x2 = 4.1472MB

Each master requires following minimum memory bandwidth to get the frame rate of 60 FPS.

- Memory bandwidth: Frame data/ Frame period = 4.1472e6/16.67e-3 = 248.8MBps

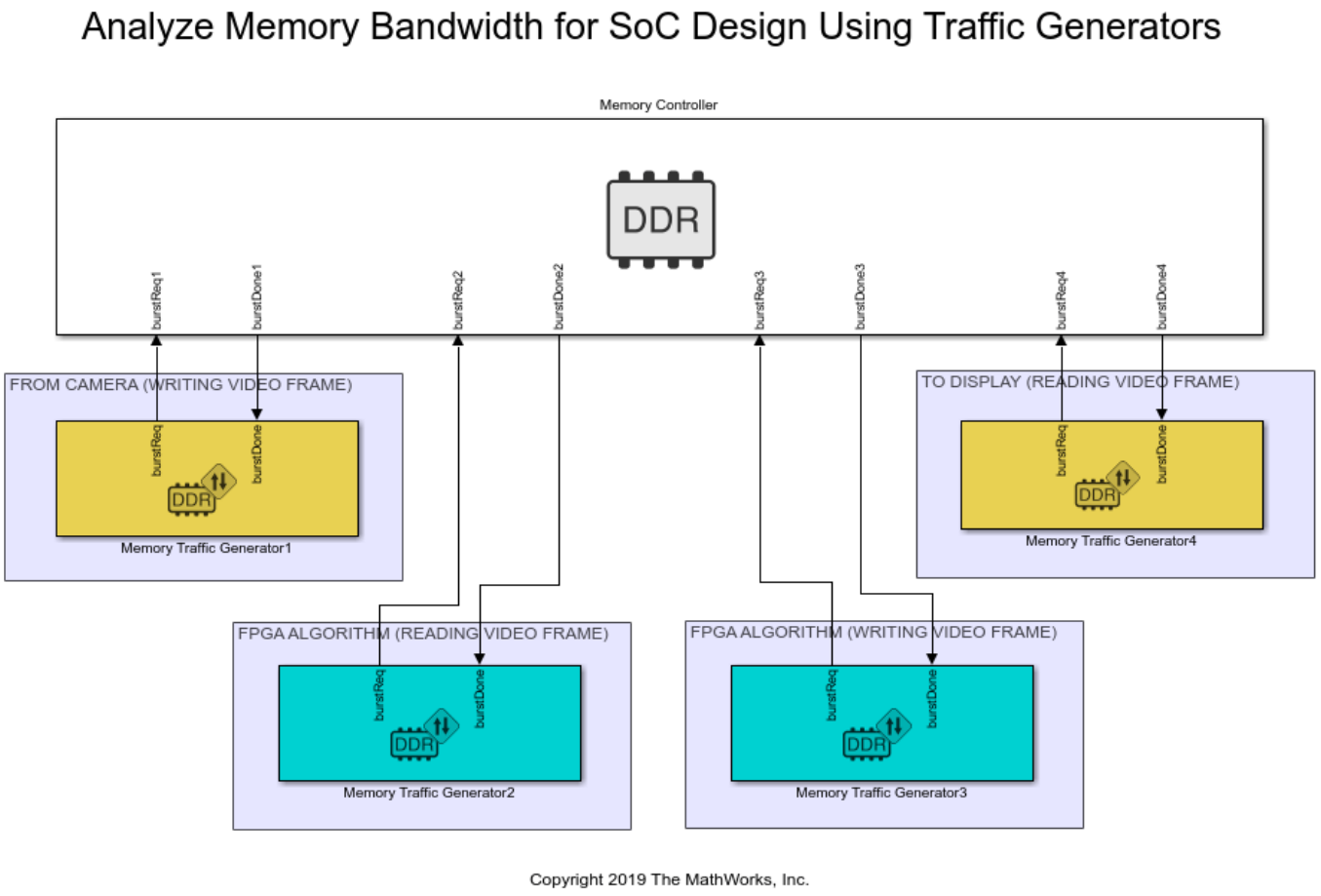
Assume the memory controller characteristics are as follows:

- Clock frequency: 200 MHz
- Data width: 32 bits
- Burst transaction length: 128

Design Using SoC Blockset

Create a model using Memory Controller and Memory Traffic Generator blocks to model four memory masters.

Memory Controller: Set the memory controller parameters in **Configuration Parameters > Hardware Implementation > Target Hardware Resources**. Under **FPGA Design (mem Controllers)** tab, set the clock frequency to 200 MHz and data width to 32. Under **FPGA Design (debug)** tab, select **Include AXI interconnect monitor**.



Memory Traffic Generators 1 & 4: Memory traffic characteristics for Master 1 and 4 are same as they represent streaming of video frames to and from memory. Set the memory traffic characteristics for masters 1 and 4 as follows:

- **Burst size (in bytes)**: Burst transaction length * (Data width/8) = 128* 32/8 = 512
- **Total burst requests**: 4 frames data for simulation = 4 * Traffic data/Burst size = 4*8100 = 32400

Burst inter access time: Frame period/Number of Burst requests = 16.67e-3/8100 = 20.58e-7 sec. As a constant data traffic, the data is continuously received at a constant rate. Set the burst times as below:

- **First burst time** = 20.58e-7
- **Random time between the bursts** = [20.58e-7 20.58e-7]

Update the Memory Traffic Generator1 and Memory Traffic Generator4 block mask with above values. Set the **Request type** for Memory Traffic Generator1 with writer and Memory Traffic Generator4 with reader. Clear the **Wait for burst done** option in both the block masks as these masters represent the masters with continuous traffic, such as HDMI Camera and display.

Memory Traffic Generators 2 & 3: Memory Traffic Generator2 represent reader for FPGA Algorithm and Memory Traffic Generator3 represent writer from FPGA Algorithm. Set the memory traffic characteristics for masters 2 and 3 as follows:

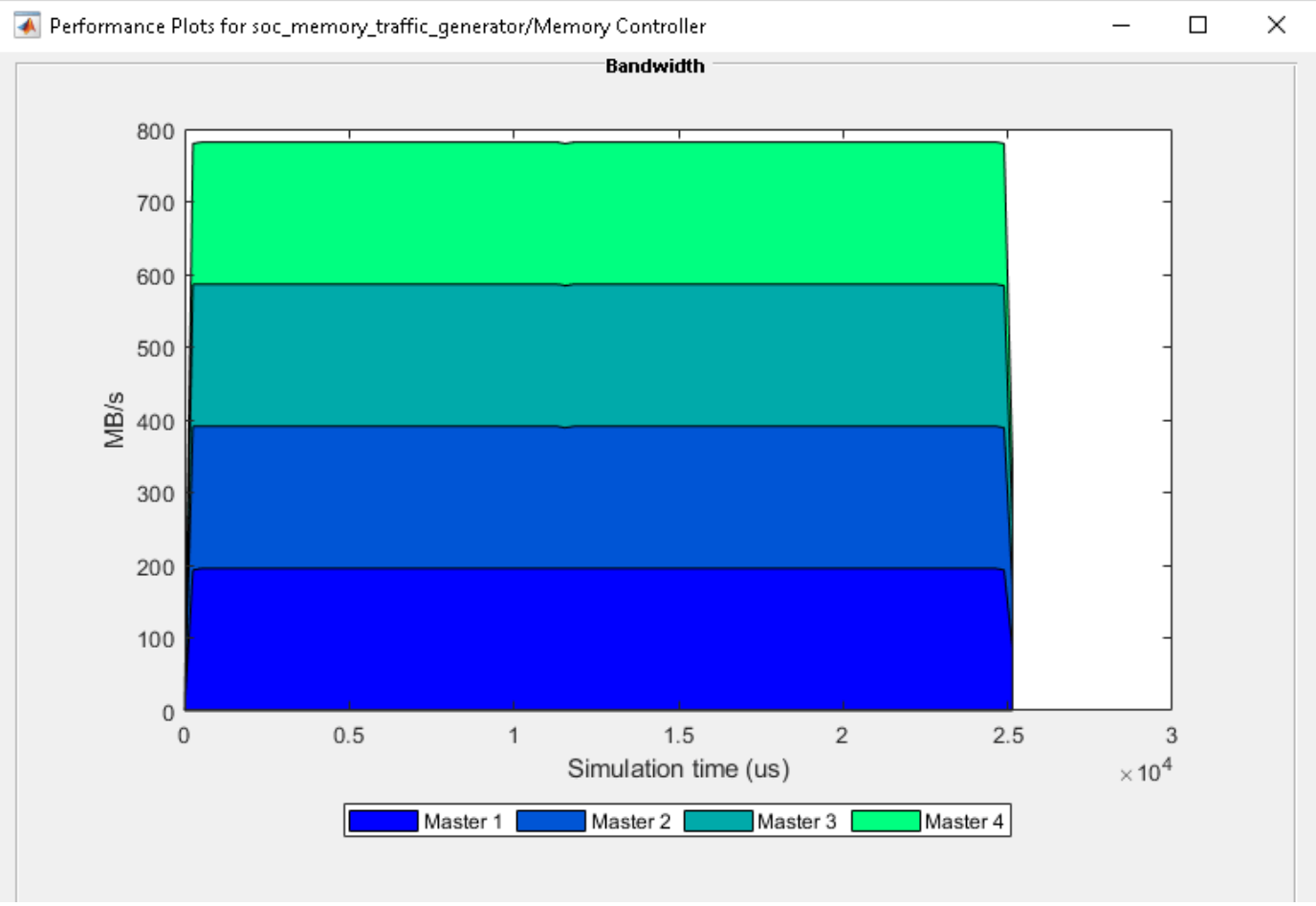
- **Burst size (in bytes)**: Burst transaction length * (Data width/8) = 128* 32/8 = 512
- **Total burst requests**: 4 * Traffic data/Burst size = 4*8100 = 32400(4 frames data for simulation)

Burst inter access time: (Burst Length + 10)/Clock period = 6.9e-7(0.69us). To allow some randomness in burst times for read and write request of data, due to variation in demands of algorithm, set the burst times as below:

- **First burst time**: 7.2e-7
- **Random time between the bursts**: [7.2e-7 7.4e-7]

Simulate

Run the model. After completion of simulation, open the Memory Controller block and click on **View performance plots** under **Performance** tab. Select all the masters under **Bandwidth** tab and click **Create Plot**. You can notice that all masters roughly achieved a bandwidth of 190 MBps and did not meet the required 248 MBps. It is also observed by the warnings in the diagnostic viewer.



To meet the required bandwidth, modify the data width of controller from 32 to 64 in configuration parameter settings under **Target Hardware Resources**. This requires changing the Memory Traffic Generator settings accordingly as follows:

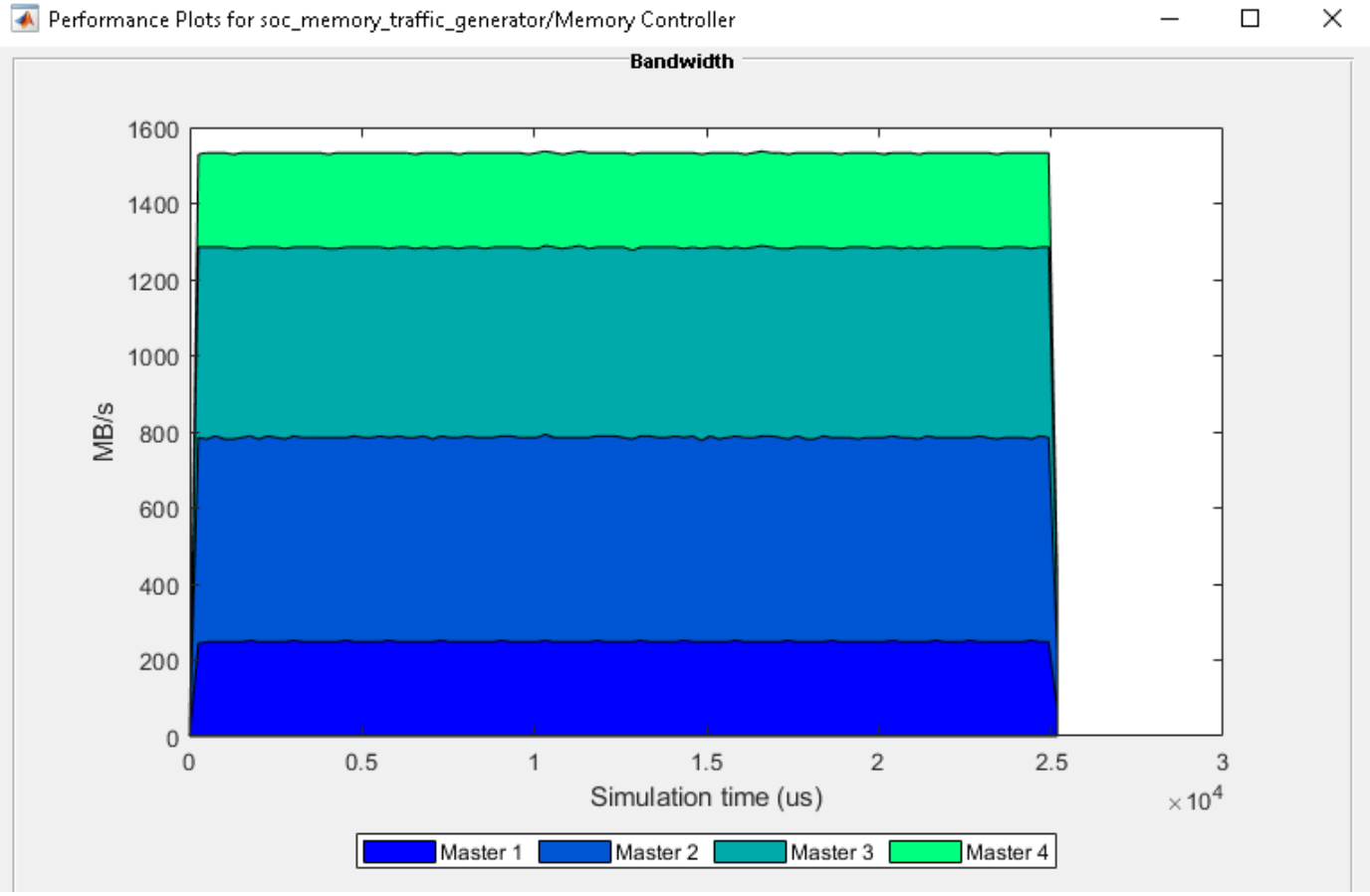
- **Burst size (in bytes)**: Burst transaction length * (Data width/8) = 128* 64/8 = 1024
- **Total burst requests**: 4 * Traffic data/Burst size = 4*4050 = 16200(4 frames of data for simulation)

Burst inter access time for Memory Traffic Generators 1 & 4: Frame Period/Number of Burst requests = 16.67e-3/4050 = 41.16e-7 sec. Set the burst times as below:

- **First burst time**: 41.16e-7
- **Random time between the bursts**: [41.16e-7 41.16e-7]

There is no change in First burst time and Random time between the bursts for Memory Traffic Generators 2 and 3, since they are determined based on algorithm requirements.

Simulate the model and open the **Bandwidth** plot from Memory Controller as mentioned earlier. Notice that Memory bandwidth achieved by Memory Traffic Generator 1 and 4 is 248 MBps. The memory bandwidth from Generator 2 and 3 is around 500 MBps. This meets the design requirement as all the masters are able to meet the real-time requirement of 248 MHz. Observe that there are no warnings on the diagnostic viewer as burst requests are not dropped.



Implement and Run on Hardware

“SoC Blockset Support Package for Xilinx Devices” is required for this section.

To implement the model on a supported FPGA board, use the SoC Builder application. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board.

AXI Traffic Generator(ATG), the hardware IP Core for Memory Traffic Generator block does not support random burst inter access times and it differentiates Reader and Writer masters in arbitration policy unlike the Memory Traffic Generator block for simulation. Therefore, before implementing on hardware, modify the Memory block settings as follows:

- Make all the Memory Traffic Generators as 'Writers'
- For Memory Traffic Generator 2 and 3, set [7.2e-7 7.2 e-7] for Random time between burst to make it fixed inter burst time of 7.2e-7

To open SoC Builder, select the **System on Chip** tab in the Simulink toolstrip, and click the **Configure, Build, & Deploy** button. Once SoC Builder opens, follow these steps:

- Select **Build Model** on **Setup** screen. Click **Next**.
- Click **View/Edit Memory Map** to view the memory map on **Review Memory Map** screen. Click **Next**.

- Specify project folder on **Select Project Folder** screen. Click **Next**.
- Select **Build, load and run** on **Select Build Action** screen. Click **Next**.
- Click **Validate** to check the compatibility of model for implementation on **Validate Model** screen. Click **Next**.
- Click **Build** to begin building of the model on **Build Model** screen. An external shell will open when FPGA synthesis begins. Click **Next** to **Load Bitstream** screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following steps:

- Close the external shell to terminate synthesis.
- Copy the pre-generated bitstream to your project folder and rename by running the below command.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
        'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
        'soc_memory_traffic_generator-zc706.bit'), './soc_prj');
```

- Click **Load** button to load pre-generated bitstream.

To run this example, copy the example test bench to your project folder.

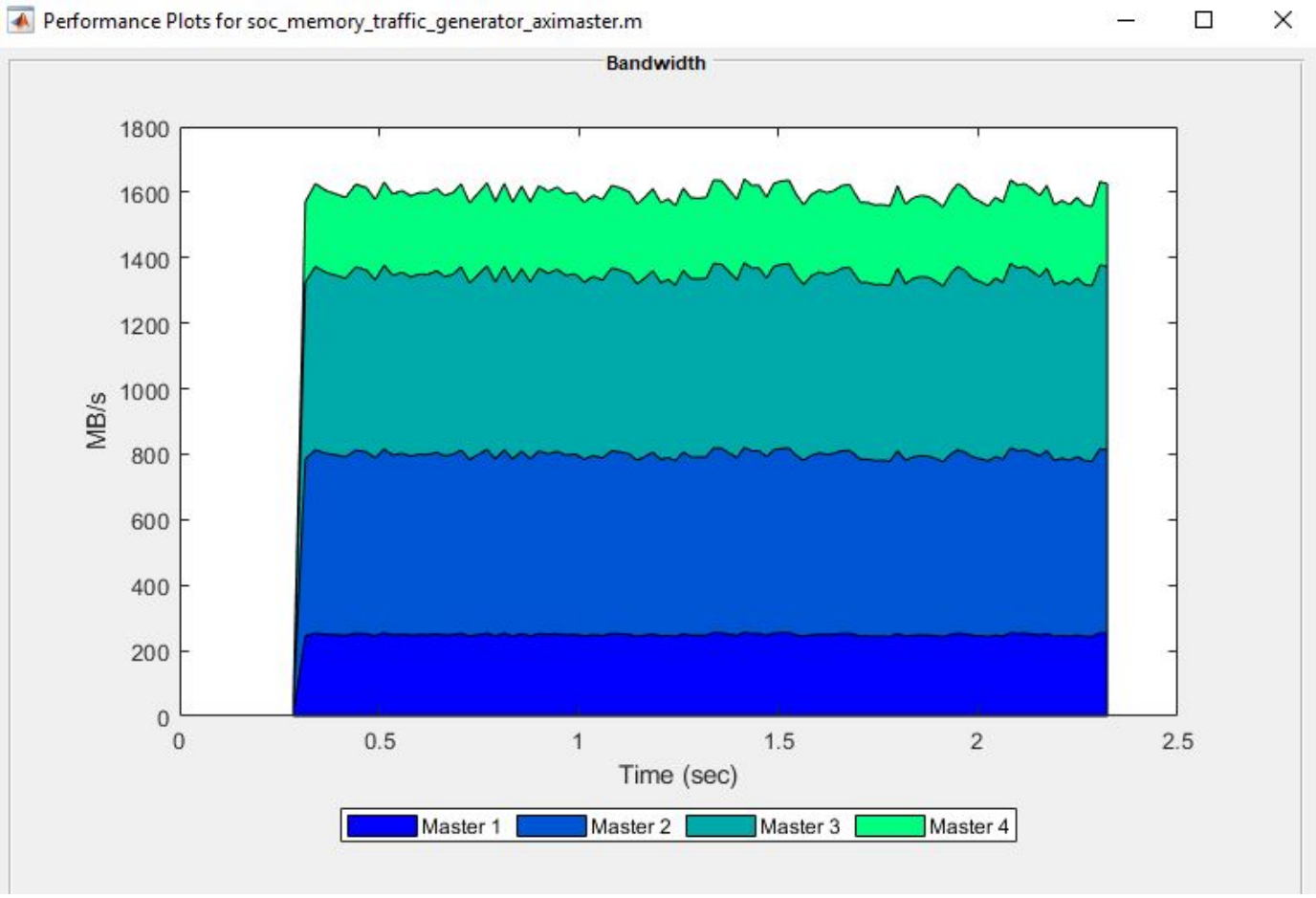
```
copyfile(fullfile(matlabroot,'toolbox','soc','socexamples',...
        'soc_memory_traffic_generator_aximaster.m'), './soc_prj','f');
```

The testbench configures the generated hardware ATG IP cores for Memory Traffic Generators. To run on hardware, increase the number of burst requests by 100 times since it uses *MATLAB® as AXI Master* IP to get the samples back to MATLAB®, which involves substantial delay in accessing hardware. Load soc_memory_traffic_generator_zc706_aximaster.mat file and increase the number of burst requests for all the masters in ATG configuration to 100 times. Save the .mat file requests in ATG configuration.

Enter the following command to run the test bench soc_memory_traffic_generator_aximaster.

```
soc_memory_traffic_generator_aximaster
```

After running the test bench, the following output is generated showing the memory traffic. All masters passing the bandwidth requirements.



Implementation on Xilinx® Kintex® 7 KC705 development board: To implement the model on KC705 development board, you must first configure the model to Xilinx® Kintex® 7 KC705 development board and set the following example parameters. Open **Model Configuration Parameters**, navigate to **Hardware Implementation** tab and perform the following:

- Select **Xilinx® Kintex® 7 KC705 development board** from the drop-down list under **Hardware board**.
- Navigate to **Target hardware resources > FPGA design (top level)** tab and enable **Include MATLAB as AXI Master IP for host-based interaction**.
- Navigate to **Target hardware resources > FPGA design (mem controllers)** tab and set **Controller data width (bits)** to **64**.
- Navigate to **Target hardware resources > FPGA design (debug)** tab and enable **Include AXI interconnect monitor**.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the copyfile command to match Kintex® 7 KC705 development board bitstream as below.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc',...
        'supportpackages','xilinxsoc','xilinxsocexamples','bitstreams',...
        'soc_memory_traffic_generator-kc705.bit'), './soc_prj');
```

In summary, you simulated the memory traffic for a prospective design before designing the algorithms. You analyzed memory bandwidth and modified memory parameters to meet the design requirement. You verified the results on hardware.

Determine and Use Task Timing Information

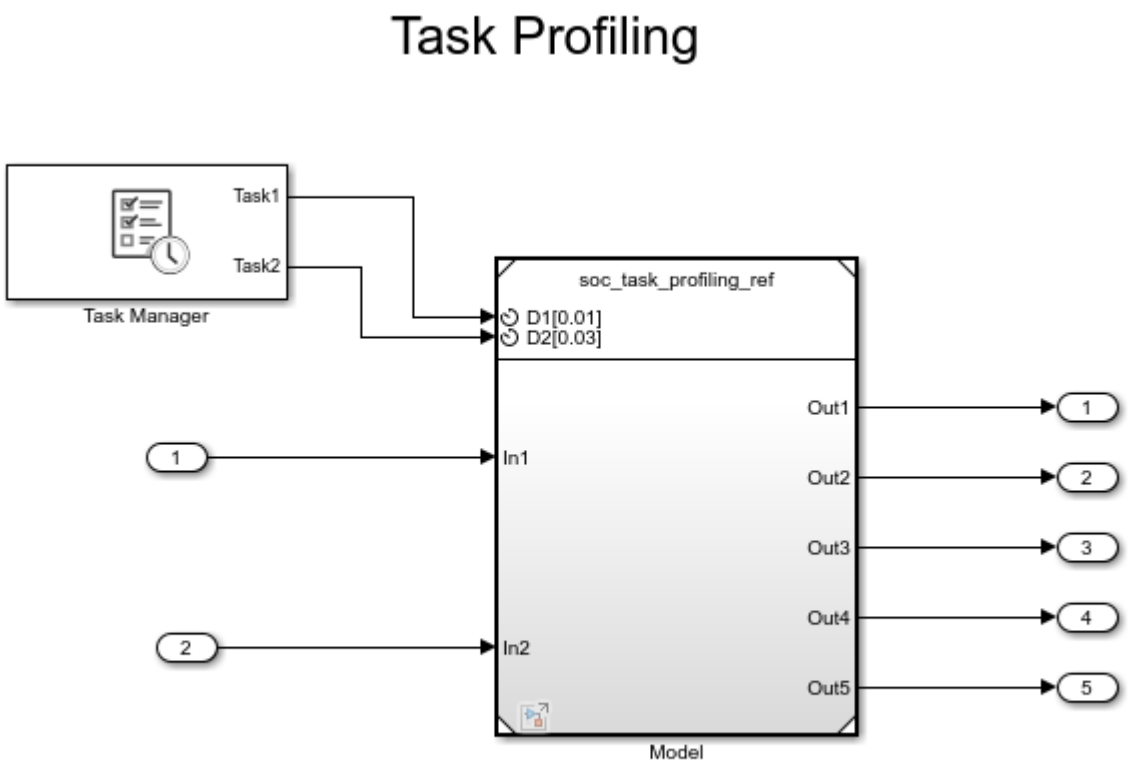
This example shows how to choose an available method for determining timing information for the tasks on your processor. The Task Manager block uses task timing information to simulate task preemption, overruns, and parallel execution. The accuracy of task timing information provides confidence that the task simulation reflects the actual behavior on your processor.

You can use a variety of methods can be used to determine task timing information. Each method has specific hardware and software requirements and offers different degrees of timing information accuracy. This table lists these methods and their respective characteristics.

Method	Requirements	Accuracy of Timing Information
Use system specifications	None	Low (Derived from system specifications)
Profile using Software-in-the loop	Embedded Coder®	Low/Medium (Measured on host computer)
Profile using Processor-in-the loop	Hardware Board Embedded Coder®	Medium/High (Measured for isolated task on hardware board)
Profile using SoC Blockset	Hardware Board SoC Blockset™, Embedded Coder® Complete system model	High (Measured from complete system on hardware board)

The sections in this example explain and demonstrate each method using an example system model. While this example uses the Xilinx Zynq ZedBoard™, these techniques can be used with any supported SoC Blockset™ hardware board or platform. To learn more about simulating task execution, see the “What is Task Execution?” on page 3-2 topic and the “Task Execution” on page 7-78 example.

```
open_system('soc_task_profiling');
```



Use Algorithm Timing Specifications

When only system requirements are available, use the worst-case execution time (WCET). The WCET should be set as a percentage of the task period (for example, 80%).

The example model has two tasks with periods 0.01s and 0.03s. Using the WCET, the average execution times should be set to:

- 8e-03s for Task1
- 24e-03s for Task2

This approach produces timing information of low accuracy. When you underestimate WCET, the processor resources can be used inefficiently. Similarly, when you overestimate WCET, unwanted task preemptions or overruns can occur.

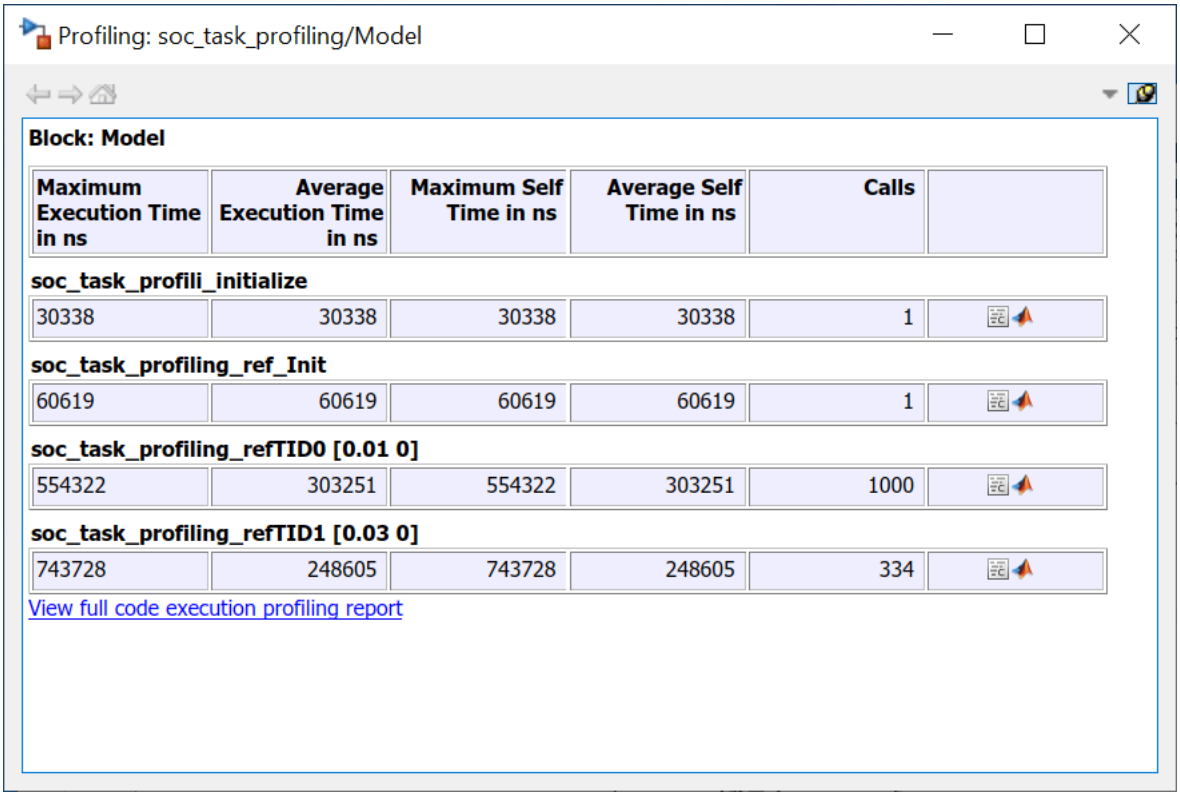
Profile Algorithm Using Software-in-the-Loop (SIL)

SIL simulation compiles generated source code and then runs the code on your host computer. During simulation, execution-time metrics for the generated code get collected. SIL simulation provides low to medium accuracy timing information, as the host computer generally has different architecture than your processor. This approach can be useful, especially, for comparative analysis. For more information on SIL, see “Configure and Run SIL Simulation” (Embedded Coder).

These steps show how to use SIL profiling to determine task information for the example model.

1. Right-click the Model block, click **Model Parameters (ModelReference)**, and select **Software-in-the-loop (SIL)** in the **Simulation mode** drop-down. Click **OK**.

2. On the **Simulation** tab, click **Run** to run the SIL simulation. When the simulation completes, click the Model block to get the execution-time metrics. This figure shows the execution-time metrics report.



3. The average execution times for Task1 and Task2, which correspond to the rates of these tasks are:

- 0.30e-03s for Task1
- 0.25e-03s for Task2

4. Use this task timing information in the Task Manager block to set the task duration mean. You can use other task timing information to set the other task timing parameters in the Task Manager block.

Profile Algorithm Using Processor-in-the-Loop (PIL)

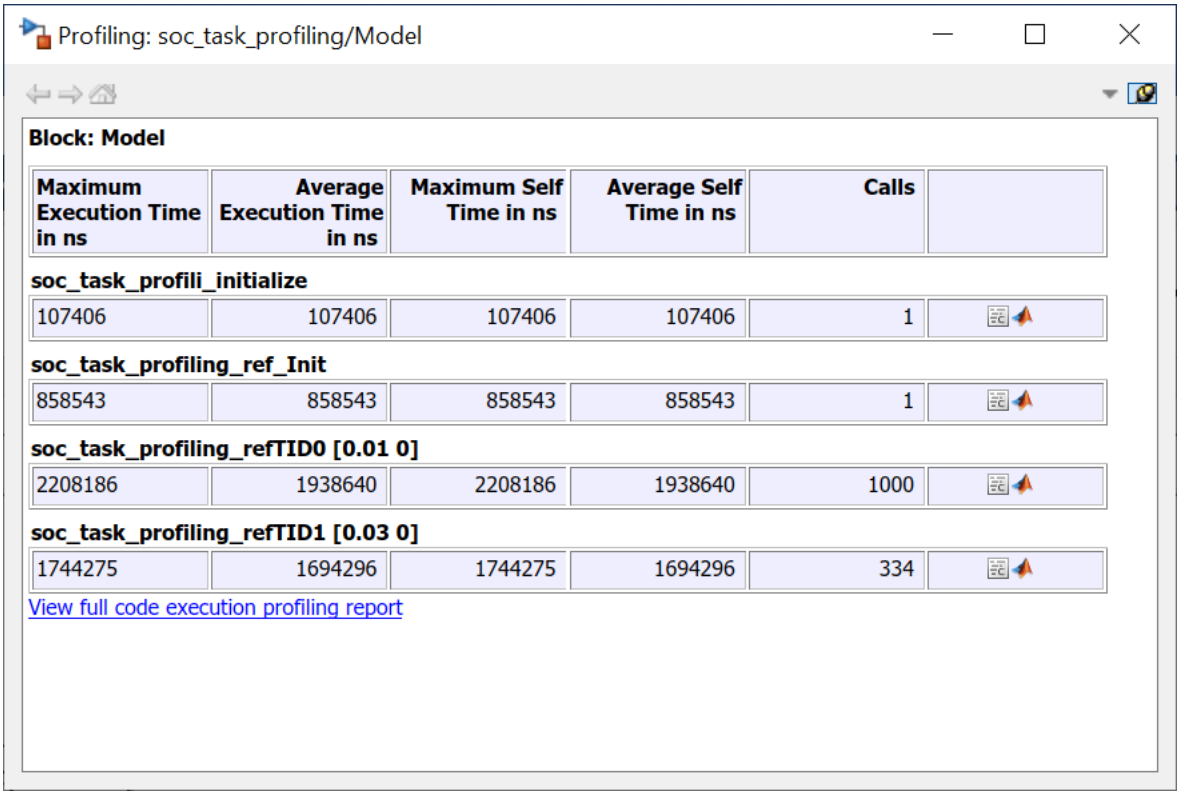
PIL simulation compiles generated source code and then runs the code on your target hardware. During simulation, execution-time metrics for the generated code get collected. PIL simulation provides medium to high accuracy timing information, as it profiles the task algorithm on your processor. With this approach, the timing of a single task is accurate but does not account for subtle effects such as sharing cache memory. For more information on PIL, see “Configure and Run PIL Simulation” (Embedded Coder).

These steps show how to use PIL profiling to determine task information for this example model.

1. Right-click the Model block, click **Model Parameters (ModelReference)**, and select **Processor-in-the-loop (PIL)** in the **Simulation mode** drop-down. Click **OK**.

2. On the **System on Chip** tab, click **Hardware Settings**. Expand the **Target hardware resources** parameter panel and in the **Board Parameters** group set **Device Address**, **Username** and **Password**.

3. On the **Simulation** tab, click **Run** to run the PIL simulation. When the simulation completes, click the Model block to get the execution-time metrics. This figure shows the execution-time metrics report.



4. The average execution times for Task1 and Task2, which correspond to the rates of these tasks are:

- 1.93e-03s for Task1
- 1.69e-03s for Task2

5. Use this task timing information in the Task Manager block to set the task duration mean. You can use other task timing information to set the other task timing parameters in the Task Manager block.

Profile Task Execution on Hardware

SoC Blockset profiling provides the timing information of every task in your model, and captures other related events such as preemptions, overruns, and task drops. Profiling the whole application on hardware provides the most accurate processor task timing information.

These steps show how to use SoC Blockset profiling to determine task information for the example model.

1. Right-click the Model block, click **Model Parameters (ModelReference)**, and select **Normal** in the **Simulation mode** drop-down. Click **OK**.

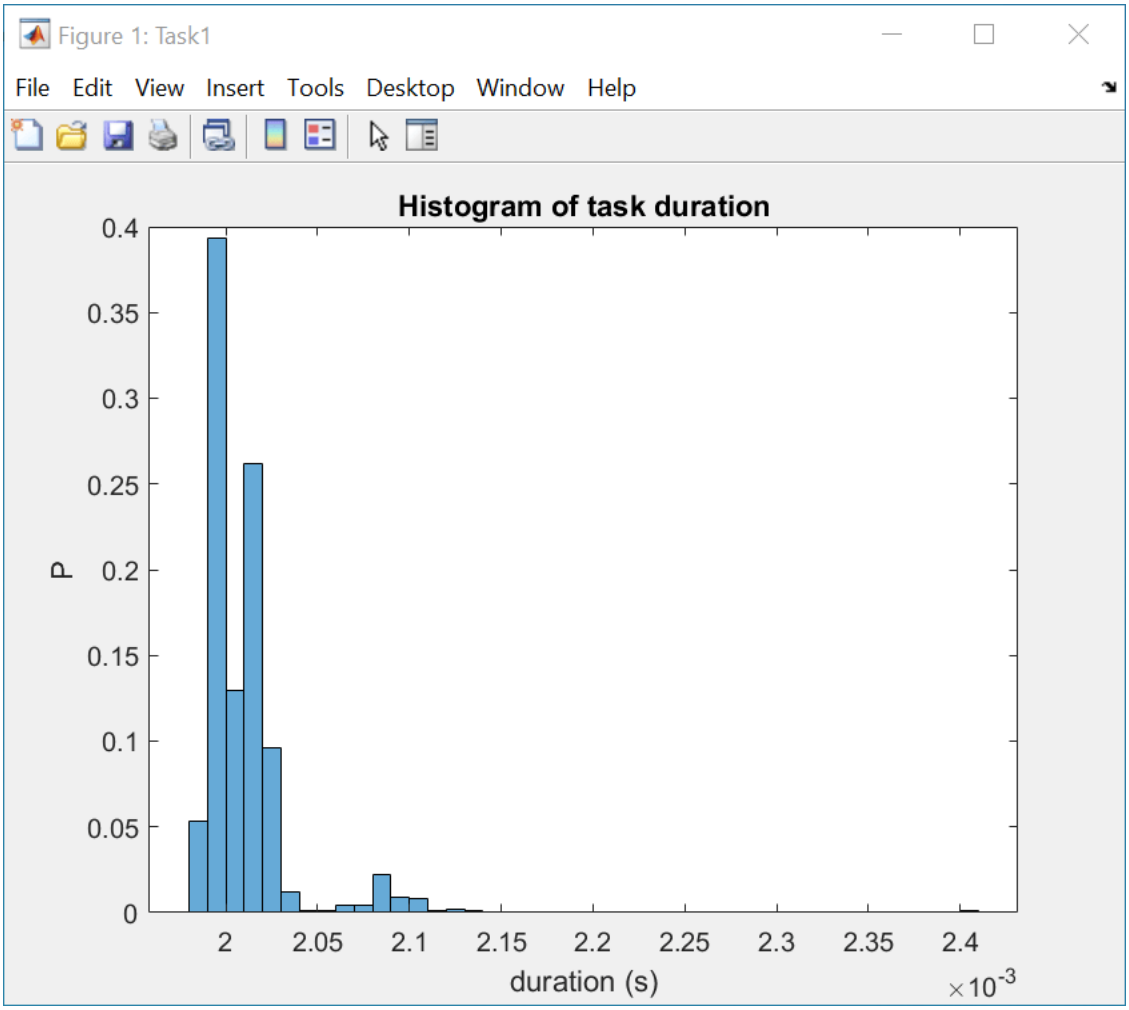
2. On the **System on Chip** tab, click **Configure, Build & Deploy**. Follow the steps provided to prepare the model to build and load for external mode, and click **Monitor & Tune**. When the external mode completes, run these commands in MATLAB to get the execution times for Task1 and Task2:

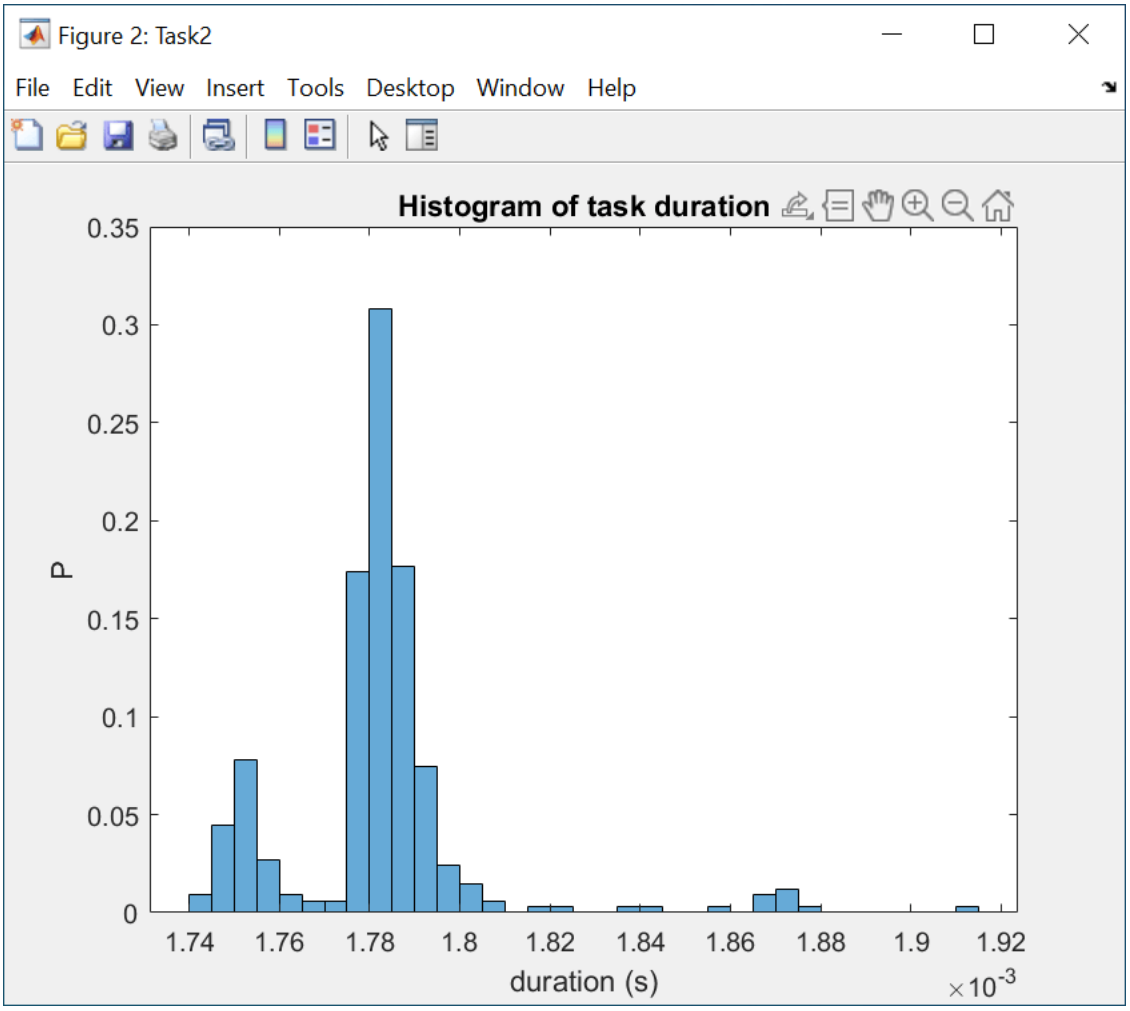
```
exectime = socTaskTimes('soc_task_profiling','Run 1: soc_task_profiling')
exectime.Mean
```

The execution times for Task1 and Task2 are:

- 2.00e-03s for Task1
- 1.80e-03s for Task2

3. The `socTaskTimes` function also shows the distribution of execution times for each task as shown in these figures.





4. Use this task timing information in the Task Manager block to set the task duration mean. You can use other task timing information to set the other task timing parameters in the Task Manager block.

Record I/O Data from SoC Device

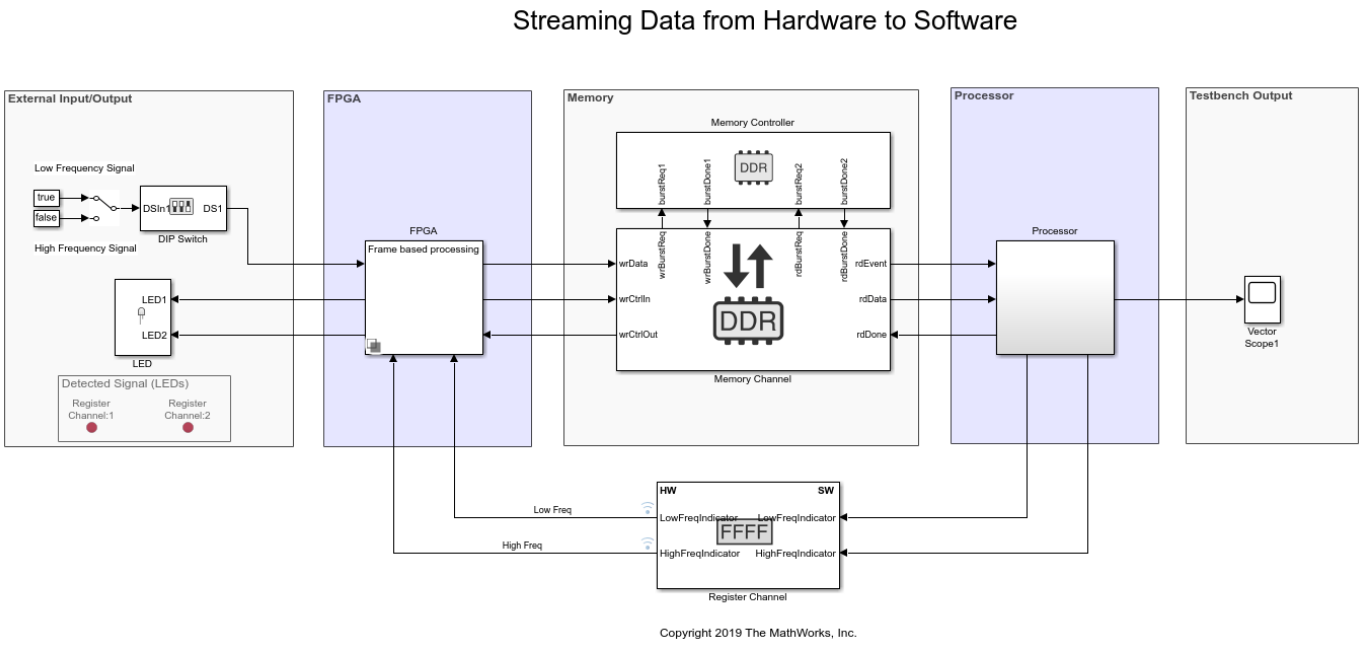
This example shows you how to record real-world data from hardware for use in simulation.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

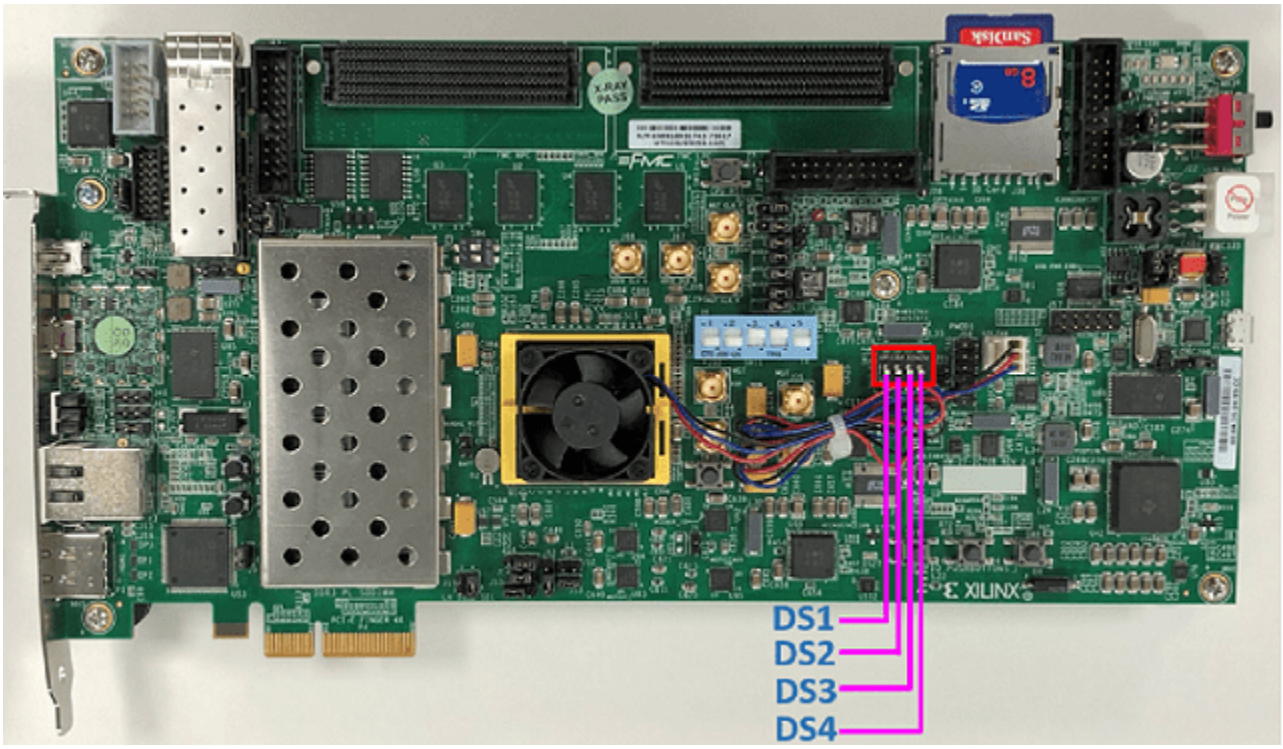
In many situations you may want to verify your algorithm against real-world data. This example, using the Streaming Data from Hardware to Software model, shows how to record signals from the AXI4 interface on a SoC device. This workflow allows you to focus on the processor side of the algorithm by substituting a pre-recorded data stream in place of the Simulink® FPGA design.

We recommend completing “Streaming Data from Hardware to Software” on page 7-31 example.



Record Data from FPGA

In this section, you will record data generated by the FPGA subsystem in the Streaming Data from Hardware to Software model. In this model, the FPGA subsystem generates a sinusoidal signal with frequency 1kHz or 10kHz, controlled via a DIP switch (DS1). The FPGA algorithm filters the signal and sends it to the processor through AXI4 Stream Memory Channel.



