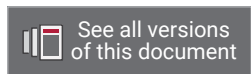


# SDSoC Environment Debugging Guide

UG1282 (v2019.1) May 22, 2019



# Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
Entire document	Editorial updates.
12/05/2018 Version 2018.3	
Entire document	Editorial updates.
01/24/2019 Version 2018.3	
Entire document	Editorial updates.
12/05/2018 Version 2018.3	
Entire document	Editorial updates.
07/02/2018 Version 2018.2	
Entire document	Editorial updates.
06/06/2018 Version 2018.2	
General updates	Initial Xilinx release.

# Table of Contents

<b>Revision History</b> .....	<b>2</b>
<b>Chapter 1: Introduction to Debugging in SDSoC</b> .....	<b>4</b>
SDSoC Environment Overview.....	4
SDSoC Debug Flow Overview.....	11
<b>Chapter 2: SDSoC Debug Features</b> .....	<b>16</b>
SDx Environment Debug Tools.....	16
System Emulation.....	25
Hardware Execution Features Available to All Platforms.....	31
Hardware/Software Event Tracing.....	40
<b>Chapter 3: Debug Techniques</b> .....	<b>45</b>
Debugging System Hangs and Runtime Errors.....	46
Peeking and Poking IP Registers.....	52
Event Tracing.....	53
Debugging with Software/Hardware Cross Probing.....	55
Tips for Debugging Performance.....	57
Troubleshooting Compile and Link Time Errors.....	58
Troubleshooting Performance Issues.....	59
<b>Appendix A: SDSoC Environment Troubleshooting</b> .....	<b>61</b>
<b>Appendix B: Additional Resources and Legal Notices</b> .....	<b>62</b>
Documentation Navigator and Design Hubs.....	62
References.....	62
Training Resources.....	63
Please Read: Important Legal Notices.....	63

# Introduction to Debugging in SDSoC

The SDSoC™ environment includes an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems. SDSoC supports Arm® Cortex™-based applications using the Zynq®-7000 SoC and Zynq® UltraScale+™ MPSoC devices, as well as MicroBlaze™ processor-based applications on all Xilinx® SoCs and FPGAs.

This user guide introduces the debugging capabilities of the SDSoC environment, and provides you with detailed instructions on how to analyze any failure encountered within the SDSoC flow.

**Note:** This user guide does *not* cover performance issues. If no tool problems are encountered, and the behavior of the design is deemed functionally correct, you can look for answers in the *SDSoC Environment Profiling and Optimization Guide (UG1235)* to examine whether the performance of the design can be further improved.

---

## SDSoC Environment Overview

The SDSoC environment includes a system compiler that transforms C/C++ programs into complete hardware/software systems with select functions compiled into the programmable logic (PL). The SDSoC system compiler analyzes a program to determine the data flow between software and hardware functions, and generates an application-specific system-on-chip (SoC) to realize the program.

To achieve high performance, each hardware function runs as an independent thread; the system compiler generates hardware and software components that ensure synchronization between hardware and software threads, while enabling pipelined computation and communication. Application code can involve many hardware functions, multiple instances of a specific hardware function, and calls to a hardware function from different parts of the program.

The SDx integrated development environment (IDE) supports software development workflows including profiling, compilation, linking, system performance analysis, and debugging. It also provides a fast performance estimation capability to enable exploration of the hardware/software interface before committing to a full hardware compile.

The SDSoC system compiler targets a base platform and invokes the Vivado® High-Level Synthesis (HLS) tool to compile synthesizable C/C++ functions into programmable logic. The system compiler then generates a complete hardware system, including DMAs, interconnects, hardware buffers, other IP, and the FPGA bitstream by invoking the Vivado Design Suite tools. To ensure that all hardware function calls preserve their original behavior, the SDSoC system compiler generates system-specific software stubs and configuration data. The program includes the function calls to drivers required to use the generated IP blocks. Application and generated software is compiled and linked using a standard GNU toolchain.

By generating complete applications from a single source, the system compiler lets you iterate over design and architecture changes by refactoring at the program level, which reduces the time needed to achieve working programs running on the target platform.

## Terminology

The following terms are widely used while designing in the SDSoC environment. The terms and their definitions are provided below.

- **Accelerator:** Portions of the application code that have been implemented in the hardware in the FPGA general interconnect. These are also called hardware functions.
- **Data Mover:** The data mover transfers data between accelerators, and between the processing system (PS) and accelerators. The SDSoC environment can generate various types of data movers based on the properties and size of the data being transferred.
- **Pipelining:** Pipelining is a technique to increase instruction-level parallelism in the hardware implementation of an algorithm by overlapping independent stages of operations or functions. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle.
- **Pragma:** Special directives that can be inserted into the source code to guide the system compiler. In the SDSoC environment, you control the system generation process by structuring hardware functions and calls to hardware functions in a way that balances communication and computation, and by inserting pragmas into your source code to guide the system compiler.
- **Processor:** Processors in the context of the SDSoC environment mean a soft processor such as a MicroBlaze processor, or a hard processor such as the Arm processors on Zynq-7000 SoCs and Zynq UltraScale+ MPSoCs.
- **System Port:** A system port connects a data mover to the PS. It can be an ACP, AFI (corresponding to high-performance ports), MIG (corresponding to a PL-based DDR memory controller), or a stream port on the Zynq.

## Elements of SDSoC

The SDSoC environment includes the following features:

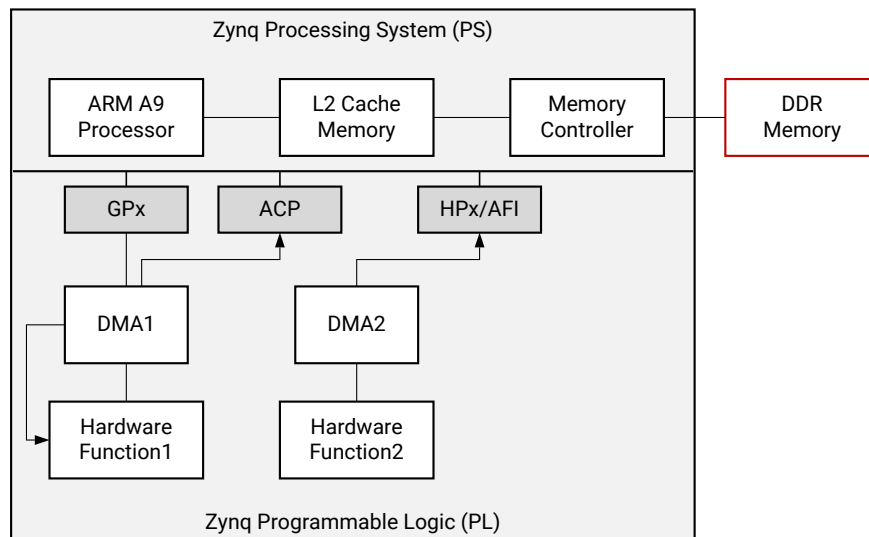
- The `sds++` system compiler, which generates complete hardware/software systems. The `sds++` system compiler employs underlying features from the Vivado Design Suite System Edition, including the Vivado High-Level Synthesis (HLS) tool, Vivado IP integrator, IP libraries for data movement and interconnect, and tools for RTL synthesis, placement, routing, and bitstream generation.
- An Eclipse-based integrated development environment (IDE) to create and manage application projects and workflows.
- A system performance estimation capability to explore different scenarios for the hardware/software interface.

The SDSoC environment also inherits many of the tools in the Xilinx Software Development Kit (SDK), including GNU toolchains for Zynq-7000 SoCs and Zynq UltraScale+ MPSoCs, standard libraries (for example, glibc), and the Target Communication Framework (TCF) for communicating with embedded processor targets. It also features a performance analysis perspective within the Eclipse/CDT-based IDE.

The `sds++` system compiler generates an application-specific system-on-chip for a targeted platform. The environment includes a number of standard base platforms for application development, and other platforms can be developed by third-party partners, or by SDSoC design teams. The *SDSoC Environment Platform Development Guide* ([UG1146](#)) describes how to create a hardware platform design in the Vivado Design Suite, configure platform interfaces, and define the corresponding software runtime environment to build a platform for use in the SDx™ IDE.

The SDx™ IDE lets you customize a target platform with application-specific hardware accelerators, and data motion networks connecting accelerators to the platform. A simplified Zynq and DDR configuration with memory access ports and hardware accelerators is shown below.

Figure 1: Simplified Zynq + DDR Diagram Showing Memory Access Ports and Memories



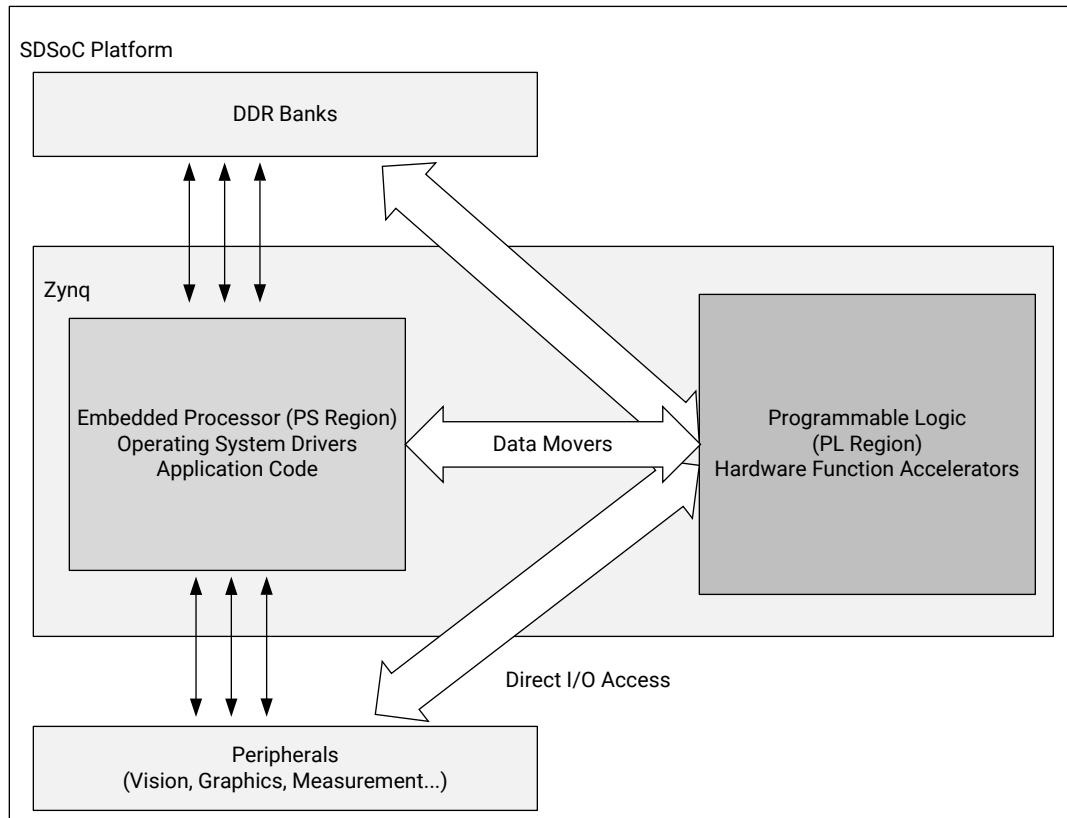
X14709-061518

## Execution Model of an SDSoC Application

The execution model for an SDSoC environment application can be understood in terms of the normal execution of a C++ program running on the target CPU after the platform has booted. It is useful to understand how a C++ binary executable interfaces to hardware.