Following products are required for this section:

- SoC Blockset Support Package for Xilinx® Devices

Follow the steps below to record data from FPGA:

1. Create a hardware communication object executing the following on the MATLAB® command prompt.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','10.10.10.15',...
                      'username','root','password','root')
```

Enter the appropriate hardware board name, IP address and the user credentials in the command above. The hardware object hw, is a communication gateway that provides control commands and I/O exchange.

2. Open Streaming Data from Hardware to Software model. Load the provided pre-generated FPGA bitstream for this model to hardware.

```
socLoadExampleBitstream(hw,'soc_hwsw_stream_top')
```

3. Create a data recorder for your hardware board.

```
dr = soc.recorder(hw);
```

4. Create an **AXI Stream Read** input source object and configure the source properties.

```
src = soc.iosource(hw,'AXI Stream Read');
src.devName = 'mwfpga_algorithm_wrapper_ip0:s2mm0';
samplingFrequency = 1e5;
src.dataTypeStr = 'uint32';
```

```
src.SamplesPerFrame = 1000;
src.SampleTime = src.SamplesPerFrame/samplingFrequency;
```

The `samplingFrequency` represents the sine wave sampling rate in the Streaming Data from Hardware to Software model.

5. Add the **AXI Stream Read** source to the data recording session.

```
addSource(dr,src,'AXI4 stream interface')
```

6. Initialize the I/O sources on the hardware board for recording.

```
setup(dr)
```

7. Use the `record` function to record 10 seconds of data.

```
record(dr, 10)
while isRecording(dr)
    pause (0.1);
end
```

During the recording, toggle the DIP switch (DS1) to change the frequency of signal generated by the FPGA.

8. Save the recorded data to a file:

```
save(dr,'sine_wave_data')
```

Record RF Signals

In this section, you will capture RF signals from an AD - FMCOMMS2/3/4 RF card connected to the FPGA. The data will be streamed from the RF card to the processor using AXI4 stream interface.

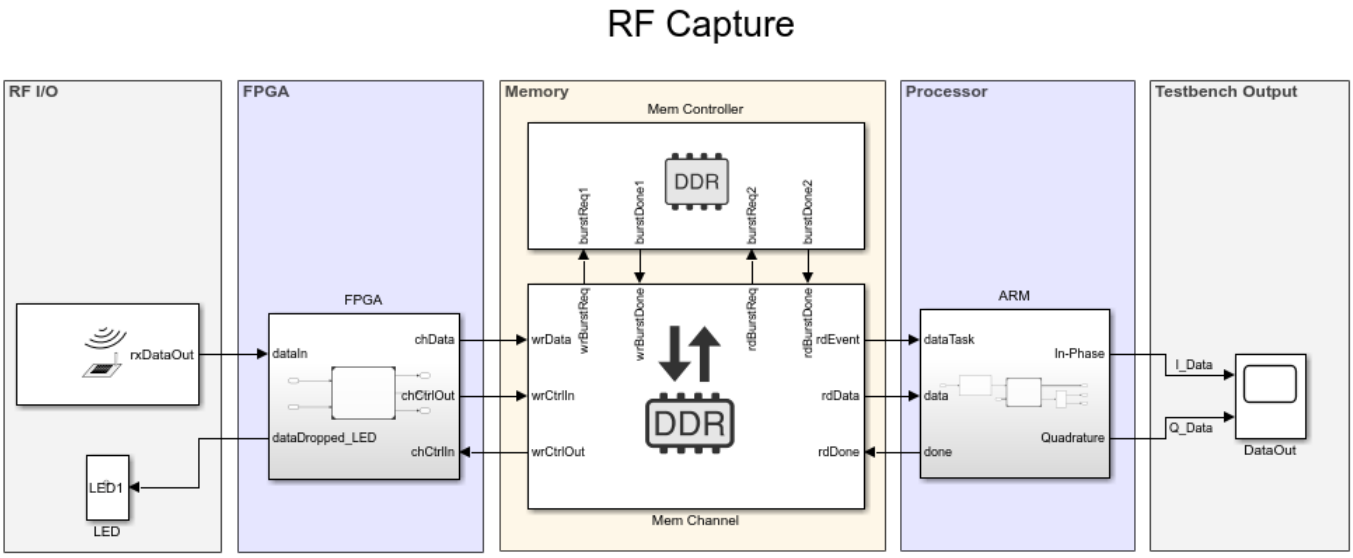
Following products are required for this section:

- SoC Blockset Support Package for Xilinx® Devices

Supported hardware platforms for this section are:

- Xilinx® Zynq® ZC706 evaluation kit
- ZedBoard™ Zynq-7000 Development Board

To configure RF card refer to “Manual Host-Radio Hardware Setup” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)



1. Open RF Capture model. Load the provided pre-generated FPGA bitstream for this model to hardware.

```
socLoadExampleBitstream(hw,'soc_rfcapture')
```

2. Configure radio card.

```
rf = rfcard(hw);
rf.CenterFrequency = 1090e6;
rf.GainSource = 'AGC Fast Attack';
rf.BasebandSampleRate = 4e6;
rf.ShowAdvancedProperties = true;
rf.ShowInternalProperties = true;
rf.BISTToneMode = 'Tone Inject Rx';
rf();
```

3. Setup data recorder.

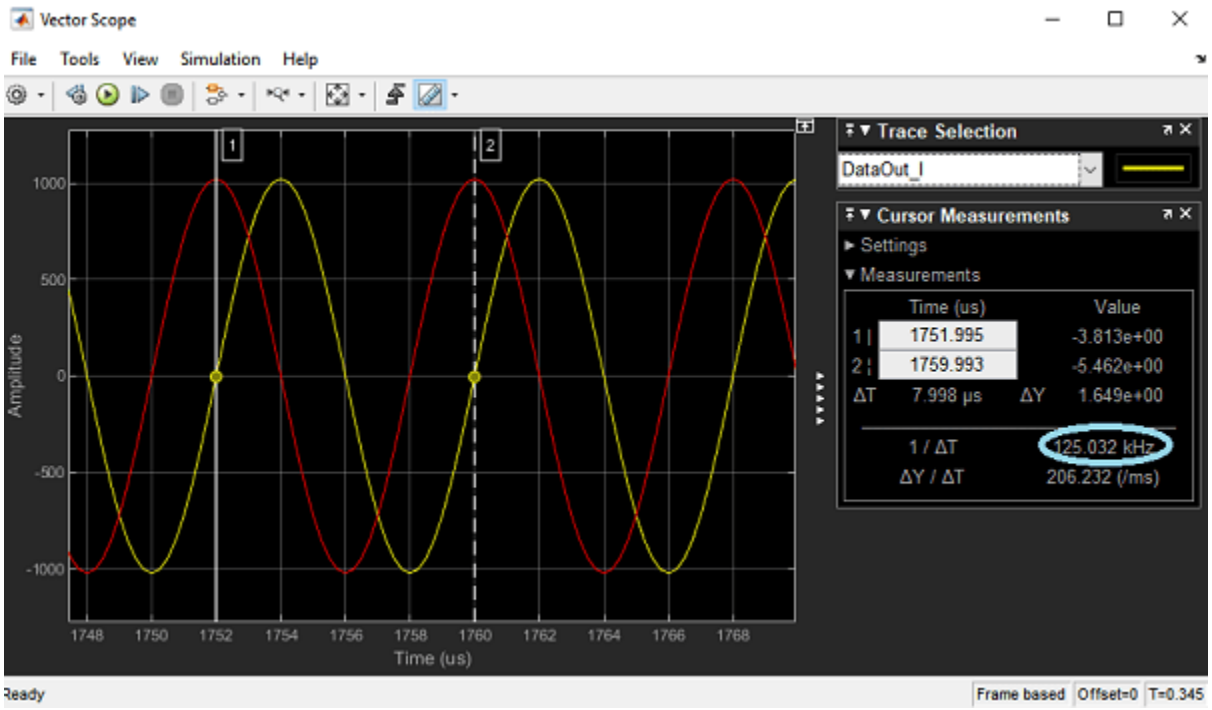
```
dr = soc.recorder(hw);
src = soc.iosource(hw,'AXI Stream Read');
src.devName = 'mwfpga_data_capture_ip0:s2mm0';
src.dataTypeStr = 'uint32';
src.SamplesPerFrame = 4000;
src.SampleTime = src.SamplesPerFrame/rf.BasebandSampleRate;
addSource(dr,src,'AXI4 stream interface');
```

4. Record radio signals.

```
setup(dr)
system(hw,'devmem 0x40010100 32 1');
record(dr,1)
while isRecording(dr)
     pause (0.1);
```

```
end
save(dr,'zynq_rf_data')
```

5. To playback the recorded RF data, open RF Playback model. Enter the dataset name and the source name on the **IO Data Source** block and simulate the model.



A pre-recorded dataset file **zynq_rf_data.tgz** is available at **matlab\toolbox\soc\socexamples**.

See Also

“Simulate with I/O Data Recorded from SoC Device” on page 7-76

Simulate with I/O Data Recorded from SoC Device

This example shows you how to use recorded real-world data in simulation.

Supported hardware platforms:

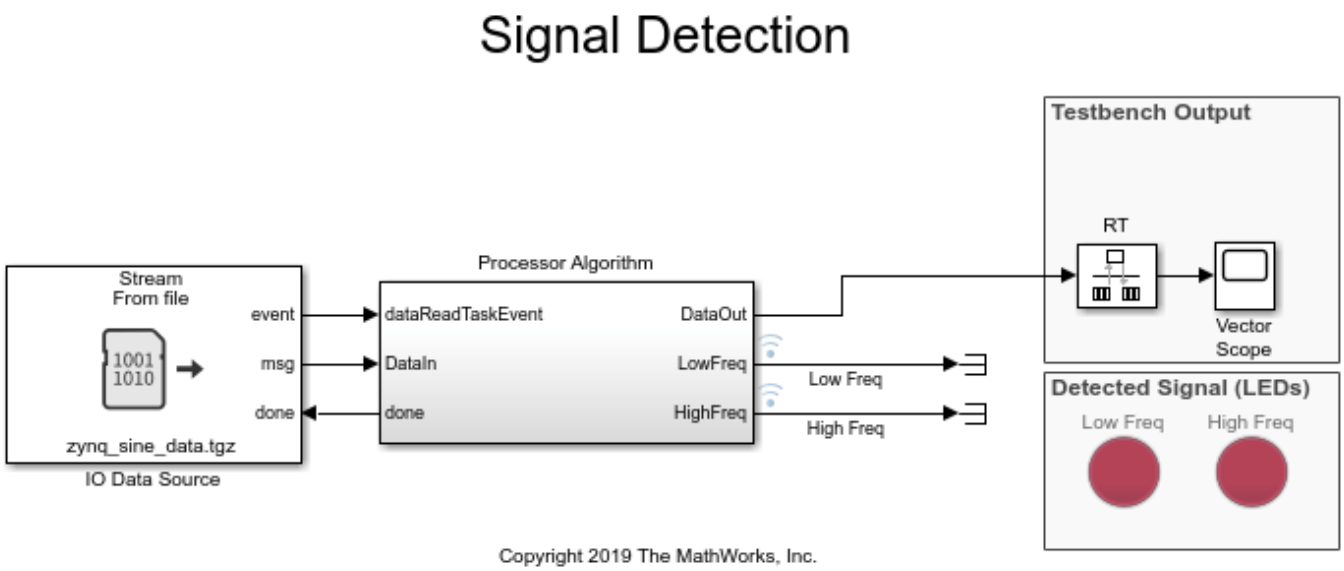
- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

In many situations you may want to verify your algorithm against real-world data. This example shows how to use the recorded data signal in a simulation of the generated processor system model of the complete SoC application.

We recommend completing “Streaming Data from Hardware to Software” on page 7-31 example.

Use Recorded Data in Simulation

In this section, you will simulate the processor subsystem of the SoC application model with recorded data as input. The processor subsystem of the SoC application uses AXI4 protocol to stream data from external memory and determine if the signal contained in the data is either high or low frequency. An **IO Data Source** block replaces the external memory and the FPGA subsystem of the model with playback of the AXI4 stream data. You will use data recorded in “Record I/O Data from SoC Device” on page 7-71 example.



1. Open Signal Detection model.

2. Open **IO Data Source** block mask.

3. Click **Browse...** and select the **matlab\toolbox\soc\socexamples\zynq_sine_data.tgz** file containing recorded data.

4. Click **Select...** and choose the data source within the data file to playback. Click **OK** to close the block mask dialog.

5. Run the Simulink® model and open **Vector Scope** to observe the recorded data.

6. To access the recorded data in MATLAB®, use `socFileReader`.

```
h = socFileReader('zynq_sine_data.tgz');
data = getData(h,'AXI4 stream interface');
```

The returned data is a time series object of 'uint32'. To plot the data in MATLAB convert 'uint32' to 'int32'.

```
plot(data.Time, typecast(data.Data,'int32'));
```

See Also

“Record I/O Data from SoC Device” on page 7-71

Task Execution

This example shows how to simulate task execution and how to generate code and run it on an SoC hardware board.

Application development often includes simulating an algorithm to ensure the correct behavior. Such simulations usually ignore the real-time aspects of an embedded system environment. This may allow certain timing problems to remain undiscovered until the application runs on hardware.

The timing problems often lead to incorrect application behavior. SoC Blockset helps you detect these problems in simulation rather than on hardware. This can help you avoid costly debugging on hardware.

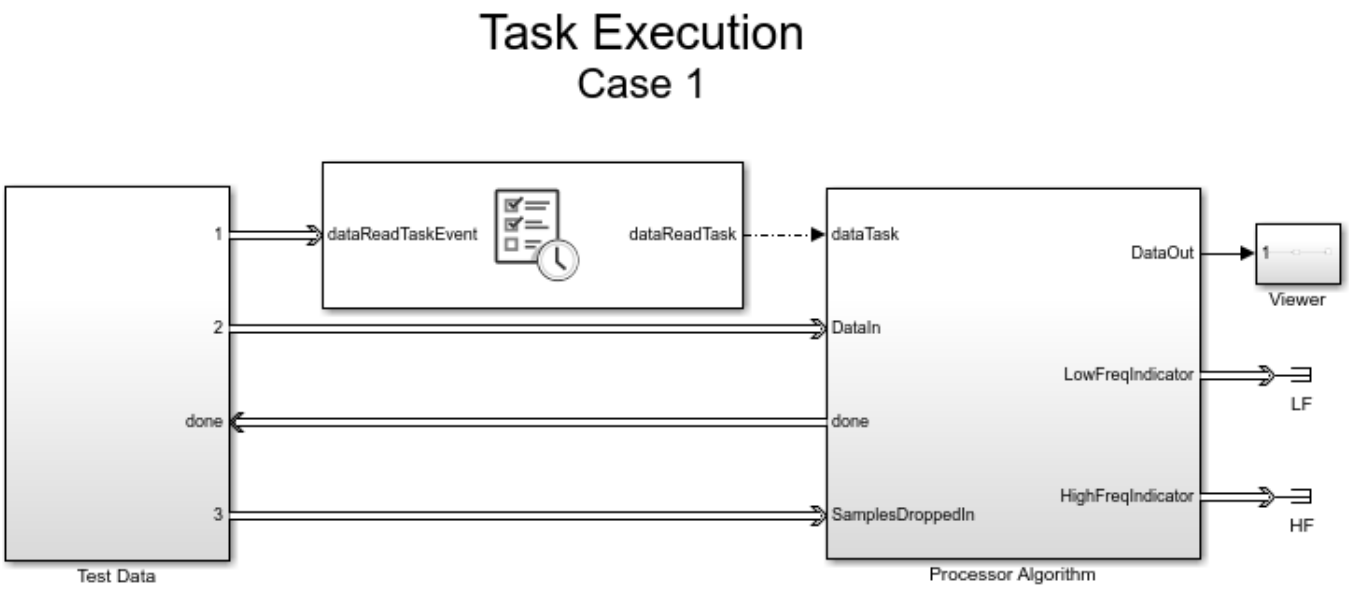
Timing problems are more likely to occur as applications become more complex. For example, rate overruns and undesired rate preemption are more frequent in applications with multiple tasks due to resource constraints and task dependencies. Simulating multitasking applications with SoC Blockset will help you in detecting these problems early.

In this example, task execution is simulated using SoC Blockset. You will learn about different techniques for simulating task duration and when to use them. You will also learn how to verify the timing specifications on hardware.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

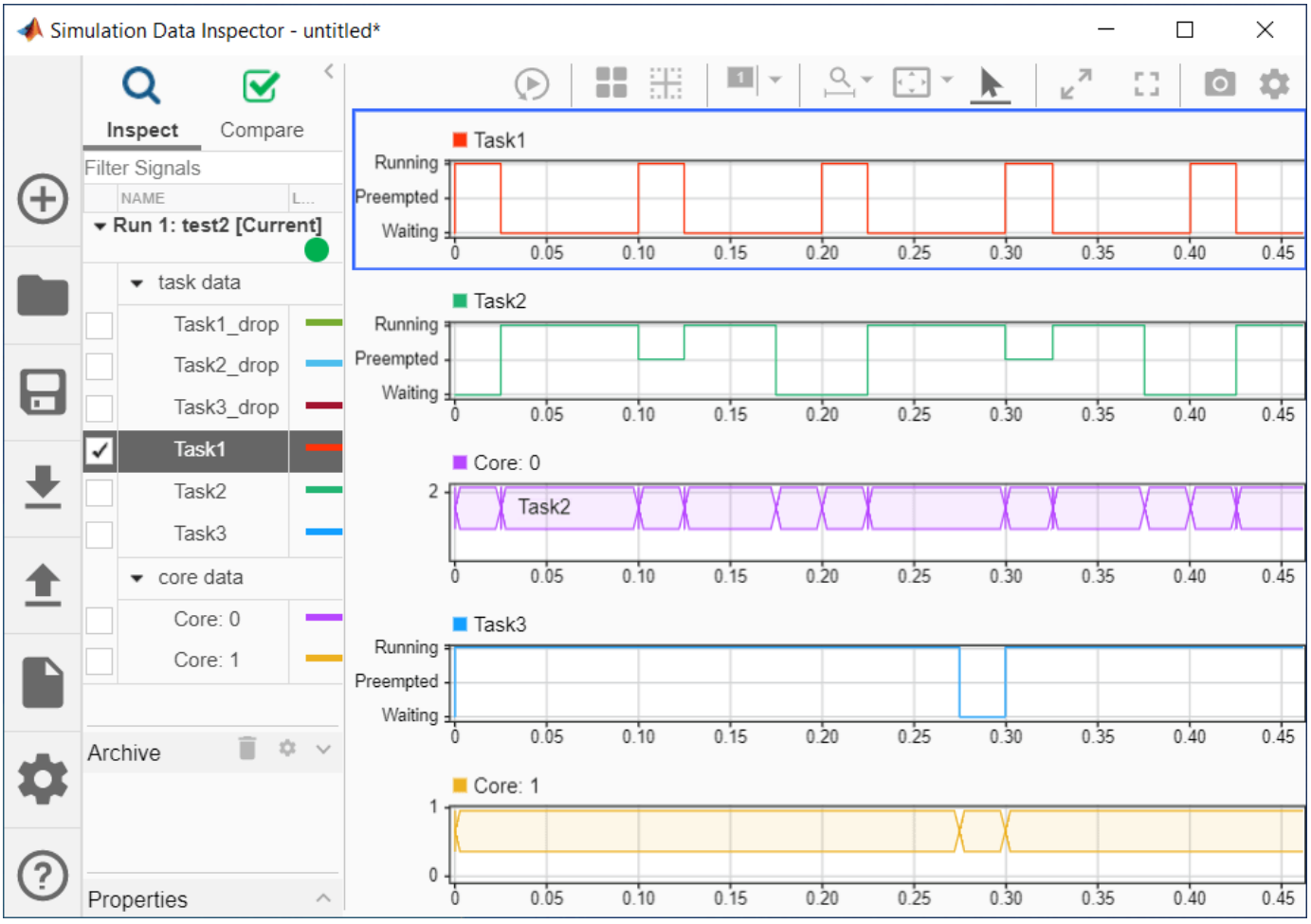
The models used in this example are set for **Xilinx Zynq ZC706 evaluation kit board**. To use a different hardware board, select one of the hardware boards listed in the **Hardware Board** on the **System on Chip** tab. Do the same for the top model and the referenced model.



Introduction

SoC Blockset simulates the execution of software tasks as they would execute on an SoC processor. The simulation honors the parameters of the task, such as period, priority and processor core. SoC Blockset simulates task preemption, task overruns, and concurrent task execution.

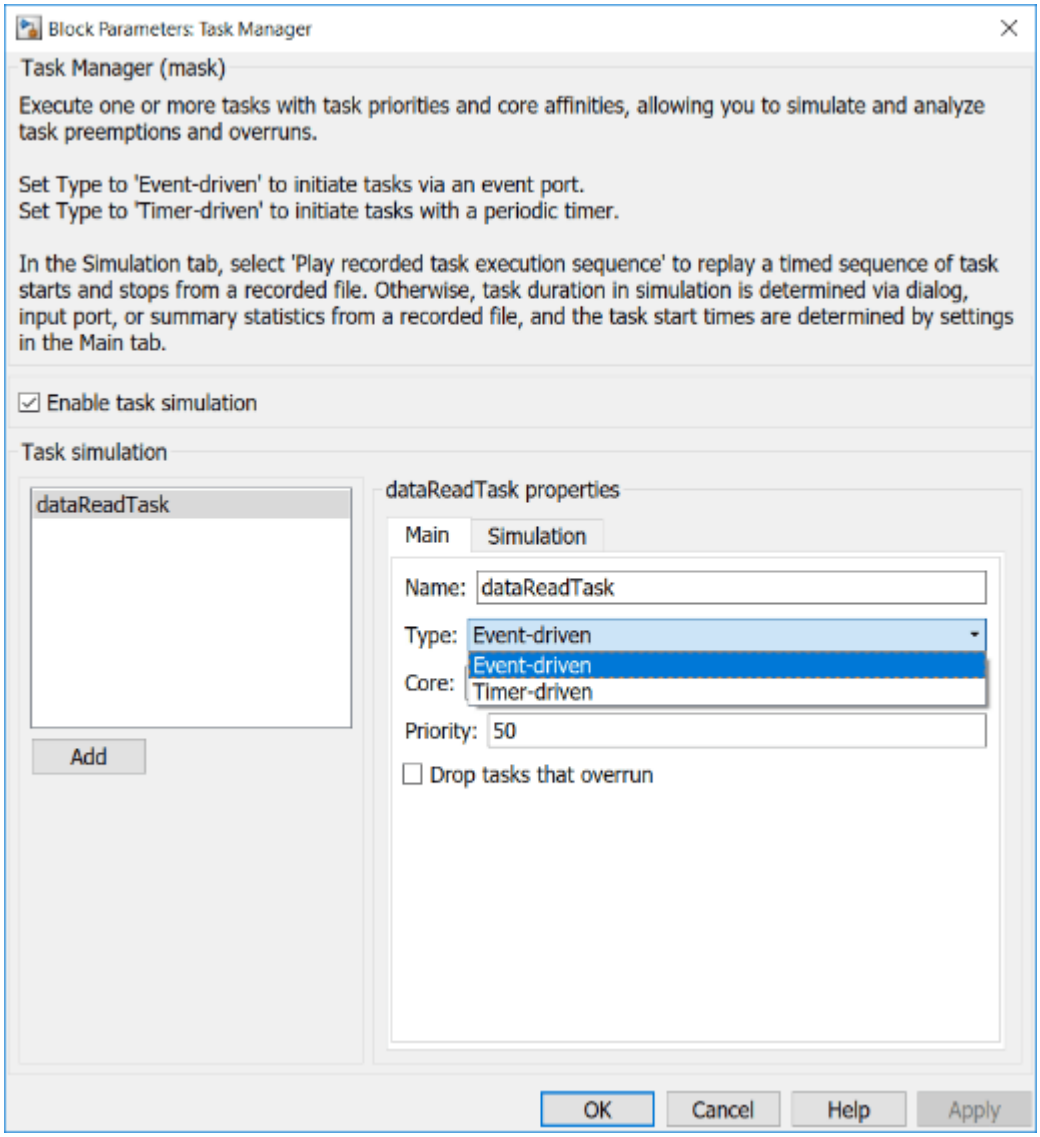
The following diagram illustrates the above-mentioned task execution simulation aspects. In the first two subplots, you can observe that Task1 executes every 0.1 s and, since they both share Core 0, Task1 preempts Task2 that executes every 0.2 s. In the third subplot, you can observe that Core 0 still has some idle time. The last two subplots show Task3 running every 0.3 s on Core 1.



To learn more about simulating task execution, see “What is Task Execution?” on page 3-2

The Task Manager block allows you to configure execution of the tasks in your model. In the block dialog, you define how many tasks you need in your system using **Add** and **Delete** buttons. On the **Main** tab of the dialog, you set the main task properties, while on the **Simulation** tab you set the simulation task properties.

The following figure illustrates the **Main** tab of the **Task Manager** block.



A task has a name so that it can be identified in the model and the various associated plots. Port labels on the **Task Manager** block use the task names for easy identification.

A task can be of two types. An event-driven task executes when triggered by an event. An event line from an IO data source block connected to the **Task Manager** block triggers the task. A timer-driven task executes with a defined period as defined in the **Main** tab of the **Task Manager**.

You define the priority of event-driven tasks in the **Main** tab of the **Task Manager**. Timer-driven task priority is assigned automatically.

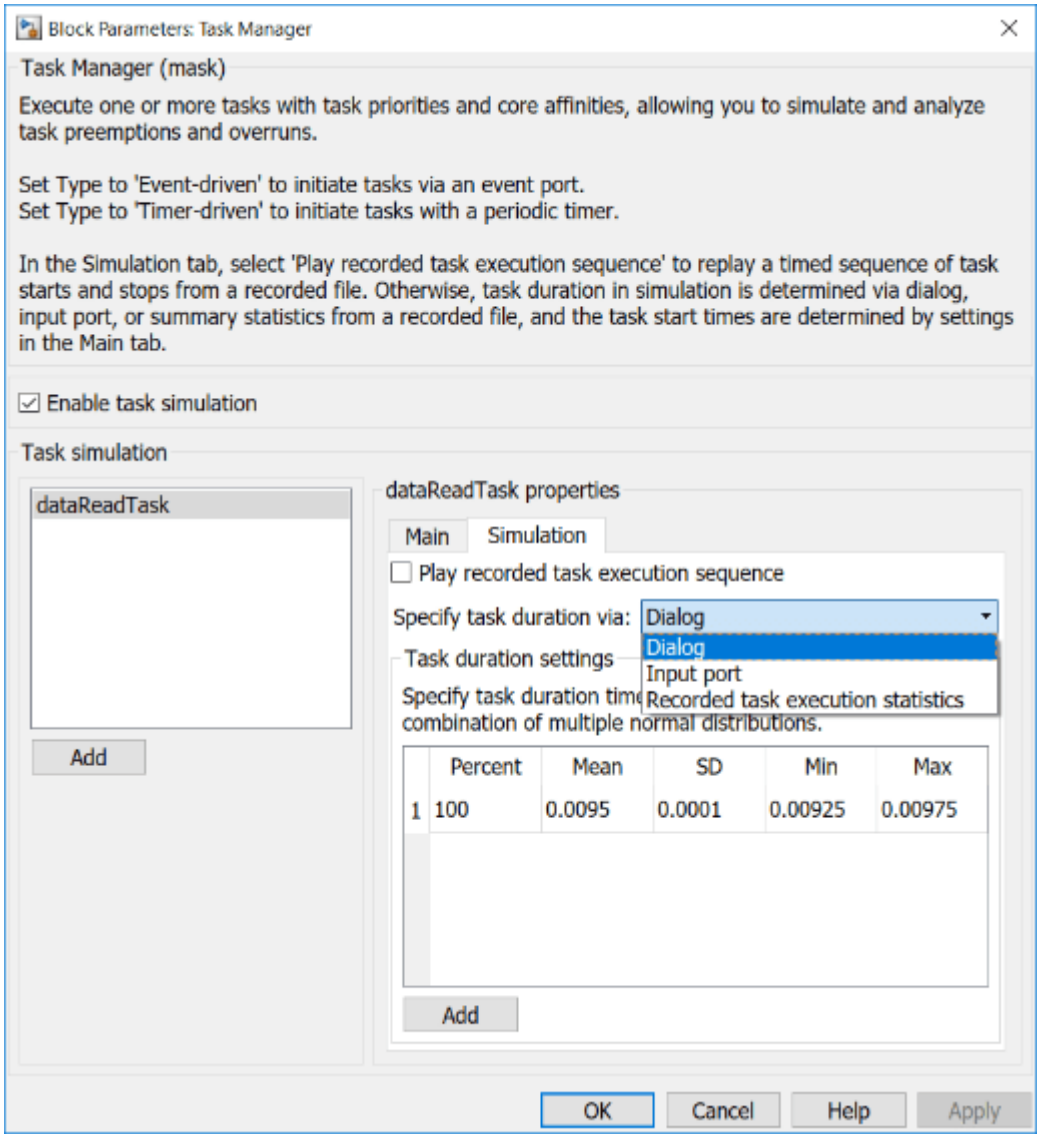
In the **Task Manager** dialog you may also set the processor core on which to execute a task so that, if your hardware board has multiple cores, you may set the tasks to execute concurrently.

The **Task Manager** block also allows you to configure how task overruns are handled. For example, you may decide to drop an instance of a task if the previous task instance has not started or completed. Or, you may decide to try to catch up with the task schedule despite overruns.

To simulate real-time task effects, such as preemption and overruns, SoC Blockset requires you to provide the duration of each task. The duration is defined as the time elapsed between the task start and the task end. Ideally, you will measure the task duration on your hardware board. If that is not possible, look up the task duration in the data sheets provided by the task algorithm developers. As a last resort, you should set the duration relative to the task period or the shortest recurrence interval for aperiodic tasks.

SoC Blockset has several choices for setting the task duration. As the task duration is applied only to simulation, these choices are found in the **Simulation** tab of the **Task Manager** dialog.

The following figure illustrates the **Simulation** tab of the **Task Manager** dialog.

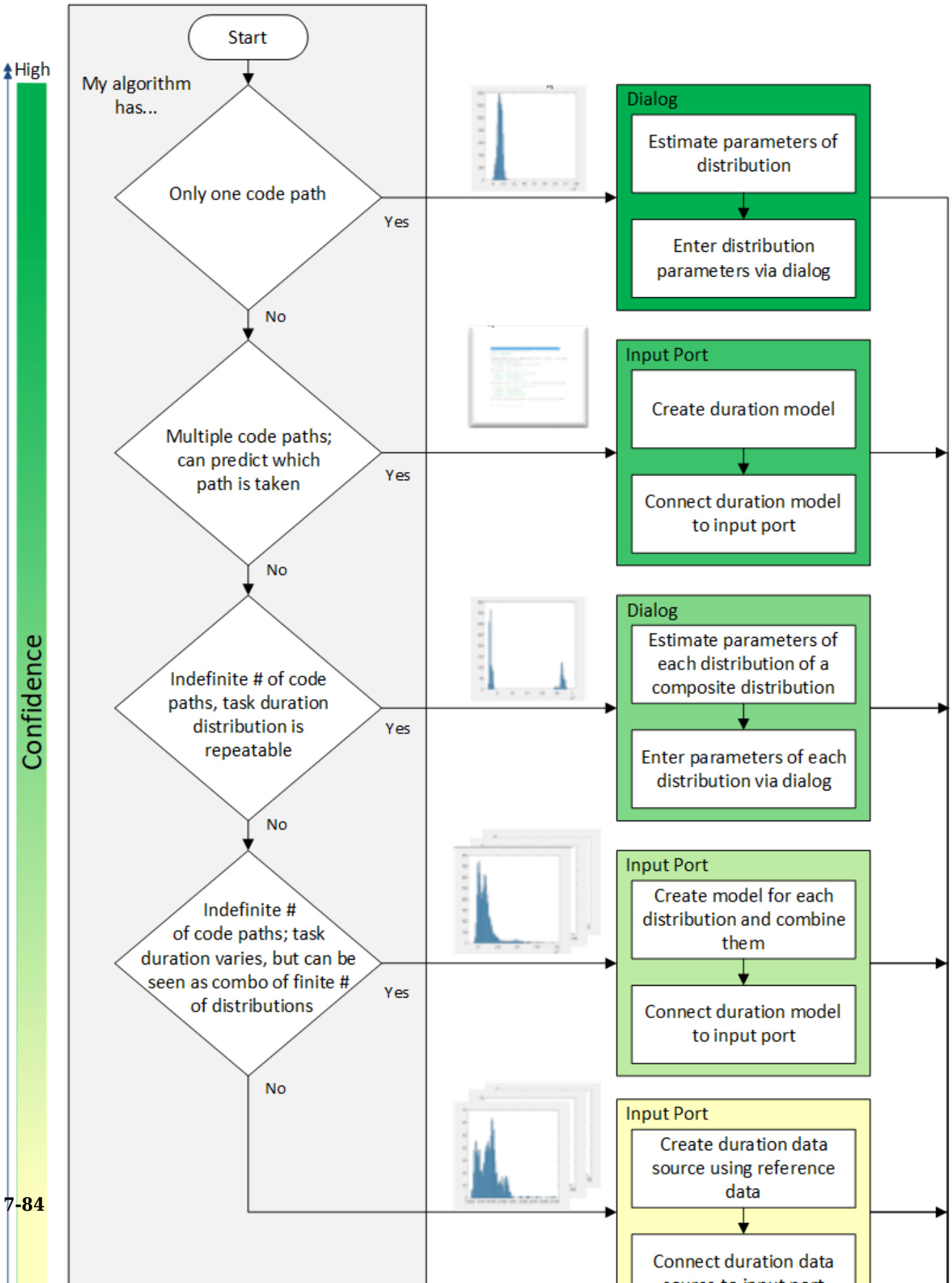


The most commonly used options are:

- **Dialog** - Allows you to specify task duration via a normal distribution, or a combination of multiple normal distributions, using the mean and the standard deviation parameters.

- **Input port** - Allows you to specify task duration on an instance basis. For example, you may create a model that calculates task duration and connect it to the **Task Manager** input port.

The following flowchart will guide you in selecting the most appropriate option.

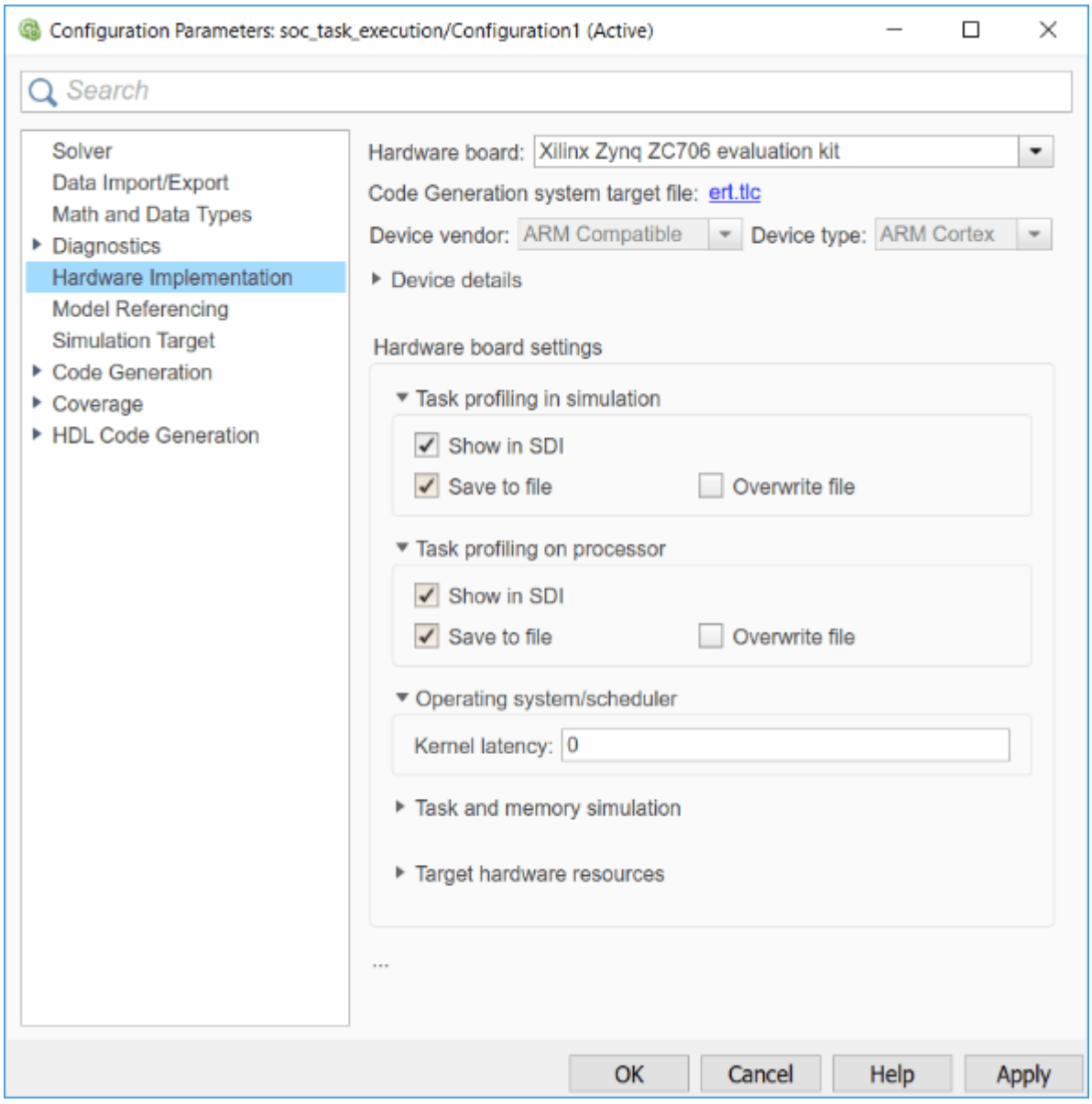

High
Confidence
Start
My algorithm has...
Only one code path
Yes
No
Dialog
Estimate parameters of distribution
Enter distribution parameters via dialog
Multiple code paths; can predict which path is taken
Yes
No
Input Port
Create duration model
Connect duration model to input port
Indefinite # of code paths, task duration distribution is repeatable
Yes
No
Dialog
Estimate parameters of each distribution of a composite distribution
Enter parameters of each distribution via dialog
Indefinite # of code paths; task duration varies, but can be seen as combo of finite # of distributions
Yes
No
Input Port
Create model for each distribution and combine them
Connect duration model to input port
Input Port
Create duration data source using reference data
Connect duration data

If the duration times for your task have different distributions and causes, select the most fitting options using the flowchart as general guidance.

You can configure additional simulation and execution parameters for SoC Blockset in the model configuration dialog. Task profiling, in simulation and on processor, allows you to profile task execution, stream results to Data Inspector and save them into a file.

You can also set the kernel latency value to affect task execution in simulation. This value varies a lot but is typically much smaller than task duration. Therefore, we recommend you leave the value set to 0 s unless you can deterministically find the appropriate value for your hardware board.

The following figure shows SoC parameters related to task execution in the model configuration parameters dialog. Note that the **Task profiling on processor** panel shows only if you install all required products and hardware support packages.



The remaining steps of this example will illustrate some of the options shown in the above flowchart.

Simulating an Algorithm with Single Code Path

This case requires you to simulate a DSP algorithm that processes a frame of data. The following product is required for that:

- DSP System Toolbox

If you do not have this product, proceed to the next case after reviewing the description of this case.

In this case, you will learn how to model the task duration when the task algorithm has a single code path.

Assume that you are tasked with developing an application that processes RF (radio frequency) data on an SoC board. After being preprocessed in the FPGA core, the data is streamed to the processor core using the AXI4 protocol. The algorithm running on the processor core should determine whether the data contains a high-frequency or a low-frequency signal. To that end, a low-pass and a high-pass filter are applied to the data. The resulting signals are then compared to a selected threshold. Based on this description, this task has a single code path, with no major code branches. The source code for the task function might have the following form.

```
double dataReadTask(double in[])
{
    /* Frame size is always 1000 */
    int signalType; /* 0 - LP, 1 - HP */
    double out1[1000], out2[1000];
    filterLP(in, out1, 1000);
    filterHP(in, out2, 1000);
    signalType = thresholding(out1, out2, 1000);
}
```

1. Open the model. Note the **Test Data** subsystem. The **RF Data Source** block in the subsystem represents the external memory and the FPGA core. The **RF Data Source** block has two output ports, **Stream Data** and **event**. They output the RF data and a notification when new data frame is available, respectively.

2. Note that the **RF Data Source** block generates frames of 1000 samples every 0.01 s. The frames are samples of a 1 kHz sine waveform.

3. Click the **Task Manager** block. Observe that it sets an event-driven task **dataReadTask**. The task is triggered by the arrival of a new data frame.

4. Click the **Simulation** tab in the **Task Manager** dialog to define the task duration for simulation.

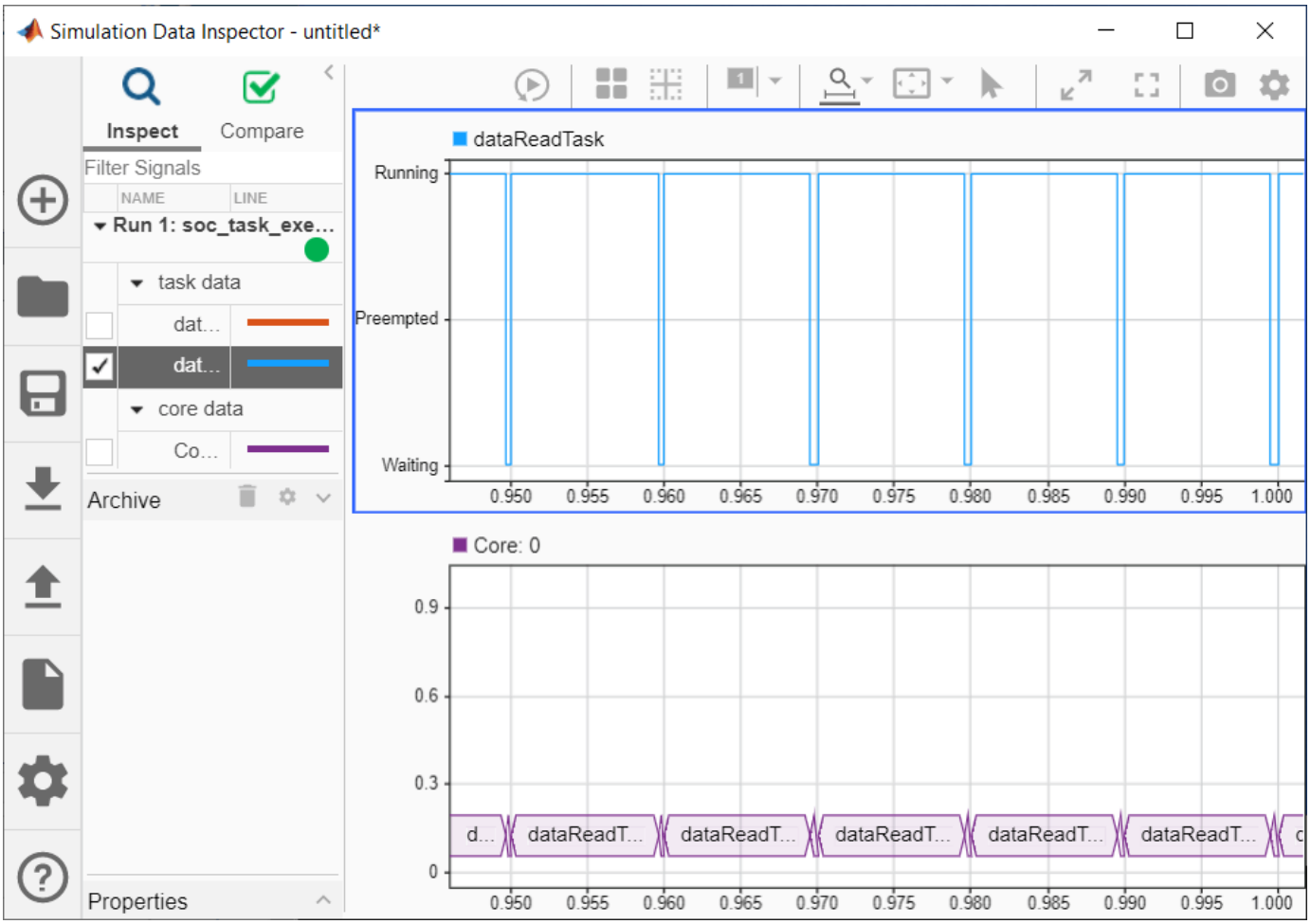
Since the algorithm consists of two filters executing without conditions, the application has a single code path. Therefore, you follow the first left branch in the flowchart shown in the introduction and you expect that the algorithm execution times have a normal distribution.

Based on the information given by the algorithm developer, you determine that the mean execution time is 0.0095 s and that the standard deviation is 0.0001 s. To represent the real-time limits, you also decide to set the min and the max execution times to 0.00925 s and 0.00975 s, respectively.

Set the duration parameters in the **Task Manager** dialog in the **Simulation** tab as described above.

5. In the model, click **Run** to start the simulation. Wait until the simulation completes.

6. From the model toolbar, open the **Data Inspector** and inspect the **dataReadTask**. Zoom in to inspect the task execution times more closely.

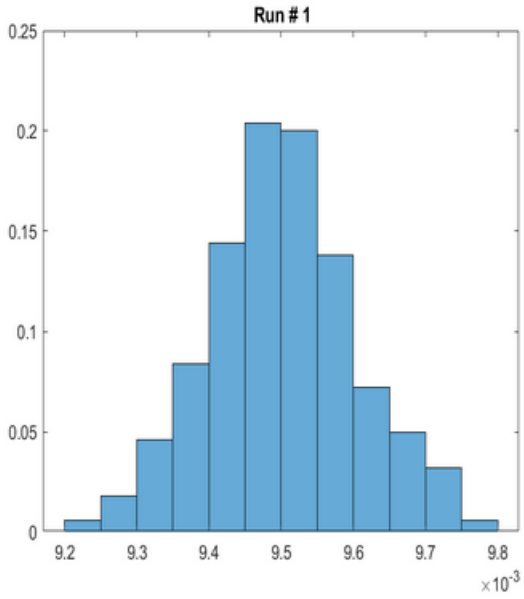


7. Run the following command to perform the statistical analysis of the task execution times. Observe the **Data Inspector** run numbers. Modify the command if your run numbers are different.

```
socTaskTimes('soc_task_execution', 'Run 1: soc_task_execution')
```

Task: dataReadTask

Histogram of the execution times



Statistics of the execution times

	Mean	SD	Min	Max
Run # 1	0.0095026	0.00010217	0.00925	0.00975

Observe that the task durations vary. As expected, the histogram of the task duration times indicates that the algorithm has one code path. The duration values are clustered around the mean value of 0.0095 s.

8. Close the model without making any changes.

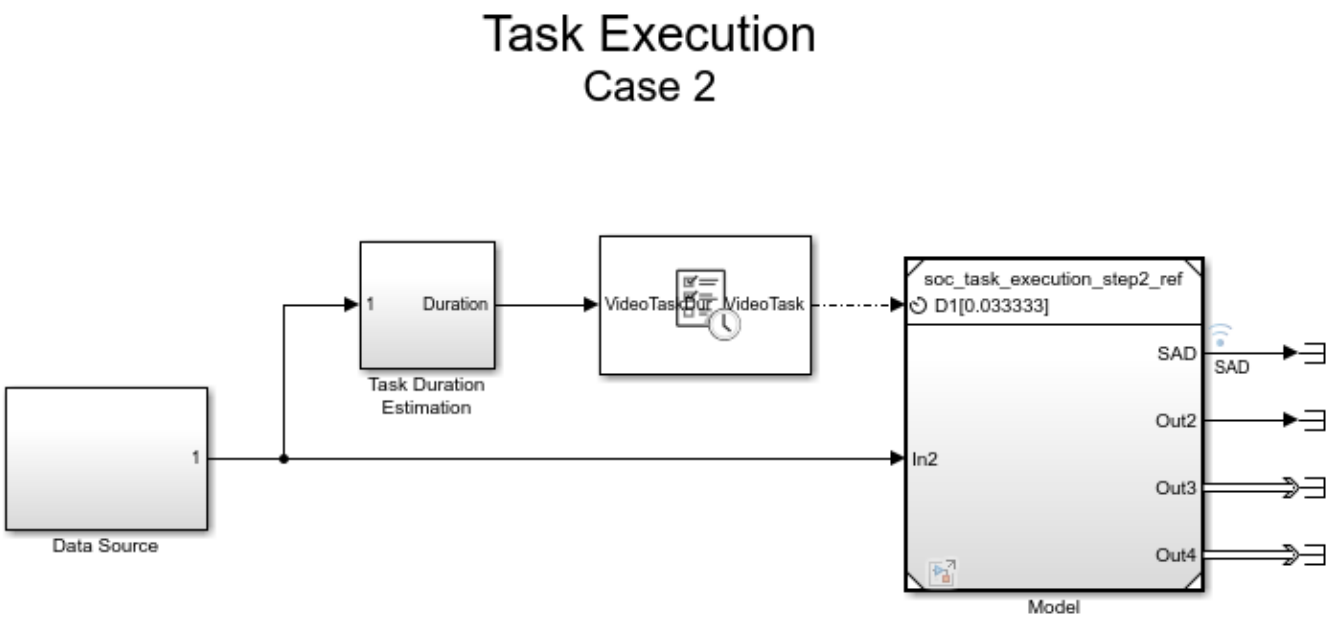
Simulating an Algorithm with Two Code Paths

In this case, you will learn how to model the task duration when the task algorithm has two code paths and it can be predicted which path will be taken.

Assume that you are developing a video surveillance application. The task is to constantly process video data to determine if there was intrusion in the system. The algorithm calculates the amount of scene change between consecutive video data frames. If the scene change exceeds the selected threshold, such frames are recorded as they may be used as evidence of potential intrusion. Thus, this algorithm has two code paths. The source code of this algorithm may be represented in the following form.