The set of declared hardware functions within a program is compiled into hardware accelerators that are accessed with the standard C runtime through calls into these functions. Each hardware function call in effect invokes the accelerator as a task and each of the arguments to the function is transferred between the CPU and the accelerator, accessible by the program after accelerator task completion. Data transfers between memory and accelerators are accomplished through data movers, such as a DMA engine, automatically inserted into the system by the `sds++` system compiler taking into account user data mover pragmas such as `zero_copy`.

Figure 2: Architecture of an SDSoC System



X21358-082418

To ensure program correctness, the system compiler intercepts each call to a hardware function, and replaces it with a call to a generated stub function that has an identical signature but with a derived name. The stub function orchestrates all data movement and accelerator operation, synchronizing software and accelerator hardware at the exit of the hardware function call. Within the stub, all accelerator and data mover control is realized through a set of send and receive APIs provided by the `sds_lib` library.

When program dataflow between hardware function calls involves array arguments that are not accessed after the function calls have been invoked within the program (other than destructors or `free()` calls), and when the hardware accelerators can be connected using streams, the system compiler transfers data from one hardware accelerator to the next through direct hardware stream connections, rather than implementing a round trip to and from memory. This optimization can result in significant performance gains and reduction in hardware resources.

The SDSoC program execution model includes the following steps:

1. Initialization of the `sds_lib` library occurs during the program constructor before entering `main()`.



2. Within a program, every call to a hardware function is intercepted by a function call into a stub function with the same function signature (other than name) as the original function. Within the stub function, the following steps occur:
  - a. A synchronous accelerator task control command is sent to the hardware.
  - b. For each argument to the hardware function, an asynchronous data transfer request is sent to the appropriate data mover, with an associated `wait()` handle. A non-void return value is treated as an implicit output scalar argument.
  - c. A barrier `wait()` is issued for each transfer request. If a data transfer between accelerators is implemented as a direct hardware stream, the barrier `wait()` for this transfer occurs in the stub function for the last in the chain of accelerator functions for this argument.
3. Clean up of the `sds_lib` library occurs during the program destructor, upon exiting `main()`.




---

**TIP:** Steps 2a–2c ensure that program correctness is preserved at the entrance and exit of accelerator pipelines while enabling concurrent execution within the pipelines.

---

Sometimes, the programmer has insight of the potential concurrent execution of accelerator tasks that cannot be automatically inferred by the system compiler. In this case, the `sds++` system compiler supports a `#pragma SDS async(ID)` that can be inserted immediately preceding a call to a hardware function. This pragma instructs the compiler to generate a stub function without any barrier `wait()` calls for data transfers. As a result, after issuing all data transfer requests, control returns to the program, enabling concurrent execution of the program while the accelerator is running. In this case, it is your responsibility to insert a `#pragma SDS wait(ID)` within the program at appropriate synchronization points, which are resolved into `sds_wait(ID)` API calls to correctly synchronize hardware accelerators, their implicit data movers, and the CPU.




---

**IMPORTANT!** Every `async(ID)` pragma requires a matching `wait(ID)` pragma.

---

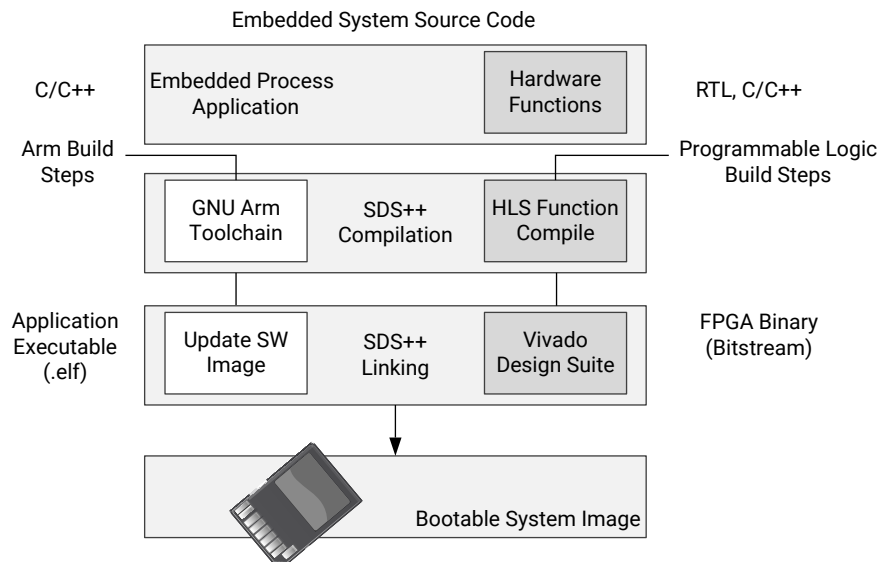
## SDSoC Build Process

The SDSoC build process uses a standard compilation and linking process. Similar to `g++`, the `sds++` system compiler invokes sub-processes to accomplish compilation and linking.

As shown in the following figure, compilation is extended not only to object code that runs on the CPU, but it also includes compilation and linking of hardware functions into IP blocks using the Vivado High-Level Synthesis (HLS) tool, and creating standard object files (`.o`) using the target CPU toolchain. System linking consists of program analysis of caller/callee relationships for all hardware functions, and the generation of an application-specific hardware/software network

to implement every hardware function call. The `sds++` system compiler invokes all necessary tools, including Vivado HLS (function compiler), the Vivado Design Suite to implement the generated hardware system, and the Arm compiler and `sds++` linker to create the application binaries that run on the CPU invoking the accelerator (stubs) for each hardware function by outputting a complete bootable system for an SD card.

**Figure 3: SDSoc Build Process**



X21126-041119

The compilation process includes the following tasks:

1. Analyzing the code and running a compilation for the main application on the Arm core, as well as a separate compilation for each of the hardware accelerators.
2. Compiling the application code through standard GNU Arm compilation tools with an object (.o) file produced as final output.
3. Running the hardware accelerated functions through the HLS tool to start the process of custom hardware creation with an object (.o) file as output.

After compilation, the linking process includes the following tasks:

1. Analyzing the data movement through the design and modifying the hardware platform to accept the accelerators.
2. Implementing the hardware accelerators into the programmable logic (PL) region using the Vivado Design Suite to run synthesis and implementation, and generate the bitstream for the device.
3. Updating the software images with hardware access APIs to call the hardware functions from the embedded processor application.
4. Producing an integrated SD card image that can boot the board with the application in an Executable and Linkable Format (ELF) file.

---

## SDSoC Debug Flow Overview

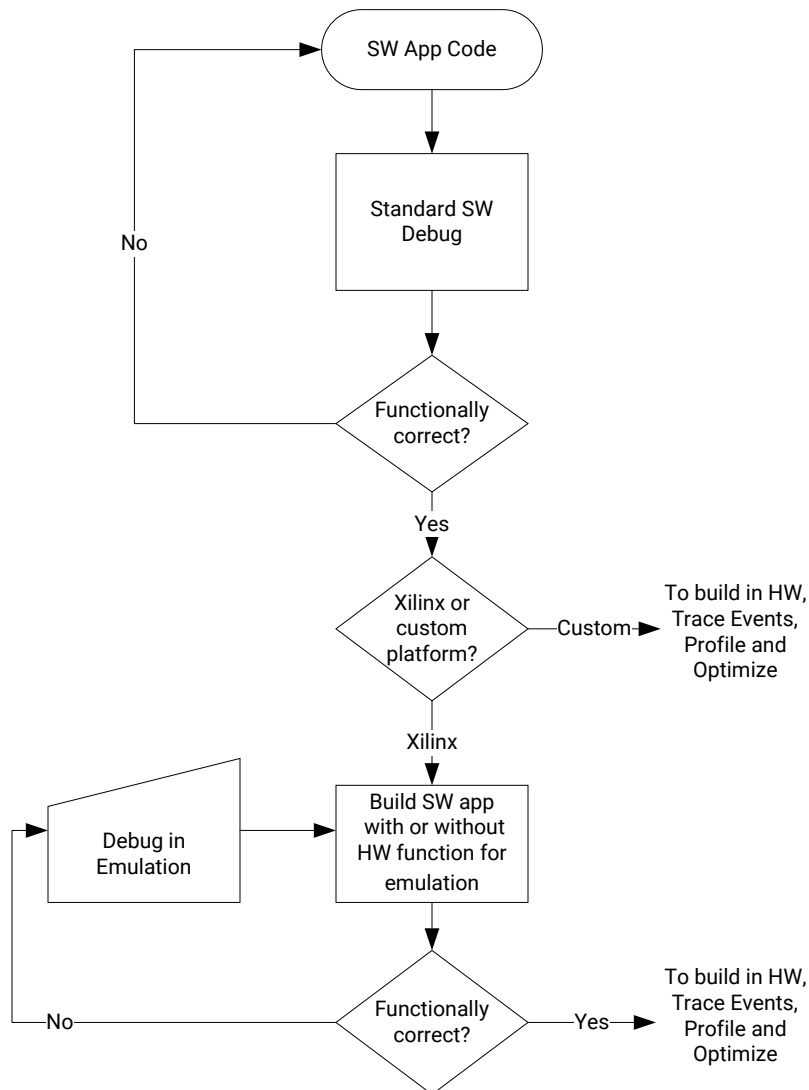
The systems produced by the SDSoC environment are high-performance, complex, and composed of hardware and software components. It can be difficult to understand the execution of applications in such systems with portions of software running in a processor, hardware accelerators executing in the programmable fabric, and many simultaneous data transfers between them. The SDSoC environment lets you create and debug projects using the Xilinx System Debugger (XSDB), and provides sophisticated hardware/software event tracing, offering an integrated timeline view of data transfers and accelerator tasks, including driver software setup and execution in hardware. Outside the SDx IDE, you can use command line or scripting options to debug your projects.

The SDSoC development environment lets you target the build process of the compilation, linking commands to either a system emulation target, or to the hardware target of the specified platform. As an alternative to building a complete system, you can create a system emulation model that consists of the target platform and application binaries. For the emulation target, the `sds++` system compiler creates a simulation model using the source files for the accelerator functions.

System emulation is one of the most capable debug features in the SDSoC environment. It can help debug functional issues and determine why an application is hanging. This feature is only available on Xilinx base platforms, including the ZC702, ZC706, ZCU102, ZCU104, ZCU106, and ZedBoard base platforms.

After you identify the hardware functions, you can use system emulation to quickly compile the logic, and verify the entire system. This provides a Quick Emulator (QEMU)-based emulator that runs the cross-compiled Arm code, interacting with the hardware accelerator being run in the Vivado simulator. The RTL simulator can display waveforms, or it can be run without waveforms for faster simulation. The emulator can be run within the SDx IDE or on the command line (`sdsoc_emulator`), providing accurate visibility of the final hardware implementation without the need to compile the system into a bitstream, and program the device on the board.