```
void VideoTask(single in[], in length, double threshold)
{
    double energy;
    energy = calcSceneChange(in, length);
    if (energy > threshold)
```

```
            recordFrame(in, length);
        }
    }
```



1. Open the model. Note the **Data Source** block that outputs the frames of video data.

2. Click the Model block and observe that the algorithm calculates motion energy between consecutive frames of data. If the calculated motion energy exceeds the threshold, the **Main Algorithm** is executed.

3. Click the **Task Manager** block. Observe that it sets a timer-driven task **VideoTask**. This task runs every 0.33333 s, which is the video frame rate.

4. Click the **Simulation** tab in **Task Manager** dialog to define the task duration for simulation.

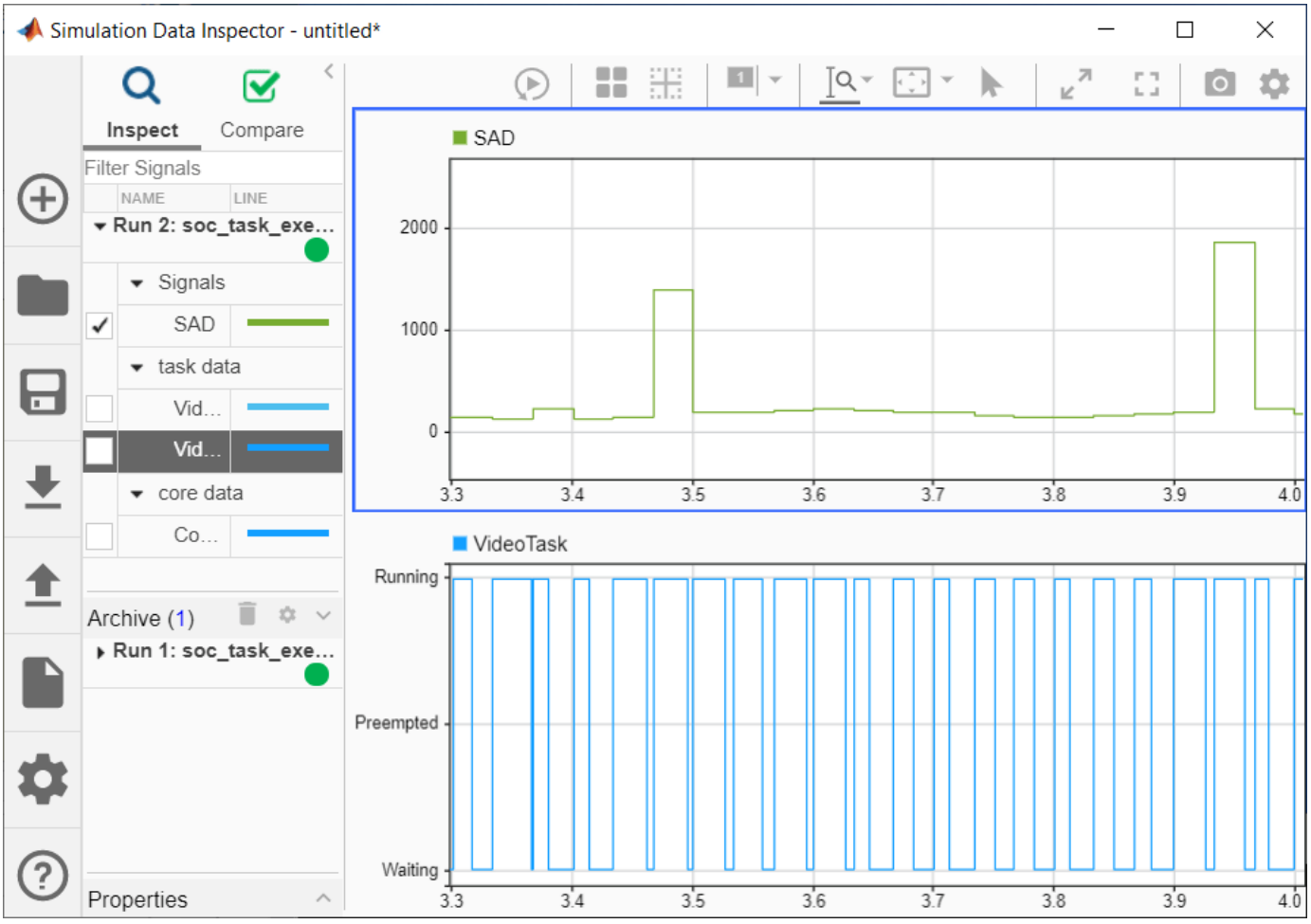
Since the algorithm has two code paths and it can be predicted which code path will be taken, follow the second left branch in the flowchart.

Model task duration to depend on motion energy. Depending on whether the motion energy threshold is exceeded or not, you will assign the task duration with the mean of 75% or 50% of the frame rate, respectively.

Click the **Task Duration Estimation** subsystem to understand how to model task duration.

5. In the model, click **Run** to start the simulation. Wait until the simulation completes.

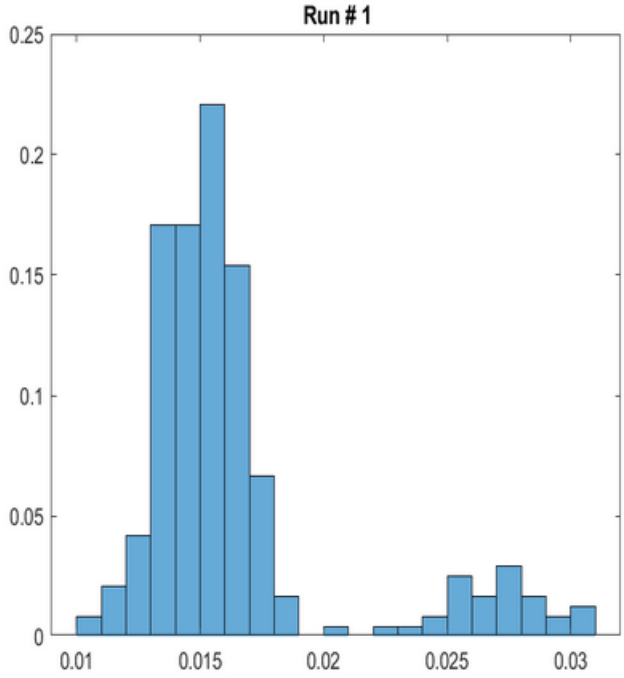
6. From the model toolbar, open the **Data Inspector** and inspect **VideoTask**. Zoom in to inspect the task execution times more closely.



7. Run the following command to perform the statistical analysis of the task execution times. Observe the **Data Inspector** run numbers. Modify the command if your run numbers are different.

```
socTaskTimes('soc_task_execution_step2', 'Run 2: soc_task_execution_step2')
```

Histogram



Statistics

	Mean	SD	Min	Max
Run # 1	0.016538	0.0043039	0.01069	0.030948

Observe that the task durations vary. As expected, the histogram of the task duration times indicates that the algorithm has two code paths.

8. Close the model without making any changes.

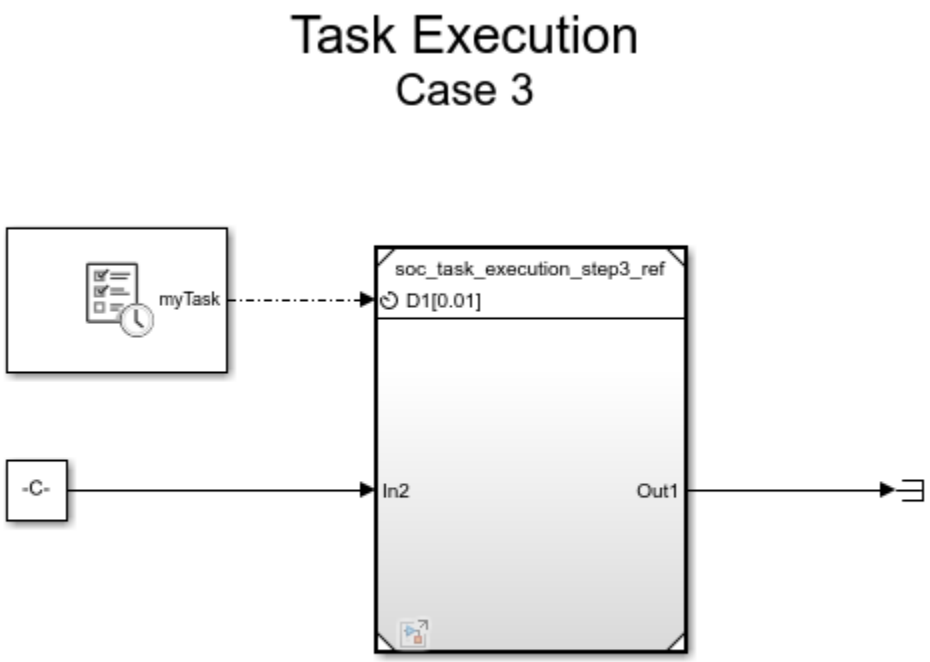
Simulating an Algorithm with Indeterminate Number of Code Paths

In this case, you will learn how to model the task duration when the task algorithm has an indeterminate number of code paths, but the code paths are repeatable for the given set of data.

In this case, assume that you are developing a complex application that processes data on an SoC board. Due to the complexity of the processing, the algorithm has an indeterminate number of code paths. As a result, it is not possible to predict which code path will be taken. However, it is known that the distribution of task durations is repeatable in multiple experiments. The source code for such an algorithm might have the following form.

```
int myTask(int arr[], int length)
{
    int i = 0;
    int sum = 0;
    while (i < length) {
```

```
        if (arr[i] > 0)
            sum = sum + arr[i]
        i++;
    }
}
```



1. Open the model.

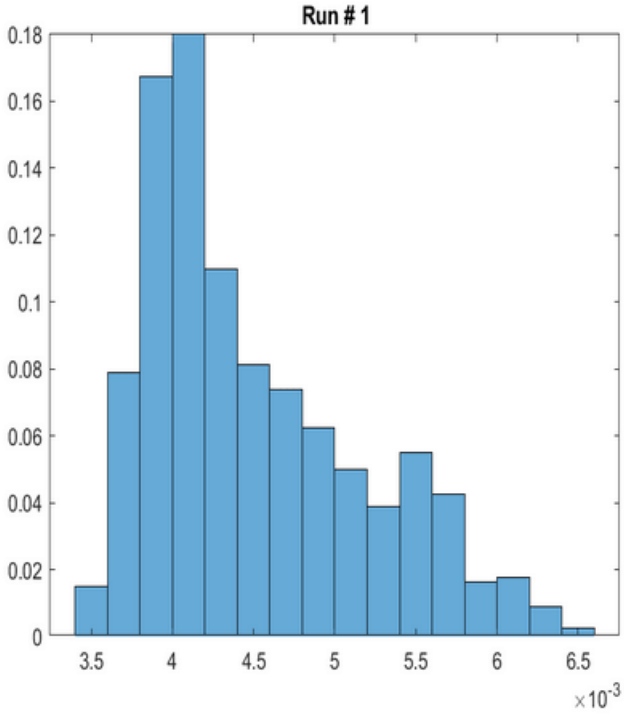
2. Click the **Task Manager** block and select the task **myTask**. Click the **Simulation** tab. Observe that we define the probability distribution as a combination of two normal distributions.

3. Click **Run** to start the simulation. The task execution data will be streamed to the **Data Inspector**.

4. Run the following command to perform the statistical analysis of the task execution times obtained in simulation. Observe the **Data Inspector** run number. Modify the command if your run number is different.

```
socTaskTimes('soc_task_execution_step3', 'Run 3: soc_task_execution_step3')
```

Histogram



Statistics

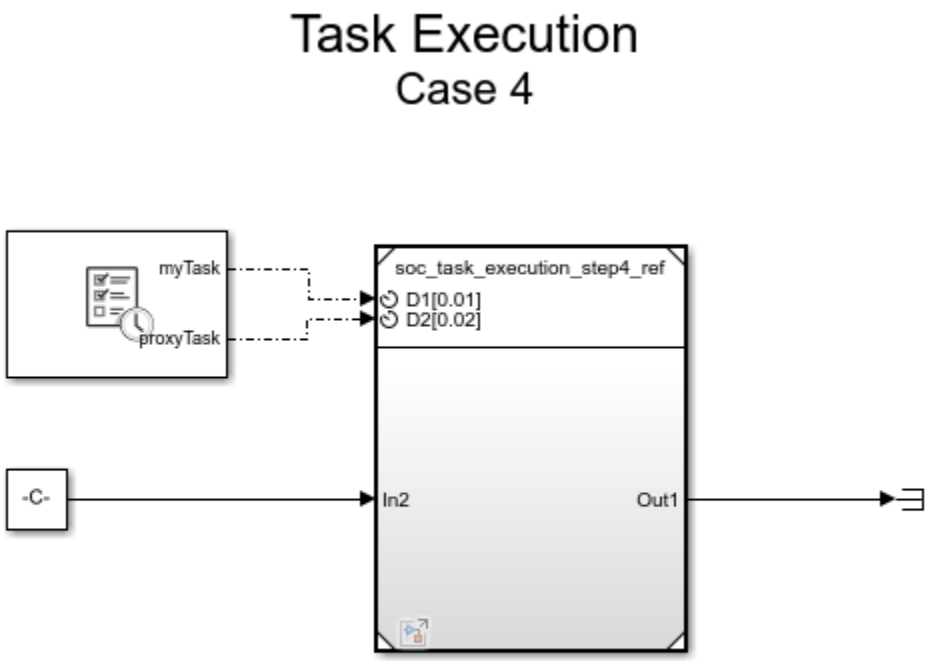
	Mean	SD	Min	Max
Run # 1	0.0045092	0.0006698	0.0035	0.0065432

Notice that the task duration distribution obtained in simulation matches the expected results.

5. Close this model without making any changes.

Simulating an Application using Proxy Tasks

Assume that you are developing a complex application that adds one more task to the model developed in the previous case. The implementation of this task is not currently available, but the timing specification for this task is known. The task executes every 0.02 s with the duration described by a normal distribution. The distribution has a mean of 0.008 s and standard deviation of 0.0009 s.



1. Open the model.

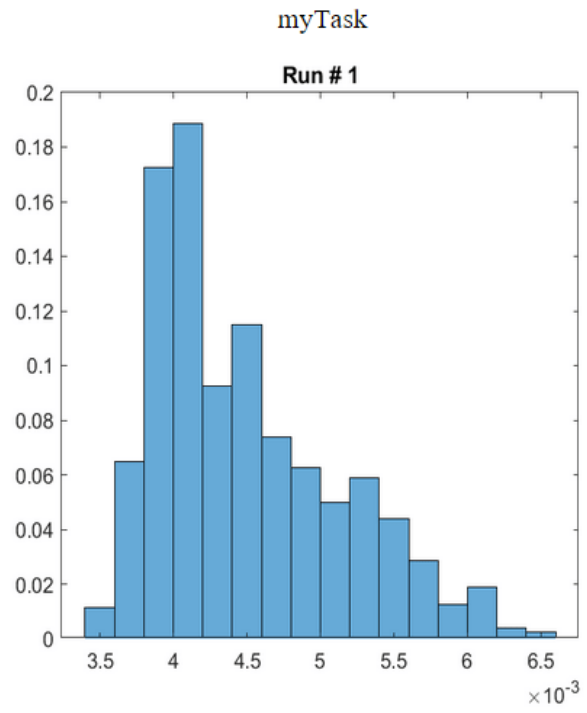
2. Click the **Task Manager** block and select the task **proxyTask**. Click the **Simulation** tab. Observe that we define the probability distribution as a normal distribution with the parameters mentioned in the introduction of this task.

3. Inside the Model block open the Proxy Task block and inspect the sample time value. The sample time value must match the period value entered in the Task Manager block. Click Cancel.

4. Click **Run** to start the simulation. The task execution data will be streamed to the **Data Inspector**.

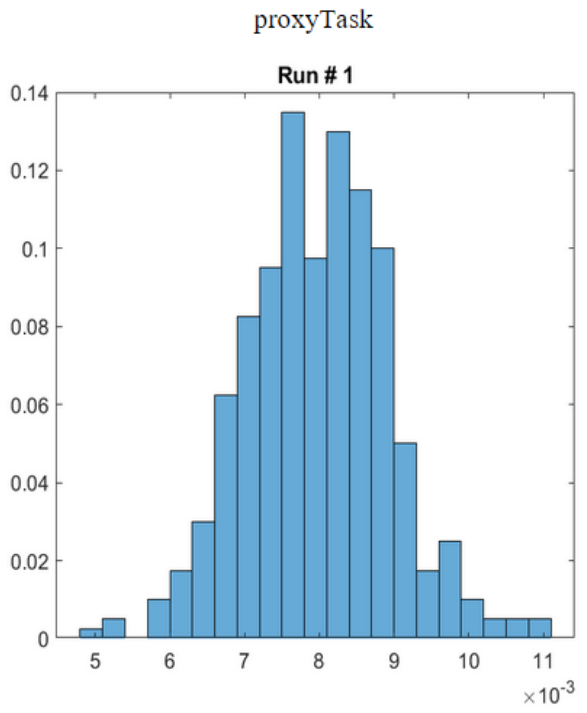
5. Run the following command to perform the statistical analysis of the task execution times obtained in simulation. Observe the **Data Inspector** run number. Modify the command if your run number is different.

```
socTaskTimes('soc_task_execution_step4', 'Run 4: soc_task_execution_step4')
```



Statistics

	Mean	SD	Min	Max
Run # 1	0.0044951	0.00062669	0.0035	0.0064749



Statistics

	Mean	SD	Min	Max
Run # 1	0.0079778	0.00095311	0.0050912	0.011067

Notice that the task durations obtained in simulation for both application and the proxy tasks.

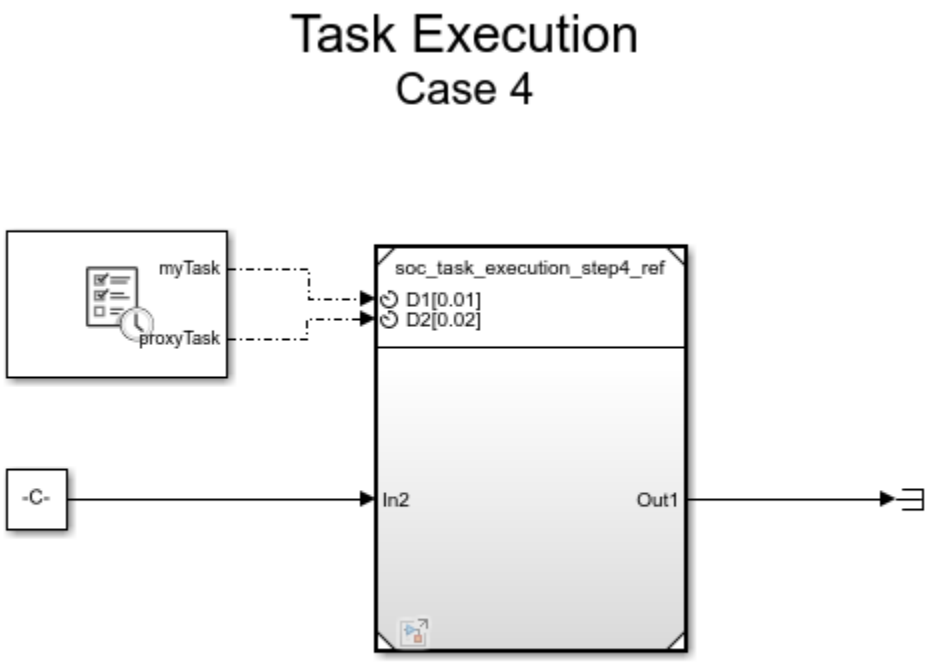
6. Close this model without making any changes.

Compare the Simulation Results with the Results on Hardware

In this section, you will compare the timing results obtained in the previous simulation to the timing results obtained on your hardware board.

Required products:

- Embedded Coder
- SoC Blockset Support Package for Xilinx Devices, or
- SoC Blockset Support Package for Intel Devices



1. On the **System on Chip** tab, click **Configure, Build & Deploy**.

2. Follow the SoC Builder workflow until you get to the **Select Build Action** screen.

3. Select **Build and load for external mode** and continue until you complete the workflow.

4. Click on **Monitor & Tune** to deploy the model to the hardware. The model is already set to profile task execution as it runs on hardware and stream the profiling data to **Data Inspector** in real-time.

5. Run the following command to perform the statistical analysis of the task execution times obtained on hardware. Observe the **Data Inspector** run number. Modify the command if your run number is different.

```
socTaskTimes('soc_task_execution_step4', 'Run 5: soc_task_execution_step4')
```

Notice that the task durations obtained on hardware match the results obtained in simulation.

6. Close this model without making any changes.

Summary

This example showed you how to simulate task execution in a multitasking operating system, how to generate code and run it on a hardware board, and how to collect the real-time task execution data.

In this example, we used simple applications, each with one task. In a typical application, however, multiple tasks must be performed. Embedded applications must run each task per defined schedule.

To allow for using the processor most efficiently and to react quickly to external events, a priority-based preemptive scheduling algorithm is used.

With priority-based preemptive scheduling, when a task gets preempted, a task switch occurs. The data used by the task (task context) is saved so that it can be restored when the task resumes executing. In this example, the task switching times are dwarfed by the task duration and are not simulated. In applications with much shorter task duration, you may need to consider them.

If a hardware board has multiple processor cores, embedded applications typically attempt to use all cores for the most efficient implementation. SoC Blockset uses a priority-based preemptive scheduling algorithm even when the processor has multiple cores. SoC Blockset honors assignment of tasks per core in both simulation and generated code.

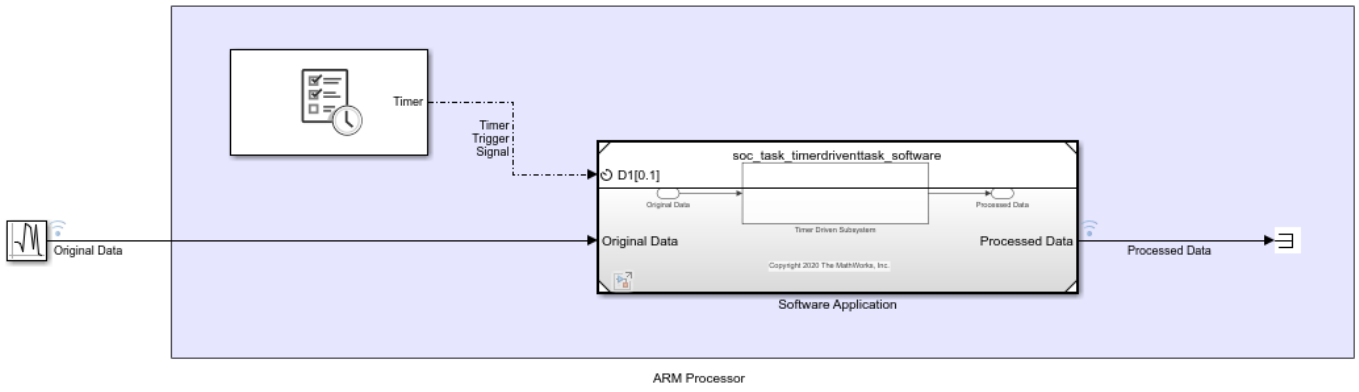
Next, we recommend completing “Streaming Data from Hardware to Software” on page 7-31 example that illustrates a systematic approach to designing a complex SoC application using SoC Blockset.

Timer-Driven Task

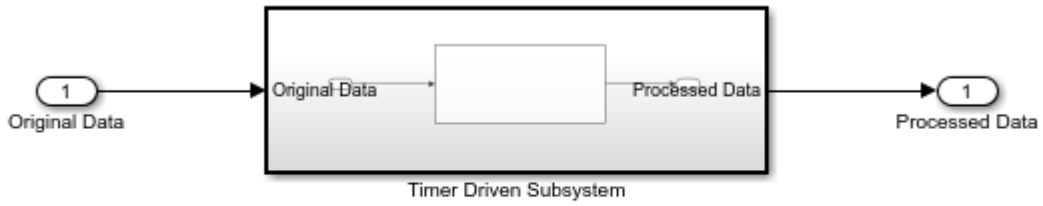
This example shows how to use the Task Manager block in a simple system where a timer-driven task samples and modifies data generated from a random number source.

Task Manager and Software Application Model

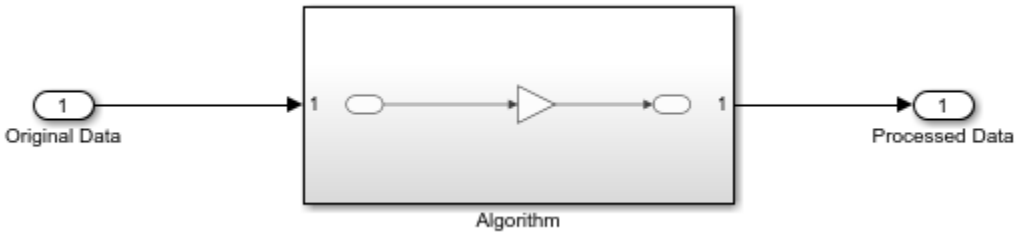
The following model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the `Timer Driven Subsystem`, inside the `Software Application` Model block. A Random Number block simulates an data source that the timer driven task samples.



The following model shows the Software Application model. This model contains the Timer-Driven Subsystem that executes based on the Timer Task events from the Task Manager block in the top-level model.

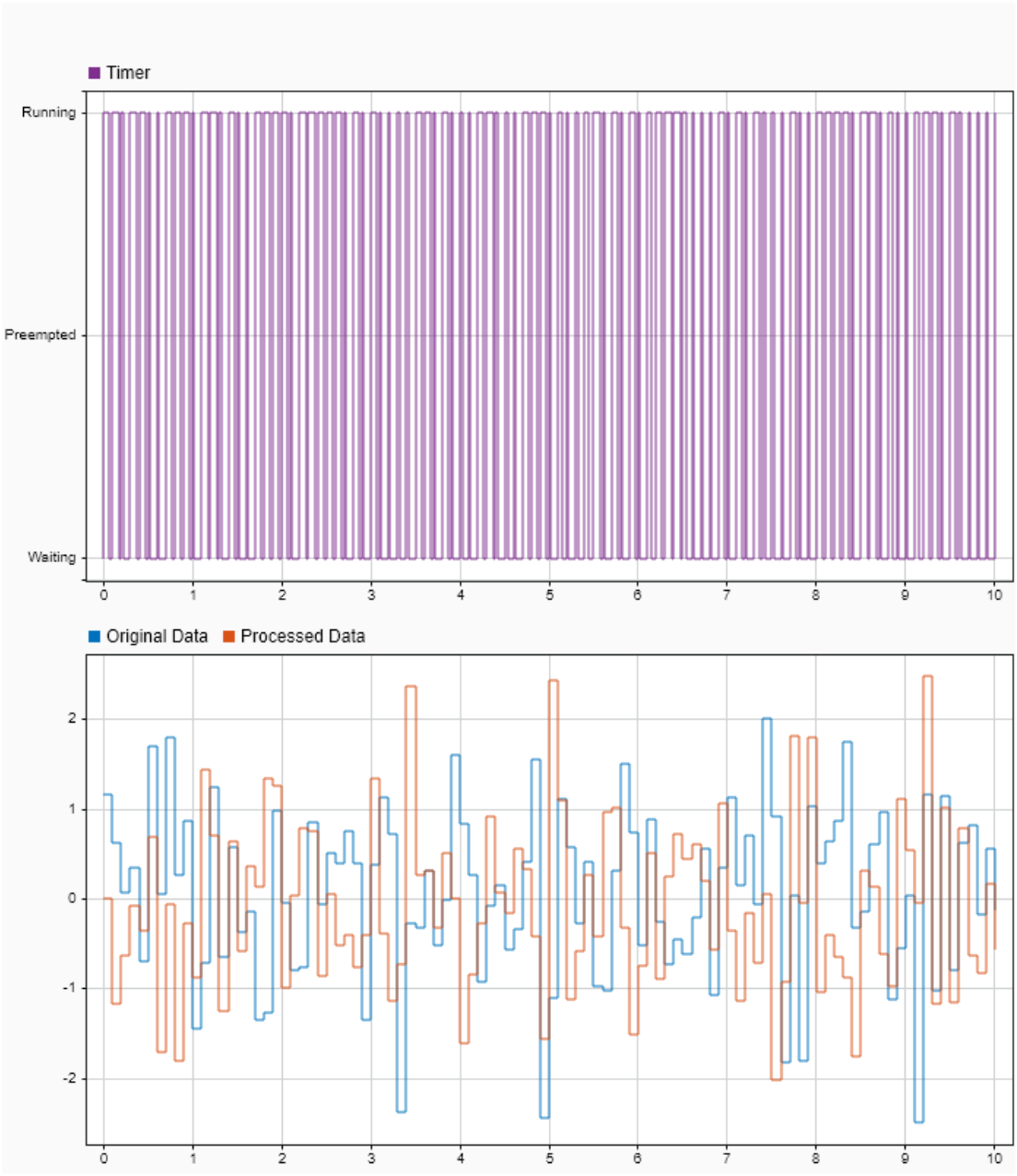


The Timer-Driven Task Subsystem, a Subsystem, samples a data value every `0.1` seconds from the Random Number block and applies the Algorithm. In this model, the algorithm outputs the negative scalar value of the sampled data value. The following model shows the Algorithm subsystem contained in the Timer-Driven Subsystem. The Inport block defines the 0.1 second sampling time for the Timer Driven Subsystem visible on the Software Application model when the Schedule rates parameter is enabled.



Simulation and Results

Click the Run button to build and run the model. When the model finishes running, open the Simulation Data Inspector to see the results of the simulation. Select the Timer_Task, original data, and processed data signals to see the effect of asynchronous task execution.



As shown in the Simulation Data Inspector, the running time of the Timer_Task varies at each instance. In some cases, the duration of the previous task execution delays the start of the next task execution. Additionally, the processed data from the task outputs at a the same time as the completion of the task execution, resulting in observed delay in the processed Data compared to the original data. As a result, despite the specified time step of 0.1 seconds, the start of execution now

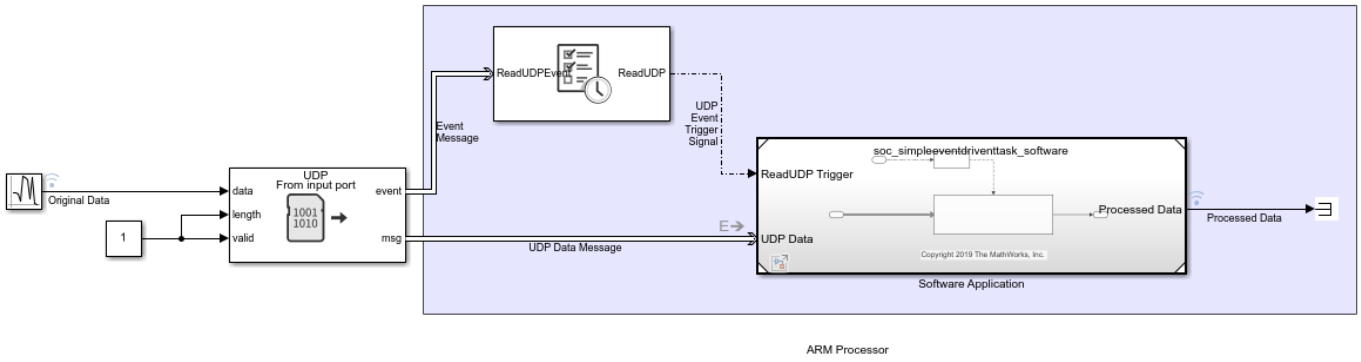
behaves as if the subsystem were executed on an SoC device processor with the associated real world processing limitations.

Event-Driven Task

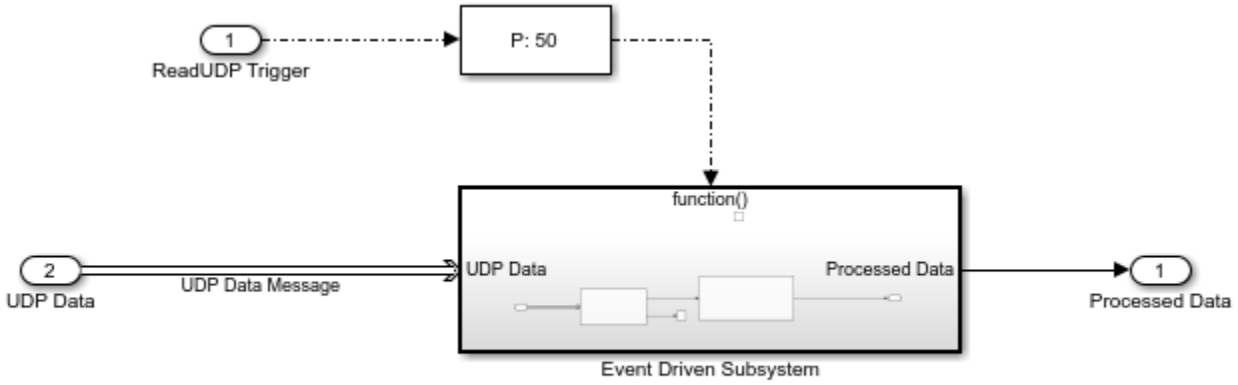
This example shows how to use the task manager block to a simple system where data from UDP source gets processed asynchronously each time a data packet arrives. The task manager block

Task Manager and Software Application Model

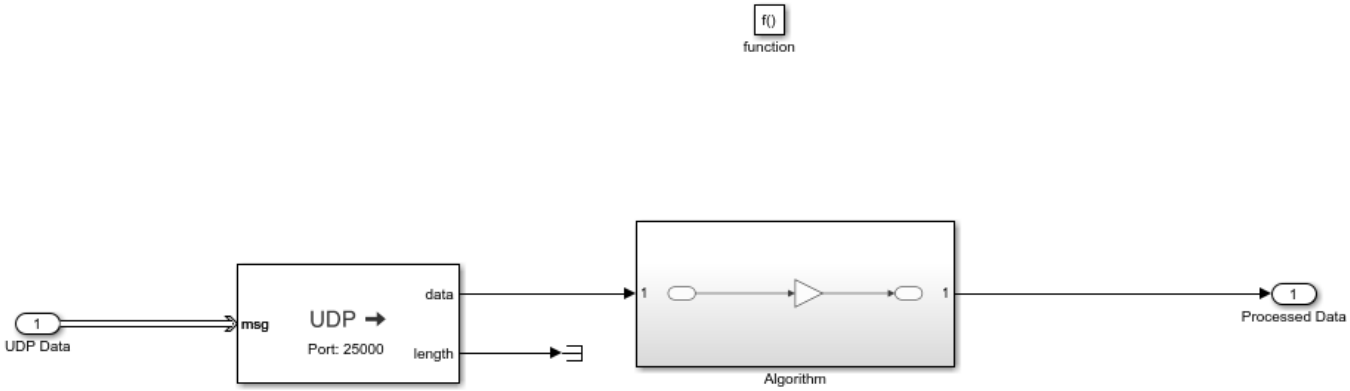
The following model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the Asynchronous Subsystem, inside the Software Application Model block. An IO Data Source block simulates the network transmission of UDP packets.



The Software Application contains the Asynchronous Task Subsystem, a Function-Call Subsystem, that executes each time an event trigger occurs. An Asynchronous Task Specification block specifies the priority of the UDP Task to match the priority set in the Task Manager block. A Rate Adaptor block allows sampling of the output signal of the Asynchronous Task Subsystem at the time step of the Simulink(c) model.

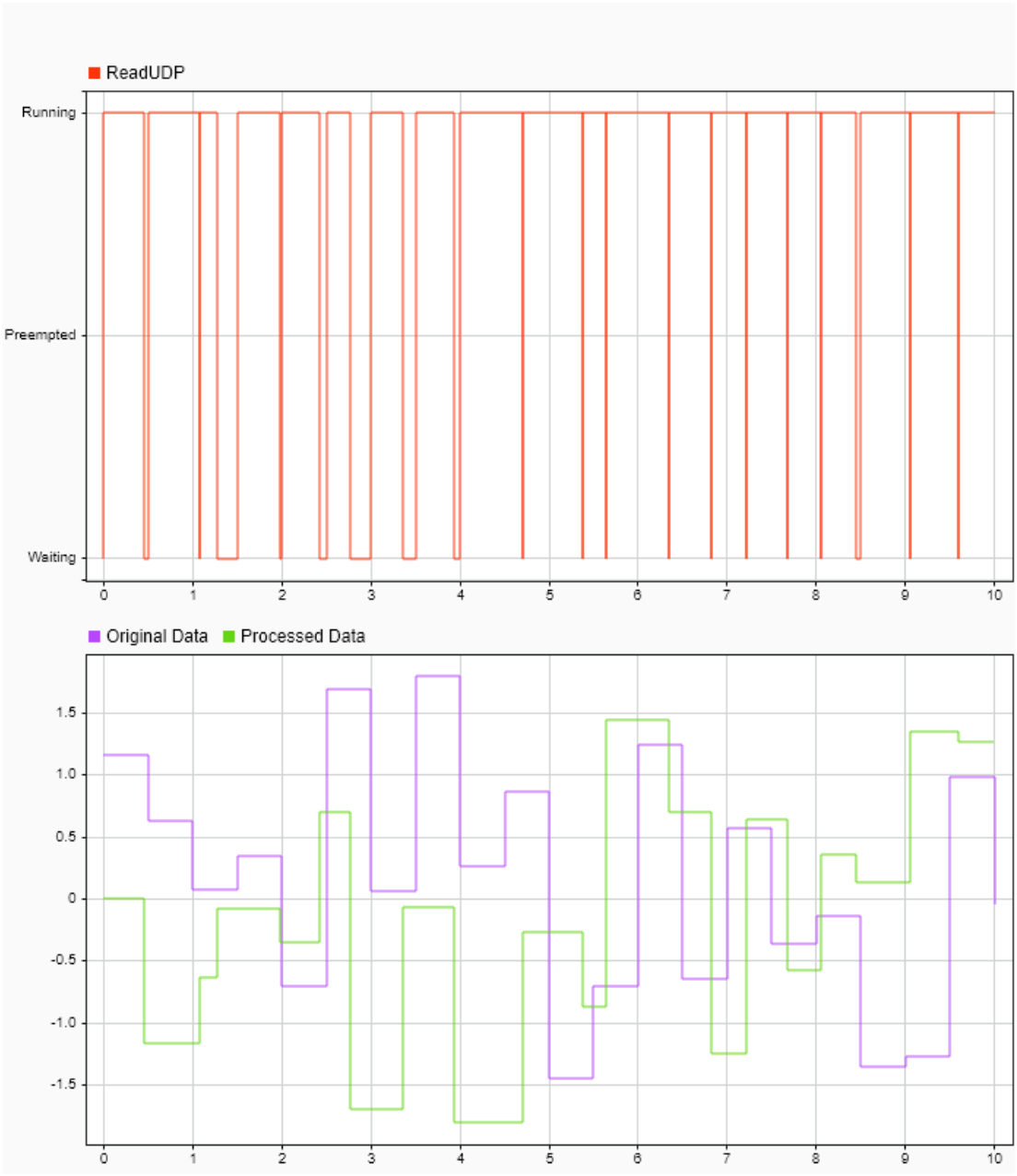


The Asynchronous Task Subsystem, a Function-Call Subsystem, reads a data value from a UDP Read block and applies the Algorithm each time a new UDP data value arrives. In this model, the algorithm outputs the negative scalar value received from the UDP Read block. The following model shows the UDP block and Algorithm subsystem contained in the function-call subsystem.



Asynchronous Simulation and Results

Click the Run button to build and run the model. When the model finishes running, open the Simulation Data Inspector to see the results of the simulation. Select the ReadUDP, original data, and processed data signals to see the effect of asynchronous task execution.



As shown in the Simulation Data Inspector, the Running time of the ReadUDP varies at each instance of receiving a UDP data packet. In some cases, the previous task execution delays the start of the next task execution. While, in this example, the UDP packets arrive at a fixed rate relative to the Simulink sample time, the start of the task execution is not directly dependent on the sample time.

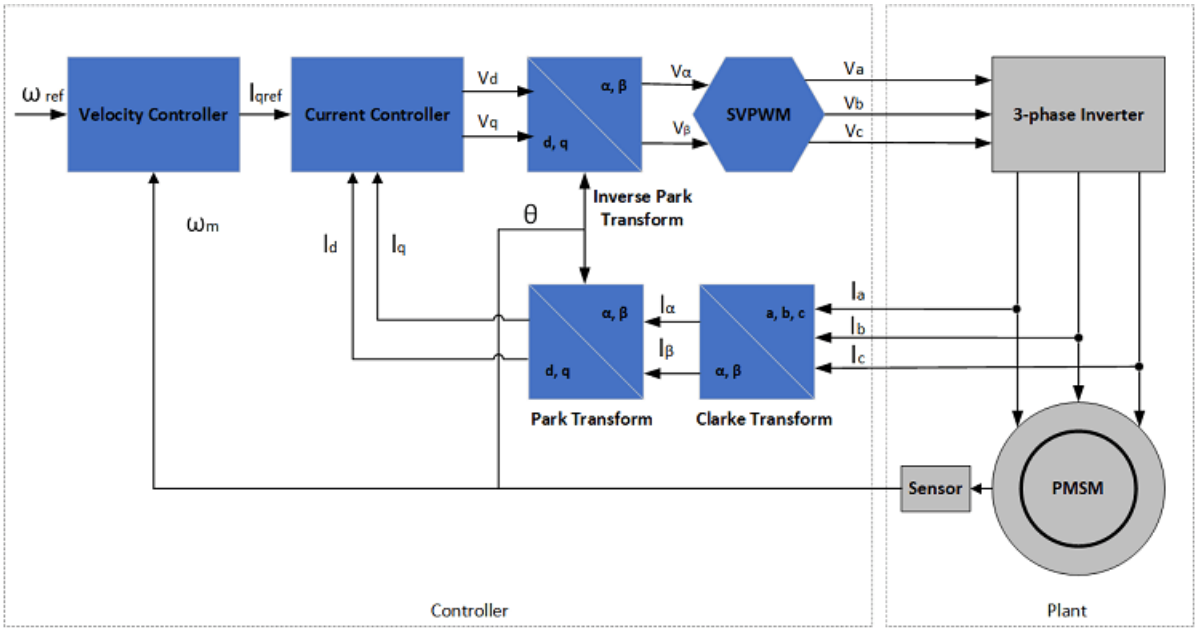
The processed data from the task outputs at a the completion of the task execution, resulting in observed delay in the processed Data compared to the original data.

Hardware-Software Partitioning of a Motor Control Algorithm

This example shows how to model a motor controller for SoC devices by partitioning the control and calibration algorithms between the FPGA and processor of the SoC.

Introduction

This example shows how to partition a Field-Oriented Controller (FOC) for a Permanent Magnet Synchronous Motor (PMSM) onto an SoC device. The following diagram shows a conceptual closed-loop FOC of PMSM.

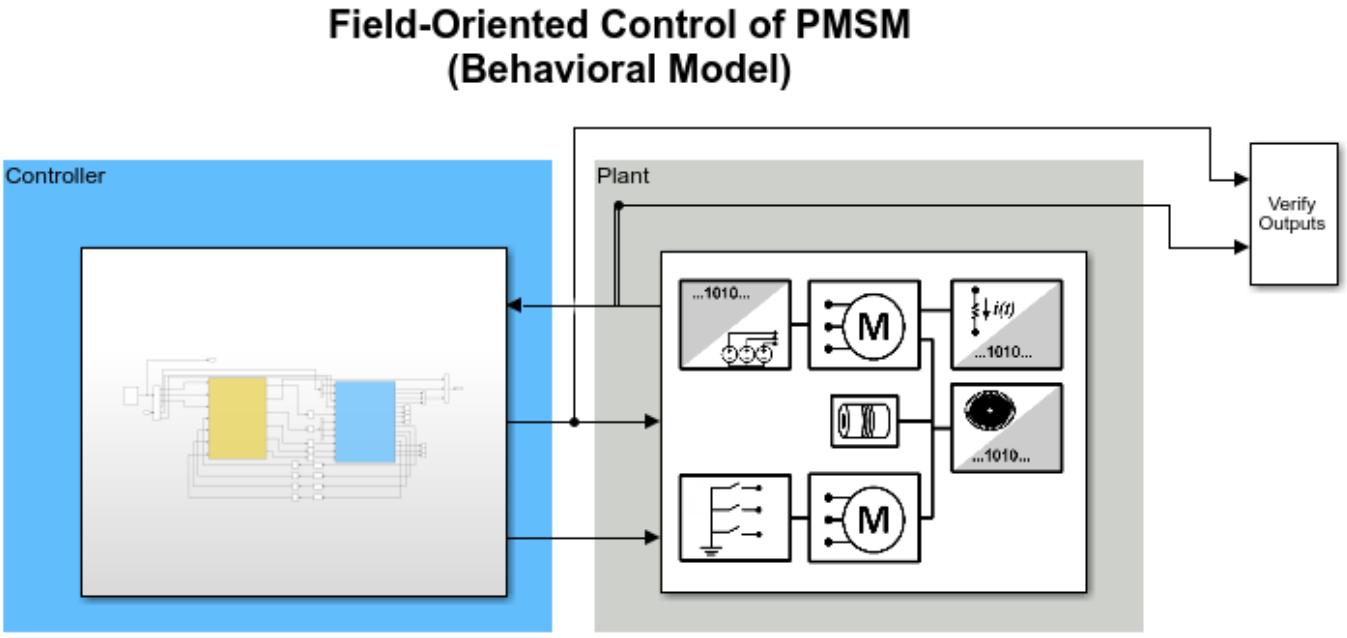


In an FOC running in closed-loop, the current control loop needs to run at a high rate, typically microseconds. In contrast, the velocity control can run at lower rates, typically milliseconds, but must react to external events, such as commanded velocity updates. By partitioning the current and velocity controllers onto the FPGA and processor cores, respectively, both control loops in the FOC can meet the above requirements.

The first model in this example is used for behavioral simulation of a closed-loop FOC with an open-loop calibration controller for a PMSM. The second model shows how the open-loop calibration controller, closed-loop velocity controller, and closed-loop current controller can be partitioned into an SoC device using SoC Blockset. A comparison of the simulation results between the behavioral and SoC models shows the expected behavior of the controller is maintained.

Behavioral Model

The top-level structure of the behavioral model is shown below. The Plant subsystem models a PMSM with load with simulated measurements from a motor shaft encoder and current sensors. The model parameters of the motor, load, and sensors are based on the AD-FMCMOTCON2-EBZ Evaluation Board from Analog Devices®. The Controller subsystem contains the closed-loop FOC and the open-loop calibration controllers.



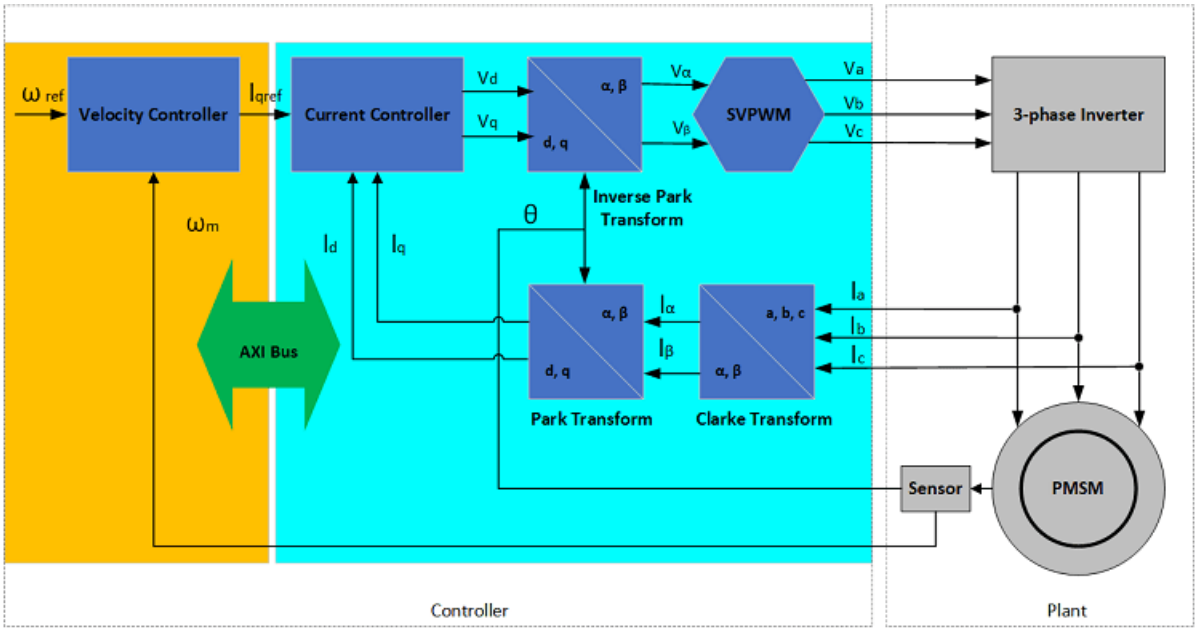
The Controller is split into two subsystems, an inner Current Control loop and outer Velocity and Calibration Control loop.

The Current Control subsystem takes a command current value from the Calibration and Velocity Control subsystem. The current controller uses consecutive Clarke and Park transforms to convert the AC current and voltage waveform into DC signals. A Proportional-Integral (PI) controller uses the DC signals to drive PWM switching signals to the power MOSFETs driving the PMSM.

The Velocity Control subsystem takes external commands to set the mode of the controller as either calibrating or closed-loop velocity tracking. In the calibration mode, the Mode_Scheduler spins the motor using an open-loop velocity controller to identify the zero index of the shaft encoder. Then the controller commands and holds a zero position to identify the encoder offset. After determining the encoder offset, the velocity controller is calibrated and can be switched into closed-loop velocity control. The closed-loop velocity control also uses a PI controller, similar to the current controller.

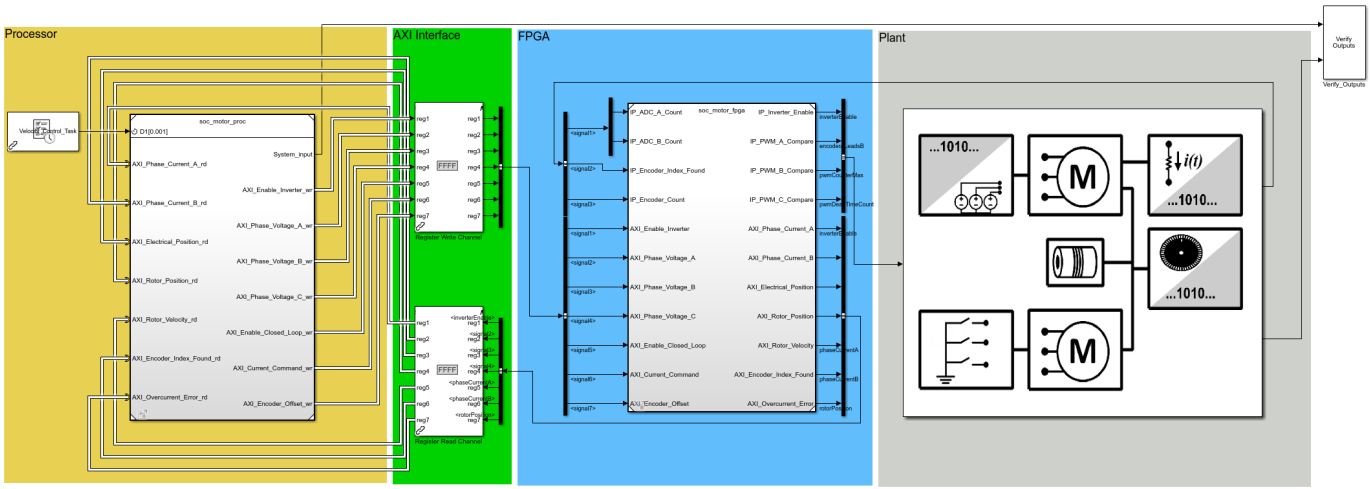
Hardware-Software Partitioned SoC Model

The structure of the partitioned SoC model is based on the partitioning scheme shown below. The fast current controller is running on the FPGA and the slow velocity controller on the processor. The FPGA and processor communicate via AXI interface.



The original Controller subsystem from the behavioral model has been partitioned into the processor and FPGA models, which are connected with Register Channel blocks.

Field-Oriented Control of PMSM (SoC Model)



- Processor

The open-loop calibration and the closed-loop velocity controllers are now inside a Model block and operate as a task driven by the Task Manager block. As part of the task iteration, the controller first reads from the AXI registers using Register Read blocks, iterates the control algorithm, and then writes the updated outputs to the AXI register using the Register Write blocks. The Task Manager executes the controller task at a rate of 1kHz with an average execution duration of 0.2ms.

- AXI Interface

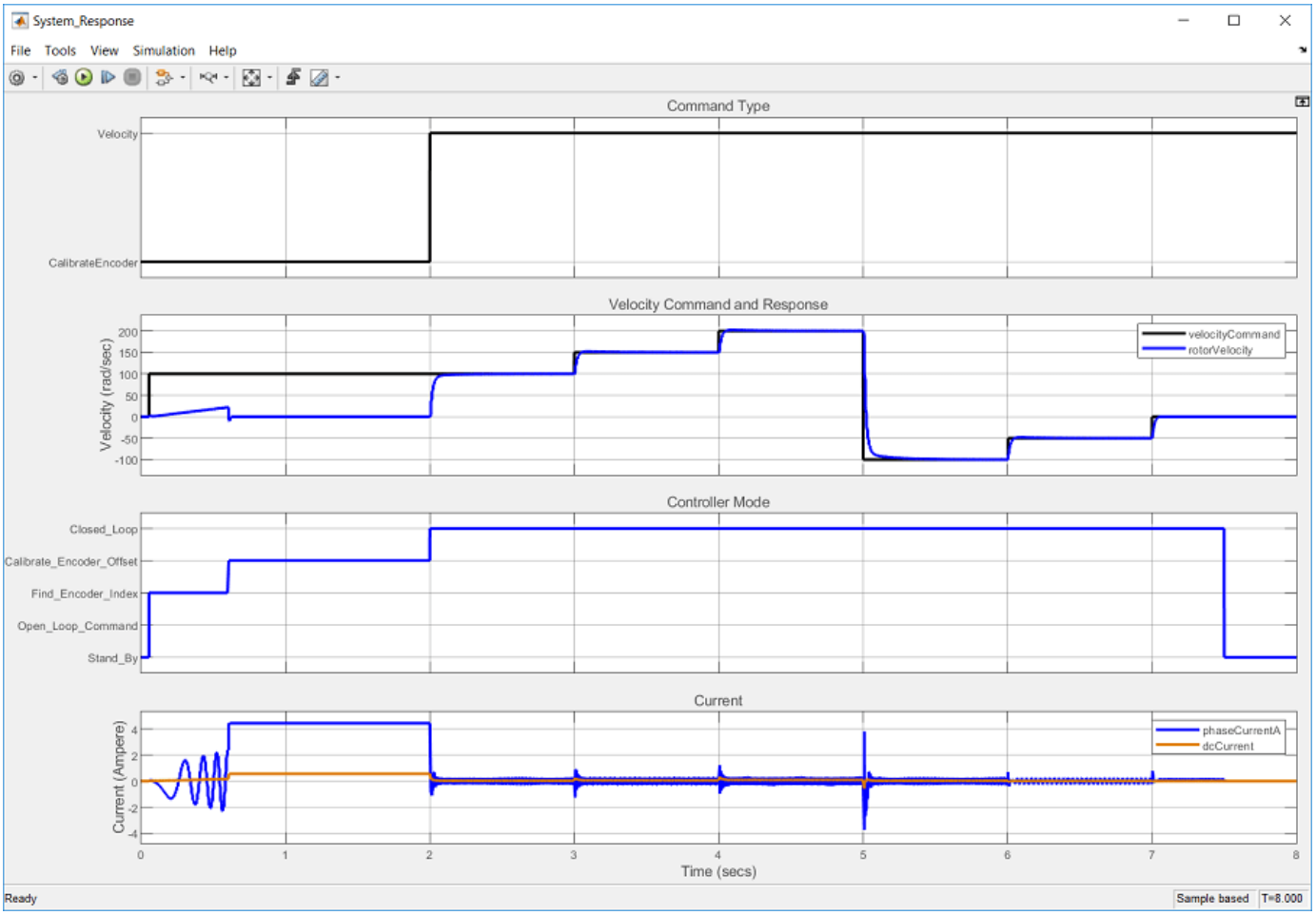
Register Channel block models the AXI communication between FPGA and Processor for register read and write operations. The corresponding AXI4-Lite driver blocks, Register Read/Register Write, are used in Processor Model to represent AXI4-Lite interface.

- FPGA

The closed-loop current controller is contained in the Model block representing the FPGA of the SoC device. Since the current controller exists in the FPGA, it can write and read directly from the AXI hardware registers. The FPGA uses a 40us clock.

Comparison of Behavioral and SoC Model Simulations

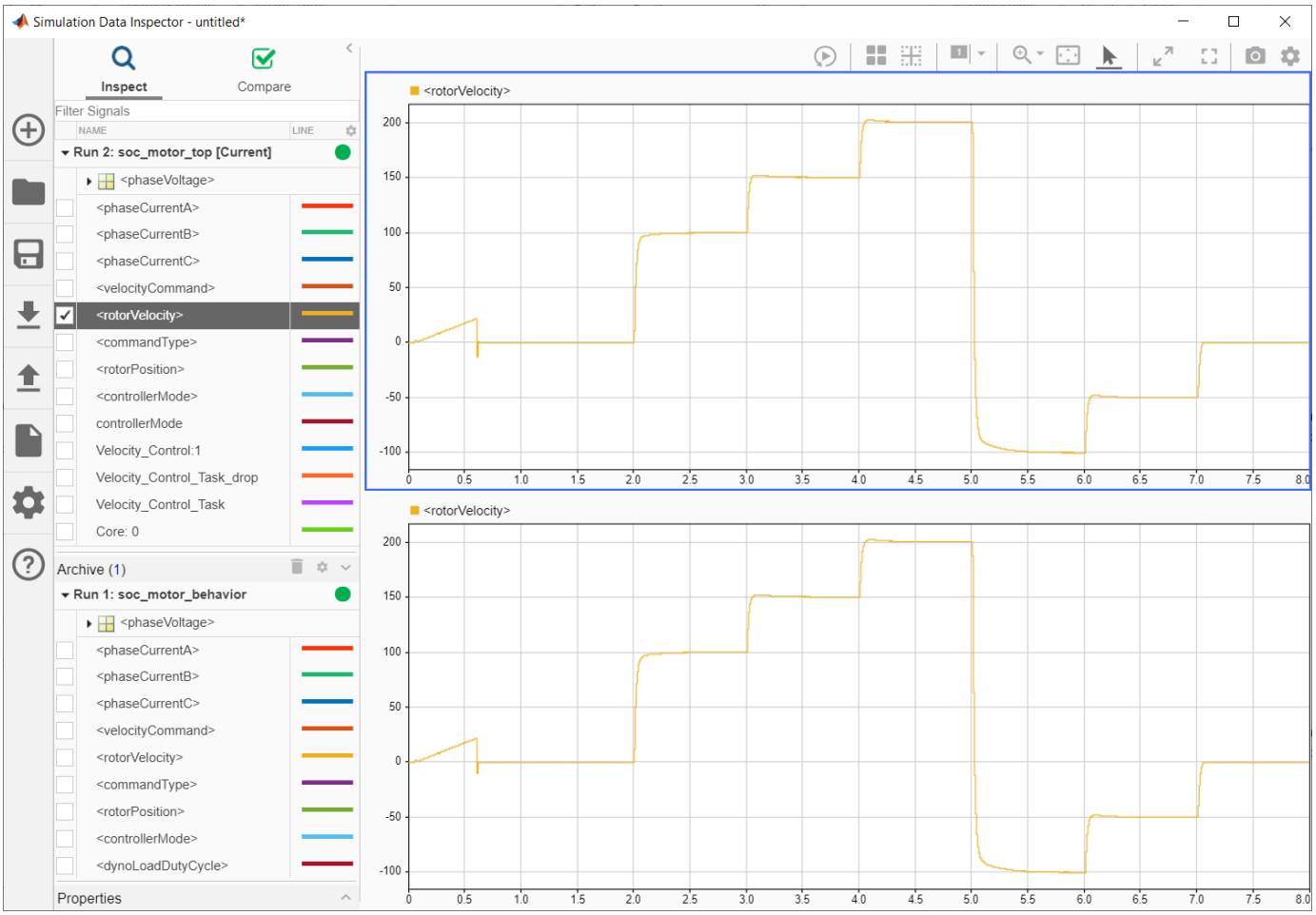
1. Open and run the behavioral model. Observe the controller and motor behavior from the System_Response scope.



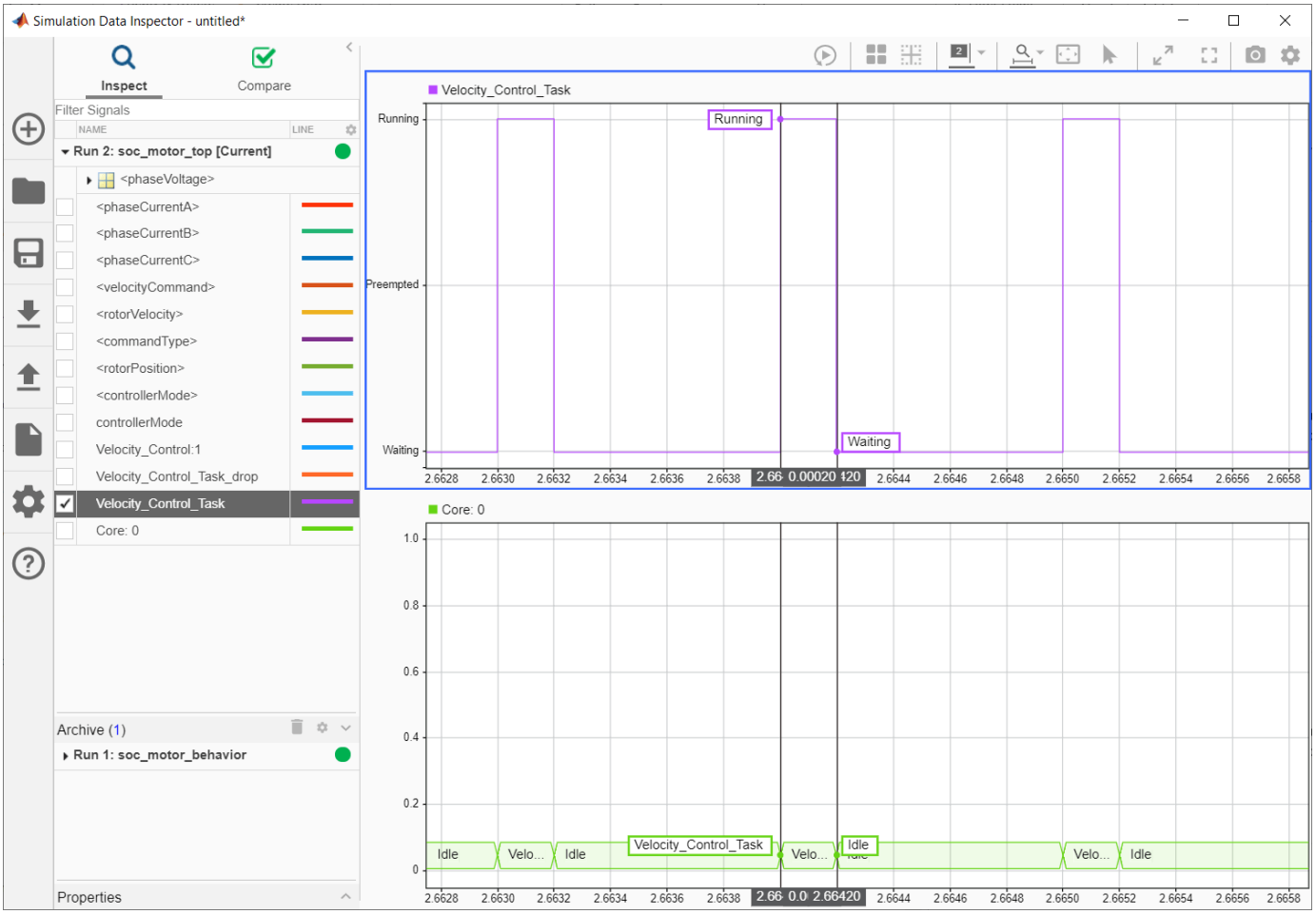
2. Open and run the partitioned SoC model. Observe that the controller and motor behavior matches.

3. Click **Data Inspector** to open the Simulation Data Inspector (SDI). Signal data for the previous model runs was automatically captured and archived in the SDI.

4. Select the **rotorVelocity** from **Run 1: soc_motor_behavior** and the **rotorVelocity** from **Run 2: soc_motor_top** into each subplot to get the following plot. Both the behavioral and partitioned models demonstrate equivalent motor velocity tracking.



5. From **Run 2: soc_motor_top**, select and display the **Velocity_Control_Task** and **Core: 0** signals into each subplot to get the following plot. From the plot, you can observe the task execution time of the velocity controller and the CPU utilization.



Other Things to Try

You can use this model as a template to develop an SoC model specific for your motor control hardware, e.g. FMC motor driver board - Analog Devices AD-FMCMOTCON2-EBZ. It will generate SoC design (FPGA bitstream and executable software) using HDL coder and Embedded coder. Refer to custom board support documentation for supporting customized SoC board and I/O devices.

Export Custom Reference Design

This example shows how to export a custom reference design from an SoC model by using the Soc Blockset™ `socExportReferenceDesign` function. After creating the custom reference design, use the **HDL Workflow Advisor** tool from HDL Coder™ to integrate an IP core into the reference design.

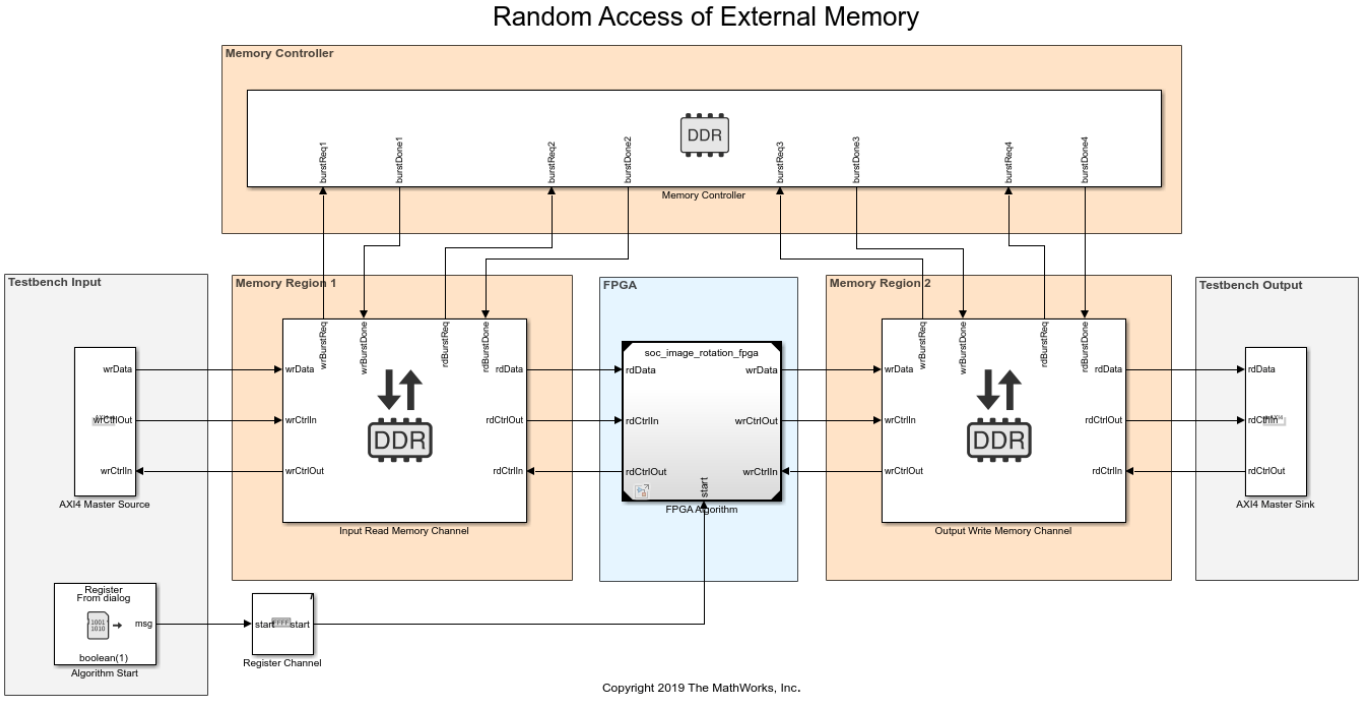
Design Task

This example uses the model `soc_image_rotation` to generate a custom reference design. The model has an external memory and an FPGA DUT. The DUT contains an AXI4 master read interface and an AXI4 master write interface to perform read and write operations to memory. For a full description of the model, see “Random Access of External Memory” on page 7-2. The model also uses an `socAXIMaster` to read and write the external memory from the host computer.

When exporting a custom reference design from this model, the DUT is not included in the reference design, and the interface to the DUT is exposed. After generating the reference design you can integrate your custom IP by using the **HDL Workflow Advisor** tool. Your custom IP must have the same interface as the FPGA Algorithm block.

Open the model to view the structure of the top model and the interface to the FPGA Algorithm block.