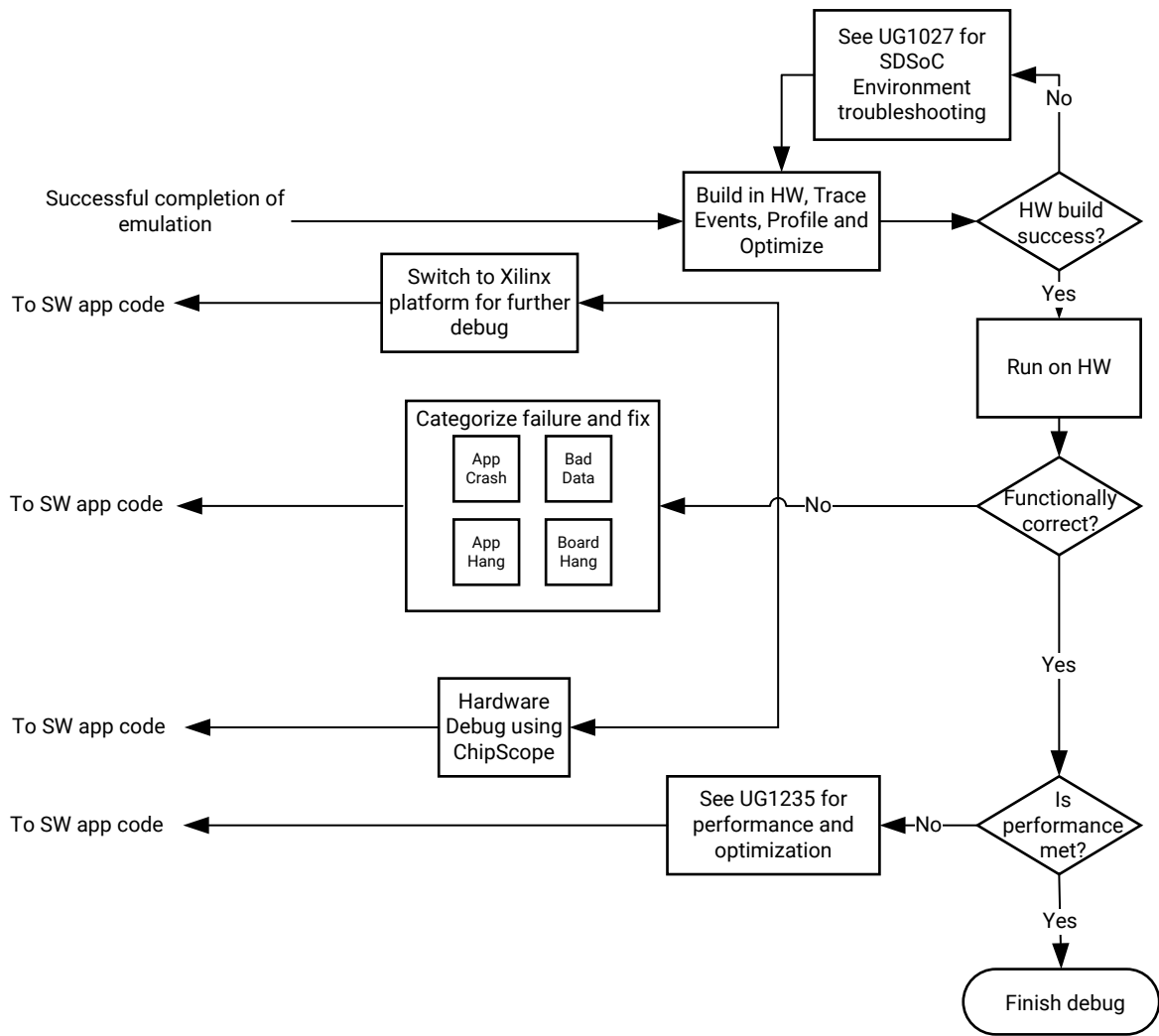
Figure 4: System Emulation Flow



X21984-112018

When targeting the hardware platform, you can also enable hardware and software event tracing to analyze the execution of events, and identify any issues (see [Hardware/Software Event Tracing](#)). If there are problems with respect to the hardware design itself, you can use hardware debug from the Vivado Lab Edition tools by inserting debug cores in the hardware functions implemented in the SDSoC environment. The following flow chart shows a typical hardware build and debug process.

Figure 5: Hardware Build and Debug Flow



X21658-100818

Xilinx base platforms support both system emulation and hardware target builds. Custom and third-party platforms, without emulation capabilities, support only the hardware build and debug flow.

**Related Information**

[Debugging System Hangs and Runtime Errors](#)

## System Emulation

On Xilinx base platforms, you can use system emulation to debug register transfer level (RTL) transactions in the entire system (PS and PL). Running your application on the SDSoC emulator (`sdsoc_emulator`) gives you visibility of data transfers with a debugger. You can debug system hangs and inspect associated data transfers in the simulation waveform view, which gives you visibility into signals on the hardware blocks associated with the data transfer.

## Hardware Execution Flow

During hardware execution, you can use the actual hardware platform to run the accelerated application. You can create a debug configuration of the hardware that includes special debug logic in the accelerators, such as the System Integrated Logic Analyzer (System ILA), Virtual Input/Output (VIO) debug cores, and AXI performance monitors. The SDSoC environment provides specific hardware debug capabilities using the Vivado hardware manager, with waveform analysis, kernel activity reports, and memory access analysis to provide visibility into these critical hardware issues.

In-system debugging lets you debug your design in real time, on your target hardware. This is an essential step in design completion. Invariably, there are situations that are extremely hard to replicate in a simulator. Therefore, there is a need to debug the problem in the running hardware. In this step, you place debug cores into your design to provide you the ability to observe and control the design. After the debugging process is complete, you can remove the debug cores to increase performance and reduce resource usage of the device.

The SDx IDE and command line options provide ways to instrument your design for debugging. The `--dk` compiler switch lets you add ILA debug cores to the interfaces of your hardware function. To debug C-callable IP that are used in your application code, you must have instantiated the required debug cores into the RTL code of the IP prior to packaging it as a C-callable IP.



---

**IMPORTANT!** *Debugging the hardware function on the SDSoC platform hardware requires additional logic to be incorporated into the overall hardware model. This means that if hardware debugging is enabled, there is some impact on resource utilization of the Xilinx device, as well as some impact on the performance of the hardware function.*

---

## Connecting to the Hardware

The board connection requirements are slightly different depending on the operating system: standalone, FreeRTOS, or Linux.

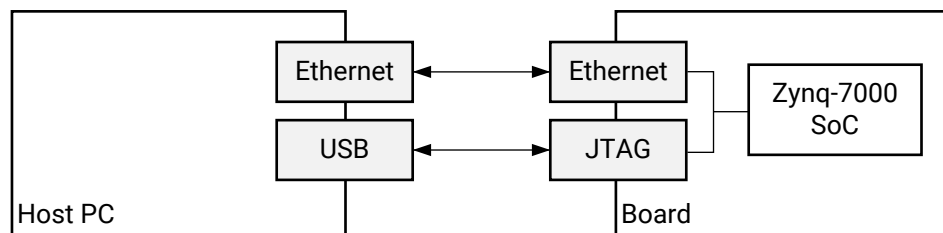
- For standalone and FreeRTOS, you must download the ELF file to the board using the USB/JTAG interface. Trace data is read out over the same USB/JTAG interface as well.

- For Linux, the SDx environment assumes the OS boots from the SD card. It then copies the `.elf` file and runs it using the TCP/TCF agent running in Linux over the Ethernet connection between the board and host PC. The trace data is read out over the USB/JTAG interface. Both USB/JTAG and TCP/TCF agent interfaces are needed for tracing Linux applications.

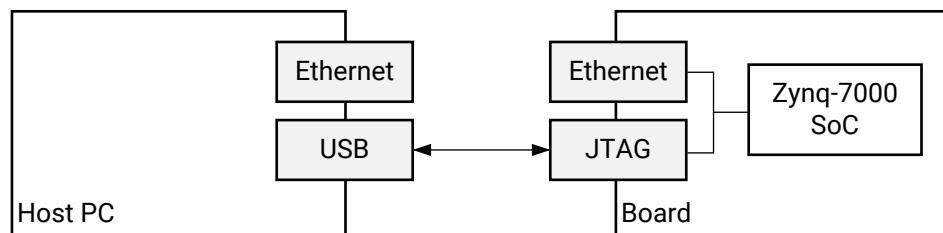
The figure below shows the connections required.

**Figure 6: Connections Required When Using Trace with Different Operating Systems**

### Linux



### Standalone/FreeRTOS



X16744-010419

## Event Tracing

The event tracing feature provides a detailed view of what is happening in the system during the execution of an application. Trace events are produced and gathered into a timeline view, giving you a perspective of the running application. This detailed view can help you understand the performance of your application given the workload, hardware/software partitioning, and system design choices. This view enables event tracing of software running on the processor, as well as hardware accelerators and data transfer links in the system. Such information helps you to identify problems, optimize the design, and improve system implementation.

Tracing an application produces a log that records information about system execution. Compared to event logging, event tracing shows the correlation between events for the duration of the event, rather than an instantaneous event at a particular time. The goal of tracing is to help debug execution by observing what happened when, and how long events took. This is best used to analyze performance and get an indication of whether there is an application hang.

# SDSoC Debug Features

This section provides details on debugging in the SDx™ environment using the Vivado® Design Suite IDE or the command line.