```
open_system('soc_image_rotation');
```



Prepare SoC Model for Custom Reference Design Export

In Simulink®, open the Configuration Parameters dialog box by clicking **Model Settings** on the **Modeling** tab. Then, follow these steps to prepare the SoC model for custom reference design export.

1 On the left pane, select **Hardware Implementation**.

2. Set **Hardware board** to match your board (if you are not using `Xilinx Zynq ZC706 evaluation kit`).
3. Under **Feature set for selected hardware board**, select **SoC Blockset**.
4. Expand **Target hardware resources**, select **FPGA design (top-level)**, and then select **Include 'MATLAB AXI Master' IP for host-based interaction**.
5. Because this SoC model does not include a processor, clear **Include processing system**. If your SoC model includes a processor subsystem, then select this option.
6. In the **IP core clock frequency (MHz)** box, specify the IP core clock frequency in MHz.
7. Select **FPGA design (mem channels)**, and set **Interconnect data width (bits)** to 32.

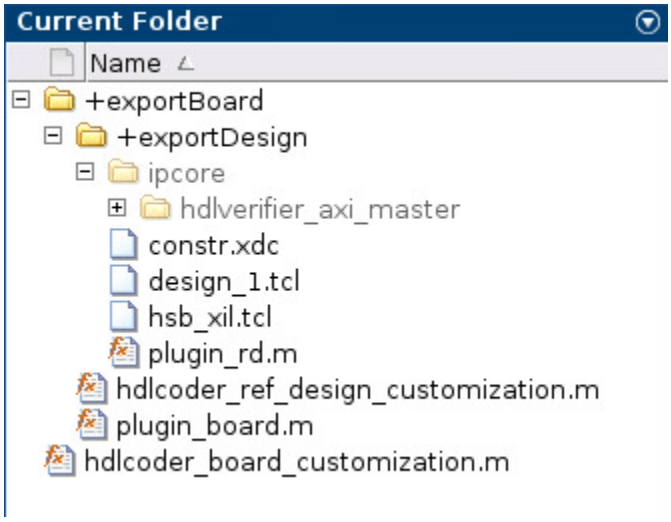
Export Custom Reference Design

Export the custom reference design for model `soc_image_rotation` by using the `socExportReferenceDesign` function. Enter this code at the MATLAB command prompt:

```
socExportReferenceDesign('soc_image_rotation')
```

The function generates these artifacts in the current folder.

- Board registration files
- Reference design registration file
- IP repository
- Design files
- Constraint files



Add Generated Design Folder to Path

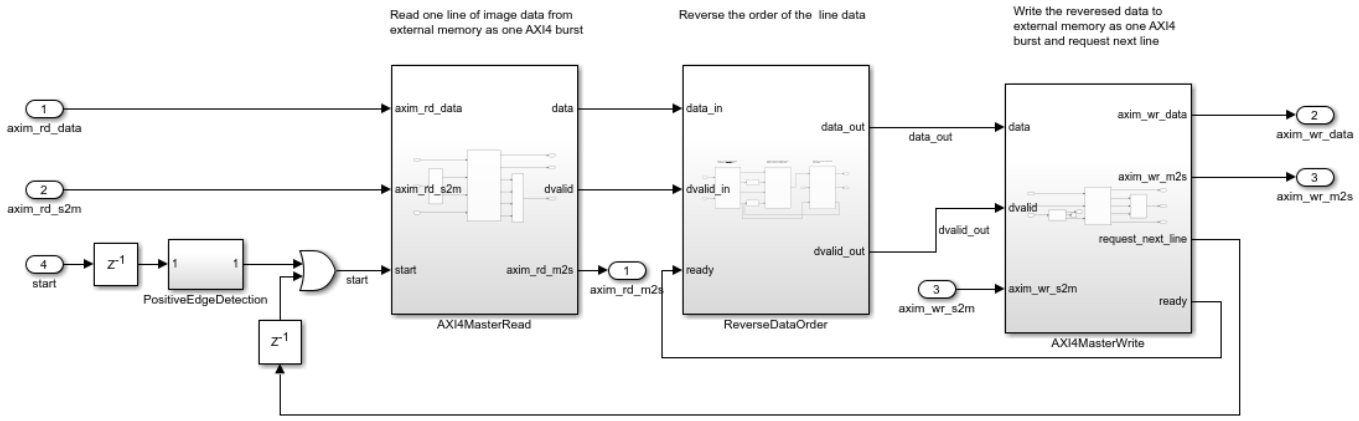
To add the generated design folder to the MATLAB path, right click on the folder named *`top-model`*`-refdesign`, where *top-model* is the name of the top SoC model. Then select **Add to Path>Selected Folders and Subfolders**.

Integrate IP Core into Custom Reference Design

After generating a reference design, you can save it or pass it to the IP developer for integration and deployment of their IP on a board.

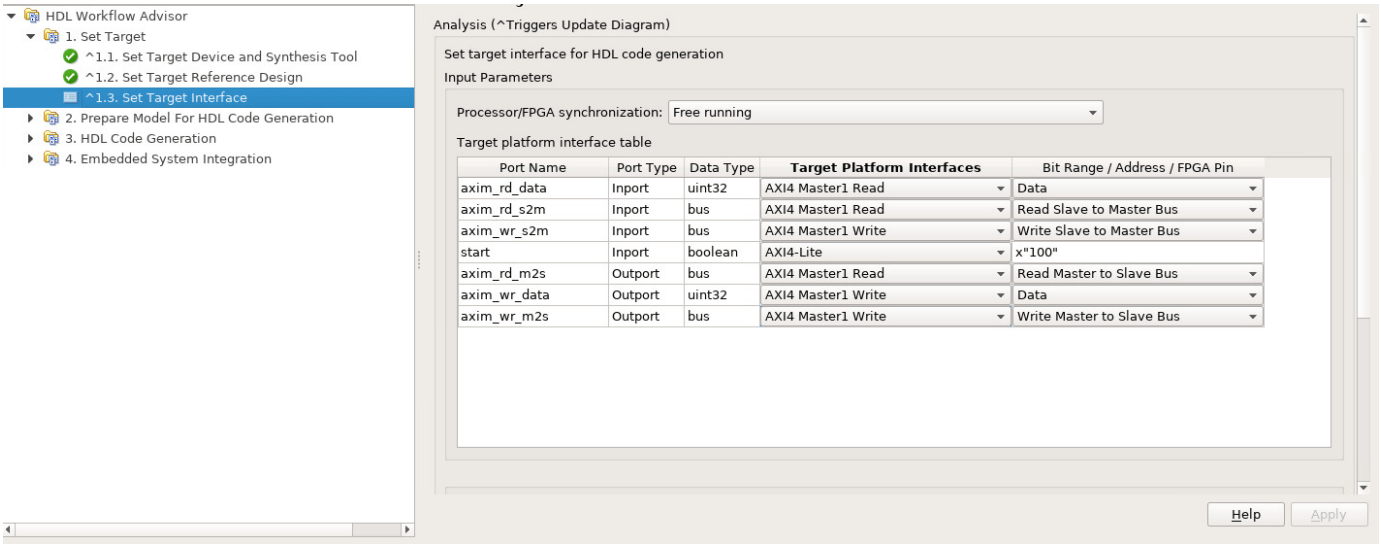
This example uses the image rotation DUT as the IP. This reference design is suitable for any IP that has the same interface.

```
open_system('soc_image_rotation_fpga');
```



In Simulink, right-click the ImageRotation block and select **HDL Code>HDL Workflow Advisor** to open the **HDL Workflow Advisor** tool.

1. In step 1.1, set **Target workflow** to `IP Core Generation` and **Target platform** to the platform generated by the `socExportReferenceDesign` function. For this example, select `Xilinx Zynq ZC706 evaluation kit (generated by SoC Blockset)`.
2. Click **Run This Task**.
3. Select step 1.2. Note that **Reference design** is set to `Design exported from 'soc_image_rotation' model`.
4. In step 1.3, set the target interface by connecting each port in your IP to the corresponding port in the reference design.



5. Continue with the remaining steps of the **HDL Workflow Advisor** tool.

6. In step 4.2, under **Generate a software interface model with IP core driver blocks for C code generation**, select **Skip this task**. For this example, select this value because the generated reference design includes only FPGA and memory components. If the reference design also includes a processing system, clear this option.

7. In step 4.4, set **Programming method** to **JTAG**.

8. Connect the host machine to a ZC706 board, and follow the workflow to load your full design (IP and custom reference design) to the FPGA.

9. Use MATLAB AXI Master to interact with the FPGA from the host machine.

Conclusion

This example covered these workflows.

- Generating a reference design from an SoC model
- Integrating an IP core into the generated reference design using the **HDL Workflow Advisor** tool

See Also

`socExportReferenceDesign` | **SoC Builder**

Related Examples

- “Export Custom Reference Design from SoC Model” on page 4-5

Estimate Number of Operators for MATLAB Algorithm

This example shows how to estimate the number of arithmetic operators in an algorithm written in MATLAB. Analyze a radix 2 FFT algorithm and generate reports showing operator usage.

Radix 2 FFT Algorithm and Testbench

Analyze the number of arithmetic operators in the `soc_analyze_FFT_radix2` function. Calculate the number of arithmetic operators used during the execution of the function. The testbench `soc_analyze_fft_tb` provides stimulus and verifies the implementation of the radix 2 FFT algorithm (`soc_analyze_FFT_radix2`) against the MATLAB FFT function (`fft`) .

Open the `soc_analyze_FFT_tb.m` file in the MATLAB editor to examine the structure of the testbench.

```
open soc_analyze_FFT_tb
```

The testbench generates a test signal with two sinusoids, one at 50 Hz with an amplitude of 0.7 and another at 120 Hz with an amplitude of 1. The signal has a sampling frequency of 1 kHz with additive random noise. The testbench calculates the FFT output for the above test signal for the FFT-length specified as the `FFTLen` argument. The testbench compares the output of the function to the output of the MATLAB FFT function (`fft`) and plots the results.

Generate Operator-Count Reports for 1024 Points Radix 2 FFT

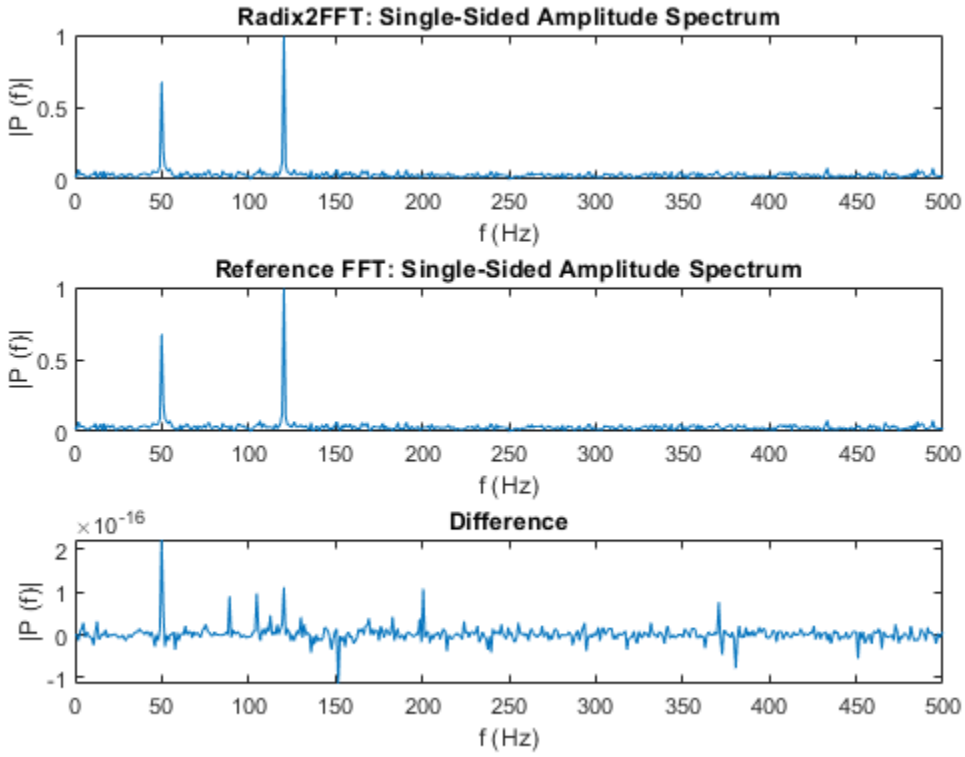
To estimate the number of operators for the radix 2 FFT, use the `socFunctionAnalyzer` function, and provide the testbench function `soc_analyze_FFT_tb` as an argument. By default, the function generates reports for all the functions called from within the testbench function and lists all the operators used.

To generate a report for only the algorithm (`soc_analyze_FFT_radix2`), and not for the testbench, use the `'RestrictFunction'` name-value pair argument with the value `'soc_analyze_FFT_radix2.m'`. Use the `'RestrictOperator'` name-value pair argument to filter the report and show only three operators by setting its value to `{'ADD','MINUS','MUL'}`. Set the `'OutputFolder'` name-value pair argument to specify a folder location for generated reports.

Execute this command to generate reports for a simulation of a 1024-points radix 2 FFT algorithm. The command simulates the design while counting operators and generating a report.

```
socFunctionAnalyzer('soc_analyze_FFT_tb.m','FunctionInputs',1024, ...
    'Folder','report_1024','IncludeFunction','soc_analyze_FFT_radix2.m', ...
    'IncludeOperator',{'ADD','MINUS','MUL'});
```

```
Generating operators analysis report for C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex65
Saving report files in C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex65369380\report_1024
Operator estimate: <a href="matlab: socAlgorithmAnalyzerReport('C:\TEMP\Bdoc21b_1757077_3096\ib2
Done.
```

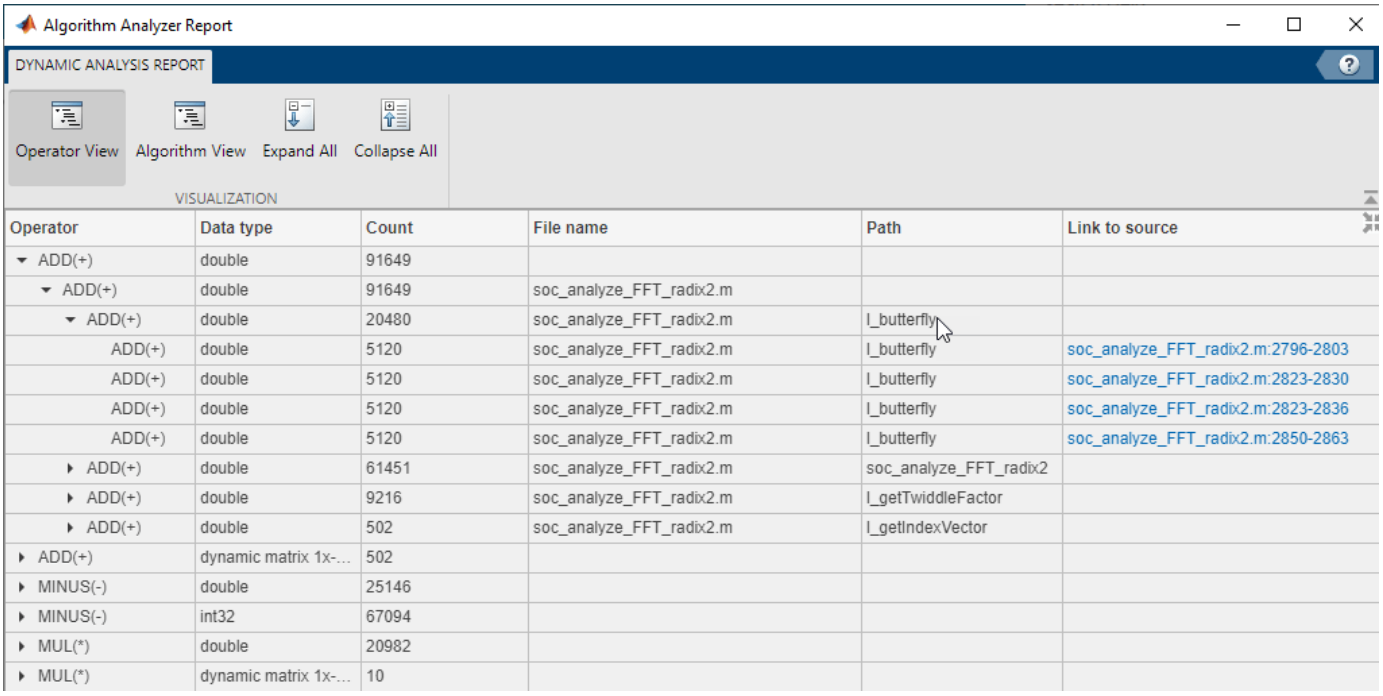


Observe the radix 2 FFT algorithm and the reference MATLAB FFT function (`fft`) results in the simulation plot above. Verify that those results are very similar and the difference between those results is in order of 10e-12.

Analyze Operator Estimation Report

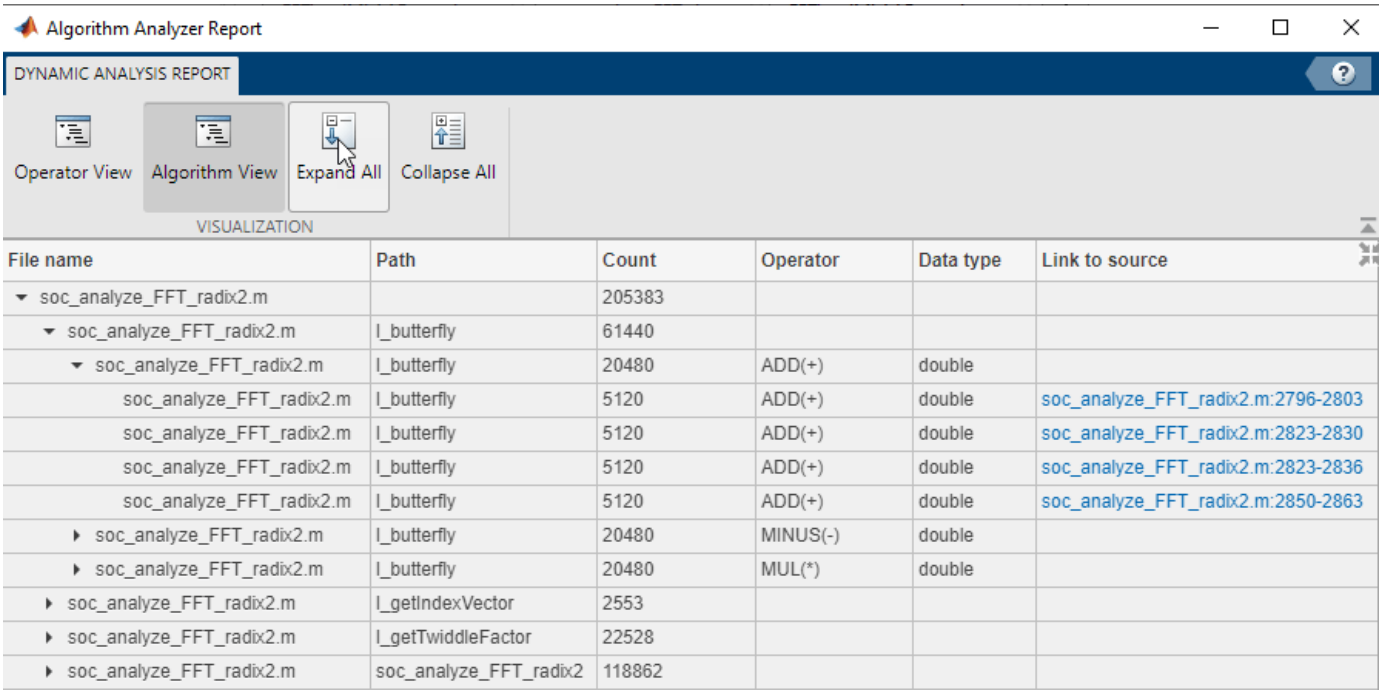
Open the report by clicking the **Open report viewer** link on the MATLAB console. Alternatively, you can use the `socAlgorithmAnalyzerReport` function. The report provides two views. The first view is the operator view, which presents the data such that each row corresponds to an operator. To use this view, click **Operator View** on the report toolstrip. The second view is the algorithm view, where each row corresponds to a MATLAB function. To use this view, click **Algorithm View** on the report toolstrip.

By default, the report opens with the operator view. The report opens the aggregate view of each operator and data type. For example, for a 1024 points radix 2 FFT there are a total of 91,649 additions **[ADD(+)]** of data type `double` and 67,094 subtractions **[MINUS(-)]** of data type `int32`. To get the detailed report for each operator, expand that operator. The report shows the operator count as used in various functions. For example, the butterfly function `l_butterfly` contains four `double` additions that executed 5,120 times each. Trace the operator by clicking on one of the links in the last column of the report to highlight the location of the operator in the `soc_analyze_FFT_radix2` file.



Operator	Data type	Count	File name	Path	Link to source
ADD(+)	double	91649			
ADD(+)	double	91649	soc_analyze_FFT_radix2.m		
ADD(+)	double	20480	soc_analyze_FFT_radix2.m	l_butterfly	
ADD(+)	double	5120	soc_analyze_FFT_radix2.m	l_butterfly	soc_analyze_FFT_radix2.m:2796-2803
ADD(+)	double	5120	soc_analyze_FFT_radix2.m	l_butterfly	soc_analyze_FFT_radix2.m:2823-2830
ADD(+)	double	5120	soc_analyze_FFT_radix2.m	l_butterfly	soc_analyze_FFT_radix2.m:2823-2836
ADD(+)	double	5120	soc_analyze_FFT_radix2.m	l_butterfly	soc_analyze_FFT_radix2.m:2850-2863
ADD(+)	double	61451	soc_analyze_FFT_radix2.m	soc_analyze_FFT_radix2	
ADD(+)	double	9216	soc_analyze_FFT_radix2.m	l_getTwiddleFactor	
ADD(+)	double	502	soc_analyze_FFT_radix2.m	l_getIndexVector	
ADD(+)	dynamic matrix 1x-...	502			
MINUS(-)	double	25146			
MINUS(-)	int32	67094			
MUL(*)	double	20982			
MUL(*)	dynamic matrix 1x-...	10			

Switch to the algorithm view, by clicking the **Algorithm View** button. Expand the report and view the operator counts for all the functions under the file `soc_analyze_FFT_radix2.m`. You can view the counts of each operator with their data types by expanding another level. You can also use the **Expand All** and **Collapse All** buttons on the report toolstrip to navigate the report. To trace a specific operator to the MATLAB code, click the corresponding link in the column **Link to source** in the report.



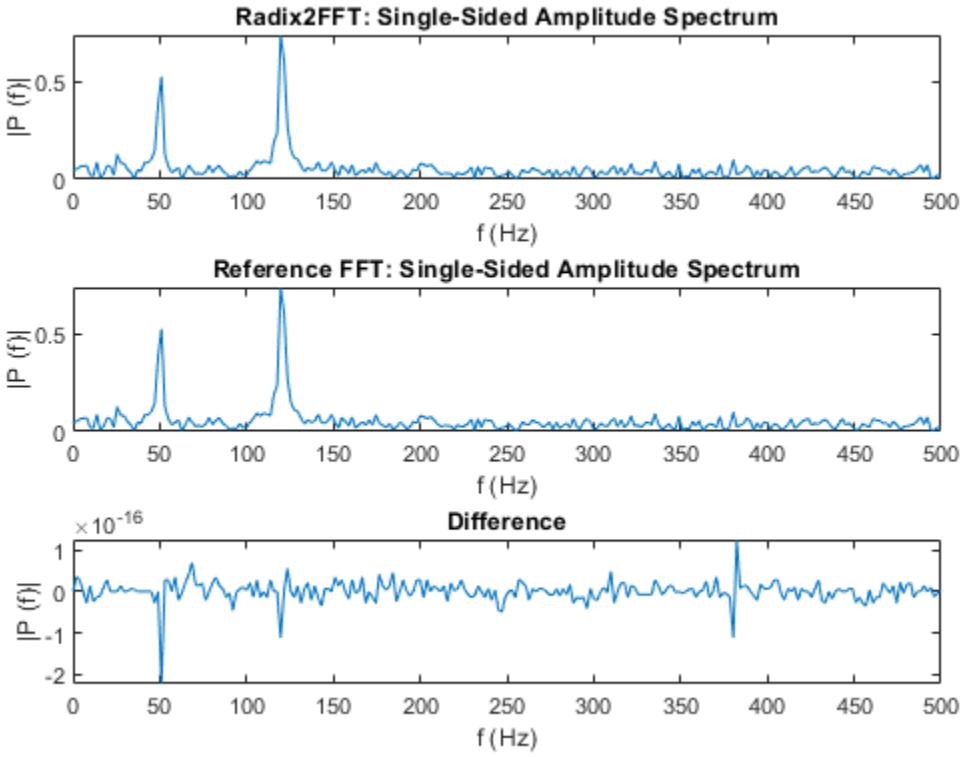
File name	Path	Count	Operator	Data type	Link to source
soc_analyze_FFT_radix2.m		205383			
soc_analyze_FFT_radix2.m	l_butterfly	61440			
soc_analyze_FFT_radix2.m	l_butterfly	20480	ADD(+)	double	
soc_analyze_FFT_radix2.m	l_butterfly	5120	ADD(+)	double	soc_analyze_FFT_radix2.m:2796-2803
soc_analyze_FFT_radix2.m	l_butterfly	5120	ADD(+)	double	soc_analyze_FFT_radix2.m:2823-2830
soc_analyze_FFT_radix2.m	l_butterfly	5120	ADD(+)	double	soc_analyze_FFT_radix2.m:2823-2836
soc_analyze_FFT_radix2.m	l_butterfly	5120	ADD(+)	double	soc_analyze_FFT_radix2.m:2850-2863
soc_analyze_FFT_radix2.m	l_butterfly	20480	MINUS(-)	double	
soc_analyze_FFT_radix2.m	l_butterfly	20480	MUL(*)	double	
soc_analyze_FFT_radix2.m	l_getIndexVector	2553			
soc_analyze_FFT_radix2.m	l_getTwiddleFactor	22528			
soc_analyze_FFT_radix2.m	soc_analyze_FFT_radix2	118862			

Generate Reports for 512 Points Radix 2 FFT

To observe the correlation between the number of operations and the number of points in the FFT, compare the previous report with the one for a 512 points radix 2 FFT. Generate reports for a 512 points radix 2 FFT by passing a value of 512 to the `'FunctionInputs'` name-value pair argument as in this command.

```
socFunctionAnalyzer('soc_analyze_FFT_tb.m','FunctionInputs',512, ...
   'Folder','report_512','IncludeFunction','soc_analyze_FFT_radix2.m', ...
   'IncludeOperator',{'ADD','MINUS','MUL'});
```

```
Generating operators analysis report for C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex65
Saving report files in C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex65369380\report_512.
Operator estimate: <a href="matlab: socAlgorithmAnalyzerReport('C:\TEMP\Bdoc21b_1757077_3096\ib2
Done.
```



For the 512 points radix 2 FFT, the aggregated report shows an estimated number of 41,473 additions of data type `double`, 32,026 subtractions of type `int32` and 11,316 subtractions of data type `double`. Previously, with the 1024 points radix 2 FFT, these values were 91,649, 70,173 and 25,146 respectively. Expand the report to get the detailed operator utilization in the `l_butterfly` function. In this case, the function is executed 2304 times for the 512 length, versus 5120 times for the 1024 length).

Conclusion

Use the `socFunctionAnalyzer` function to estimate and analyze the number of arithmetic operators in the MATLAB function for radix 2 FFT. Use various viewer options to analyze the report.

- in aggregate total view
- detailed per operator view and per MATLAB function

Analyze the report by passing different arguments as inputs to your algorithm and observing the differences.

You can use this analysis to get an estimate of the cost of implementing an algorithm on a given hardware platform.

See Also

`socFunctionAnalyzer` | `socAlgorithmAnalyzerReport`

Compare FIR Filter Implementations Using socModelAnalyzer

This example shows how to analyze and compare different implementations of a Simulink® algorithm based on the number of arithmetic operations. Use the SoC Blockset `socModelAnalyzer` function to generate reports that show the number of operators for different implementations of a FIR Filter using static and runtime execution.

Design Task and Requirements

This design task evaluates two implementations of a FIR filter and compares the implementation costs. This example uses the number of operators as a way to measure the implementation cost.

To meet system requirements such as speed, latency, and hardware resources, consider and compare several implementations of the algorithm. The number of arithmetic operators used in an implementation can help you identify resource usage and allocation.

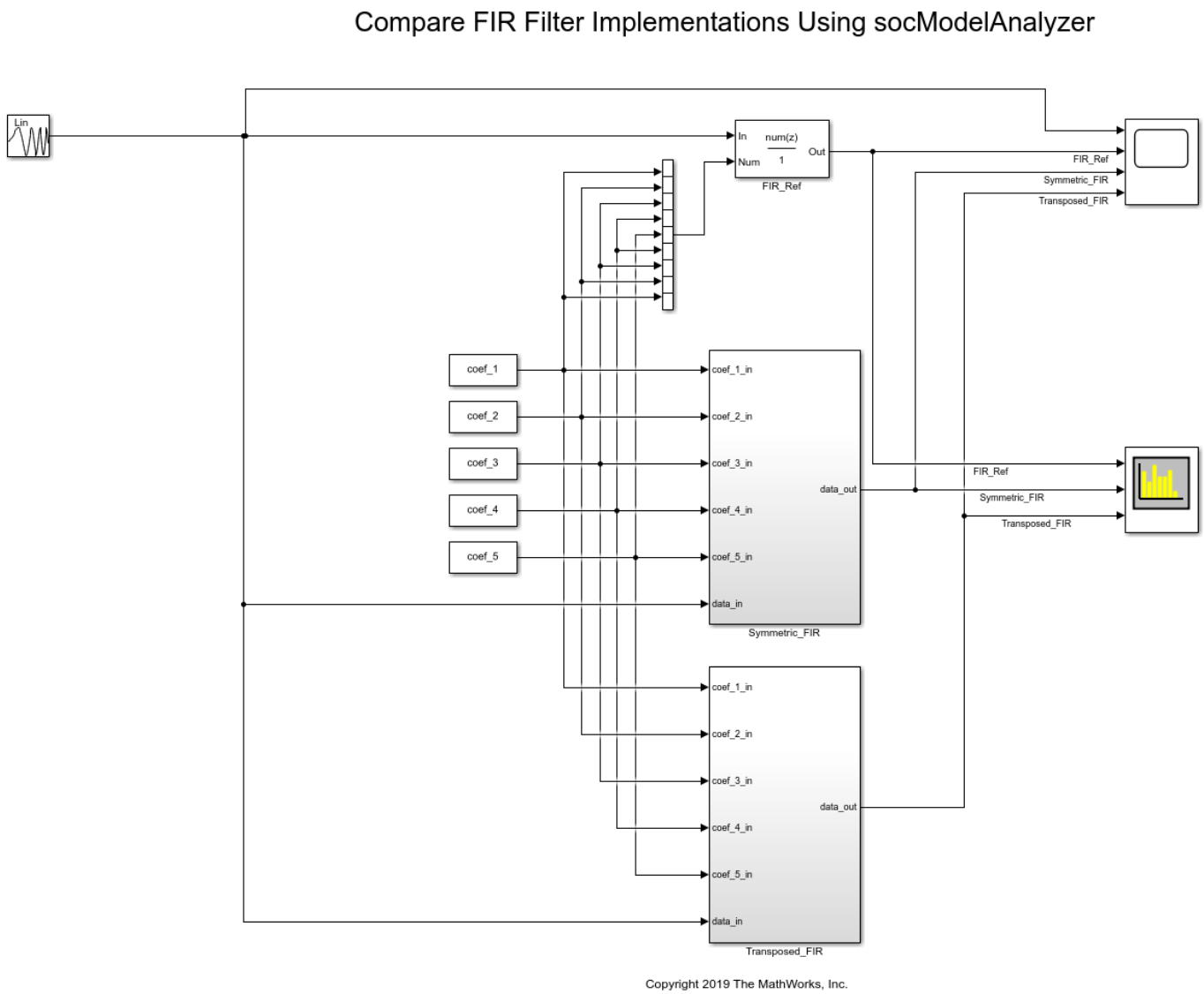
Manual analysis and calculation of the number of arithmetic operators can be tedious, error prone, and time consuming. Manual calculations can be inaccurate for an algorithm involving a branch, loop, or recursion construct and might be impossible to calculate if the execution path depends on the input data or random factors (for example, a convergence algorithm).

Structure of Model

The `soc_analyze_FIR_tb` model implements a low pass digital FIR filter in two ways. The Symmetric_FIR subsystem exploits symmetry in coefficients to optimize the resources. The Transposed_FIR subsystem employs a filter structure geared toward higher speed of operation. The model uses a chirp input signal as an input stimulus and a FIR_ref (Discrete FIR Filter) block as a reference for checking numerical correctness of the implementations.

Open the `soc_analyze_FIR_tb` model in Simulink and examine the structure of the model.

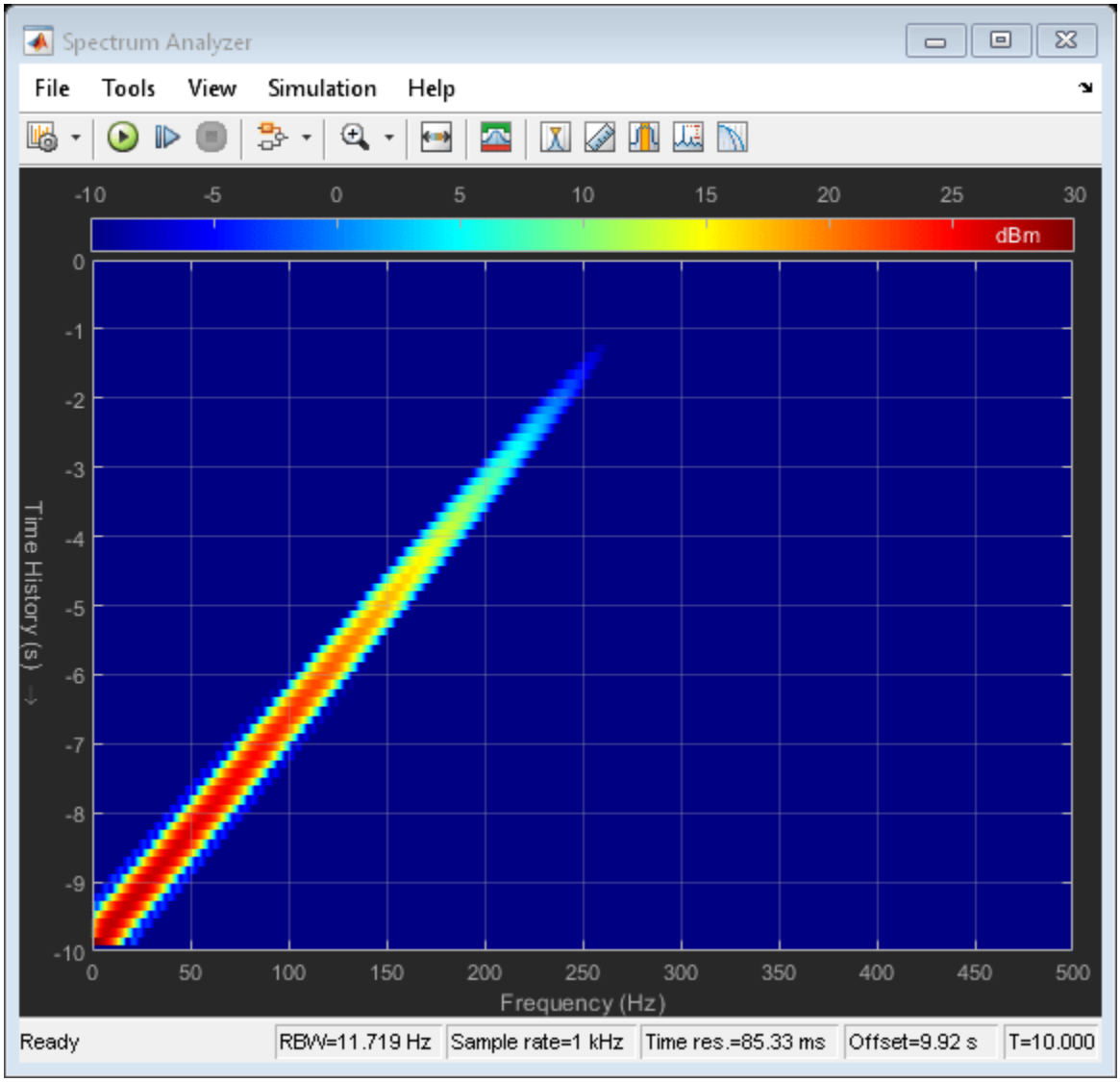
```
open_system('soc_analyze_FIR_tb');
```

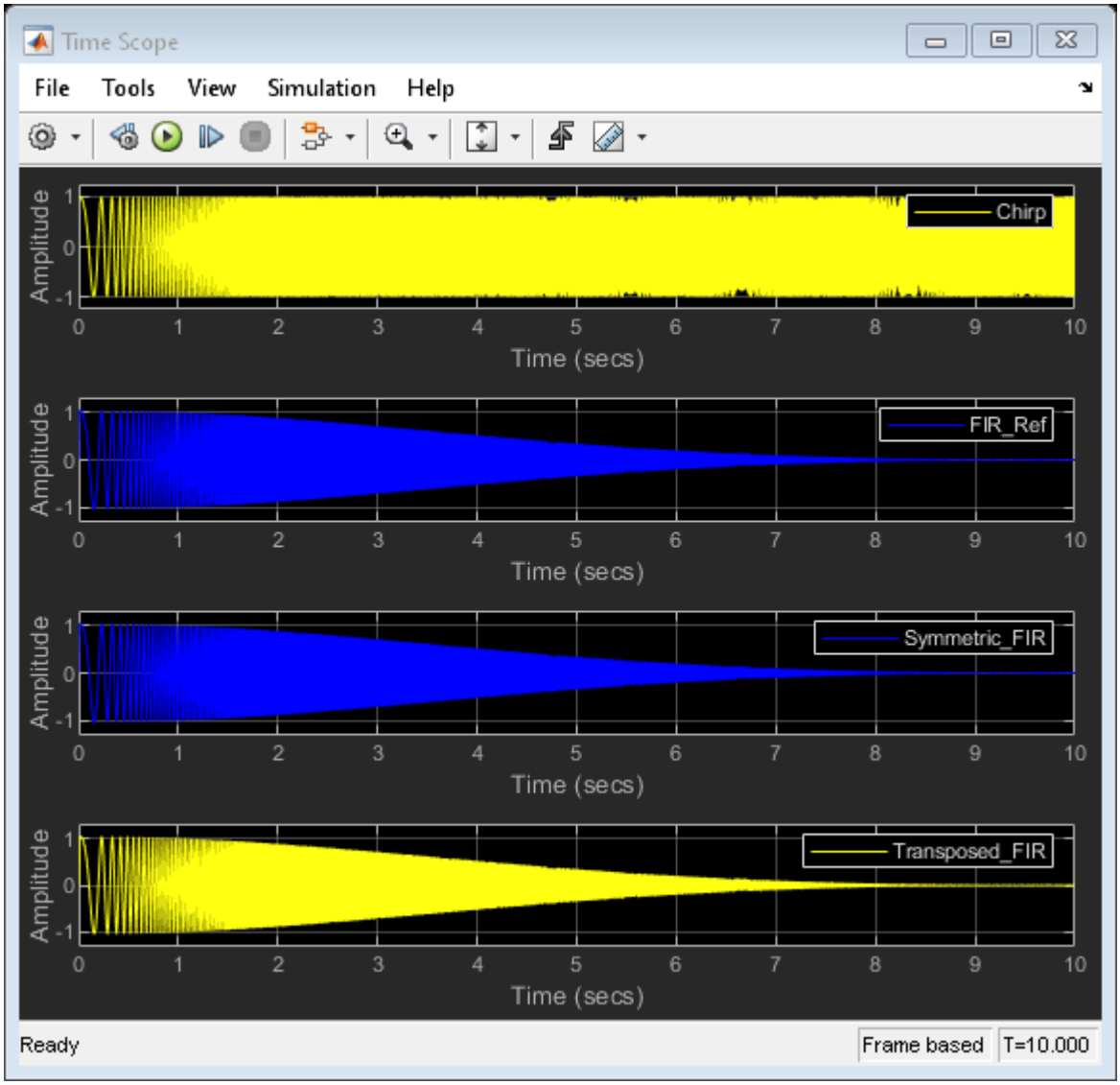


To design the low pass FIR filter we use the **`filterDesigner` (DSP System Toolbox)** app to generate coefficients for an 8th order FIR filter. The FIR filter has a cutoff frequency of 0.25 (normalized) and a passband ripple and stop band attenuation of 1 dB and 60 dB, respectively. The model sets these coefficients via the model initialize callback.

Simulate the model to validate the functionality of both implementations against the reference FIR block. The responses of the filter implementations match the reference.

```
sim('soc_analyze_FIR_tb');
```


Spectrum Analyzer
File
Tools
View
Simulation
Help
dBm
Time History (s)
Frequency (Hz)
Ready
RBW=11.719 Hz
Sample rate=1 kHz
Time res.=85.33 ms
Offset=9.92 s
T=10.000



Compare Implementations Using Model Analyzer

Use the `socModelAnalyzer` function to generate reports for the number of arithmetic operators in each implementation and compare the implementations. The reports are generated using the runtime execution of the model.

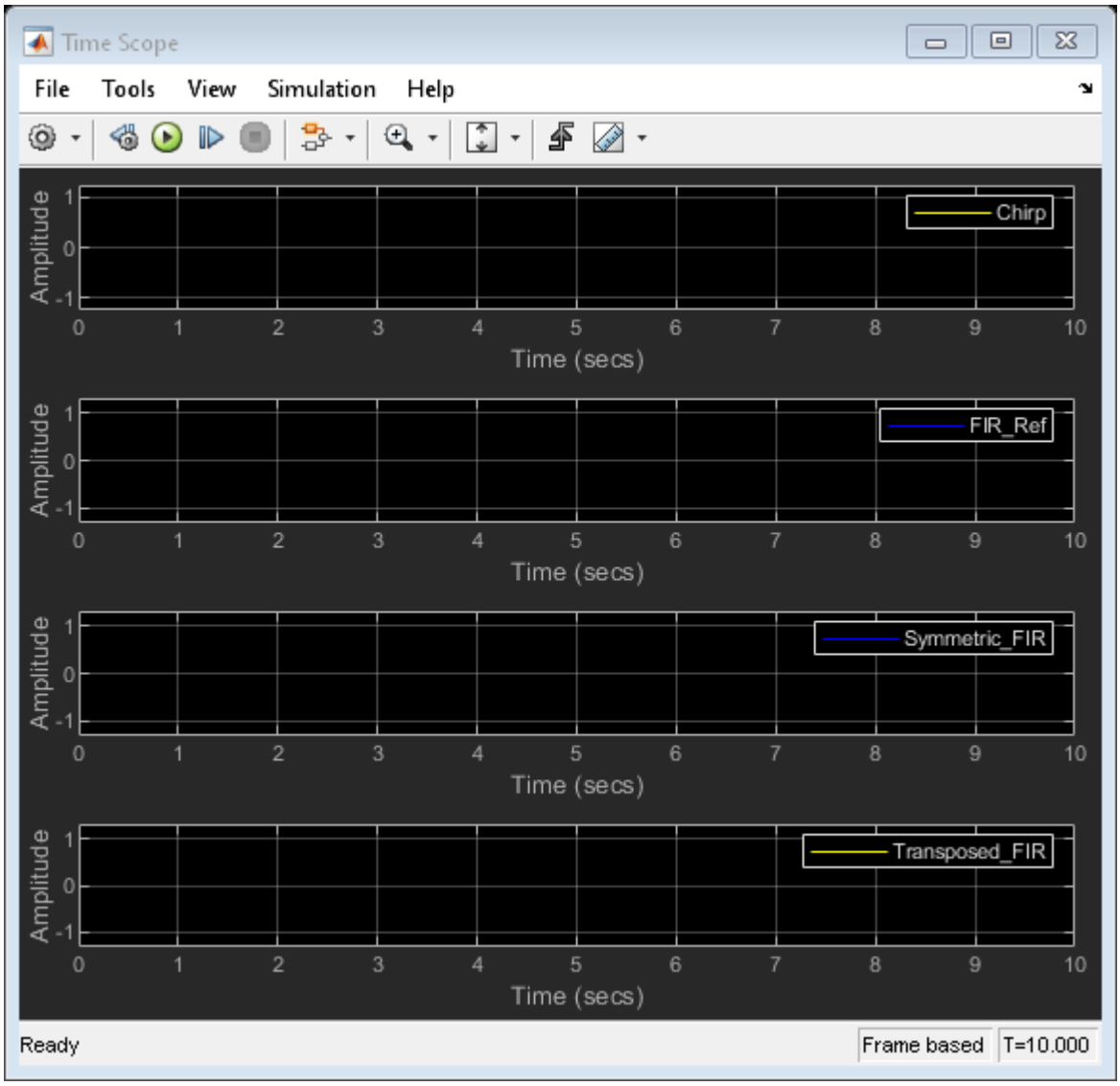
Symmetric FIR Filter:

To estimate the number of operators for Symmetric FIR filter implementation, use the `socModelAnalyzer` function. Specify the subsystem name for the `IncludeBlockPath` name-value pair argument of the function. Set the output folder to specify where to generate the reports. Enter this command at the MATLAB command prompt.

```
socModelAnalyzer('soc_analyze_FIR_tb.slx','Folder','report_sym','IncludeBlockPath',...
    'soc_analyze_FIR_tb/Symmetric_FIR');
```

```
Generating operators analysis report for C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex824
Saving report files in C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex82446029\report_sym.
```

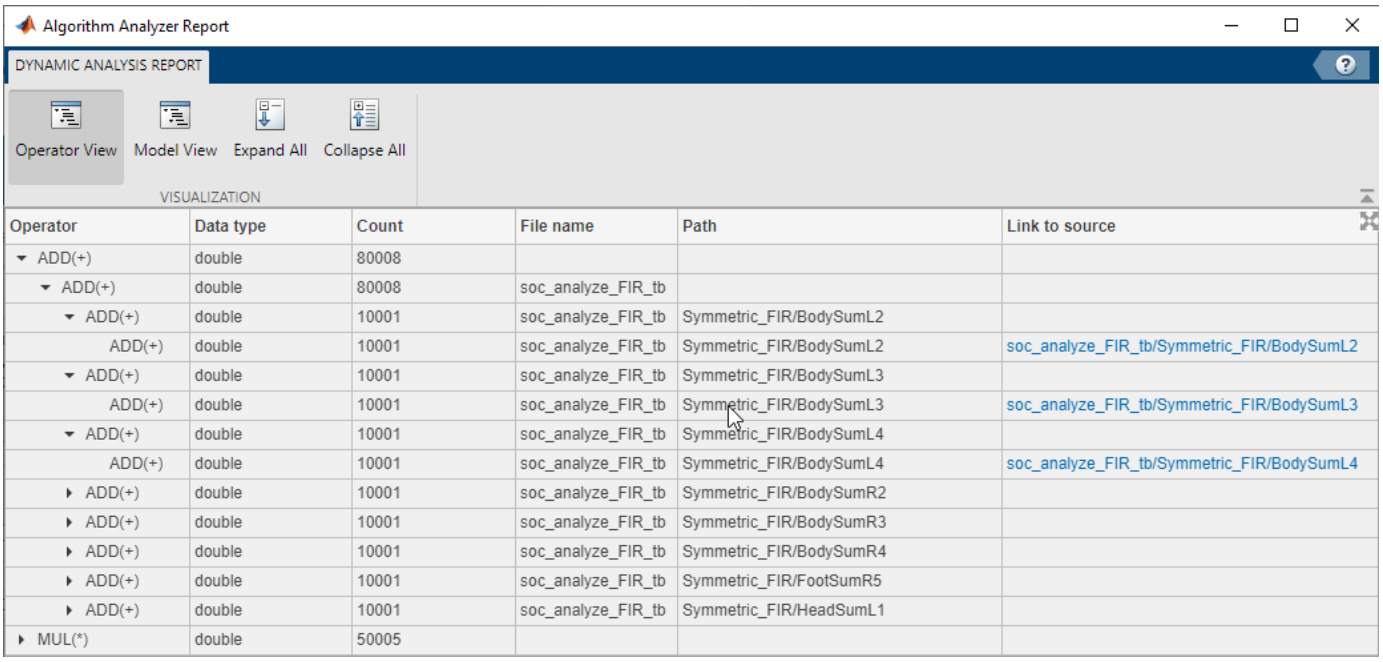
```
Operator estimate: <a href="matlab: socAlgorithmAnalyzerReport('C:\TEMP\Bdoc21b_1757077_3096\ib2
Done.
```



Open the report by clicking the **Open report viewer** link on the MATLAB console. Alternatively, you can use the `socAlgorithmAnalyzerReport` function. The report provides two views. The first view is the operator view, which presents the data such that each row corresponds to an operator. To use this view, click **Operator View** on the report toolstrip. The second view is the model view where each row corresponds to a Simulink subsystem path. To use this view, click **Model View** on the report toolstrip. Reports are also saved in the `report_sym` folder as a MAT-file (`soc_analyze_FIR_tb.mat`) and an Excel® file (`soc_analyze_FIR_tb.xlsx`).

By default, the report opens with the operator view. The viewer opens the aggregate view of each operator and data type. For example, the Symmetric FIR filter contains a total of 8 `ADD(+)` operators of data-type `double` and 5 `MUL(*)` operators of data-type `double` executed 10,001 times each. (The model simulation duration is 10 s and the base rate is 10 ms. This produces 10,000 simulation cycles plus 1 for initialization.) To get the detailed report for each operator, expand that operator. The report

shows the operator count as used in various blocks. Trace the operator by clicking on one of the links in the last column of the report to highlight the location of the operator in the `soc_analyze_FIR_tb` model.



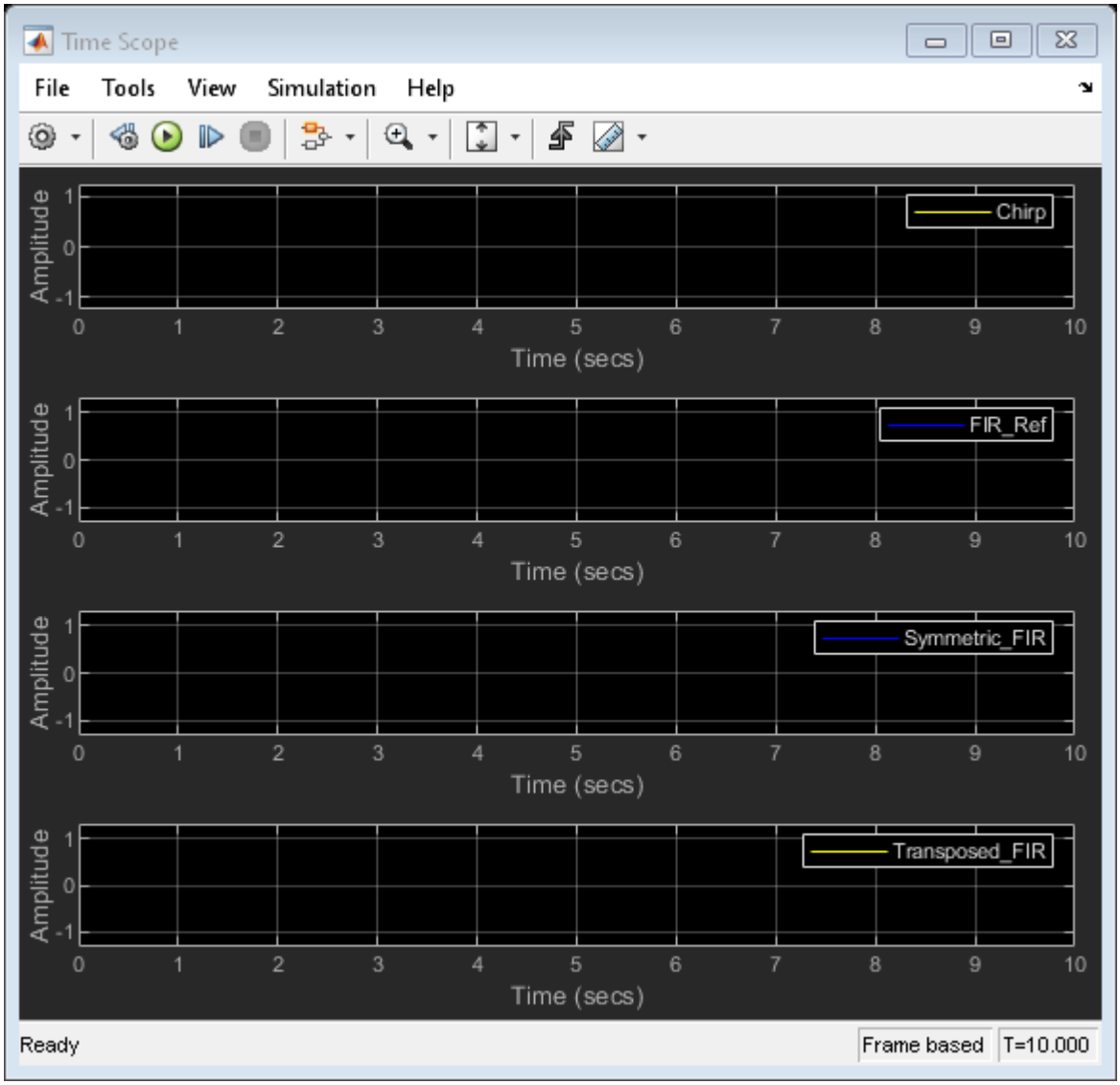
Operator	Data type	Count	File name	Path	Link to source
ADD(+)	double	80008			
ADD(+)	double	80008	soc_analyze_FIR_tb		
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL2	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL2	soc_analyze_FIR_tb/Symmetric_FIR/BodySumL2
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL3	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL3	soc_analyze_FIR_tb/Symmetric_FIR/BodySumL3
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL4	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumL4	soc_analyze_FIR_tb/Symmetric_FIR/BodySumL4
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumR2	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumR3	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/BodySumR4	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/FootSumR5	
ADD(+)	double	10001	soc_analyze_FIR_tb	Symmetric_FIR/HeadSumL1	
MUL(*)	double	50005			

Transposed FIR Filter

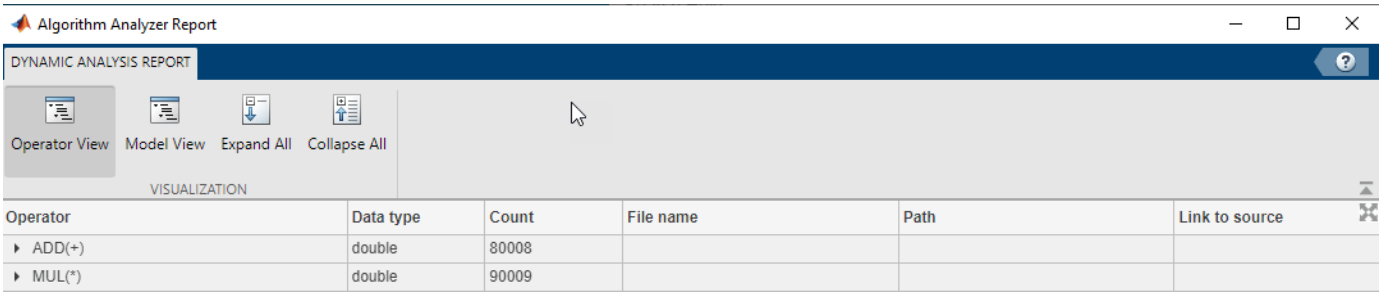
To estimate the number of operators for the Transposed FIR filter implementation, use the `socModelAnalyzer` function. Specify `'soc_analyze_FIR_tb/Transposed_FIR'` for the `'IncludeBlockPath'` name-value pair argument of the function. Set the output folder for the generated reports to `report_trans`. Enter this command at the MATLAB command prompt.

```
socModelAnalyzer('soc_analyze_FIR_tb.slx','Folder','report_trans','IncludeBlockPath',...
    'soc_analyze_FIR_tb/Transposed_FIR');
```

```
Generating operators analysis report for C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex824
Saving report files in C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\3\tp86af1a1d\ex82446029\report_trans
Operator estimate: <a href="matlab: socAlgorithmAnalyzerReport('C:\TEMP\Bdoc21b_1757077_3096\ib2
Done.
```



Open the report for the Transposed FIR filter by clicking the **Open report viewer** link on the MATLAB console.



For the Transposed FIR filter, the report shows an estimated number of 8 additions of data-type `double` and 9 multiplications of data-type `double` (each operator executed 10,001 times).

Comparison of Symmetric and Transposed Implementations

Compare the symmetric and transposed FIR filter reports generated by using the `socModelAnalyzer` function. The Symmetric FIR filter uses fewer multiplication operators (9) than the Transposed FIR filter (5). They both use the same number of add operators (8).

Conclusion

You used the `socModelAnalyzer` function to estimate and analyze the number of arithmetic operators in two FIR filter implementations. You generated operator reports for both Symmetric and Transposed FIR filters. You compared the number of multiply and add operators for two implementations.

You can use the `socModelAnalyzer` function for analyzing the number of operators in a Simulink algorithm.

See Also

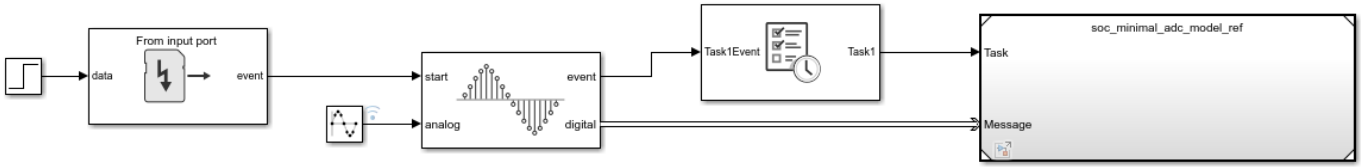
`socModelAnalyzer` | `socAlgorithmAnalyzerReport`

Simulate Analog to Digital Conversion for MCU

This example shows a minimal ADC simulation for an MCU using SoC blocks.

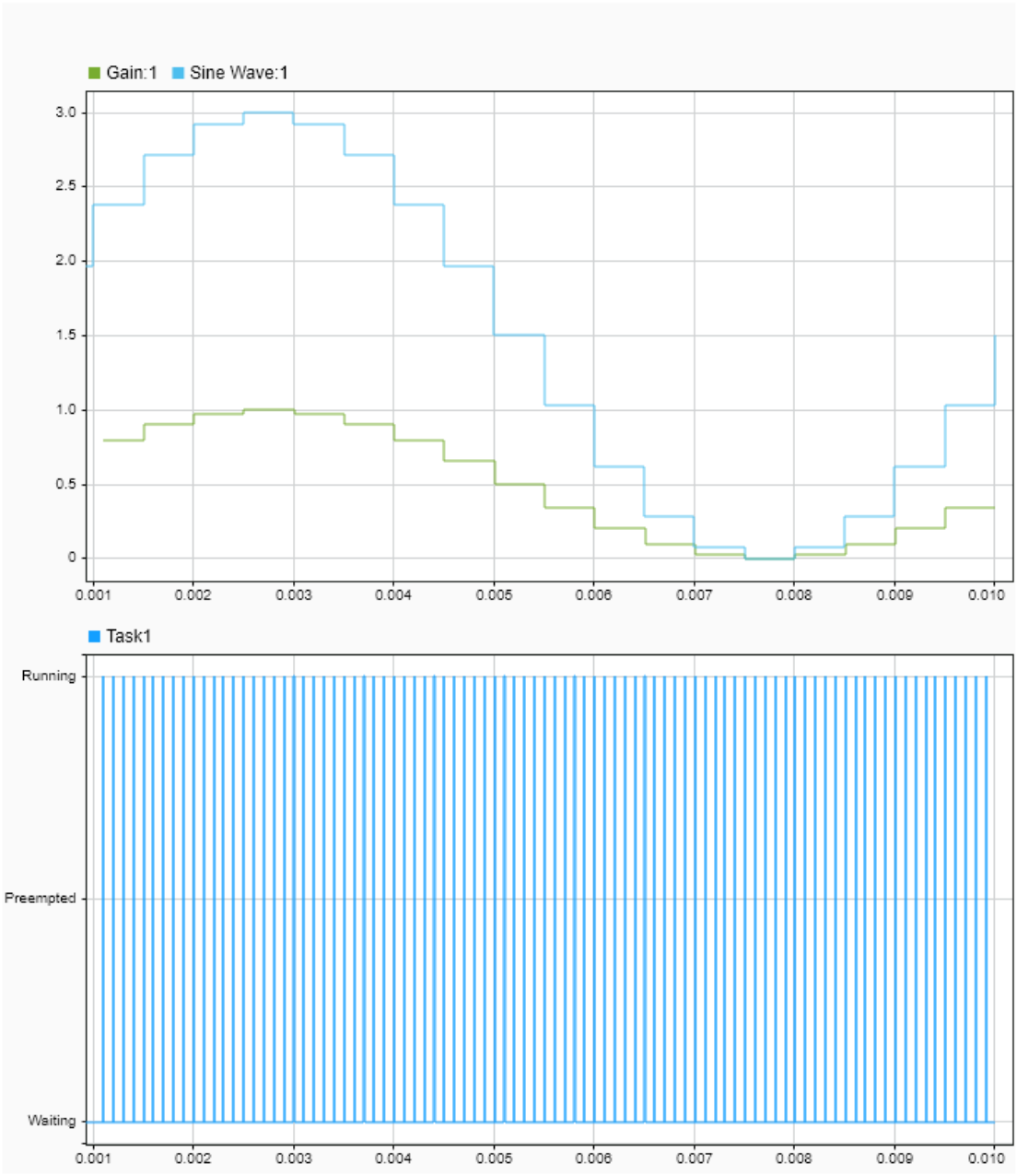
Model

The following model simulates an analog-to-digital converter (ADC) in an MCU. The original voltage signal is a sine wave with an amplitude of 1.5V with an average offset of 1.5V. An initial signal at 1 us starts the ADC sampling. The ADC Interface block samples the signal at the rate of the combined Acquisition time and Conversion time of the simulated ADC hardware, approximately 0.1 us. At each sample an asynchronous task executes to process the data sample and give the normalized ADC mesurement.



Results

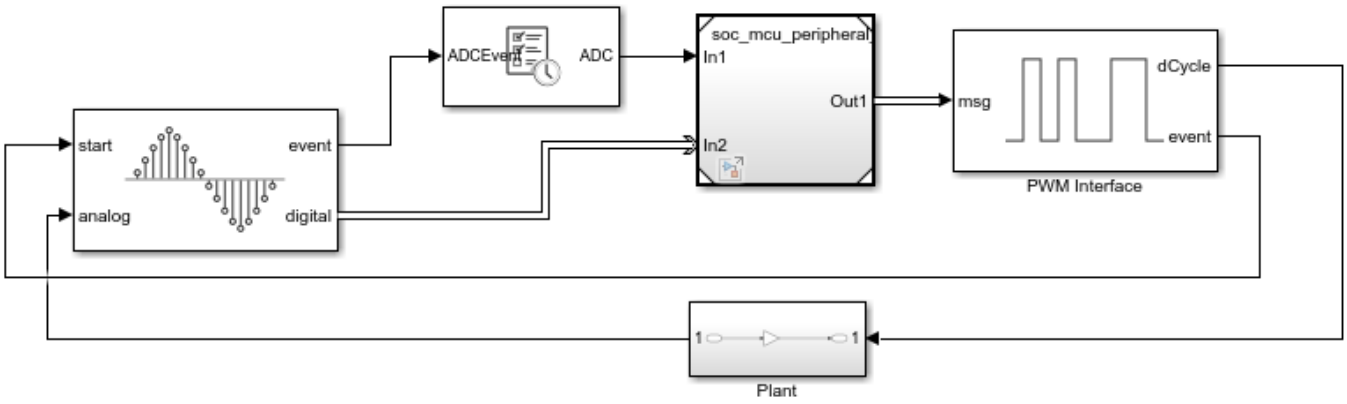
In the Simulation tab, click Run. When the simulation completes, open the Simulation Data Inspector to view the resulting signals and tasks. From the graphs, the original signal is sampled and normalized inside the process running on the MCU. The entire process run asynchrously, with Task1 executing only upon receiving a new value.


Gain:1
Sine Wave:1
3.0
2.5
2.0
1.5
1.0
0.5
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009
0.010
Task1
Running
Preempted
Waiting
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009
0.010

Map Peripherals in MCU Model

This example shows how to map the peripheral blocks in an MCU model to the peripheral pins on the MCU hardware board.

1. Open the top level MCU peripheral model.



2. On the System On Chip tab, click Hardware Settings to open the Configuration Parameters window.

3. On the Hardware Implementation tab, on Hardware board settings > Design mapping click View/Edit Peripheral Map to open the Peripheral Mapping tool.

4. Complete actions.

Get Started with SoC Blocks on MCUs

This example shows how to simulate and deploy a closed-loop feedback control algorithm on to a MCU using SoC Blockset.

SoC Blockset allows you to create a closed-loop model consisting of a plant, an algorithm running on a microcontroller and hardware peripherals interfacing microcontroller to the plant. You can easily create a high-fidelity simulation of the system by taking advantage of following capabilities:

- Model ADC and PWM peripherals with accurate real-time behavior
- Model interrupts as tasks including synchronization and scheduling
- Model task latencies due to execution and sensor delays

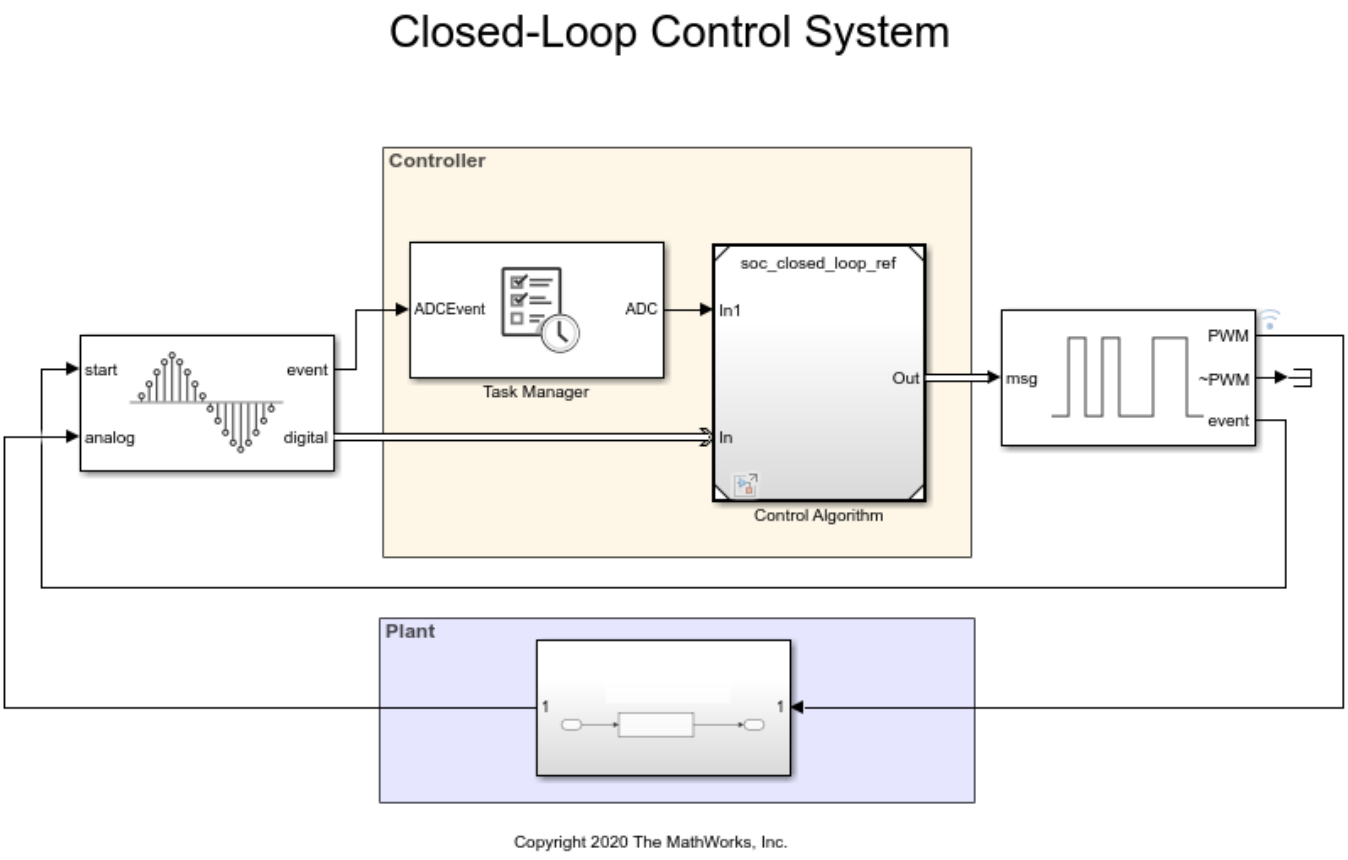
This example shows how to use SoC Blockset to deploy a Simulink model of a closed-loop application on to the TI Delfino F28379D LaunchPad.

Supported hardware platforms:

- TI Delfino F28379D LaunchPad
- TI Delfino F2837xD based board

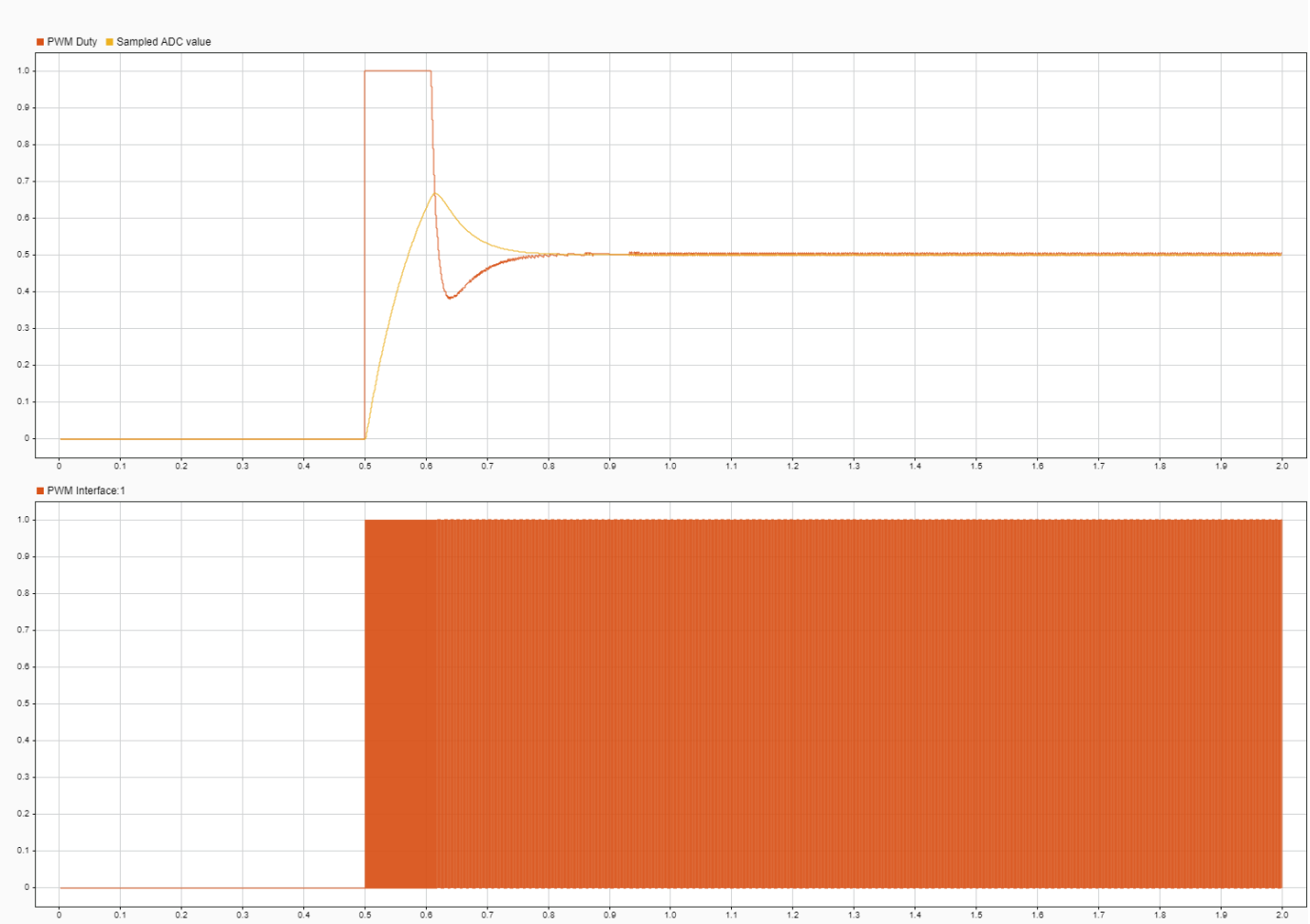
Model Using SoC Blockset