---

## SDx Environment Debug Tools

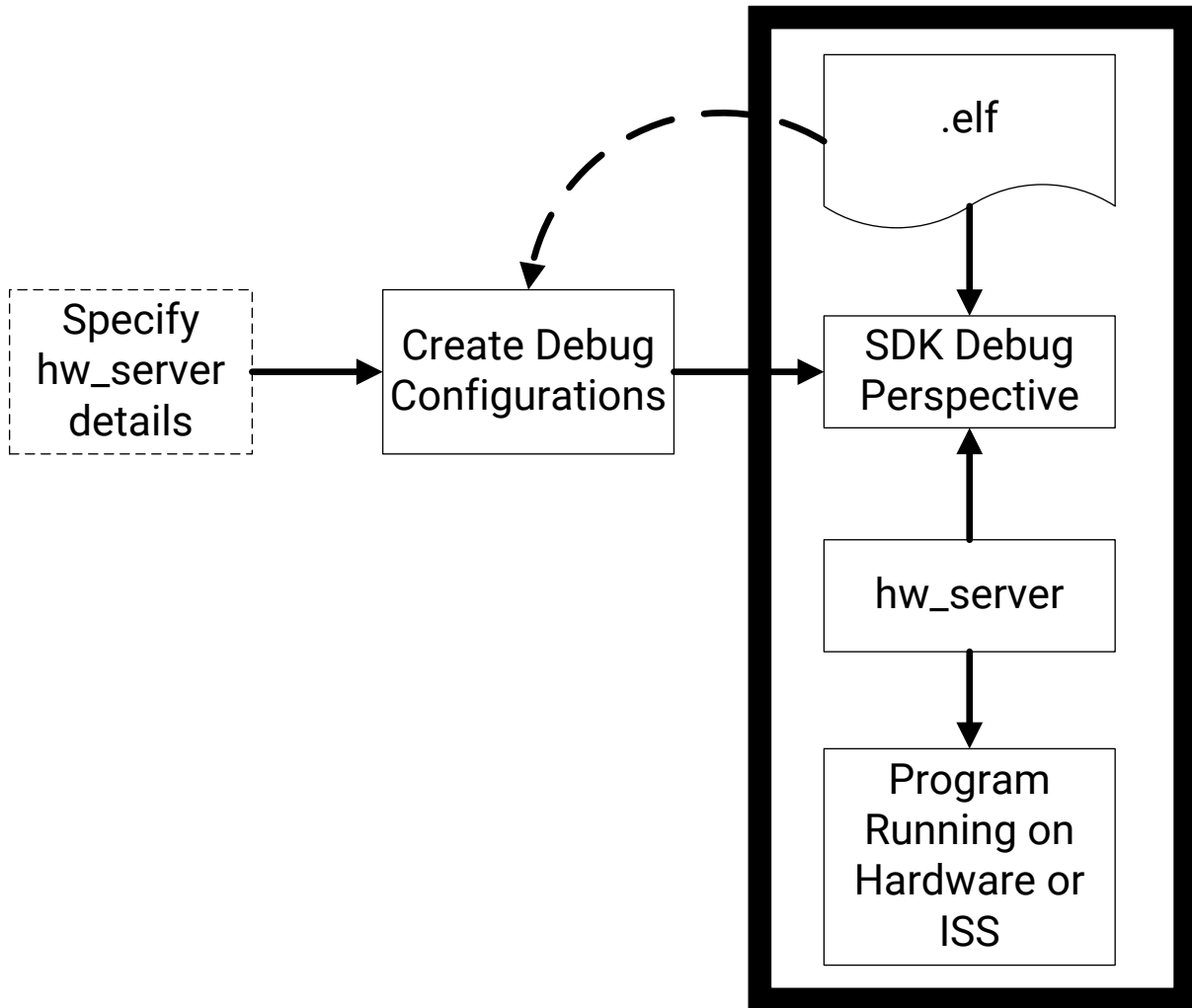
The SDx environment includes the Xilinx System Debugger (XSDB) for debugging SDSoC environment designs.

### Xilinx System Debugger (XSDB)

Xilinx System Debugger (XSDB) uses the Xilinx `hw_server` as the underlying debug engine.

The Xilinx Software Development Kit (SDK) translates each user interface action into a sequence of Target Communication Frameworks (TCF) commands. It then processes the output from System Debugger to display the current state of the program being debugged. It communicates to the processor on the hardware using Xilinx `hw_server`. You can debug multiple processors simultaneously with a single System Debugger debug configuration. This is the recommended debug engine for SDx environment designs. The System Debugger can either be launched on the hardware or the QEMU engine.





X21076-120218

The workflow is made up of the following components:

- ELF file:** To debug your application, you must use an ELF file compiled for debugging. The debug ELF file contains additional debug information for the debugger to make direct associations between the source code and the binaries generated from that original source. Refer to [Build Configurations](#) for more information.
- Debug configuration:** To launch the debug session, you must create a debug configuration in the SDx environment. This configuration captures options required to start a debug session, including the executable name, processor target to debug, and other information. Refer to [Setting Debug Configurations](#) for more information.
- SDx debug perspective:** Using the debug perspective, you can manage the debugging or running of a program in the SDx workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

You can repeat the cycle of modifying the code, building the executable, and debugging the program in the SDx environment.

**Note:** If you edit the source after compiling, the line numbering will be out of step because the debug information is tied directly to the source. Similarly, debugging optimized binaries can also cause unexpected jumps in the execution trace.