```
open_system('soc_closed_loop');
```



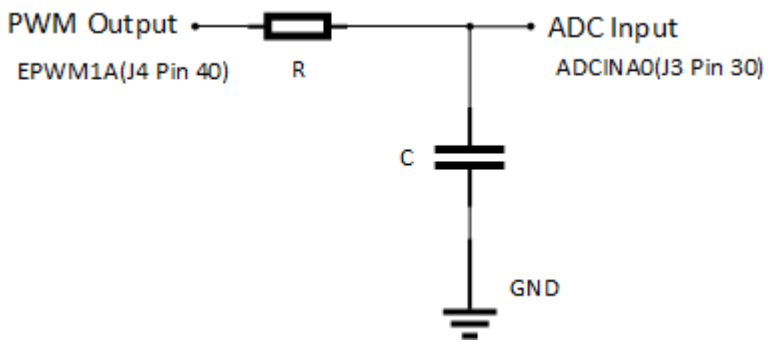
Open the closed-loop template model. This model shows a basic closed-loop control application with a low pass filter as a plant. The output of the plant is sampled by the ADC Interface generating an event on each conversion. The Task Manager executes an event-driven task called ADC upon reception of each ADC end-of-conversion event. The ADC Interrupt task contains the feedback control algorithm that executes asynchronously in response to each ADC conversion event. The control algorithm receives feedback through ADC Read and generates duty cycle values for PWM Write block. The PWM Interface block simulates PWM behavior including triggering an event to start the next ADC conversion.

Click 'Play' to simulate the model. Open the Simulation Data Inspector and view signals. Validate the models in simulation before trying deployment. The figure shows the controller response and switching PWMs generated from PWM Interface block. You can tune the PI controller parameters by adjusting the gain values in PID Controller block.



Deploy and Run SoC Model on MCU

You can create the first order plant model from the simulation using simple RC circuit. Assuming no loading at the output of the RC-circuit, you can use R = 10kOhm and C = 10uF. Connect the output of the selected EPWM1A, J4 pin 40 in F28379D launchpad with ADCINA0, J3 pin 30 as shown in the figure below.



You can directly deploy the model on to the TI Delfino F28379D LaunchPad by following the below step by step instructions and guidelines.

1. Open the SoC Builder tool from the **System on Chip** tab, by clicking **Configure, Build, & Deploy**.
2. Review Task Mapping in the next page. Observe ADCA1_isr is configured as event source for control task.
3. Review “Map Peripherals in MCU Model” on page 7-131 in the next page. Configure peripherals with same value used for simulation.
4. **Validate Model** page ensures the models are error free. If model compilation step fails, try **Update Model** (Ctrl+D) from the **Debug** tab.
5. To monitor data from hardware, select **Build and load for External mode** in **Select Build Action** page. In the next page, click **Load and Run**. Open the Simulation Data Inspector and view signals from the hardware.
6. To profile task execution on the processor, open the controller reference model and select profiling information to **Show in SDI**. Select **Build and load for External mode** in **Select Build Action** page. In the next page, click **Load and Run**. Open the Simulation Data Inspector and view task profiling data from the hardware.

See Also

- “Integrate MCU Scheduling and Peripherals in Motor Control Application” on page 7-140
- “Partition Motor Control for Multiprocessor MCUs” on page 7-135

Partition Motor Control for Multiprocessor MCUs

This example shows how to partition real-time motor control application on to multiple processors to achieve design modularity and improved control performance.

Many MCUs provide multiple processor cores. These additional cores can be leveraged to achieve a variety of design goals:

- Divide the application into real-time tasks, such as control laws, and non-real time tasks, such as external communication, diagnostics, or machine learning
- Partition the control algorithm to run on multiple CPUs to achieve higher loop rate
- Run the same application in multiple CPUs for safety critical applications

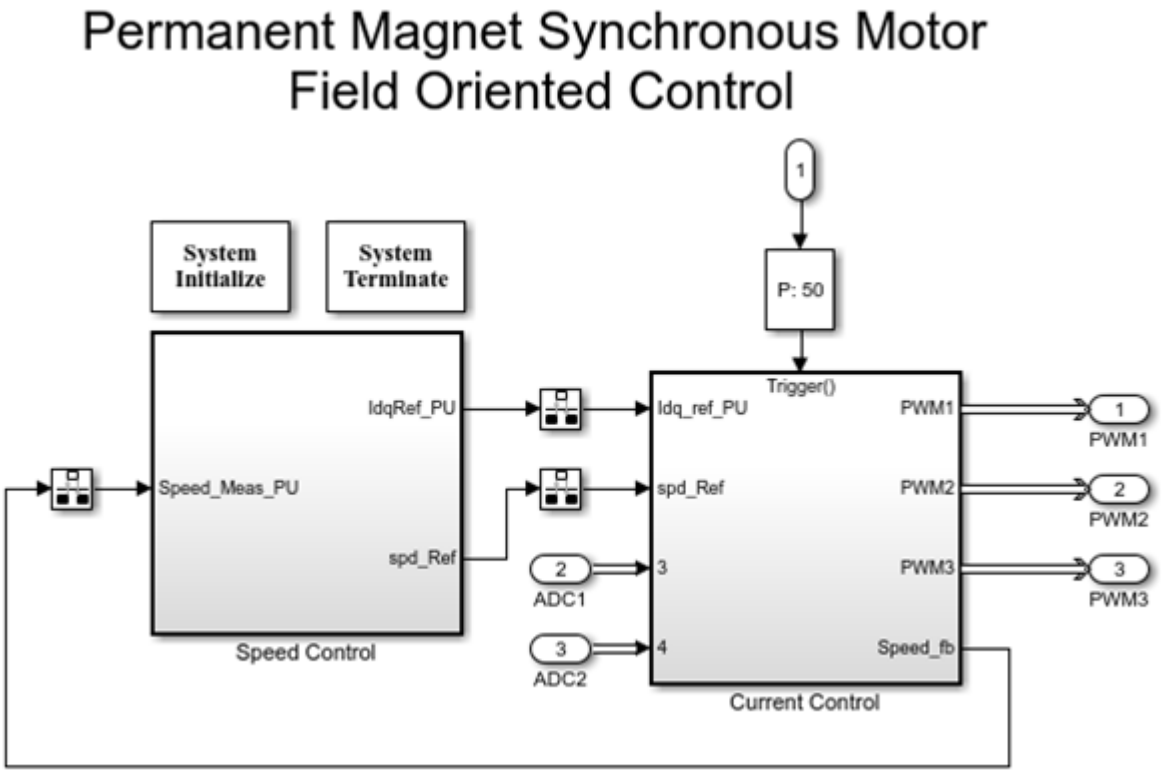
This example shows how to partition motor control application across two CPUs of the TI Delfino F28379D to achieve higher sampling time/PWM frequency.

Required hardware:

- TI Delfino F28379D LaunchPad or TI Delfino F2837xD based board
- BOOSTXL-DRV8305EVM motor driver board
- Teknic M-2310P-LN-04K PMSM motor

Partition Motor Control Algorithm

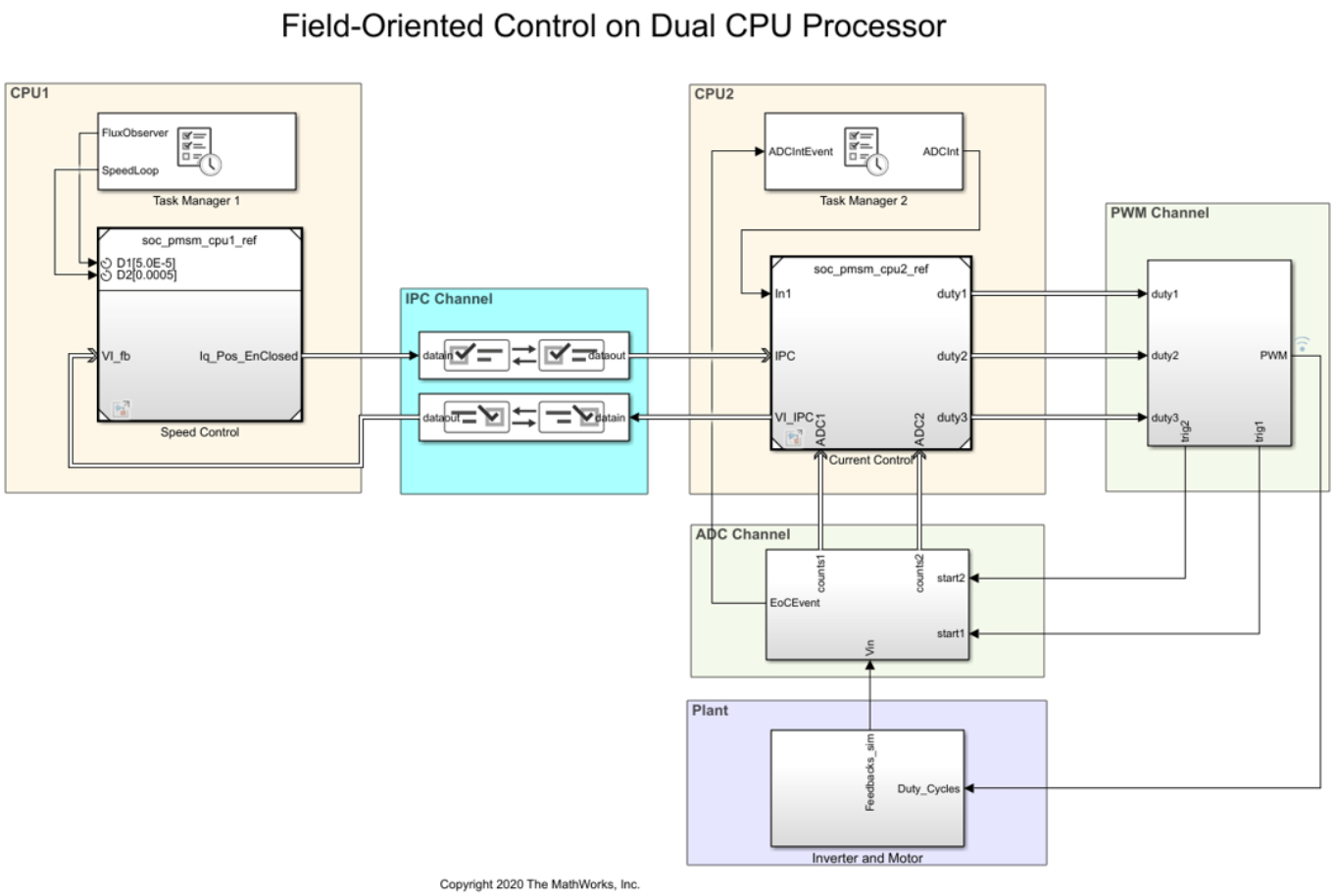
Open the soc_pmsm_singlecpu_foc model. This model simulates a single CPU motor controller, contained in the soc_pmsm_singlecpu_ref model, for a permanent magnet synchronous machine (PMSM).



We partition the control algorithm by executing current control on CPU2, and speed control and position estimation on CPU1 respectively. Data transfer between the CPU's are handled by Interprocess Data Channel block. For more information see “Interprocess Data Communication via Dedicated Hardware Peripheral” on page 3-31.

Open the soc_pmsm_dualcpu_foc model.

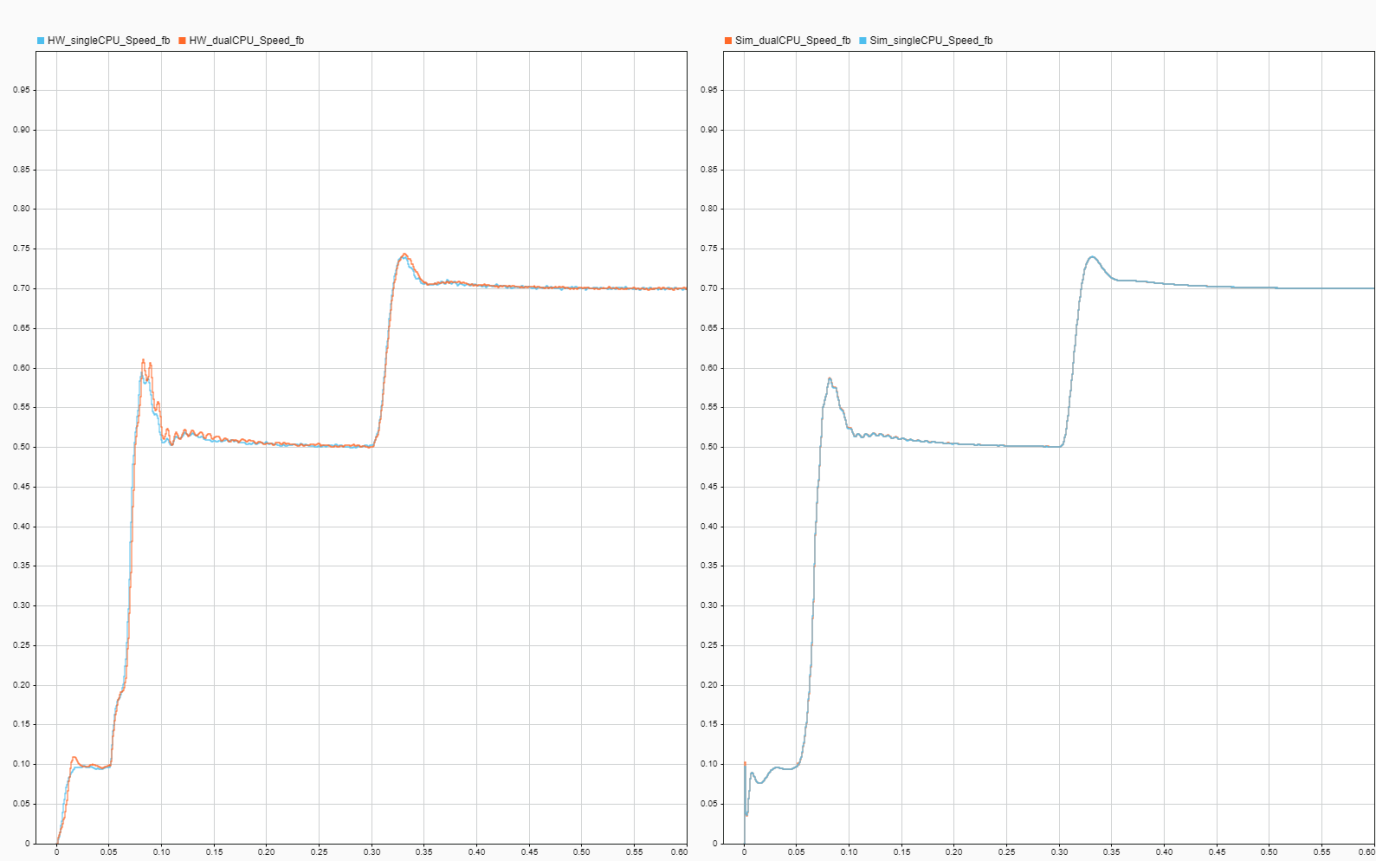
```
open_system('soc_pmsm_dualcpu_foc');
```



On the **System on Chip** tab, click **Hardware Settings** to open the **Configuration Parameters** window. In the **Hardware Implementation** tab, the **Processing Unit** parameter is configured to "None" indicating it is the top-level system model.

Open the soc_pmsm_cpu1_ref model and open the soc_pmsm_cpu2_ref model to view algorithms configured for each CPU. Model references contained within the system model are configured to run on c28xCPU1 (CPU1) and c28xCPU2 (CPU2).

On the Simulation tab, click 'Run' to simulate the model. Open the Simulation Data Inspector and view signals. This figure shows results from the single and dual CPU models in simulation and deployment.



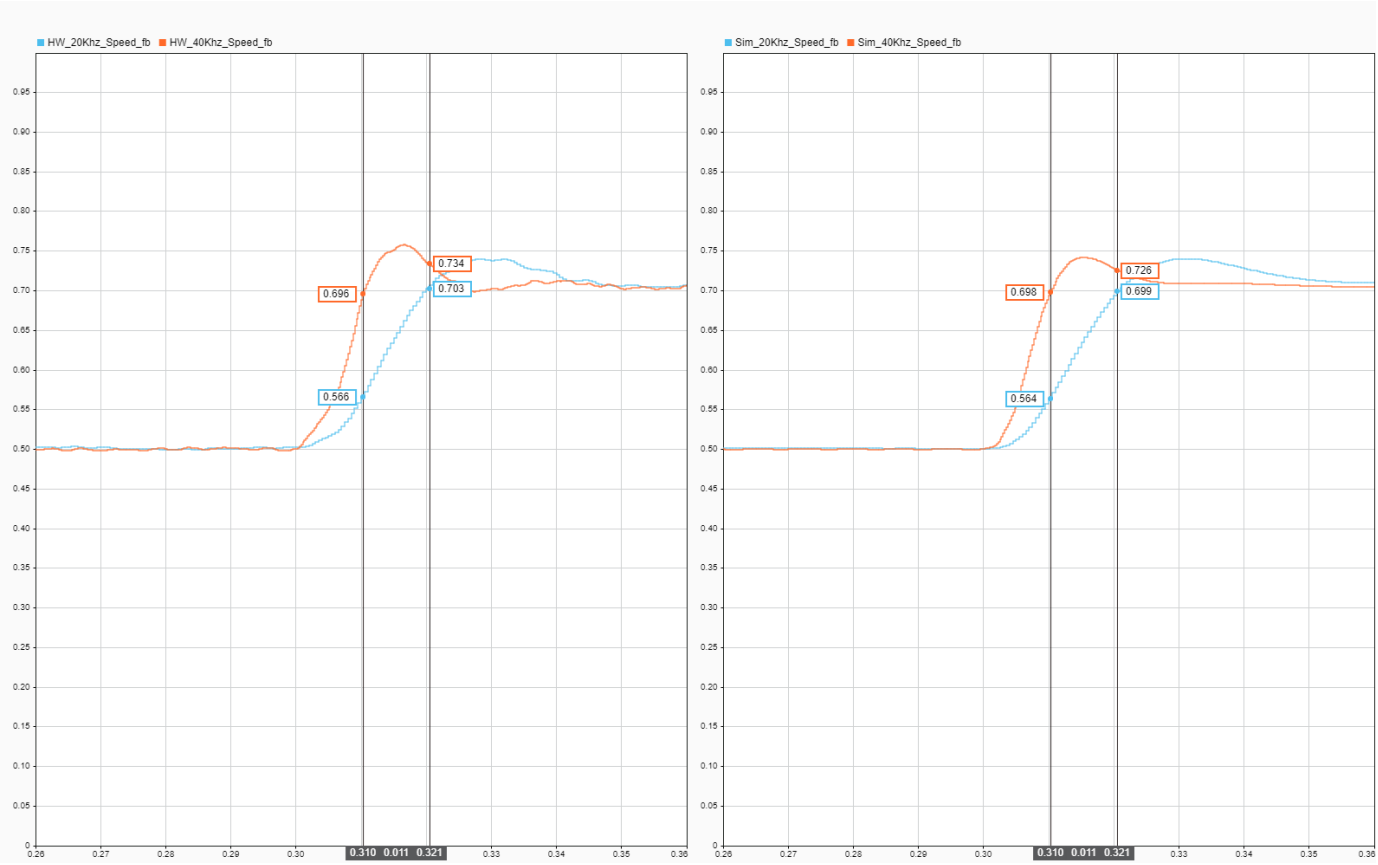
Performance Improvement with Concurrent Execution

Using both CPUs to execute control algorithms allows us to achieve higher controller bandwidth. In the original single CPU model, the control algorithm takes just over 25us to execute. To provide a safety margin, single CPU model uses a PWM frequency of 20kHz, equivalent to 50us period.

After partitioning, the CPU1 and CPU2 execution times reduce to less than 20us. Allowing the PWM frequency to be increase to 40kHz. In the `mcb_pmsm_foc_sensorless_f28379d_data.m` script, set PWM_frequency to 40e3 and run the script to configure the model to the new PWM frequency. With faster sampling of currents, controller gains can then be tuned to achieve faster response times.

Deploy the model to the TI Delfino F28379D LaunchPad using the SoC Builder tool. To open the tool, on the **System on Chip** tab, click **Configure, Build, & Deploy,** and follow the guided steps.

This figure shows the controller response from simulation and deployment at 25us current loop with 40kHz PWM frequency compared with 50us current loop at 20kHz frequency. As expected, the rise time in speed improves with faster current loop by approximately 50 percent.



Speed response is oscillatory because of sensorless algorithm, for more information see “Sensorless Field-Oriented Control of PMSM” (Motor Control Blockset)

For higher simulation granulairty, set the PWM Interface block output to Switching Mode and change the plant model variant to use the MOSFET simulation.

See Also

- “Get Started with SoC Blocks on MCUs” on page 7-132
- “Integrate MCU Scheduling and Peripherals in Motor Control Application” (Motor Control Blockset)

Integrate MCU Scheduling and Peripherals in Motor Control Application

This example shows how to identify and resolve issues with respect to peripheral settings and task scheduling early during development.

The following are typical challenges associated with MCU peripherals and scheduling:

- ADC-PWM synchronization to achieve current sensing at mid point of PWM period
- Incorporate sensor delays to achieve the desired controller response for the closed loop system
- Studying different PWM settings while designing special algorithms

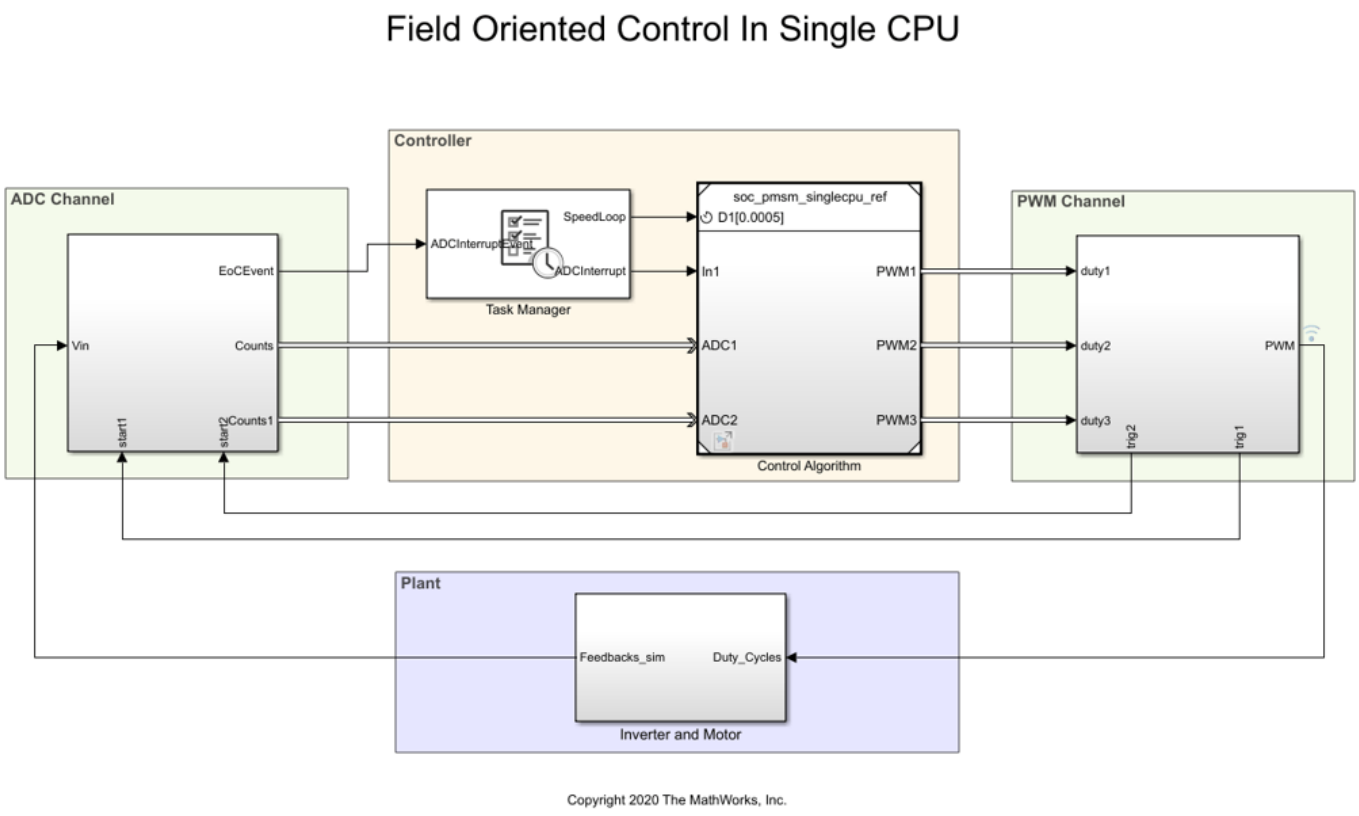
This example shows how to use SoC Blockset to address these challenges for a motor control closed-loop application in simulation and verify on hardware by deploying on to the TI Delfino F28379D LaunchPad.

Required hardware:

- TI Delfino F28379D LaunchPad or TI Delfino F2837xD based board
- BOOSTXL-DRV8305EVM motor driver board
- Teknic M-2310P-LN-04K PMSM motor

Model Structure

```
open_system('soc_pmsm_singlecpu_foc');
```

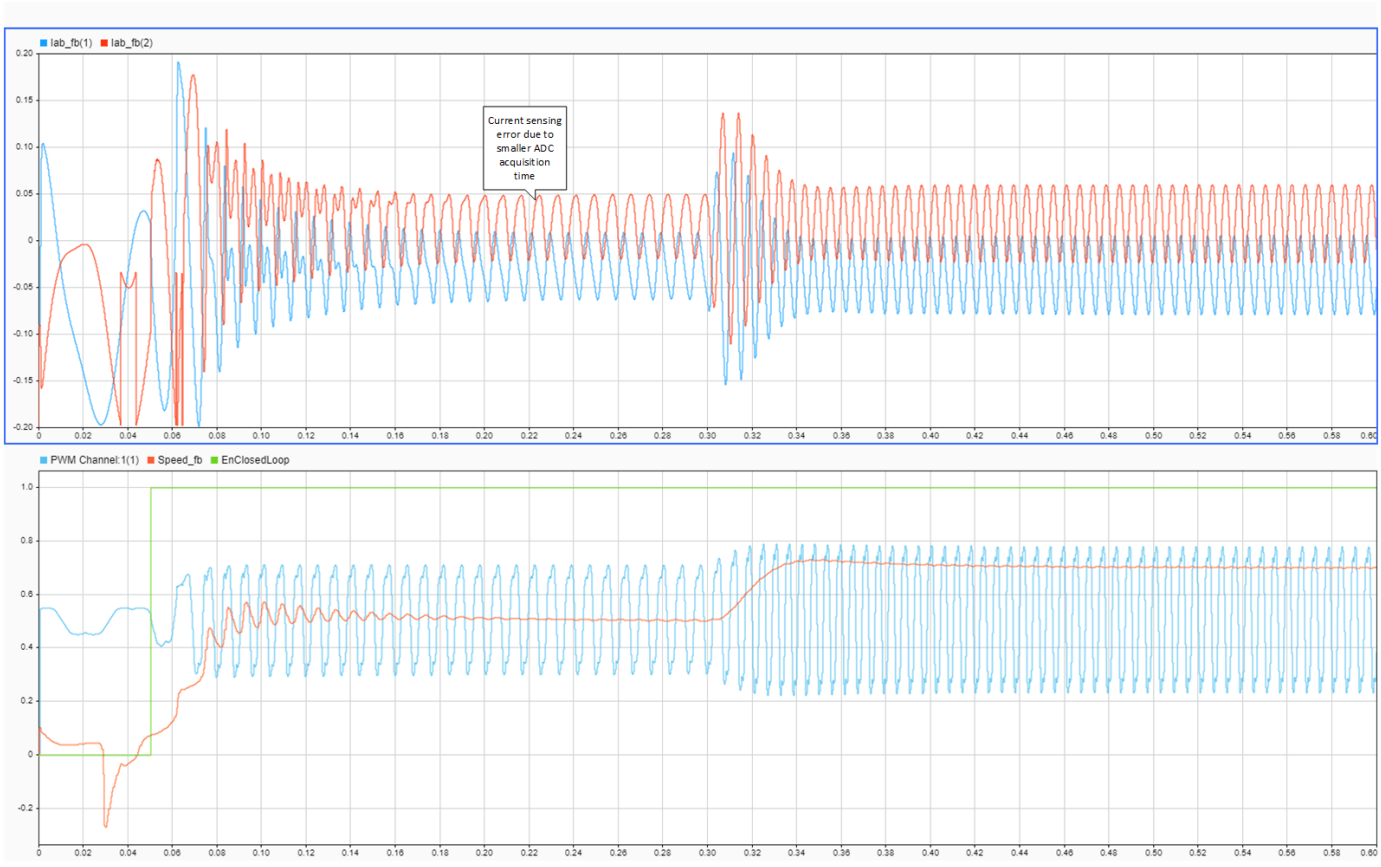


Open the soc_pmsm_singlecpu_foc model. This model simulates single CPU motor controller, contained in soc_pmsm_singlecpu_ref model, for a Permanent magnet synchronous motor inverter system. Controller senses the outputs from the plant using ADC Interface and actuates using PWM Interface that drives the inverter. Algorithm blocks from Motor Control Blockset™ is used in this example.

ADC Acquisition Time

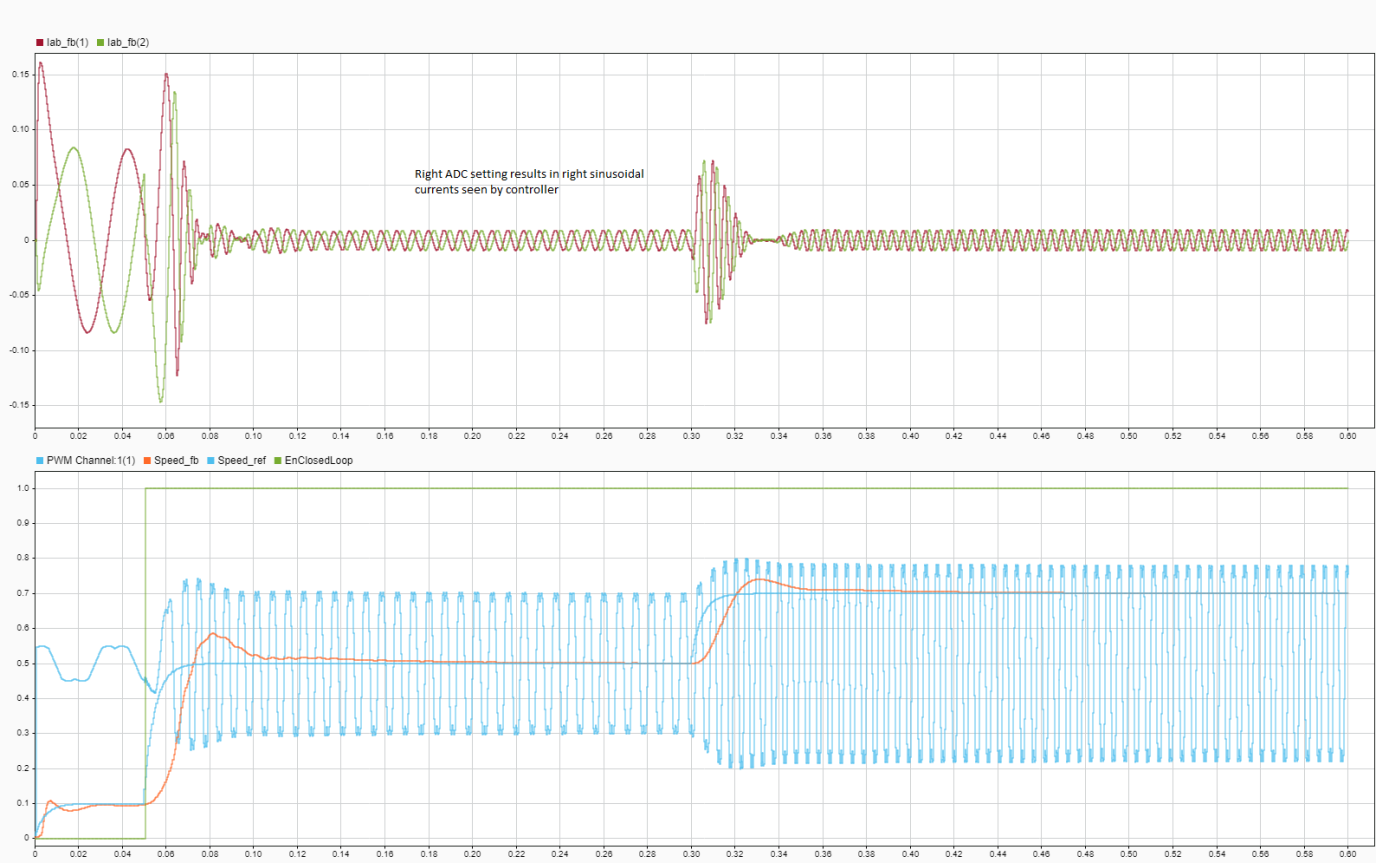
ADC hardware contains a sample and hold circuit to sense the analog inputs. To ensure complete ADC measurement, the minimum acquisition time must be selected to account for the combined effects of input circuit and the capacitor in the sample and hold circuit.

Open ADC Interface block and change the default acquisition time to 100ns. Run the simulation and view the results in Simulation Data Inspector and observe there is a distortion in current waveforms. The low acquisition time resulted in ADC measurements not reaching their true value. As a result, the controller reacts by generating a relative duty cycle causing variations in current drawn by the motor. These figures show the reaction to the incorrect ADC measurement and overdraw in the phase A current channel, with phase A current in blue and phase B current in orange. The simulated speed feedback shows significant oscillations during open loop to closed loop transition, which in real world will halt the motor.



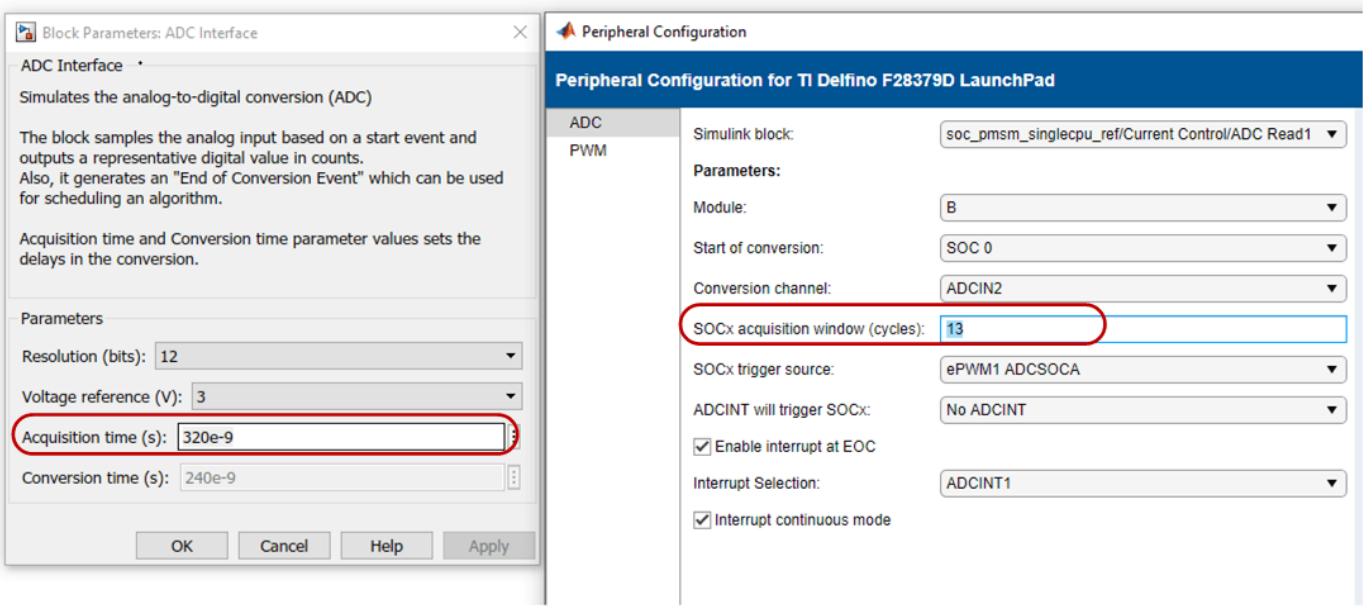
To fix this issue, open ADC Interface blocks change and change acquisition time to a larger value, 320ns. This value is the minimum ADC acquistion time recommended in Table 5-42 of the TI Delfino F28379D LaunchPad data sheet. Run the simulation and view the results in Simulation Data

Inspector. This figure shows the accurately sampled ADC values and the controller tracking the reference value as expected.

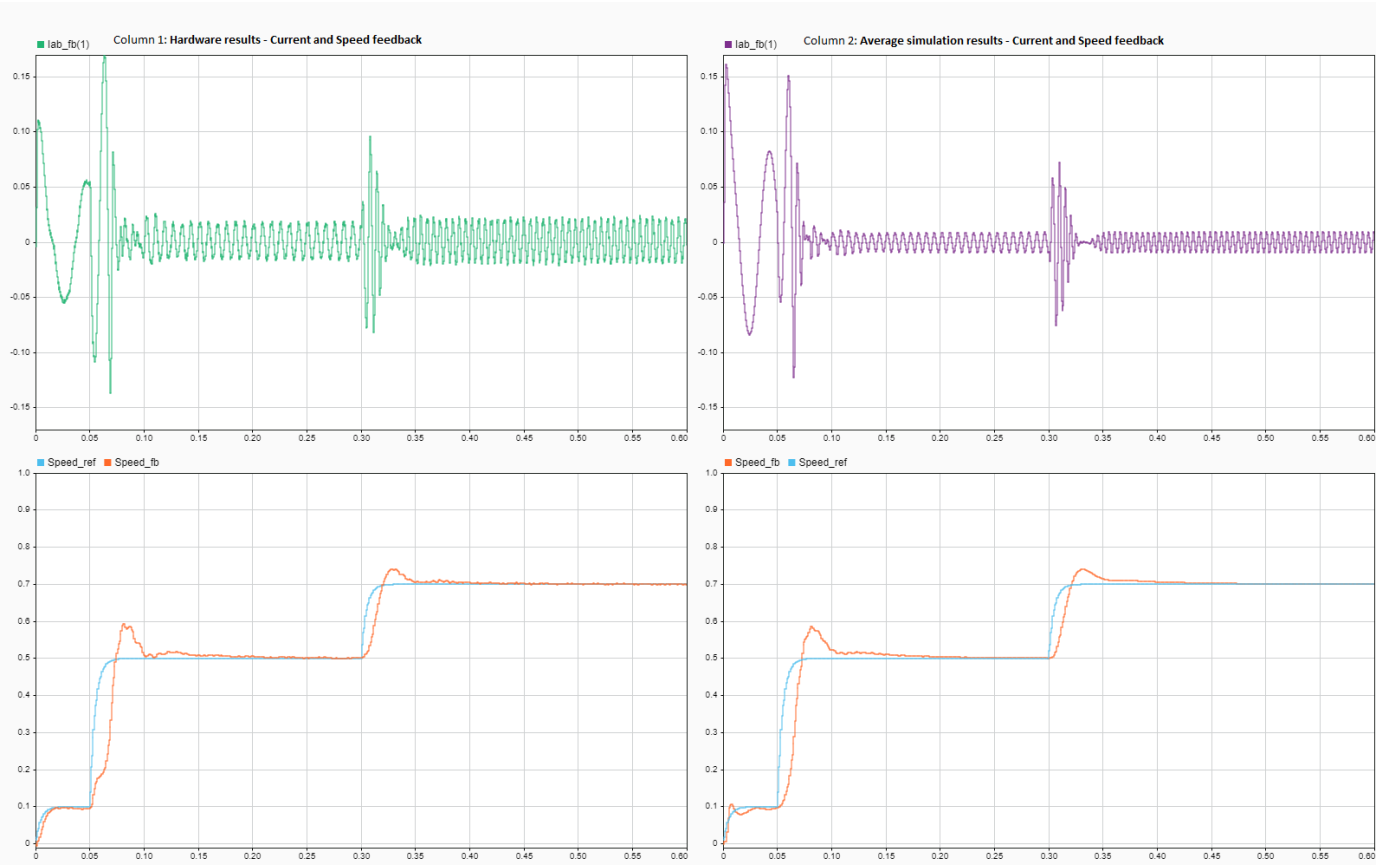


Verify simulation results against hardware by deploying the model to the TI Delfino F28379D LaunchPad. On the **System on Chip** tab, click **Configure, Build, & Deploy** to open the SoC Builder tool.

In the SoC Builder tool, on **Peripheral Configuration** tool, set **ADC > SOCx acquisition window cycles** parameter to `13 ADC clock ticks` for the ADC B and C modules. The ADC acquisition clock ticks parameter must be set to the simulation time value, set in the ADC Interface block, multiplied by the ADC clock frequency. You can get the ADC clock frequency from the model hardware settings. Open the soc_pmsm_singlecpu_ref model. On the **System on Chip** tab, click **Hardware Settings** to open the **Configuration Parameters** window. In the **Hardware Implementation > Target hardware resources > ADC_x** section, you can see the `ADC clock frequency in MHz` parameter value. This figure shows the ADC Interface block setting for simulation and peripheral app setting for deployment. Use same setting in simulation and codegen to ensure expected behavior.

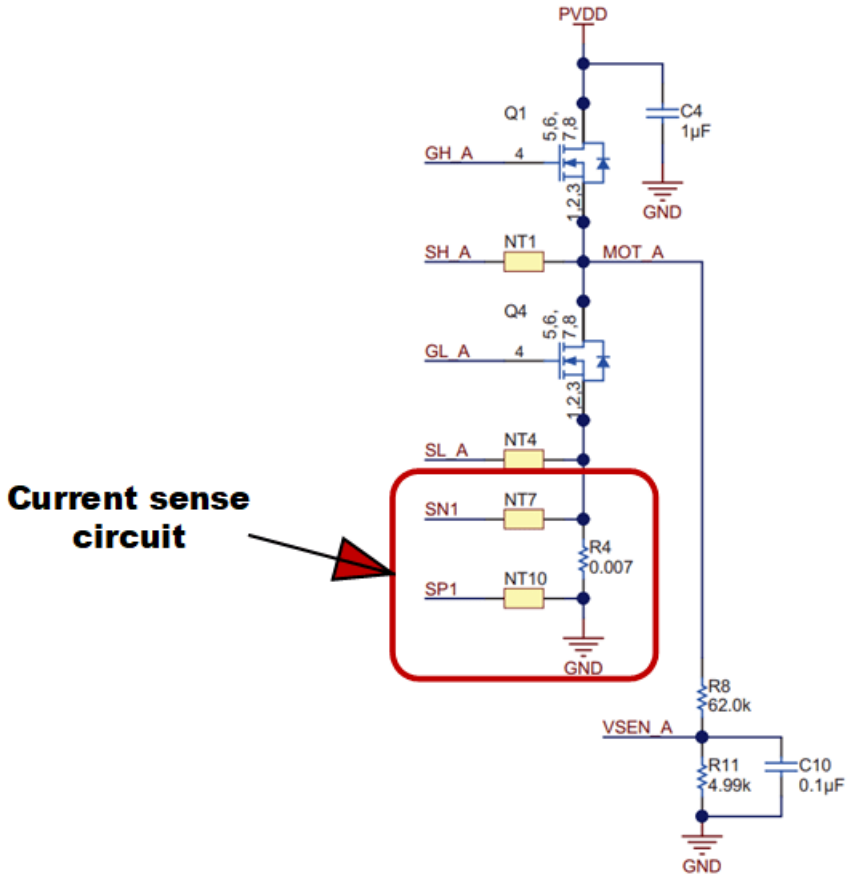


On **Select Build Action** page, to monitor data from hardware select `Build and load for External mode`. This figure shows the data from hardware with accurately sampled ADC values and the controller tracking the reference value as expected.



ADC-PWM Synchronization

The BOOSTXL-DRV8305EVM motor driver has a 3-phase inverter built using 6 power MOSFETS. This motor driver board uses a low-side shunt resistor to sense motor currents. The Current sense circuit amplifies the voltage drop across the shunt. This setup ensures low power dissipation, since the current only flows through the shunt when the bottom switches are on and away from PWM commutation noise. This figure shows the low-side shunt resistor circuit in BOOSTXL-DRV8305EVM motor drive.



For correct operation, current sensing must occur during the mid point of the PWM period when ADCs trigger. Specifically, the PWM counter must be at the maximum value when the bottom switches are active in the Up-Down counter mode. Current sampling at a different instance results in a measured currents of zero.

To analyze this case, switch the model to **high fidelity inverter simulation** mode. Change the plant variant to use detailed MOSFET based 3-phase inverter to replicate BOOSTXL-DRV8305EVM.

```
set_param('soc_pmsm_singlecpu_foc/Inverter and Motor/Average or Switching',...
'LabelModeActivechoice','SwitchingInverter');
```

Change the `Output mode` parameter of PWM Interface to `Switching` and connect 6 PWMs to the Mux block.

```
set_param('soc_pmsm_singlecpu_foc/PWM Channel/PWM Interface', 'OutSigMode', 'Switching');
set_param('soc_pmsm_singlecpu_foc/PWM Channel/PWM Interface1', 'OutSigMode','Switching');
set_param('soc_pmsm_singlecpu_foc/PWM Channel/PWM Interface2', 'OutSigMode', 'Switching');
```

Delete existing connection between PWM Interface block and Mux.

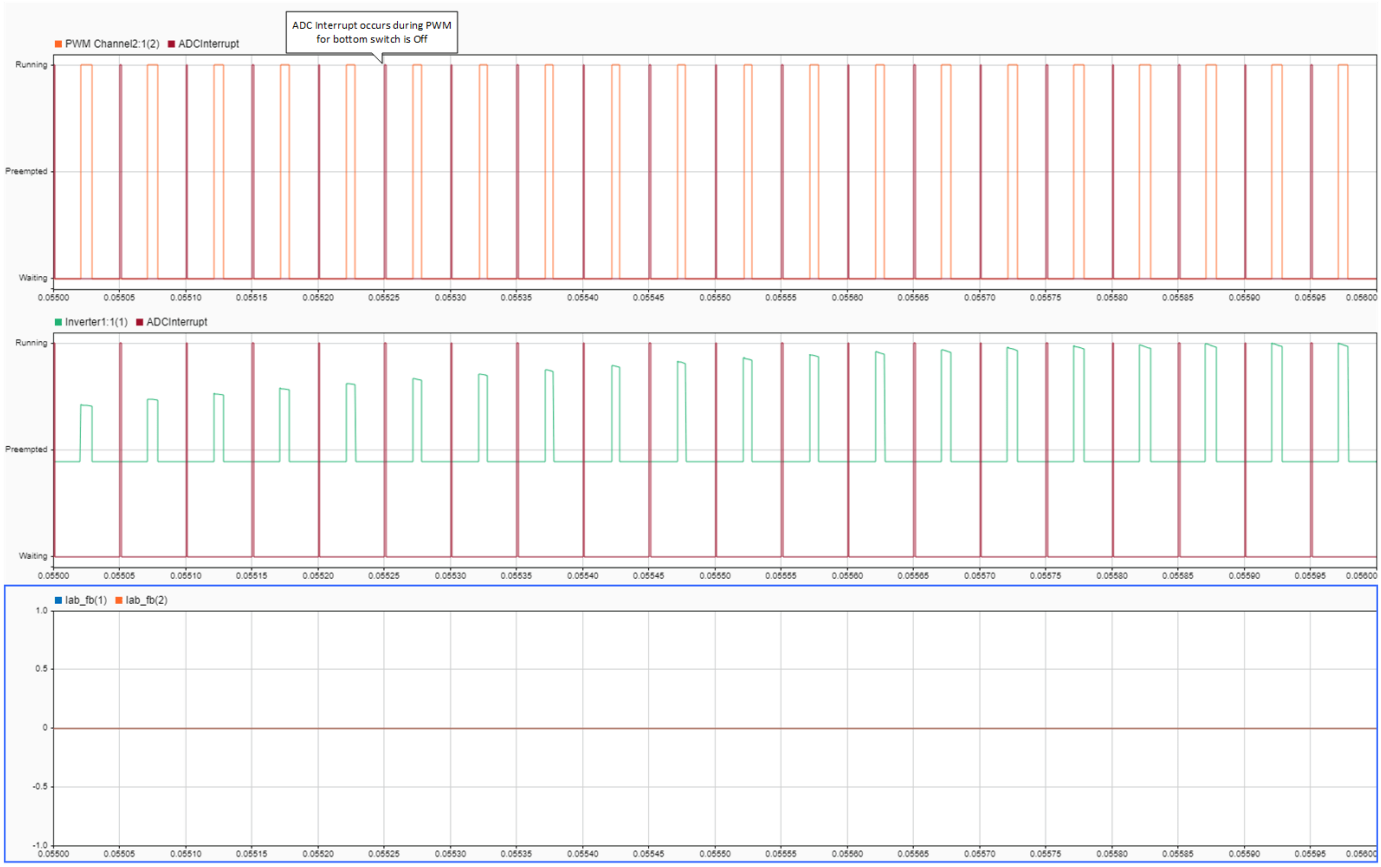
```
h = get_param('soc_pmsm_singlecpu_foc/PWM Channel/Mux','LineHandles');
delete_line(h.Inport);
```

As a last step, connect 6 PWM outputs to Mux.

```
set_param('soc_pmsm_singlecpu_foc/PWM Channel/Mux','Inputs','6');
```

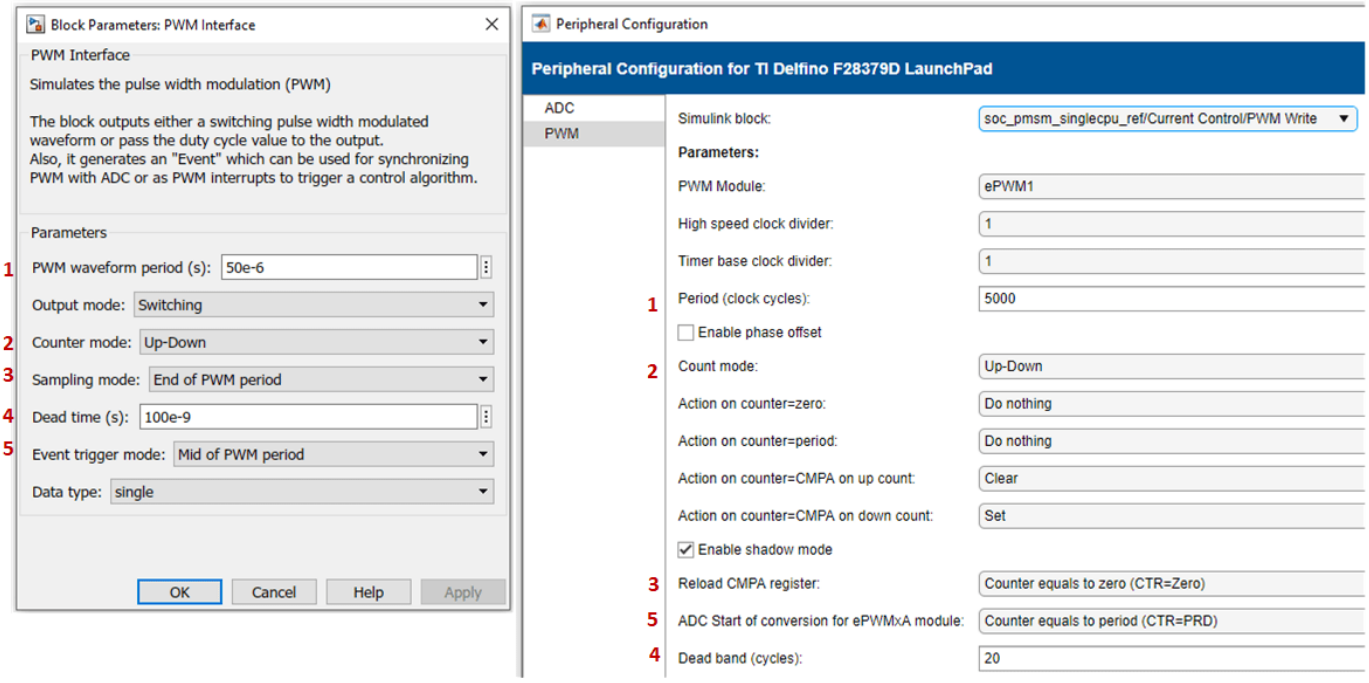
```
add_line('soc_pmsm_singlecpu_foc/PWM Channel', ...
{'PWM Interface/1', 'PWM Interface/2', 'PWM Interface1/1',...
'PWM Interface1/2', 'PWM Interface2/1', 'PWM Interface2/2'}, ...
{'Mux/1','Mux/2','Mux/3','Mux/4','Mux/5','Mux/6'}, 'autorouting', 'smart');
```

Open the PWM Interface blocks and set **Event trigger mode** to `End of PWM period`. Run the simulation and view the results in Simulation Data Inspector. In the figure, phase A and phase B currents are approximately zero current. This results in a loss of feedback and no actuation in the control loop. Select `Enable task simulation` in Task Manager block to simulate and visualize tasks in Simulation Data Inspector.

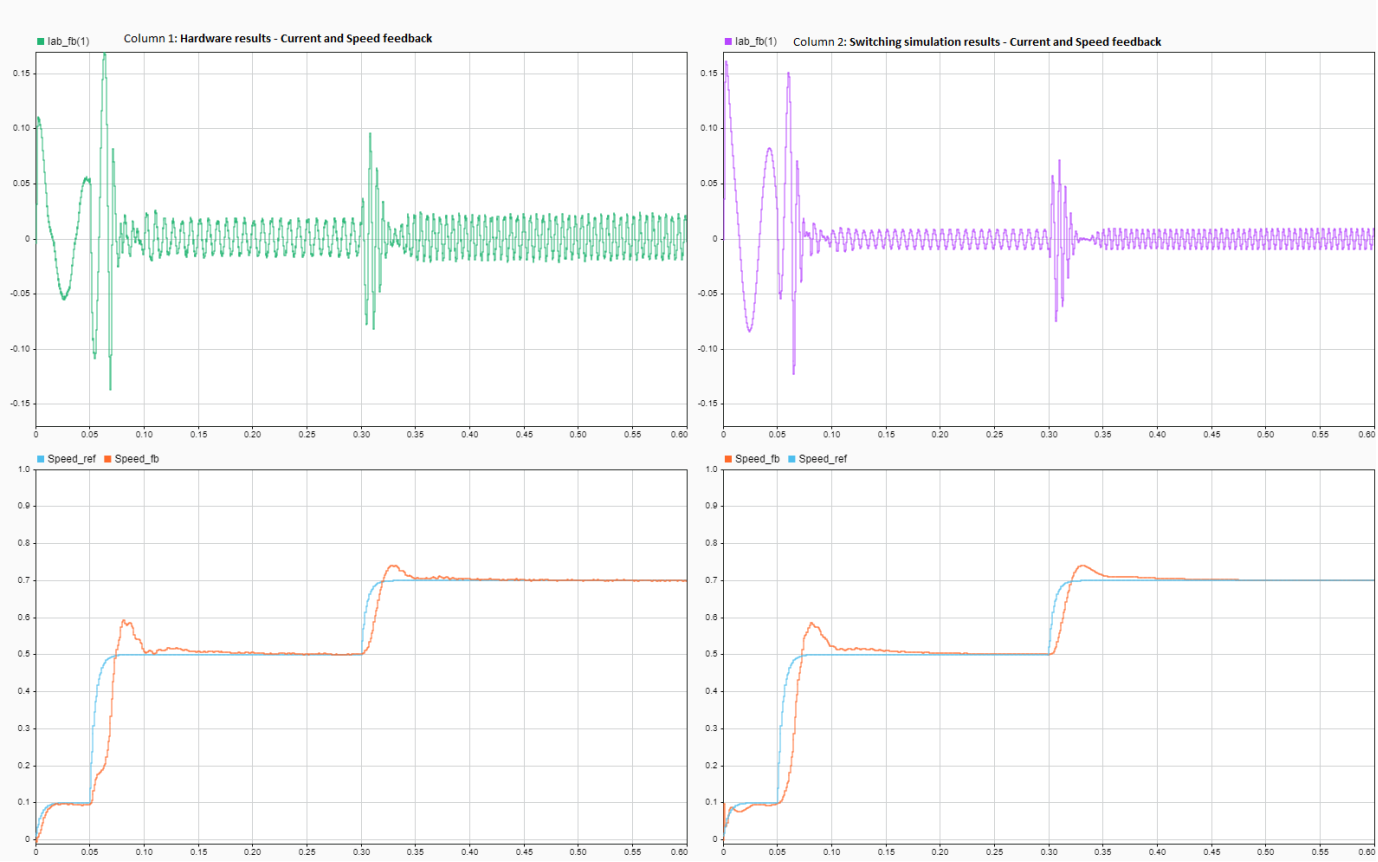


To fix this issue, change the **Event trigger mode** to `Mid point of PWM period`, equivalent to the PWM internal counter being at a maximum. Run the simulation and view the results in Simulation Data Inspector.

Deploy the model on to the TI Delfino F28379D LaunchPad using the SoC Builder tool. In the SoC Builder tool, on **Peripheral configuration** tool, set **PWM event condition** to `Counter equals to period`. Use same setting in simulation and codegen to ensure expected behavior. This figure shows the PWM Interface block setting for simulation and the Peripheral Configuration tool setting for deployment.



This figure shows the data from simulation and hardware with correct ADC-PWM synchronization and the controller tracking the reference value as expected.



See Also

- “Get Started with SoC Blocks on MCUs” on page 7-132
- “Partition Motor Control for Multiprocessor MCUs” (Motor Control Blockset)

DC-DC Buck Converter Using MCU

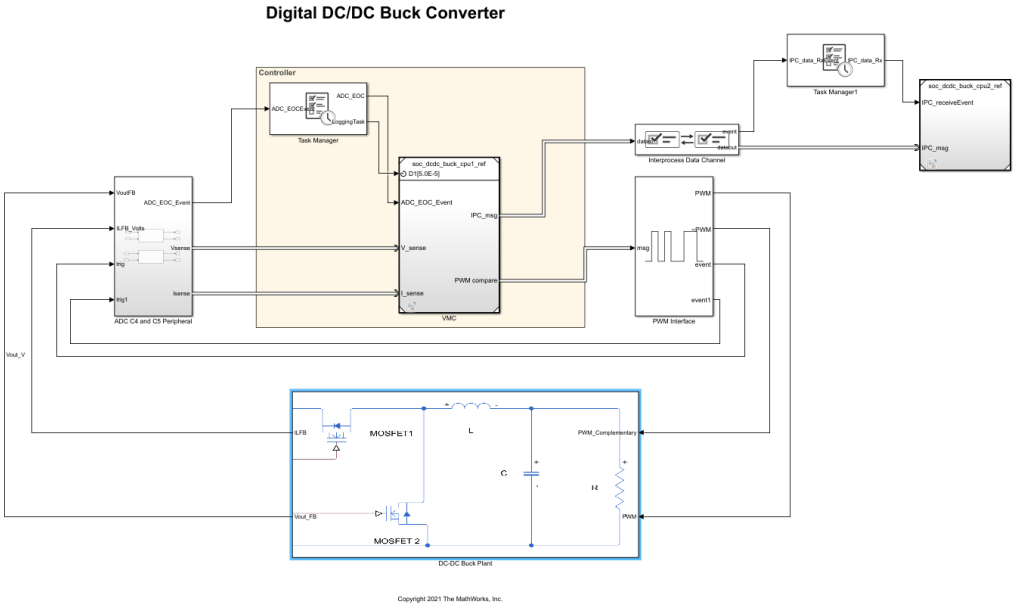
This example shows how to develop a DC-DC buck converter power regulator application.Typical challenges with power conversion simulation and deployment include:

- Modeling the analog circuit behavior of the buck converter circuit
- Modeling the timing behavior of PWM output and ADC sampling on an MCU
- Capturing signals in high CPU load controllers
- The amount of time required for controller validation, which is typically performed on hardware

These challenges are addressed in this example using SoC Blockset™ and Simscape™. Digital control type used in this example is voltage mode controller (VMC), verified on the TI Delfino F28379D LaunchPad and TI BOOSTXL-BUCKCONV kit.

This model shows the complete converter system, and the sections in this example will examine the individual challenges. To open this model, run the following code.

```
soc_dcdc_buck
```

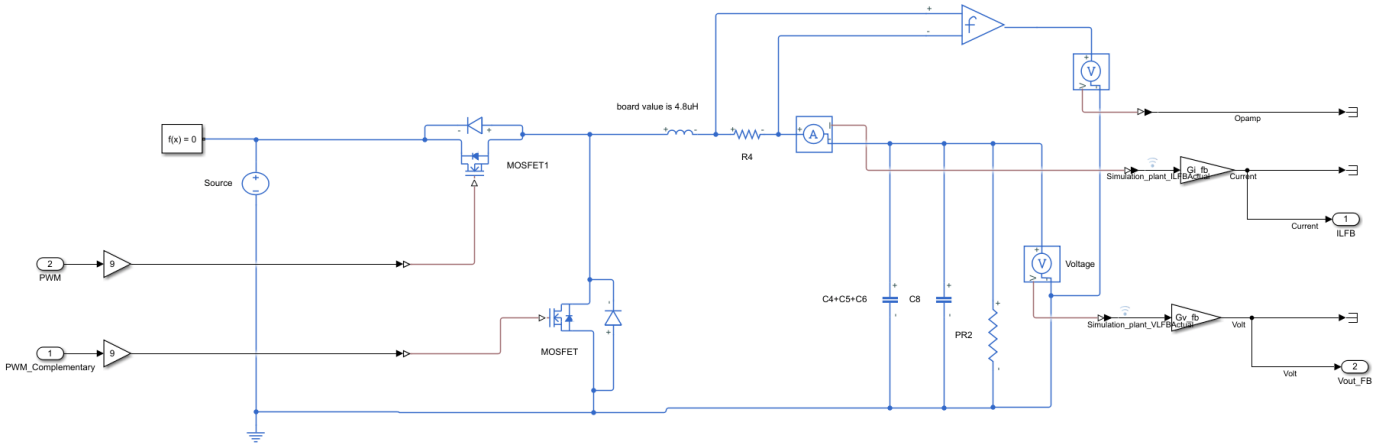


Required Hardware

- TI Delfino F28379D LaunchPad
- TI BOOSTXL-BUCKCONV kit

Model of DC-DC Buck Converter Kit

`DC-DC Buck Plant` subsystem is a Simscape reference model of the DC-DC buck converter analog circuitry.



The Simscape™ blocks in the model are selected and configured based on the original equipment manufacturer (OEM) specifications provided in the data sheet. To achieve computational efficiency in simulation without affect on behavior, the model includes these simplification relative to OEM specifications:

- Voltage and current sensing circuits are simplified to gain blocks.
- MOSFETS are simplified to ideal MOSFETS.
- Gate driver is not modeled and its propogation delays are not considered.
- Inductor is simplified to linear inductor.
- All parts are modeled with nominal values, and tolerances are not considered.
- DC supply is assumed to be constant.

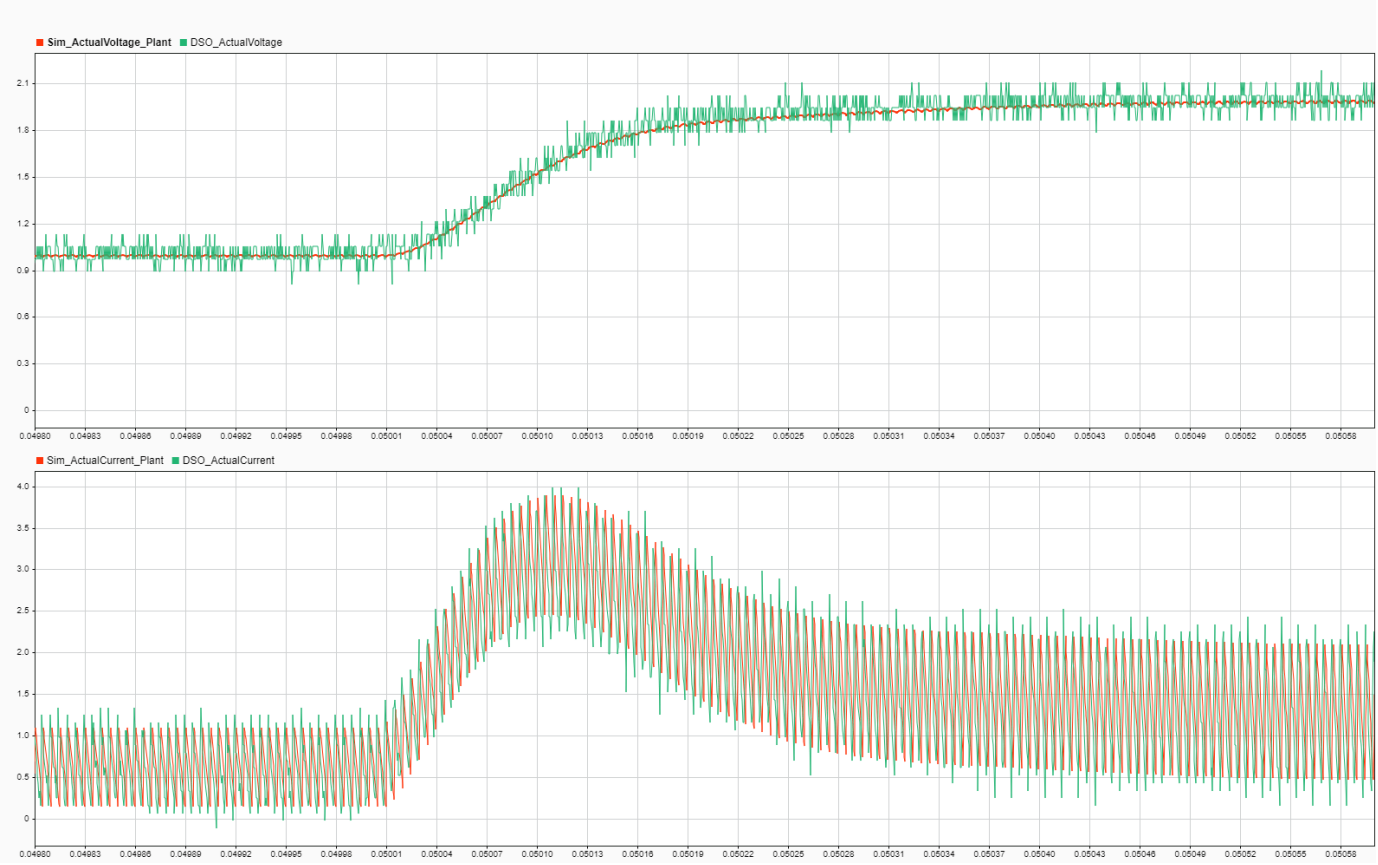
If needed, the open loop response of the Simscape model can be compared and verified against the physical hardware using a digital oscilloscope with results captured using the Data Acquistion Toolbox.

Voltage Mode Control on MCU

On the MCU, the output of the plant sampled by the ADC Interface generaring an event on each end of conversion. The Task Manager executes an event-driven task called ADC upon reception of each ADC end-of-conversion event. The ADC Interrupt task contains the feedback control algorithm that executes asynchronously in response to each ADC conversion event. The control algorithm receives feedback through ADC Read and generates duty cycle values for PWM Write block. The PWM Interface block simulates PWM behavior including triggering an event to start the next ADC conversion. PWM frequency is set to 200 kHz. The discrete proportional integral (PI) controller minimizes the error between the reference voltage and the output voltage. The duty cycle of the PI controller is limited to 40% of the PWM time period.

The system starts with an initital voltage reference of 1 volt and allowed to reach steady state. This enables a fair control between the phsyical hardware and simulation to compare with a known state. The desired voltage step of 2 volts is then triggered at 50 ms to examine the step response of the closed loop controller. Click **Play** to simulate the model. Open the **Simulation Data Inspector** and view signals.

To verify simulation results against hardware, deploy the model to the TI Delfino F28379D LaunchPad. On the **System on Chip** tab, click **Configure, Build, & Deploy** to open the SoC Builder tool. This figure shows the comparison of the controller response between the simulation and deployed model on the physical hardware. This signal on the hardware is captured using a digital oscilloscope. The high frequency operation of controller prevents the direct use of external mode on the same CPU. For this reason a digital oscilloscope is used to take these measurements.



As expected, the voltage mode controller correctly tracks the desired the voltage output. Additionally the measurements from the deployed model match the simulation with greater than 95% accuracy for this type of system. The minor differences seen between the simulation and the deployed measurements can be attributed to the simplifications made in the Simscape model.

Taking Advantage of Multicore to Log Data in CPU2

CPU2 is configured to run external mode **SoC Builder** tool, to log and transmit the high frequency signals produced by control loop on CPU1. An Interprocess Data Channel block connects CPU1 and CPU2, providing a low latency data transfer between the CPUs.

Use the SoC Builder tool to deploy the model to the TI Delfino F28379D LaunchPad. A host-target communication connection, set up by the **SoC Builder** tool, logs the signal data from the executable running on CPU2 of the hardware board and sends the data to the **Simulation Data Inspector** in Simulink. Using CPU2 to own and manage the host-target communication and data logging, data can be captured from the resource intensive, high-priority task on CPU1 without interfering with its behavior and enabling that task to consume most of the CPU resources, and with maintaining the

quality of data logging to Simulink. This figure shows the logged data signal from task 1 on CPU1, captured on task 2 on CPU2, of the model deployed to a TI Delfino F28379D LaunchPad.

ADC start of conversion trigger can be configured to generate at 1st PWM or 2nd PWM event. These settings are available in simulation and codegen. Observe simulation and codegen results match with greater than 95% accuracy.

Any resource intensive SoC Blockset model could use this setup to log data from hardware when the model is deployed to a TI Delfino F28379D LaunchPad. For more information on data logging techniques, see “Data Logging Techniques” on page 3-45.

Further Exploration

- Extend for high frequency switching applications involving Gallium Nitride (GaN) or Silicon Carbide (SiC)
- Variable PWM frequency and fixed duty cycle
- Variable phase offset
- Different PWM output schemes by using the PWM output control options
- Different PWM event generation techniques

See Also

- “Get Started with SoC Blocks on MCUs” on page 7-132

- “Partition Motor Control for Multiprocessor MCUs” on page 7-135
- “Data Logging Techniques” on page 3-45

Vertical Video Flipping Using External Memory

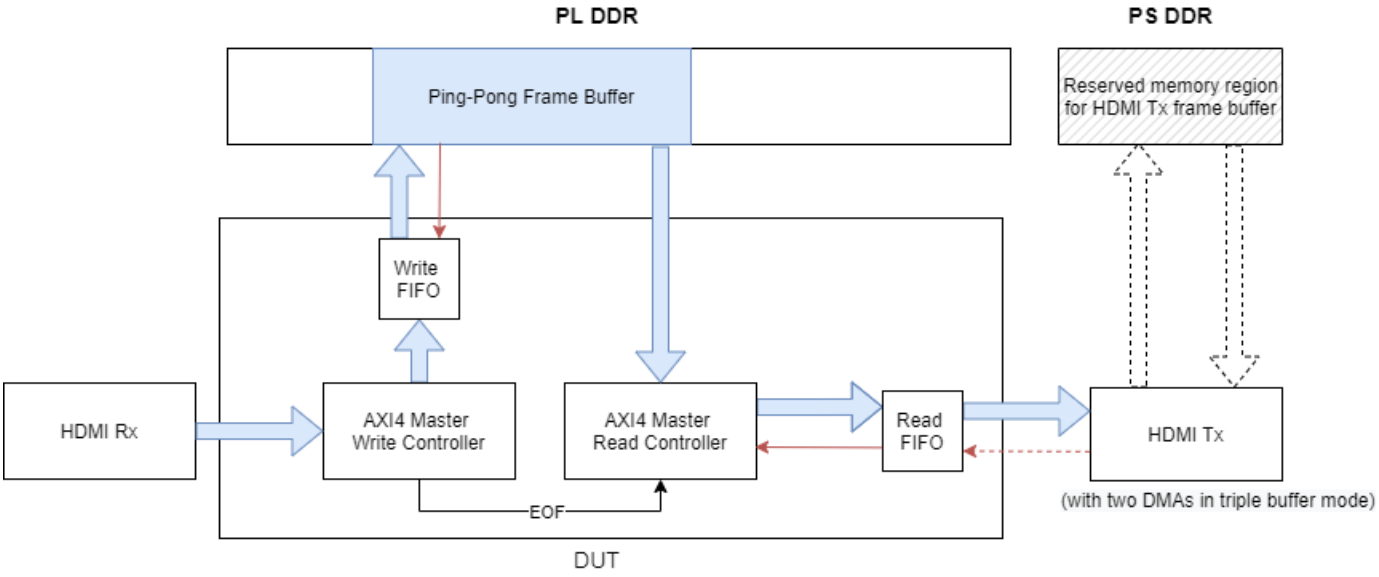
This example shows how to design an application to flip an incoming video stream vertically. You will use external memory to store a video frame to accomplish this task. You can use this technique for designing and implementing other vision applications requiring access to the external memory.

Supported hardware platform

- Xilinx® Zynq® ZC706 evaluation kit + FMC-HDMI-CAM mezzanine card

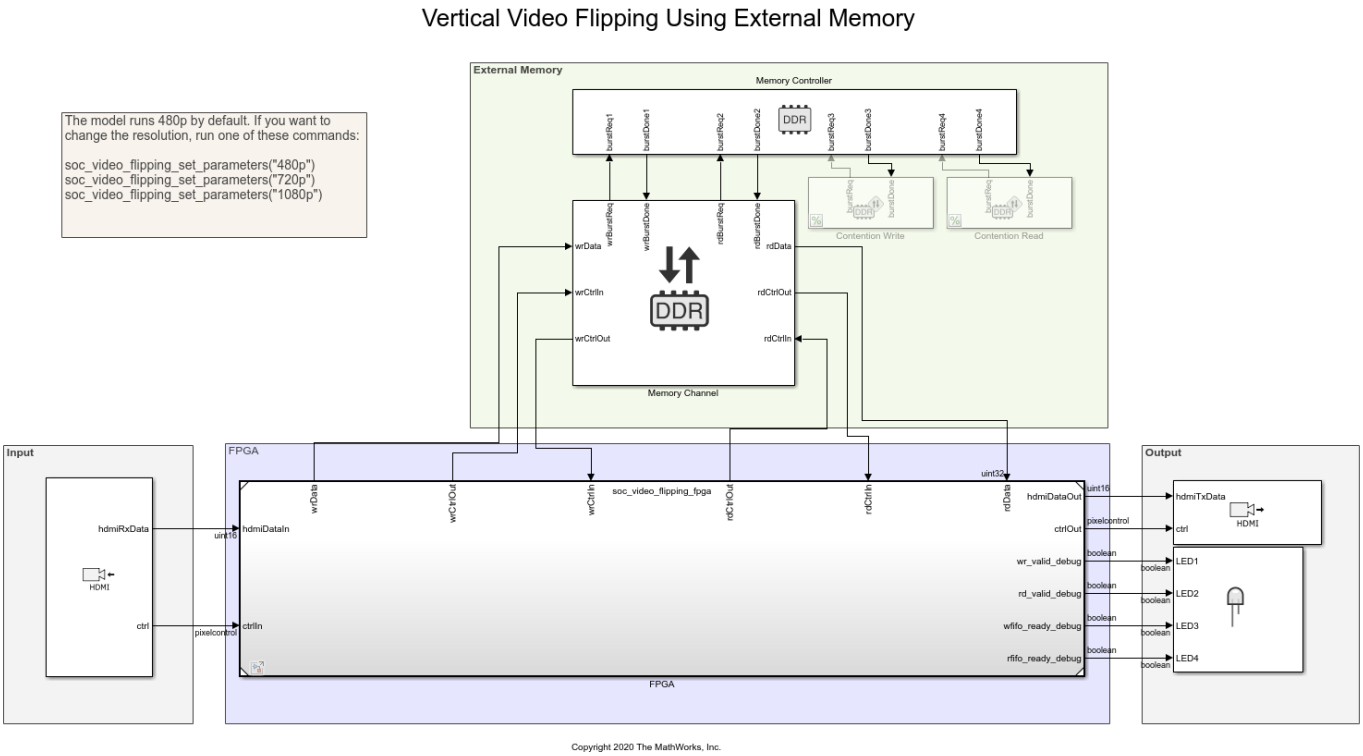
Introduction

To flip the incoming video stream from HDMI source (RX), the FPGA logic writes the video frames a line at a time to the external memory. Later, the FPGA logic reads the stored image back, a line at a time in the reverse order, thus flipping the image vertically. The read video frame is then sent to the HDMI Out (TX). Video frames are stored in a ping-pong buffer in the PL-DDR, which enables independent memory write and read operations. A separate, PS-DDR is used for storing video frames during transfer of data to the HDMI Out. The diagram below highlights the overall data flow.

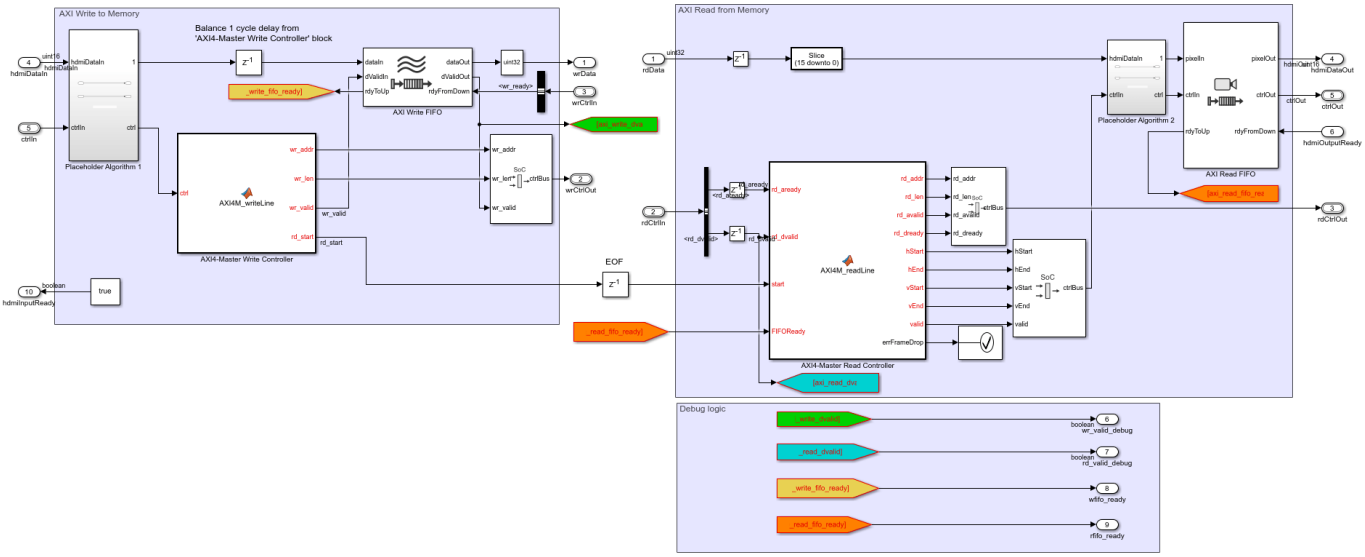


Modeling

In the top model, soc_video_flipping_top the FPGA logic is connected with the external memory and the HDMI In/Out blocks.



The soc_video_flipping_fpga model includes the FPGA logic. It is linked as a referenced model from the top model. This image shows the contents of the VideoFlipping subsystem inside the soc_video_flipping_fpga model.



The FPGA logic consists of four key components:

- **AXI4 Master Write Controller** receives video from HDMI Rx and writes the data into DDR. One line of data is written per burst. When one frame finishes, it sends the end of frame (EOF) signal to trigger AXI4 Master Read Controller.

- **AXI4 Master Read Controller** reads the lines of the video frame from the external memory ping-pong buffer in reverse order. One line of data is read per burst. New read requests are paused if the downstream Read FIFO block is not ready to process data. When a frame is fully read from memory, the Read Controller waits for the next EOF signal from the Write Controller to start reading in a new frame. If the memory controller doesn't have enough bandwidth, the read controller may be still processing the earlier frame when the write controller finishes writing in the next frame. In this case, the Read Controller will throw an error using errFrameDrop signal.
- **Write FIFO** buffers the data written to the ping-pong buffer when the DDR memory controller asserts the backpressure signal (highlighted red arrow), to allow video data from HDMI source to be processed. The Write FIFO should be large enough to prevent overflow and accommodate any delay in writing to the external memory. In this example the depth of the Write FIFO is set to 2048 to accommodate 1 HD line of backpressure.
- **Read FIFO** buffers the data from the Read Controller when the DMA in the HDMI Tx asserts the backpressure signal (highlighted red arrow). The backpressure is propagated to the upstream AXI4 Master Read Controller to stop requesting data from DDR. Notice that since the AXI4 Master Read Controller will not pause during the read burst, it is important to make sure that the Read FIFO has enough room to store data even after its ready signal de-asserts. In this example, the depth of the Read FIFO is set to 2048 and the almost full threshold is set to 128. When the FIFO has 128 samples, the Read FIFO sends 'Full' signal to upstream block to stop any new read requests. In the meanwhile, it can buffer 2048-128 = 1920 samples without an overflow. The setting is sufficient even for a 1080p frame.

In the hardware implementation, the HDMI Tx includes two DMA frame buffers and one video timing controller (VTC), for robust and tear free video output. The DMAs may send backpressure to DUT when the memory controller is busy with other read or write transactions. In the FPGA model, the backpressure signal `hdmiOutputReady` is set to always true for simulation only (which indicates that the memory controller is always available). In practice, this signal often toggles between high and low. The Read FIFO block in the DUT is used to handle this backpressure.

Simulation

The memory bandwidth requirement must be considered when designing an application that interfaces with external memory. What is the rate that you need to transfer data to/from memory to satisfy the requirements of your algorithm? Specifically, for vision applications, what is the frame-size and frame-rate that you must be able to maintain?

For the selected ZC706 board, PL DDR controller is configured with 64-bit AXI4-Slave interface running at 200 MHz. The resulting bandwidth is 1600 MB/s. Let's first evaluate if the memory bandwidth is sufficient to maintain a 1920x1080p video stream at 60 frames-per-second. As the video format is YCbCr 4:2:2, we require 2 bytes-per-pixel. However, for the DUT AXI4 read and write, each pixel is zero-padded to 4 bytes, this equates to a throughput requirement of

$$2 \times 4 \times 1920 \times 1080 \times 60 = 995.328\,\mathrm{MB/s}$$

The calculated throughput satisfies the bandwidth requirement.

To simulate 1080p 60fps case, run the following command and then simulate the model

```
soc_video_flipping_set_parameters("1080p")
```

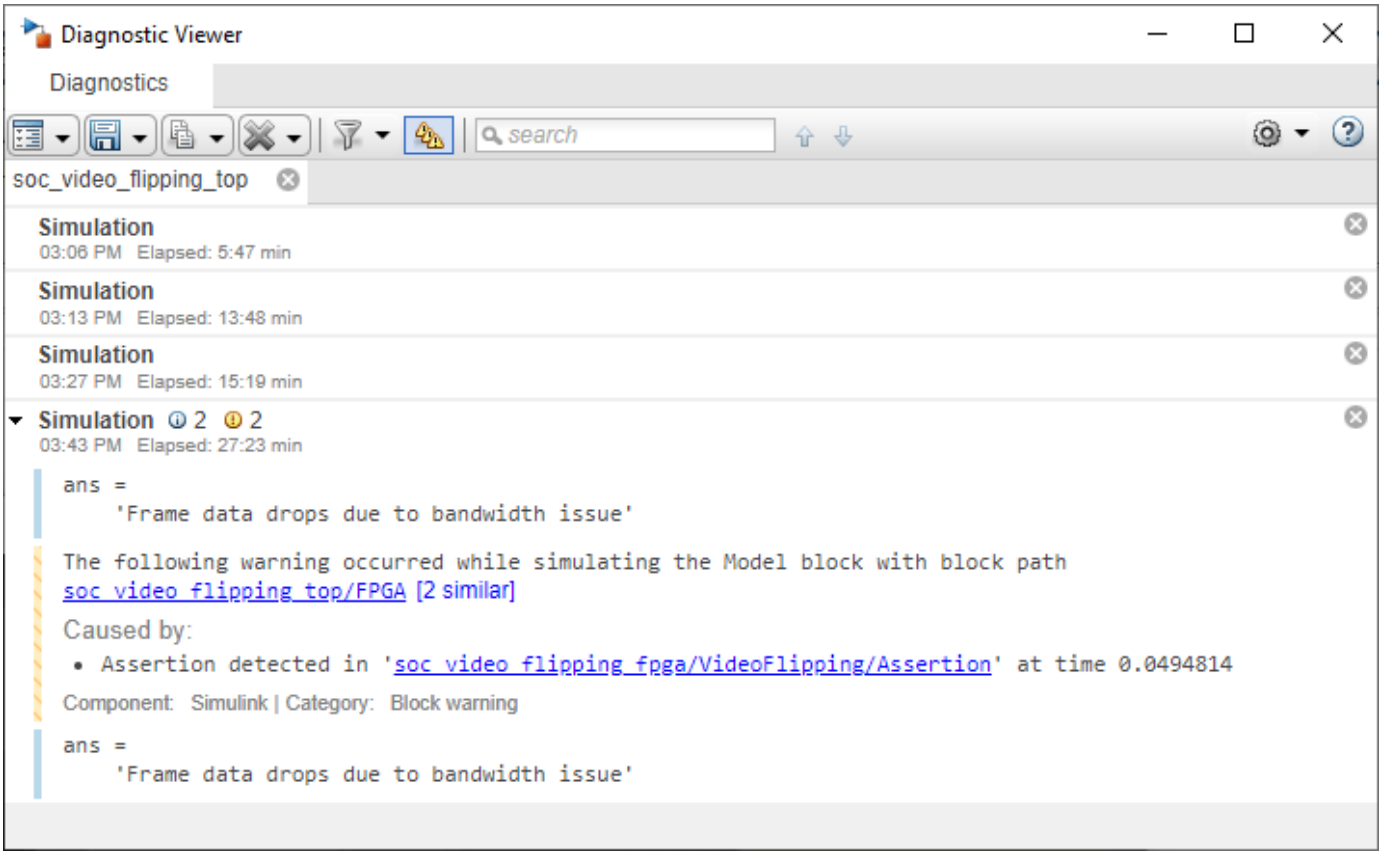
The output is shown as below.

If you want to model another DUT accessing the same external memory, you can use memory traffic generator block to simulate the contention before the implementation, so you can save effort on hardware debugging. In this model, two memory traffic generator blocks, `Contention Write` and `Contention Read` block, are modeled to mimic the AXI transactions of another frame buffer. The throughput of two memory traffic generators is calculated as

$$2 \times 2 \times 1920 \times 1080 \times 60 = 497.664\,\mathrm{MB/s}$$

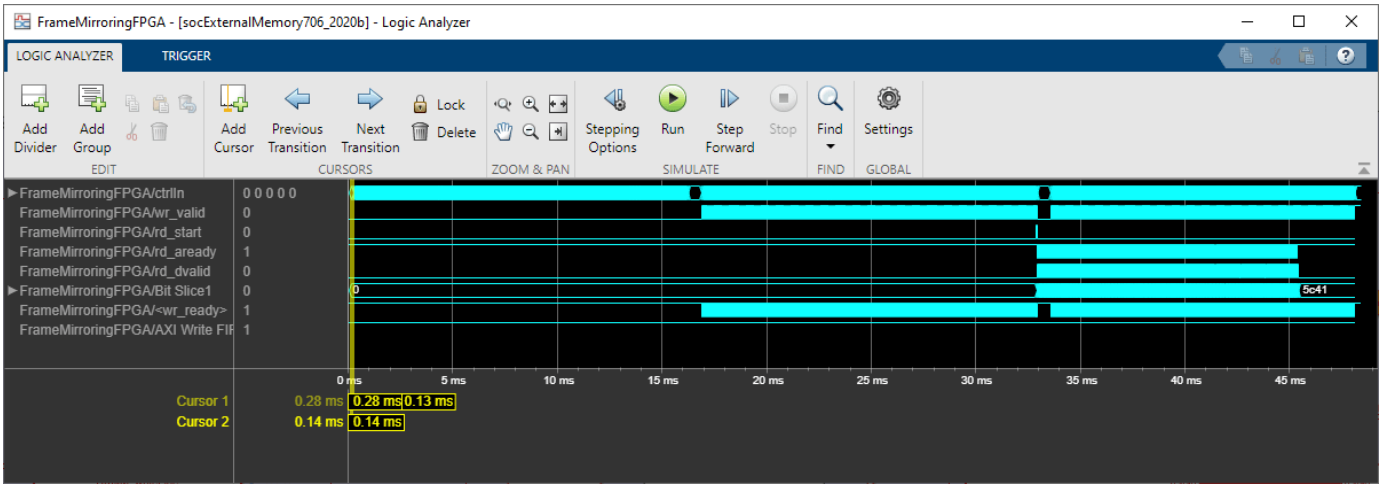
The total required bandwidth of memory controller is $497.664 + 995.328 = 1492.992\,\mathrm{MB/s}$ Which is less than its maximum bandwidth 1600 MB/s. Uncomment Contention Write and Contention Read block, simulate the model for 1080P again, the output is teared at the bottom. You will also get assertion by 'soc_video_flipping_fpga/VideoFlipping/Assertion' block that frame get dropped.


Test Sink output



Simulation shows that memory controller can not meet the bandwidth requirement for the DUT and memory traffic generators. You may need to consider reducing the frame resolution, or making DUT algorithm more efficient, for example implementing pack/unpack 16bit YCbCr pixel to 32bit, other than zero padding.

The top model runs in Accelerator mode by default. If you want to inspect the logged data in Logic Analyzer, change the top model to Normal mode. It is best to simulate with 480p or smaller frame size for faster results.



Implementation

Following products are required for this section:

- HDL Coder™

Before implementation,

- Set the simulation mode to 'Normal' on the top model.
- Comment out `Contention Write` and `Contention Read` blocks.
- Run this command if the model has been changed for the frame size other than 1080p.

```
soc_video_flipping_set_parameters("1080p")
```

To implement the model on a supported SoC board use the SoC Builder tool. Make sure you have installed required products and FPGA vendor software before implementation. To open SoC Builder click, **Configure, Build, & Deploy** button in the toolstrip and follow these steps:

1. Select 'Build Model' on 'Setup' screen. Click 'Next'.
2. Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'
3. Specify project folder on 'Select Project Folder' screen. Click 'Next'.
4. Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
5. Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
6. Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
7. Click 'Test Connection' on the 'Connect Hardware' screen to test the connectivity of the host computer with SoC board. Click 'Next' to go to the 'Run Application' screen.

The FPGA synthesis can take more than 30 minutes to complete. To save time, you can use the provided pregenerated bitstream by following these steps.

1. Close the external shell to terminate synthesis.
2. Copy pregenerated bitstream to your project folder by running this `copyfile` command below.
3. Click 'Load and Run' to load the pregenerated bitstream.

```
copyfile(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','soc','supportpackages
```

Four LEDs on the ZC706 are driven by signals and can be used for debugging the design:

- GPIO_LED_LEFT is driven by AXI4 Master write data valid. It should be on or blinking when the application is running.
- GPIO_LED_CENTER is driven by AXI4 Master read data valid. It should be on or blinking when the application is running.
- GPIO_LED_RIGHT is driven by Write FIFO ready. It should be always on, otherwise AXI4 Write data get dropped.
- GPIO_LED_0 is driven by Read FIFO ready. It could be on or off. The data won't get dropped in this FIFO because the upstream controller will handle this backpressure properly.

Conclusion

This example shows modeling of AXI4 Master interfaces for accessing external memory in random fashion using SoC Blockset. You can use this technique to model vision applications involving external memory. One such example is “Contrast Limited Adaptive Histogram Equalization with External Memory” on page 7-162 which builds further on this example.

Contrast Limited Adaptive Histogram Equalization with External Memory

This example shows how to implement the contrast-limited adaptive histogram equalization (CLAHE) algorithm for FPGA, including an external memory interface.

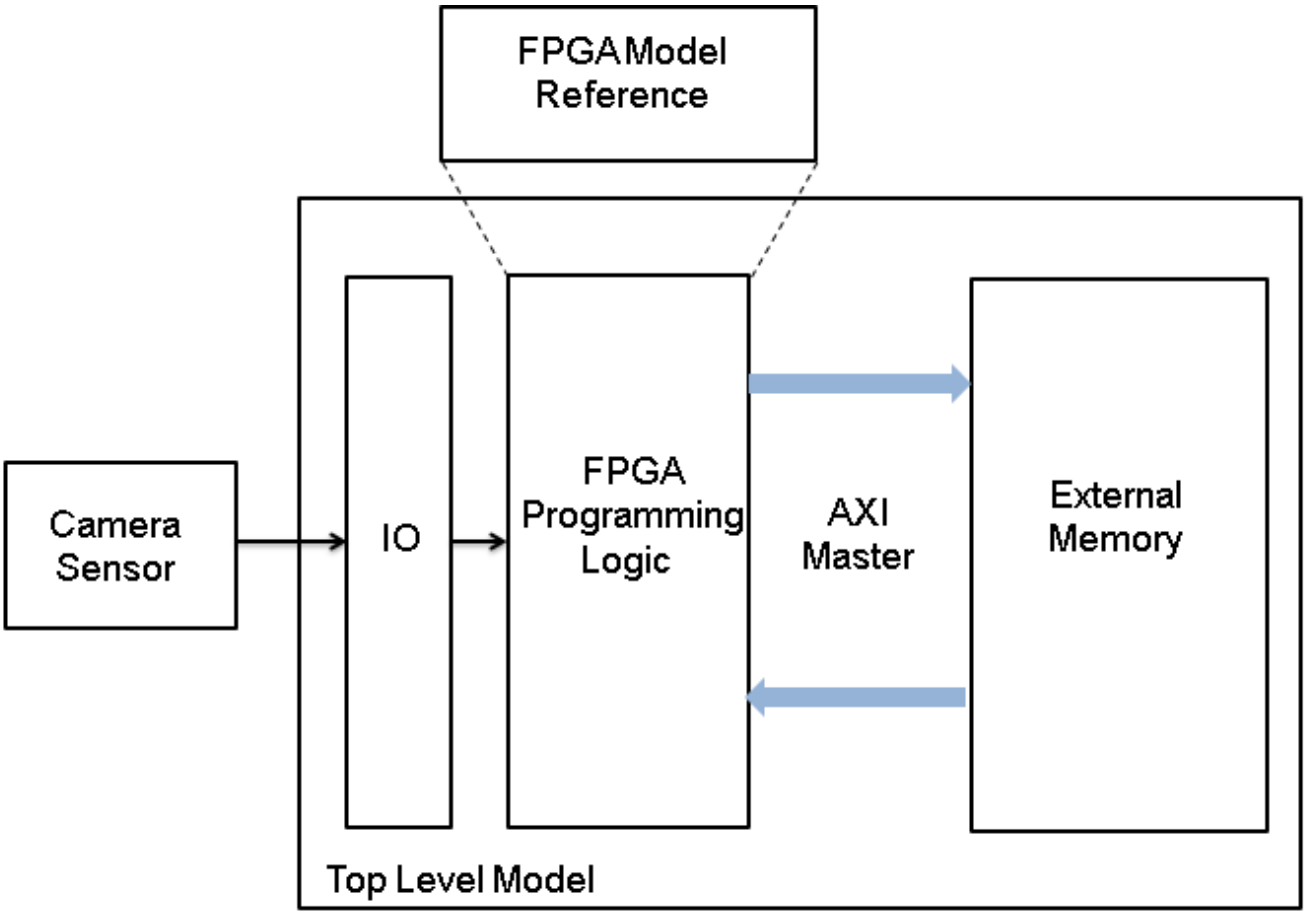
Supported Hardware

- Xilinx® Zynq® ZC706 evaluation kit + FMC-HDMI-CAM mezzanine card

Introduction

Video processing algorithms often store a full frame of video data in memory. Implementing this storage on an FPGA increases BRAM utilization and can result in input video resolution constraints. This example shows how to implement vision algorithms on FPGAs by using an external memory resource to reduce use of BRAM and enable processing of higher resolution input video.

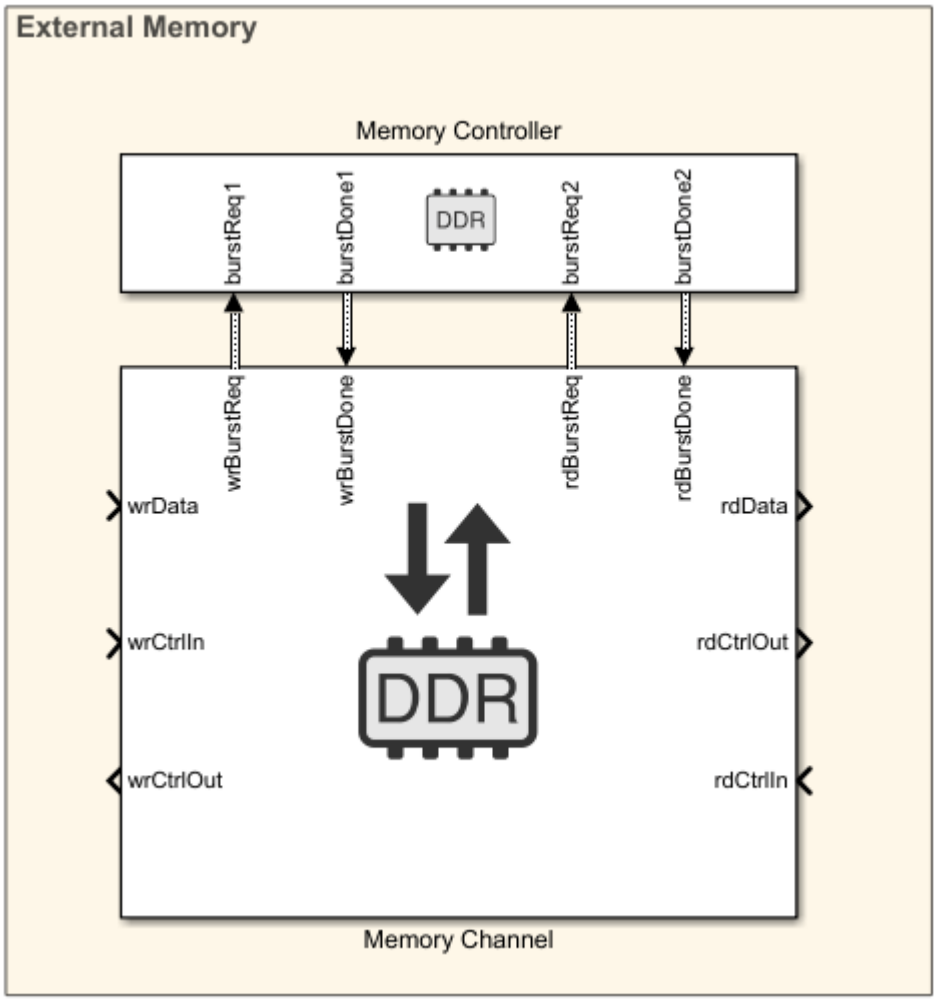
The external memory interface in this example uses AXI4 protocols and verifies the design against memory contention. The AXI4 Random Access interface provides a simple, direct interface to the memory interconnect. This protocol enables the algorithm to act as a memory master by providing the addresses and managing the burst transfer directly. The AXI4 Master Write Controller and AXI4 Master Read Controller blocks in this example model a simplified AXI-4 interface in Simulink™. When you generate HDL code using the HDL Coder™ product, the generated code includes a fully compliant AXI4 interface IP.



Model External Memory

You can use SoC Blockset™ blocks and visualization tools for modeling, simulating, and analyzing hardware and software architectures for ASICs, FPGAs, and systems on a chip (SoC). These features can help you build system architecture using memory models, bus models, and interface models and help you simulate the architecture together with the algorithms. This example models external memory using two blocks from the SoC Blockset libraries:

- Memory Channel: This block streams data through external memory. It models data transfer between the read and write master algorithms through shared memory. This example uses the AXI4 Random Access channel type.
- Memory Controller: This block arbitrates between masters and grants them unique access to shared memory. It is configured to support multiple channels with various arbitration protocols. This block also logs and displays memory performance data. This feature enables you to analyze and debug the performance of the system at simulation time.



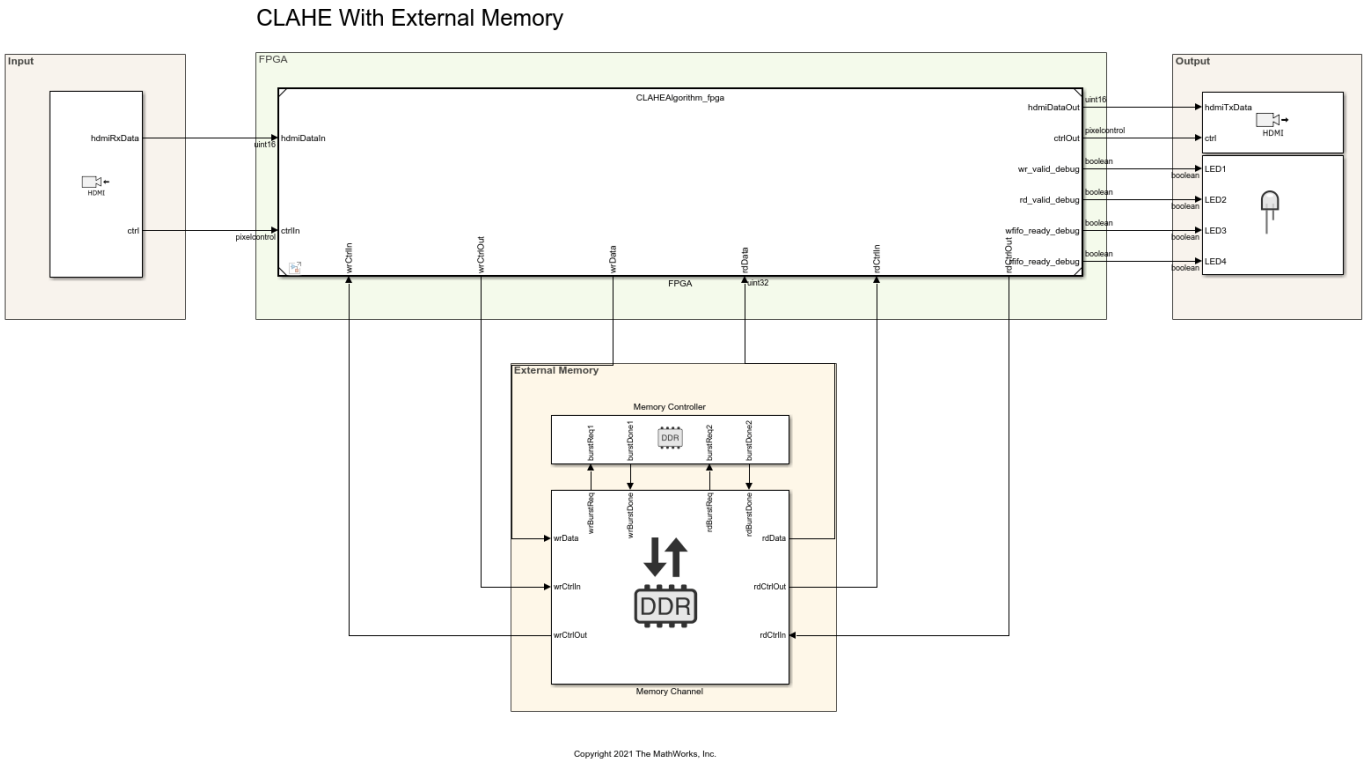
HDL Implementation

The CLAHE algorithm has three steps: tiling, histogram equalization, and bilinear interpolation. The bilinear interpolation step uses the pixel intensities from the input frame. Storing the full input frame of video data until the bilinear interpolation step requires external memory.

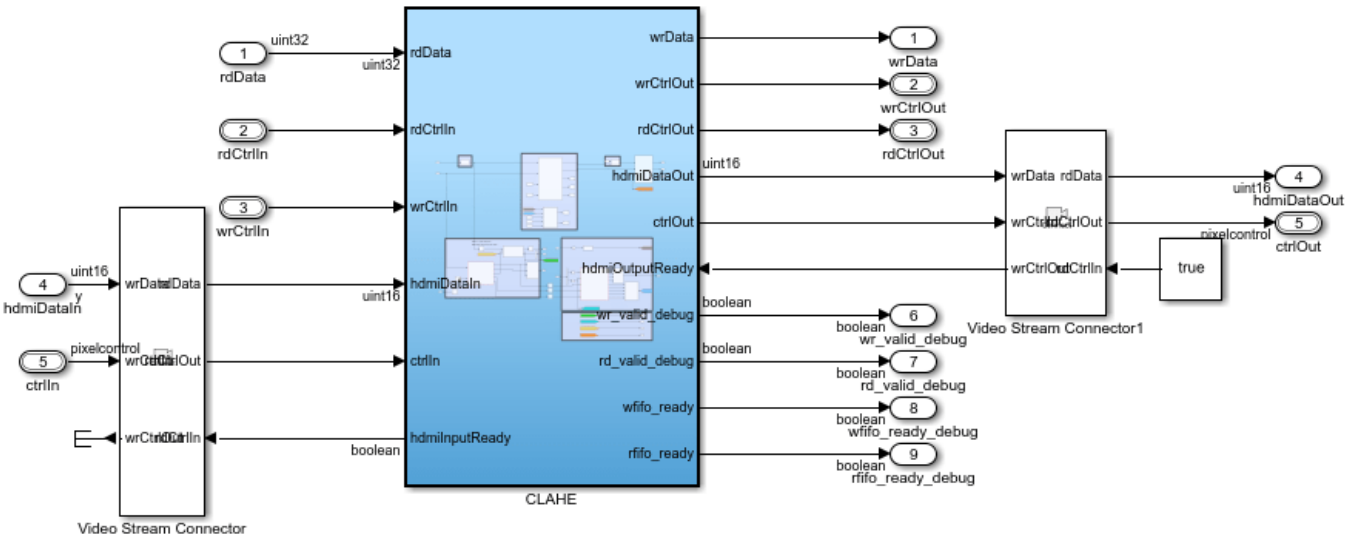
The figure shows the top level of the example model. The HDMI Rx block processes the video input and passes it to the CLAHEAlgorithm_fpga subsystem. The HDMI Rx block converts raw video data to a YCbCr 4:2:2 pixel stream format. The output data is a pixel stream suitable for hardware algorithm design. The HDMI Rx block also directs the **SoC Builder** tool to generate the IP blocks necessary to receive video data from the FMC-HDMI-CAM card that is attached to the hardware board.

In the model, the AXI4-Master Write Controller and AXI4-Master Read Controller blocks model the AXI4 memory mapped interfaces. The AXI4-Master Write Controller block writes the input frame into the external memory, and the AXI4-Master Read Controller block reads the frame from the external memory for bilinear interpolation. The AXI Read FIFO block sends the output pixel stream to the HDMI Tx block. The HDMI Tx block converts a pixel stream in YCbCr 4:2:2 format to raw video data for display during simulation. This block also directs the **SoC Builder** tool to generate the IP blocks that transmit video data back to the FMC-HDMI-CAM card. To indicate the status of the AXI Read

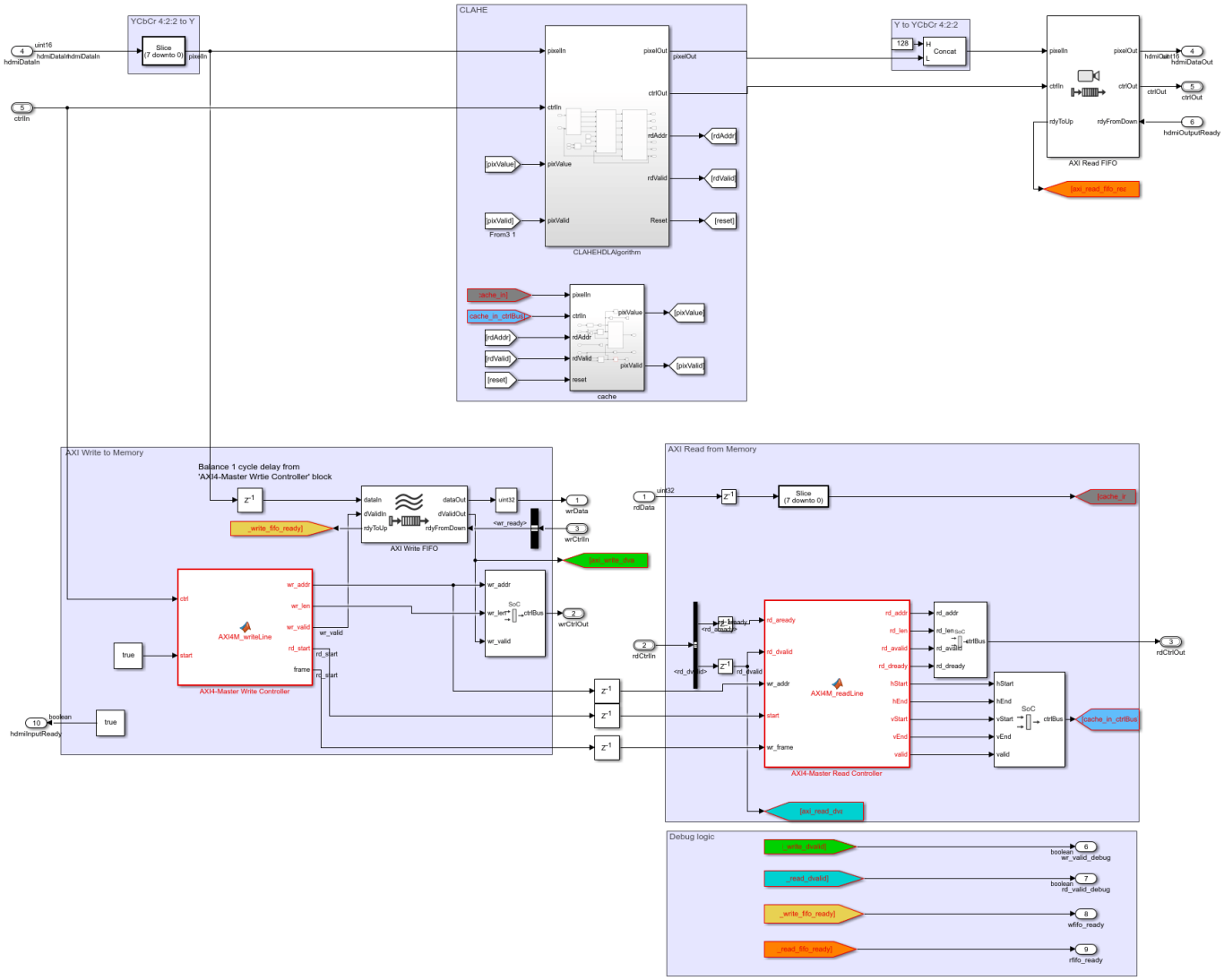
FIFO and AXI Write FIFO blocks when running the design on hardware, four debug signals from these blocks are connected to LEDs on the board.



The next figure shows the CLAHEAlgorithm_fpga reference model. The input pixel stream connects to a Video Stream Connector block. This block provides a video streaming interface to connect any two IPs in the FPGA implementation. In this example, the Video Stream Connector blocks connect the HDMI input and output blocks with the rest of the FPGA algorithm.



The next figure shows the CLAHEAlgorithm_fpga/CLAHE subsystem, which implements the AXI write and read from external memory, and the CLAHE algorithm.



The subsystem contains these areas: * AXI Write to Memory: This section writes the input data into the DDR. It consists of an AXI4 Master Write Controller block that receives the input video control information from the HDMI Rx block and models the AXI4 memory mapped interface for writing data into the DDR. It generates five signals: `wr_addr`, `wr_len`, `wr_valid`, `rd_start`, and `frame`. The `wr_valid` signal is an input to the AXI Write FIFO block, which stores the incoming pixel intensities. The SoC Bus Creator block generates the `wrCtrlOut` master to slave bus for writing the data into the DDR. The model writes one line of data per burst. After writing *tileHeight*/ 2 lines (where *tileHeight* corresponds to the height of each tile in CLAHE), the model asserts the `rd_start` signal to begin the read request. The `frame` signal indicates the input frame count.

- AXI Read from Memory: This section reads the data from the DDR. It consists of an AXI4-Master Read Controller block that receives the `rd_start` signal from the AXI4-Master Write Controller block. The AXI4-Master Read Controller block generates the `rd_addr`, `rd_len`, `rd_avalid`, and `rd_dready` signals. An SoC Bus Creator block combines these signals into a bus. The AXI4-Master Read Controller block also generates the `pixelcontrol` bus corresponding to the `rd_data`. The model slices the 32-bit `rd_data` signal to retrieve the 8-bit (LSB) luminance component and then writes it into the cache memory block of the CLAHE algorithm.

- CLAHE: For a detailed description of the implementation of the CLAHE algorithm for hardware, see the “Contrast Limited Adaptive Histogram Equalization” (Vision HDL Toolbox) example. The CLAHEHDLAlgorithm subsystem operates on 8-bit grayscale images, which is why the 8-bit luminance (Y) component is separated from the 16-bit YCbCr pixel data.

The CLAHEHDLAlgorithm subsystem performs the three steps of CLAHE: tiling, histogram equalization, and bilinear interpolation. In the first step, the input frame is divided into an 8-by-8 grid of tiles. In the second step, the histogram of each tile is calculated, and then performs distribution, redistribution, and CDF calculations. The calculated CDF values are stored in a buffer for further processing. The third step calculates the output pixel intensities by using a bilinear interpolation of the CDF values. The pixel intensities of the input frame are used as the address to the buffer that stores the CDF values. These pixel intensities are read from the external memory that stores the original input frame.

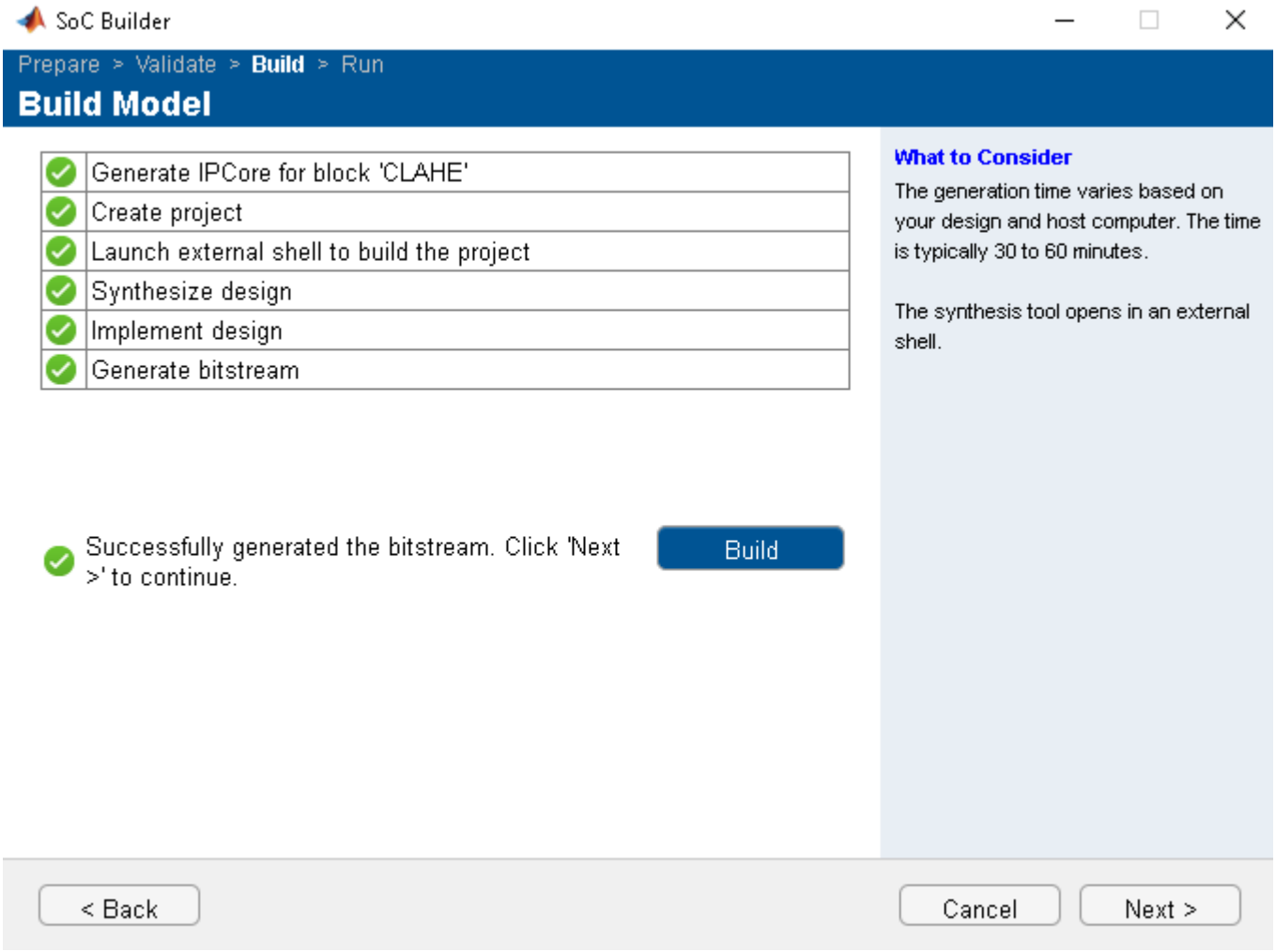
Because the data read back from the external memory is in burst mode, it cannot be used directly for bilinear interpolation. The cache buffer stores the burst of lines read from the external memory. The depth of the cache is enough to store a number of lines equal to *tileHeight*. The `rdValid` signal from the CLAHEHDLAlgorithm subsystem generates the `rd_addr` signal to read the data from the cache. The data read from the cache (`pixValue`) is then returned to the CLAHEHDLAlgorithm subsystem to complete the bilinear interpolation to calculate the output pixel intensity.

Hardware Implementation

The **SoC Builder** tool builds, loads, and executes the model on the FPGA board. The hardware board used in this example is the Xilinx® Zynq® ZC706 evaluation kit. To build, load, and execute the design on the hardware, follow these steps.

1. Set up the Vivado® tool for synthesis, implementation, and generation of the FPGA bitstream.
2. The example model runs in `Accelerator` mode by default to speed up the simulation. However, the **SoC Builder** tool requires `Normal` simulation mode. In Simulink Configuration Parameters, set **Simulation mode** to `Normal`.
3. Launch the **SoC Builder** tool by clicking **Configure, Build, & Deploy** in the Simulink Toolstrip.
4. Select **Build model**, and then review the memory map in the **Review Memory Map** pane.
5. In the **Select Project Folder** pane, specify your project folder. In the **Select Build Action** pane, select **Build, load, and run**.
6. In the **Validate Model** pane, click **Validate** to check the compatibility of the model for implementation. Then click **Build** to begin building the model. When FPGA synthesis starts ,an external shell opens.
7. When the bitstream generation is complete, in the **Connect Hardware** pane, select **Test Connection** to test the connection between the host computer and the hardware board. Load the bitstream on the hardware by clicking **Load**.

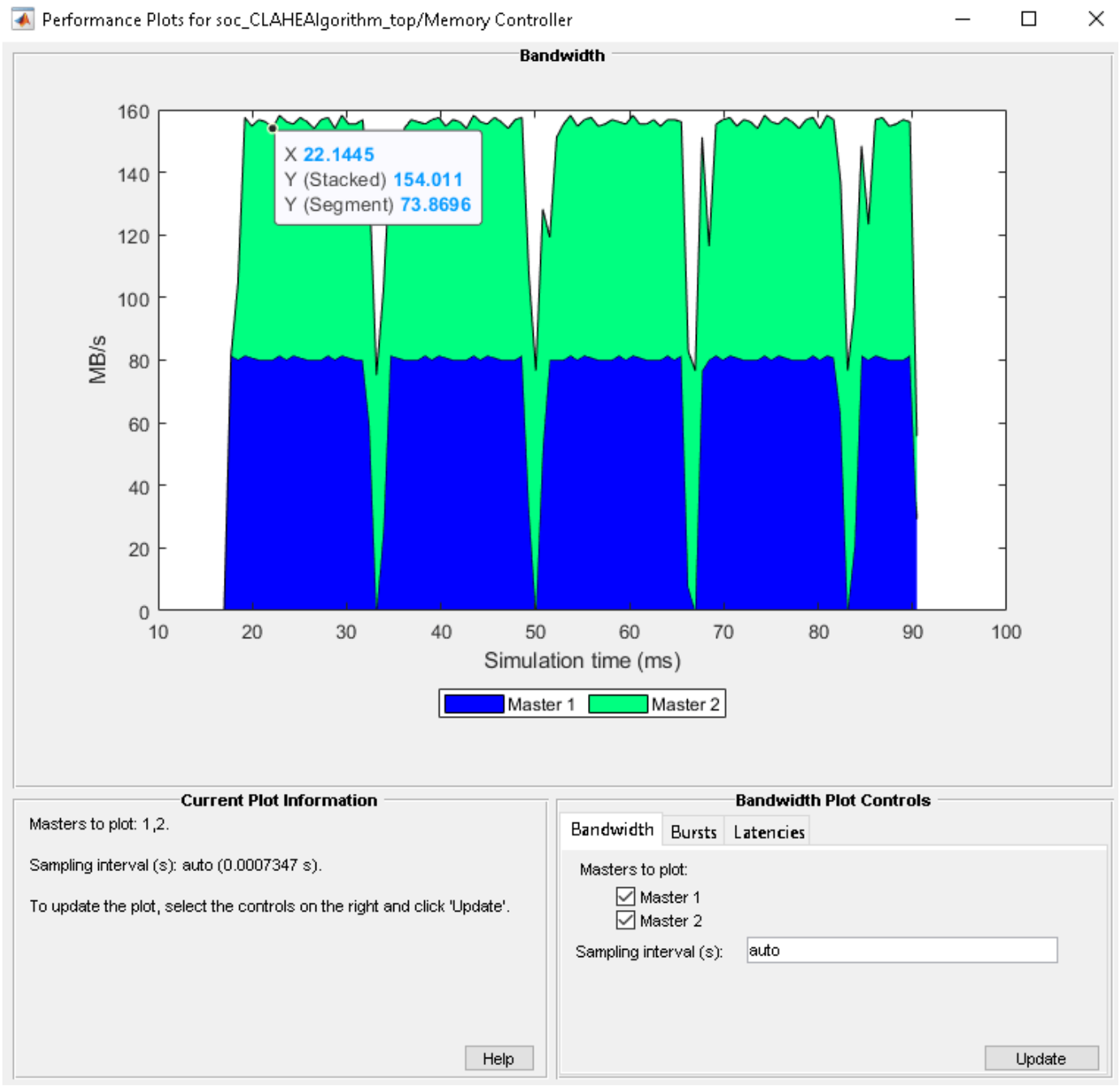
This figure shows the final **SoC Builder** results after these steps are complete.



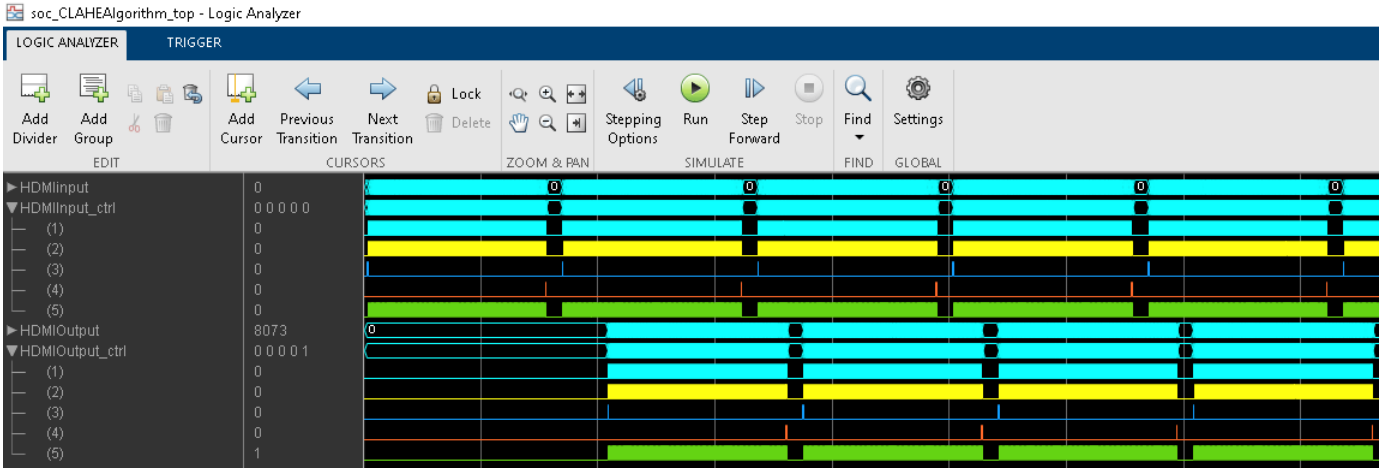
Simulation and Results

This example uses an input video of size 480-by-640 pixels. This size is configured in the HDMI Rx block. For the Xilinx Zynq ZC706 evaluation kit, the PL DDR controller is configured with a 64-bit AXI4-Slave interface running at 200 MHz. The resulting bandwidth is 1600 MB/s. This example has two AXI masters connected to the DDR controller. These AXI masters are the DUT AXI4 read and write interfaces. The YCbCr 4:2:2 video format requires 2 bytes per pixel. For the DUT AXI4 read and write interfaces, each pixel is zero-padded to 4 bytes. In this case, the read and write interfaces have a throughput requirement of 2*4*480*640*60 = 147.456 MB/s.

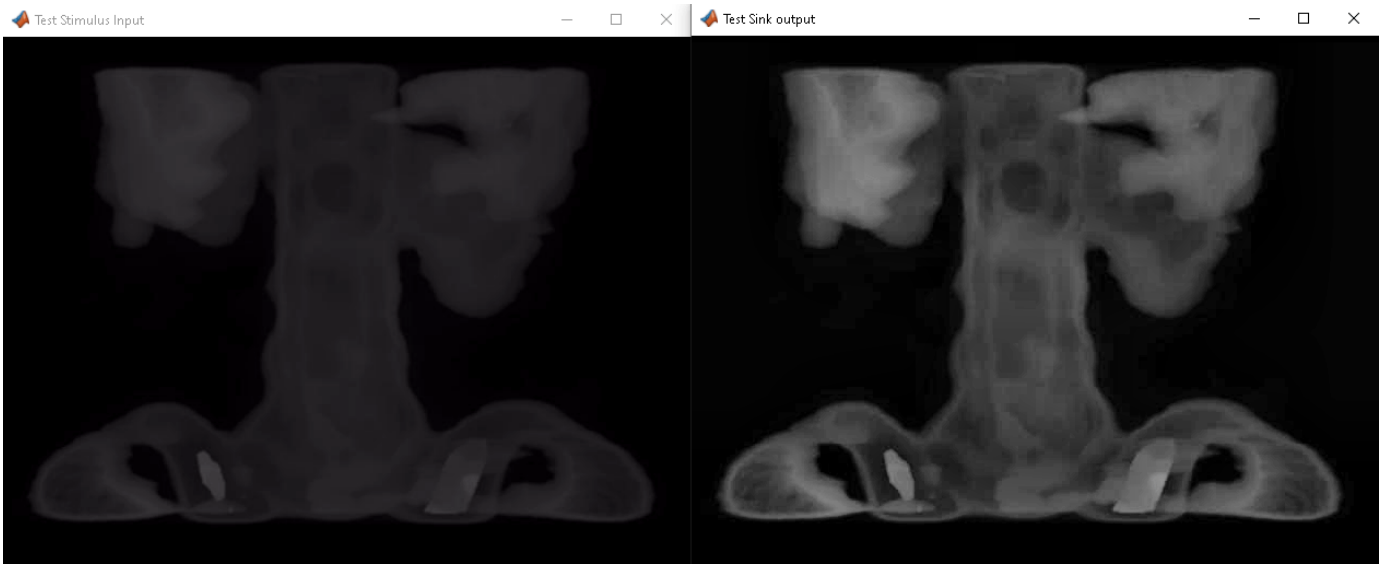
This figure shows the performance plot of the Memory Controller block. To view the performance plot, first open the Memory Controller block. Then, on the **Performance** tab, click **View performance plots**. Select all masters under **Bandwidth**, and then click **Update**. After the DUT starts writing and reading data into external memory, the throughput remains around 154 MB/s, which is within the required throughput of 147.456 MB/s.



The signals in the example model are logged during simulation. View these signals by using the **Logic Analyzer** app. This figure shows the logged data of input and output frames.



This figure shows the input and output frames from the model. The result shows the improved contrast in the output image.

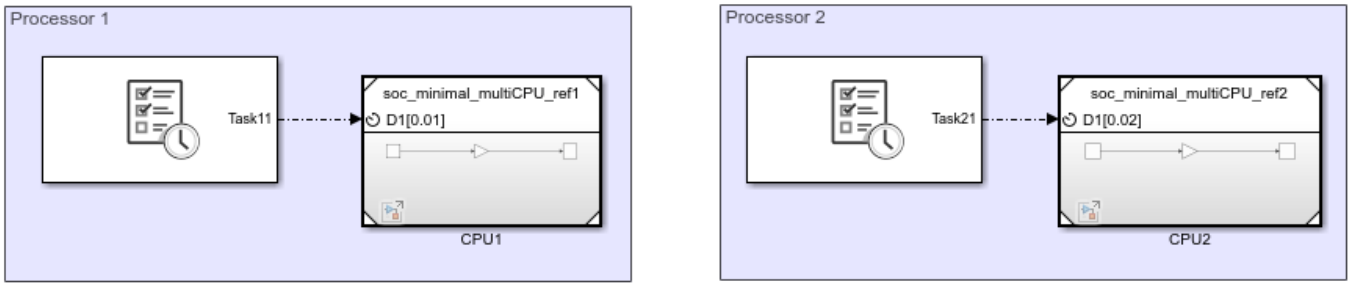


References

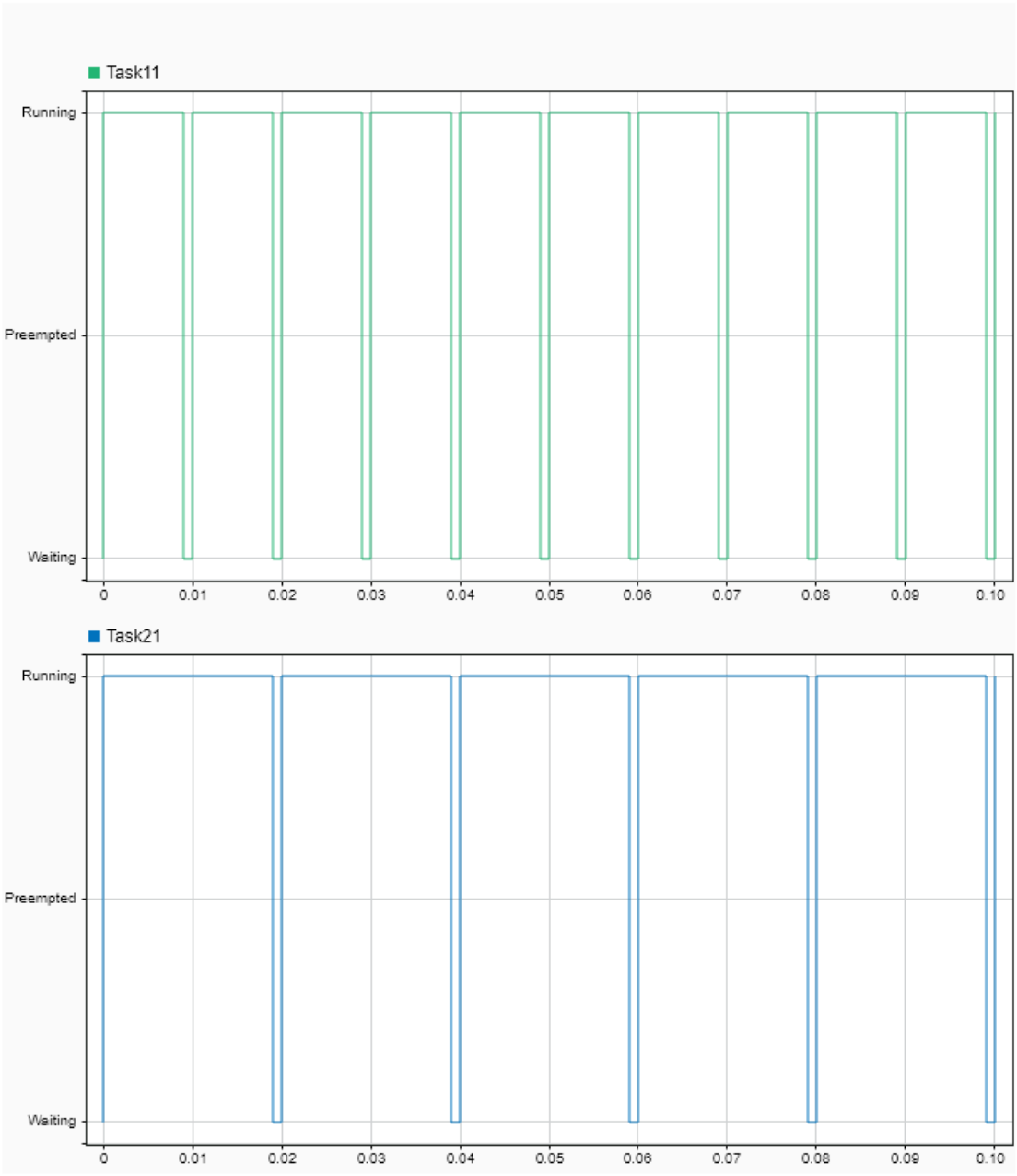
[1] Zuiderveld, Karel. "Contrast Limited Adaptive Histogram Equalization." In Graphics Gems IV, edited by Paul S. Heckbert, 474-485. AP Professional, 1994.

Multiprocessor Sample Model

This example shows a minimal multiprocessor model representing an TI Delfino F2837xD hardware board that contains a pair of C28x architectures processors in the same microcontroller die.



Each reference model, driven by the Task Manager, contains a free running counter and gain. The first model, `soc_minimal_multiCPU_ref1`, runs a timer task with a period of 0.01 and median task duration of 0.008. The second model, `soc_minimalCPU_ref2`, runs a timer driven task with a period of 0.02 and median task duration of 0.018. To run the simulation, on the Simulation tab, click Run.

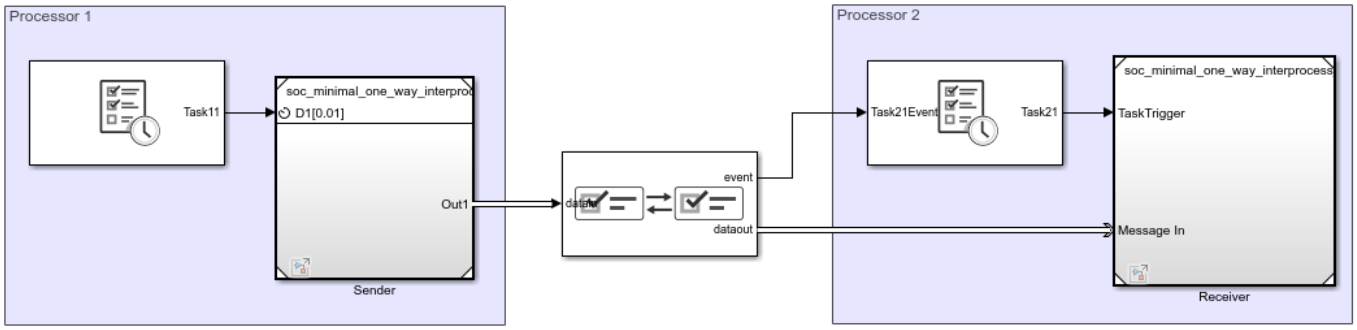


Inspecting the execution timing of the two tasks, `Task11` and `Task21`, shows that each task executes independently of the other, simulating the expected behavior of the multiprocessor TI Delfino F2837xD device.

One Way Interprocess Communication

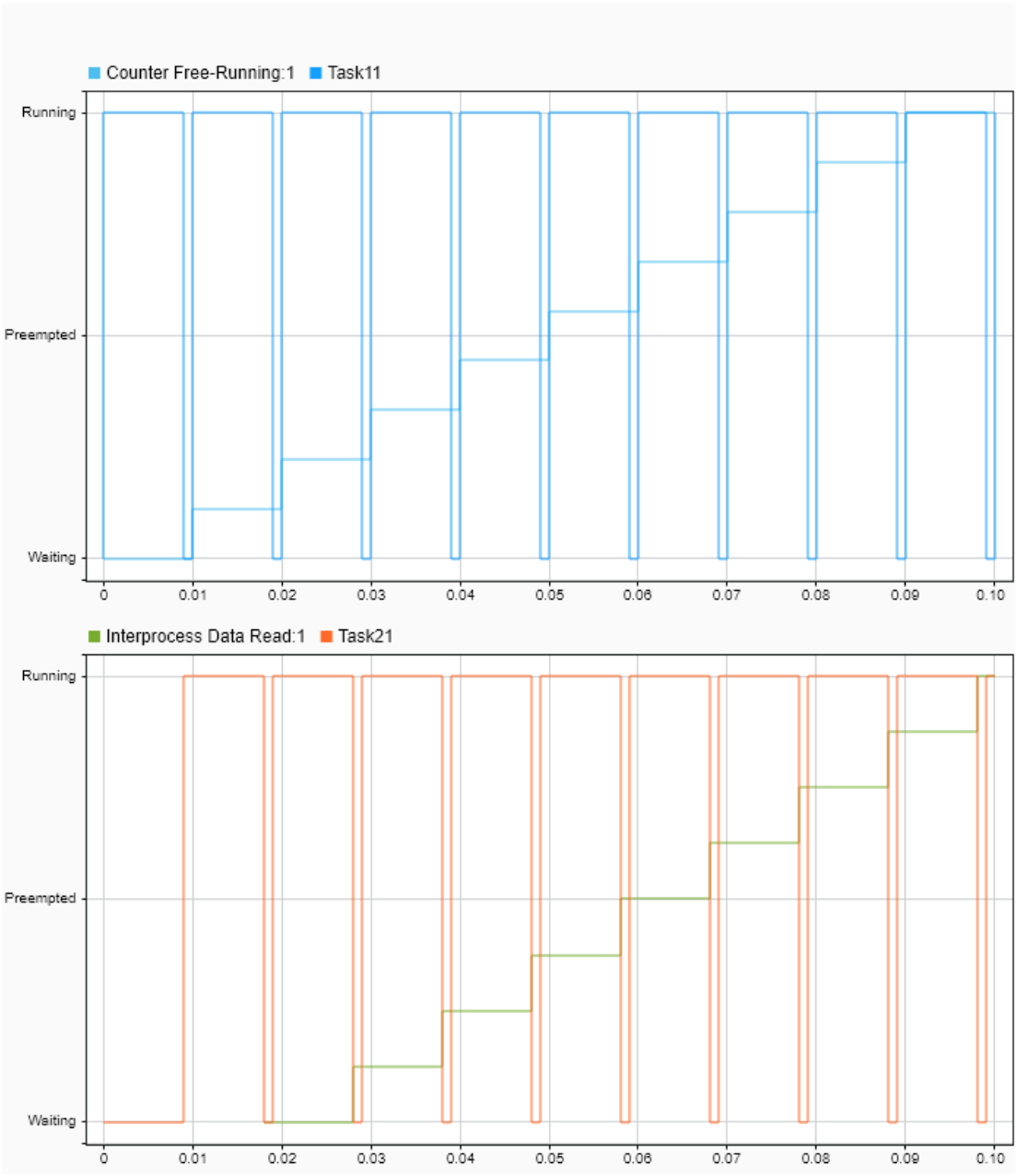
This example shows one-way interprocess data communication between two bare metal processors.

An algorithm in Processor1 sends a data message, using the Interprocess Data Write block, to the Interprocess Data Channel block at a 0.01 second interval. Processor2 two receives and processes the data messages asynchronously, using the Interprocess Data Read block.



Results

In the Simulation tab, click Run. When the simulation completes, open the Simulation Data Inspector to view the resulting signals and tasks. From the graphs, Processor1 sends the data value at the completion of the first task, Task11, instance. The data then gets received by Processor2, triggering the event driven task, Task21. At the completion of Task21 instance, the final value gets emmitted in Processor2, potentially for additional processing by other tasks.

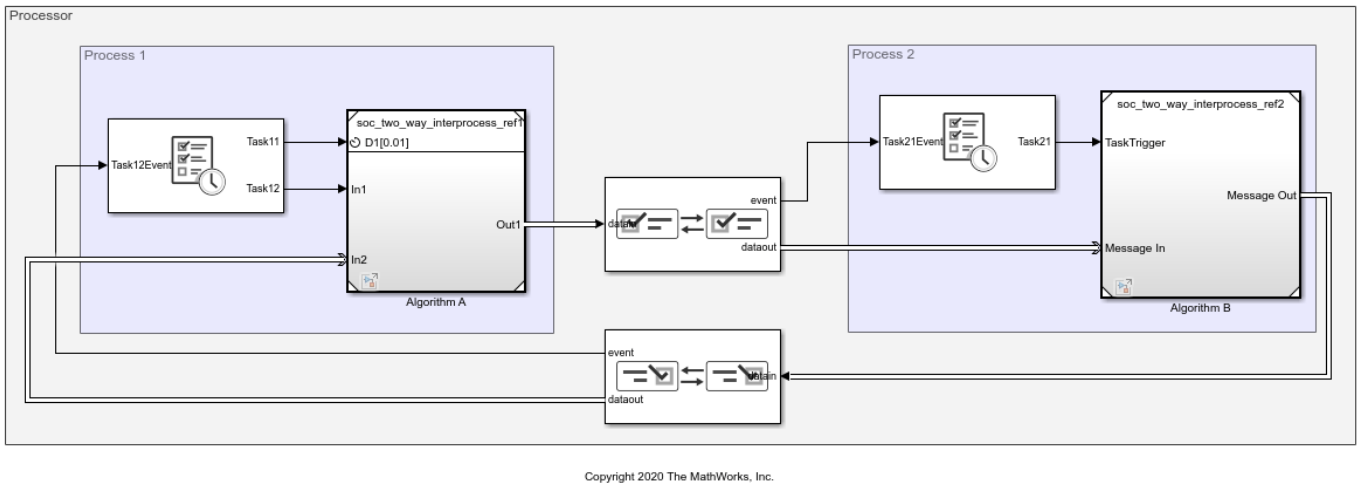

Counter Free-Running:1
Task11
Running
Preempted
Waiting
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10
Interprocess Data Read:1
Task21
Running
Preempted
Waiting
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10

Two Way Interprocess Communication

This example shows how to two-way interprocess data communication between two processors running in an operating system managed processor.

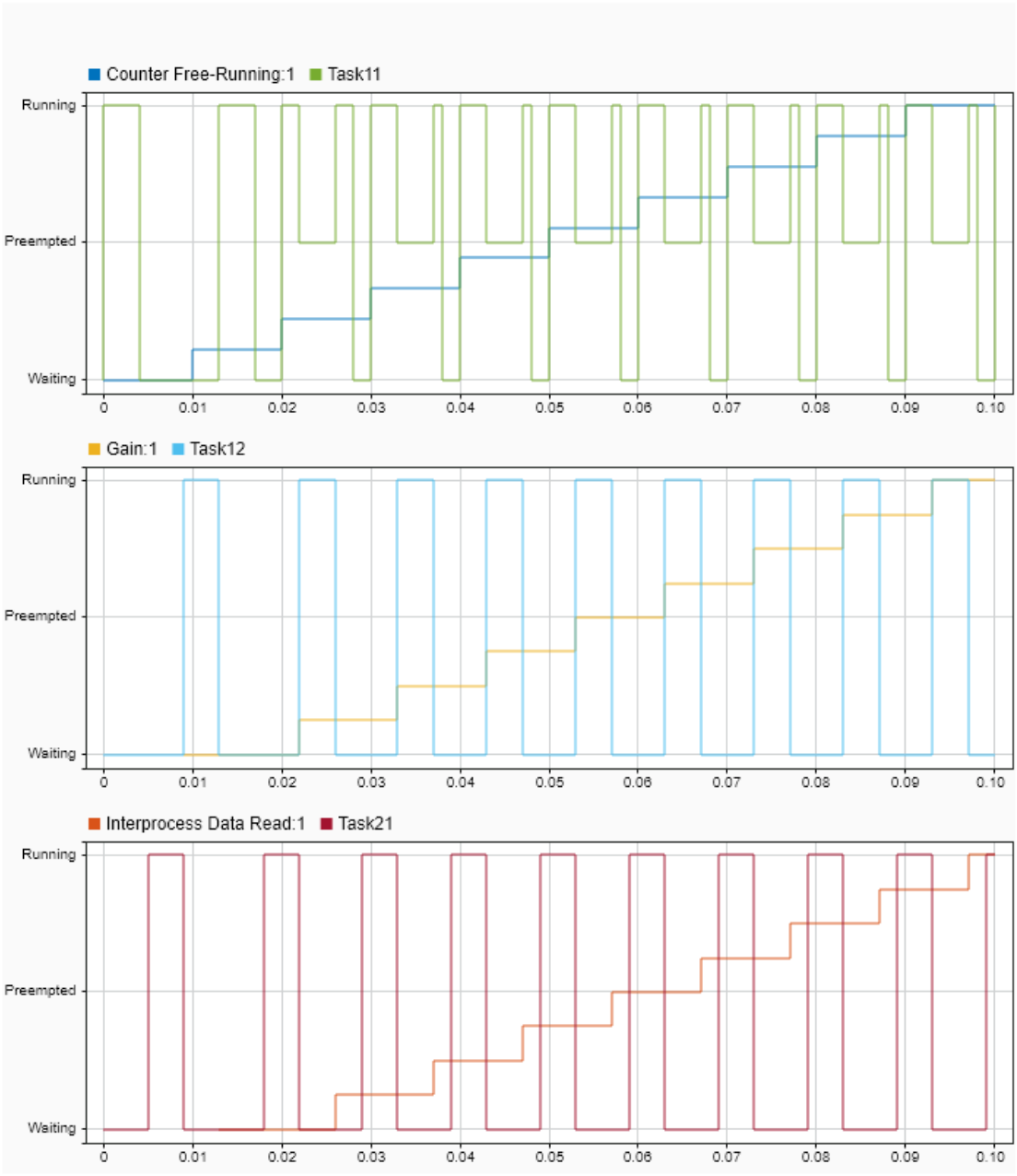
Model

Process1 sends a data message, using the Interprocess Data Write block, to the Interprocess Data Channel block at a 0.01 second intervals. Process2 receives and processes the data messages asynchronously, using the Interprocess Data Read block. The processed data returns to the first process and is received by an asynchronous task.



Results

In the **Simulation** tab, click **Run**. When the simulation completes, open the Simulation Data Inspector to view the resulting signals and task executions. Process1 sends the data packet at the completion of Task11. Task21 in Process2 triggers upon receiving the event, processes the data packet, and sends the packet back to Process1. Task12 in Process1 executes immediately upon receiving the data packet, preempting Task11.


Counter Free-Running:1
Task11
Running
Preempted
Waiting
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10
Gain:1
Task12
Running
Preempted
Waiting
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10
Interprocess Data Read:1
Task21
Running
Preempted
Waiting
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10

Systems Engineering Approach for SoC Applications

This example shows how to design a sample signal detector application on an System on Chip (SoC) platform using a systems engineering approach. The workflow in this example maps the application functions onto the selected hardware architecture.

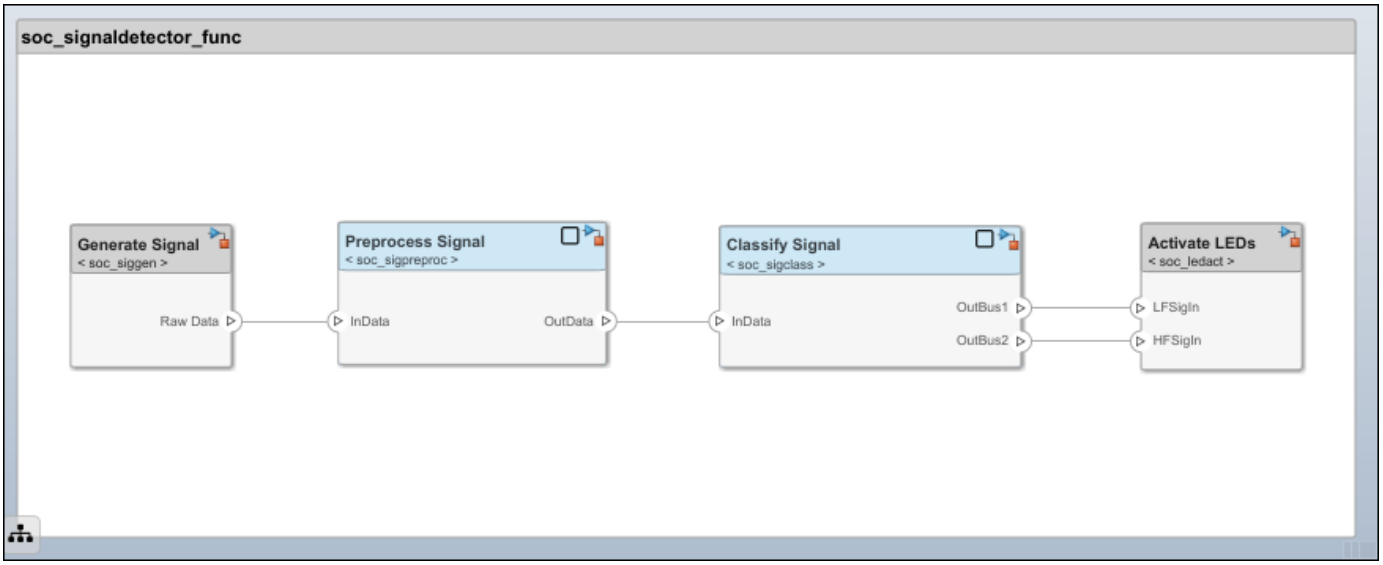
The signal detector application continuously processes the signal data and classifies the signal as either high or low frequency. The signal cannot change between high- and low-frequency classes faster than 1 ms. The signal is sampled at the rate of 10 MHz.

Functional Architecture

Define the functional architecture of the application. At this stage, the implementation of the application components is not known. You can use the System Composer™ software to capture the functional architecture.

This model represents the functional architecture with its main software components and their connections.

```
systemcomposer.openModel('soc_signaldetector_func');
```



The functional architecture of the application consists of these top-level components:

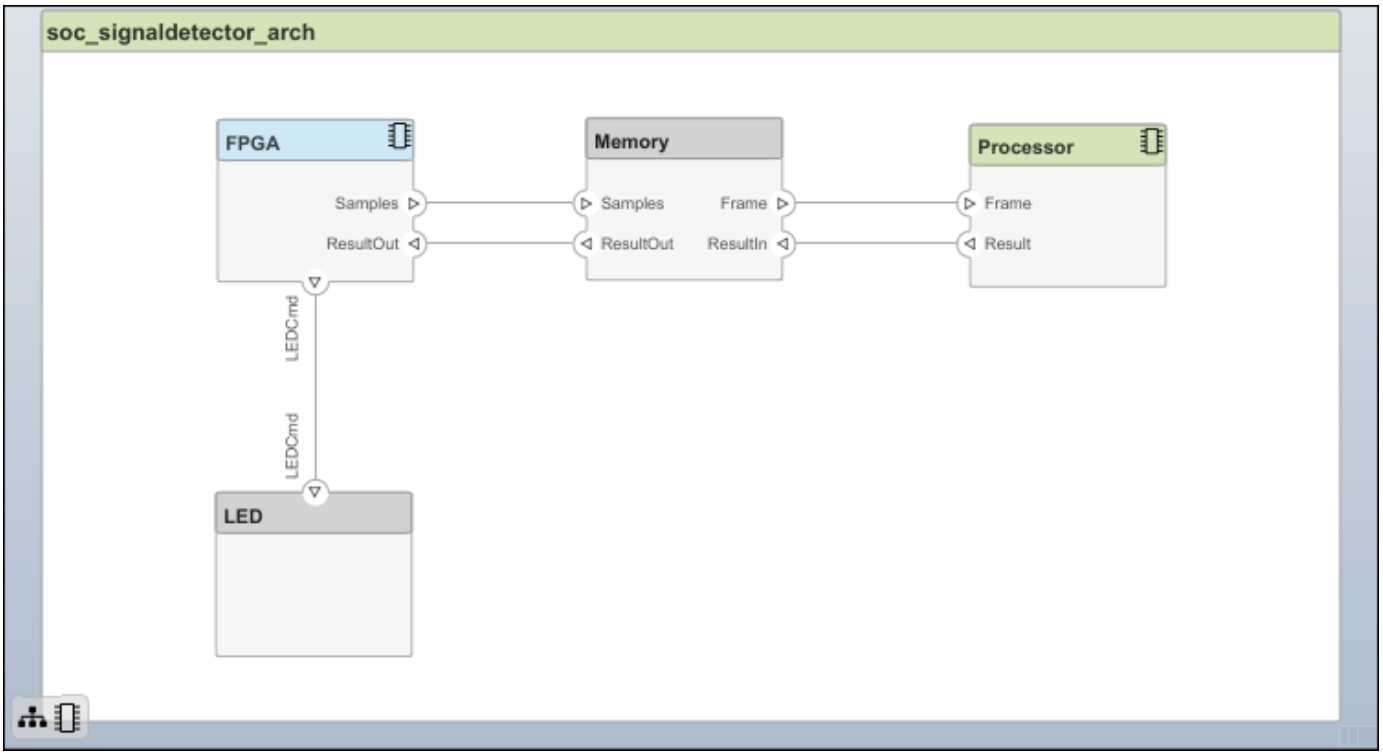
1. Generate Signal
2. Preprocess Signal
3. Classify Signal
4. Activate LEDs

Hardware Architecture

Select the hardware architecture. Due to the anticipated application complexity, choose an SoC device. The chosen SoC device has a hardware programmable logic (FPGA) core and an embedded processor (ARM) core. You can use the System Composer software to capture the details of the hardware architecture.

This model represents the hardware architecture with its main hardware components and their connections.

```
systemcomposer.openModel('soc_signaldetector_arch');
```



Behavioral Modeling

If the implementations for functional components are available, you can add them to the functional architecture as behaviors. In System Composer, for each functional component, you can link the implementation behaviors as Simulink® models. To review the component implementations, double-click each component in the functional architecture model.

After you define the behavior of each component, you can simulate the behavior of the entire application and verify its functional correctness. Select **Run** in the functional architecture model. Then, analyze the signals classification results in the **Simulation Data Inspector**. To change the signal type, select the `Generate Signal` component and then select the Manual Switch block. Confirm that the source signal is classified correctly.

Allocation of Functional and Hardware Elements

After refining the functional and hardware architecture, allocate different functional components to different hardware elements to meet desired system performance benchmarks. In this case, some functional components are constrained as to where in the hardware architecture they can be implemented. You must implement the `Generate Signal` and `Activate LEDs` components on the FPGA core in the chosen hardware architecture due to input output (I/O) connections. Comparatively, you can implement the `Preprocess Signal` and `Classify Signal` components on either the FPGA or on the processor core.

```
          Component  Constraint
Generate Signal        FPGA
```

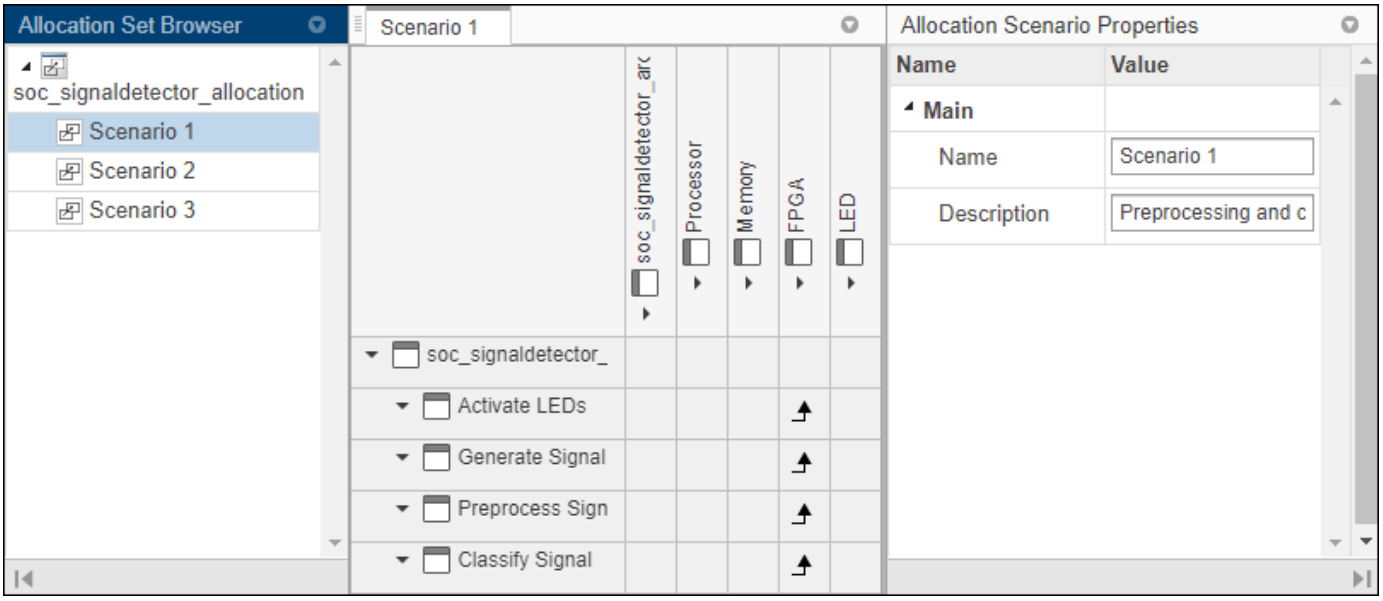
```
    Preprocess Signal           -
      Classify Signal           -
        Activate LEDs        FPGA
```

This example shows how to use three possible scenarios for allocating the application functional architecture to the hardware architecture.

- The FPGA handles preprocessing and classification.
- The FPGA handles preprocessing and the processor handles classification.
- The processor handles preprocessing and classification.

System Composer captures these scenarios as `Scenario 1`, `Scenario 2`, and `Scenario 3` using the allocation editor.

```
systemcomposer.allocation.editor
allocSet = systemcomposer.allocation.load('soc_signaldetector_allocation');
```



Choosing an allocation scenario requires finding an implementation that optimally meets the application requirements. Often you can find this implementation via static analysis without detailed simulation. In this example, use static analysis to analyze the computational costs of implementing different functional components on the processor and on the FPGA.

Implementation Cost

The implementation cost of a component depends on the required computation operations. To determine the implementation costs, consider these typical approaches.

- Component implementation is not available: Obtain the computational cost from the available reference implementations.
- The implementation and the hardware are available: Measure or profile the implementation cost on the candidate hardware.
- The implementation is available, but the hardware is not: Estimate the implementation cost by using the SoC Blockset™ algorithm analyzer function `socAlgorithmAnalyzerReport`.

The `socModelAnalyzer` function estimates the number of operations in a Simulink model and generates an algorithm analyzer report. To get the number of operations that a model executes to then analyze the implementation cost on the processor, use the dynamic analysis function option. To get the number of operators an algorithm requires to then analyze the implementation cost on the FPGA, use the static analysis function option. For an example on how to use `socModelAnalyzer`, see this sample function.

```
soc_signaldetector_costanalysis
```

```
*** Component: 'Preprocess Signal'
                                ADD(+)    MUL(*)
                                ______    ______

    FPGA Implementation            15        16
    Processor Implementation    15300     16320



*** Component: 'Classify Signal'
                                ADD(+)    MUL(*)
                                ______    ______

    FPGA Implementation            32        18
    Processor Implementation    32640     18360
```

The implementation costs for each functional component obtained in this code are entered in the corresponding stereotypes in the functional architecture. To verify the values, select each component in the functional architecture model and use the Property Inspector.

To learn more about `socModelAnalyzer`, see the “Compare FIR Filter Implementations Using socModelAnalyzer” on page 7-121 example. This example shows how to analyze the computational complexity of different implementations of a Simulink algorithm.

Allocation Choice

You can use the number of operators or operations that are required for implementing the application functional components to decide how to allocate the functional components to the hardware components. Analyze the candidate allocations by comparing the implementation cost against the available resources of the FPGA and the processor. This example uses sample values in the FPGA and the processor components in the hardware architecture model for the available computation resources. Verify the values by using the Property Inspector.

Typically, the analysis does not use the number of operators or operations directly. Rather, the number of operators or operations are multiplied by the cost of each operator or operation first. The cost of the operator or operations is hardware dependent. Determining such costs is beyond the scope of this example.

For an example on how to use the cost models, use this function. Observe that we require the capacity of the FPGA and the processor be greater than the estimated implementation cost as well as that the processor headroom be between 60 and 90 %.

```
soc_signaldetector_partitionanalysis
```

```
                    FPGA DSPs Used (out of 900)    FPGA LUT Used (out of 218600)    Processor Instr
                    ___________________________    _____________________________    _______________
```

```
Scenario 1               34                              576
Scenario 2               16                              192
Scenario 3                0                                0
```

Based on the results Scenario 2 is feasible.

Data Path Design Between FPGA and Processor

The FPGA processes data sample-by-sample, and the processor processes frame-by-frame. Because the duration of a processor task can vary, to prevent data loss, a queue is needed to hold the data between the FPGA and processor. In this case you must set these parameters that are related to the queue: frame size, number of frame buffers, and FIFO size (that is, the number of samples in the FIFO). Also, in embedded applications, the task durations can vary between different task instances (for example, due to different code execution paths or due to variations in OS switching time). As a result, data might be dropped in the memory channel. The “Streaming Data from Hardware to Software” on page 7-31 example shows a systematic approach to choosing the previously mentioned parameters that satisfy the application requirements.

See Also

`socModelAnalyzer`

Related Examples

- “Streaming Data from Hardware to Software” on page 7-31