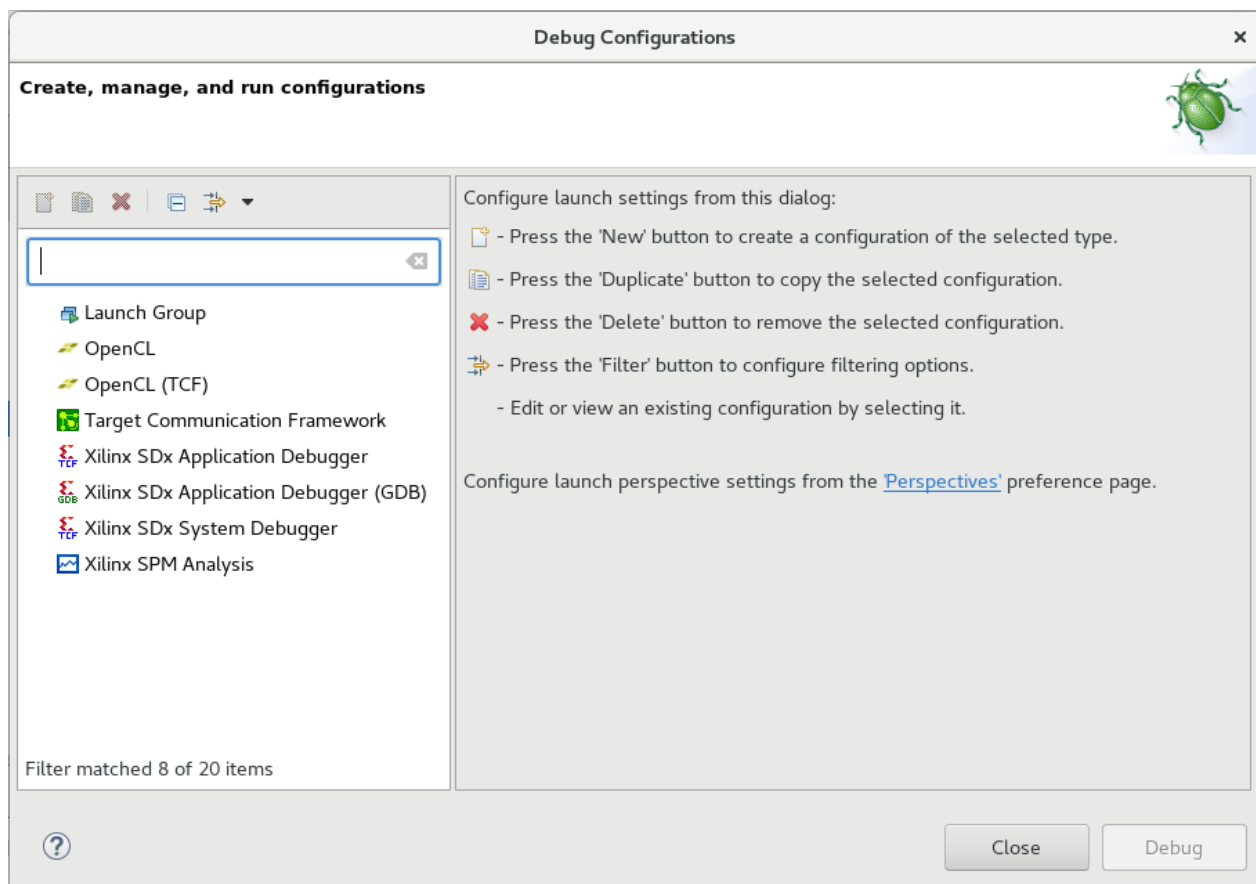
## Setting Debug Configurations

To debug, run, and profile an application, you must create a debug configuration that captures the settings for executing and debugging the application. To create a debug configuration, in the Assistant view, right-click on the **Debug** build configuration, and select **Debug → Debug Configurations** from the menu. Alternatively, you can select the **Run → Debug Configurations** command from the main menu. The Debug Configurations dialog box opens as shown below.



**TIP:** Based on the OS and system configuration of your application project, and the type of application being debugged, the tabs of the Debug Configurations dialog box can change. The tabs and options discussed here might be different from what you see.

Figure 7: Debug Configurations



In the Debug Configurations dialog box, select the **Xilinx SDx Application Debugger** to create a debug configuration for the project. A new debug configuration is created for the application project, and is opened with multiple tabs to manage the configuration.

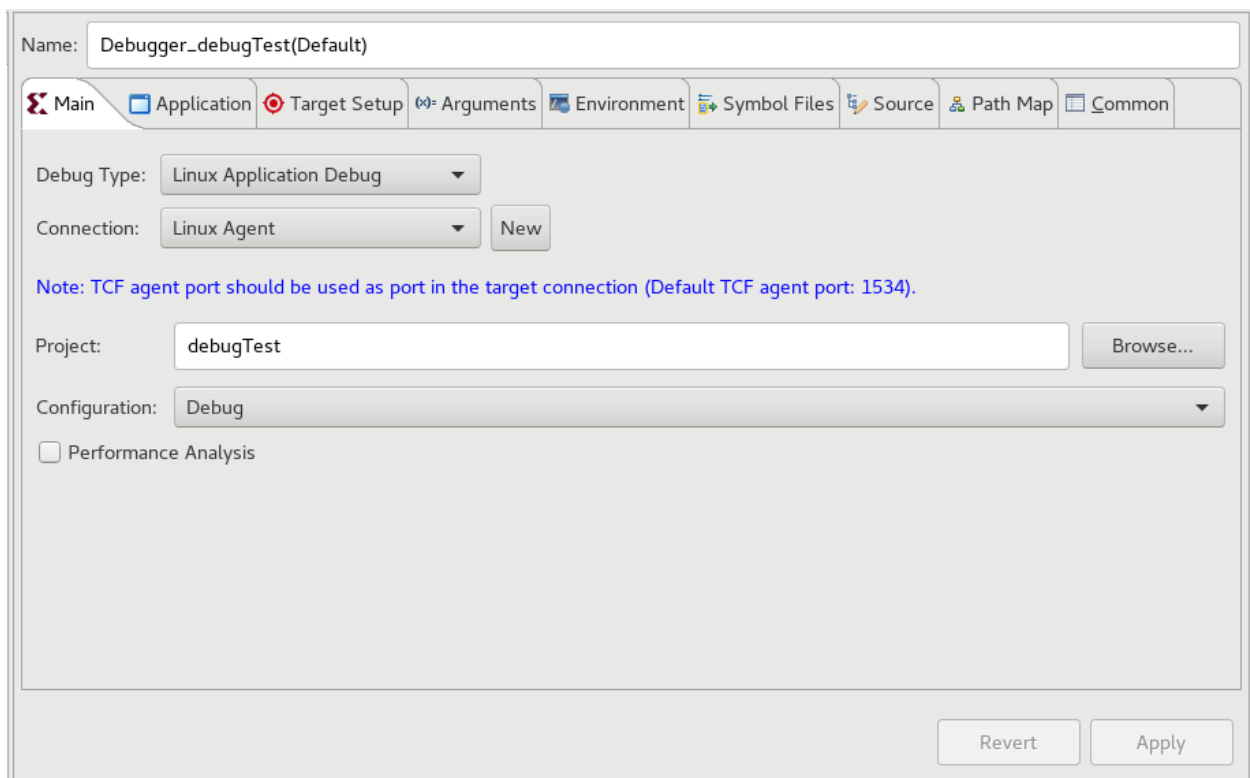
## Main Tab

The **Main** tab is automatically populated with the debug and connection type for the current application project. For example, a Linux application uses the Linux debug type and the Linux agent for connecting to the application.



**TIP:** You can change the selected Debug Type, but this also resets the application project associated with the debug configuration.

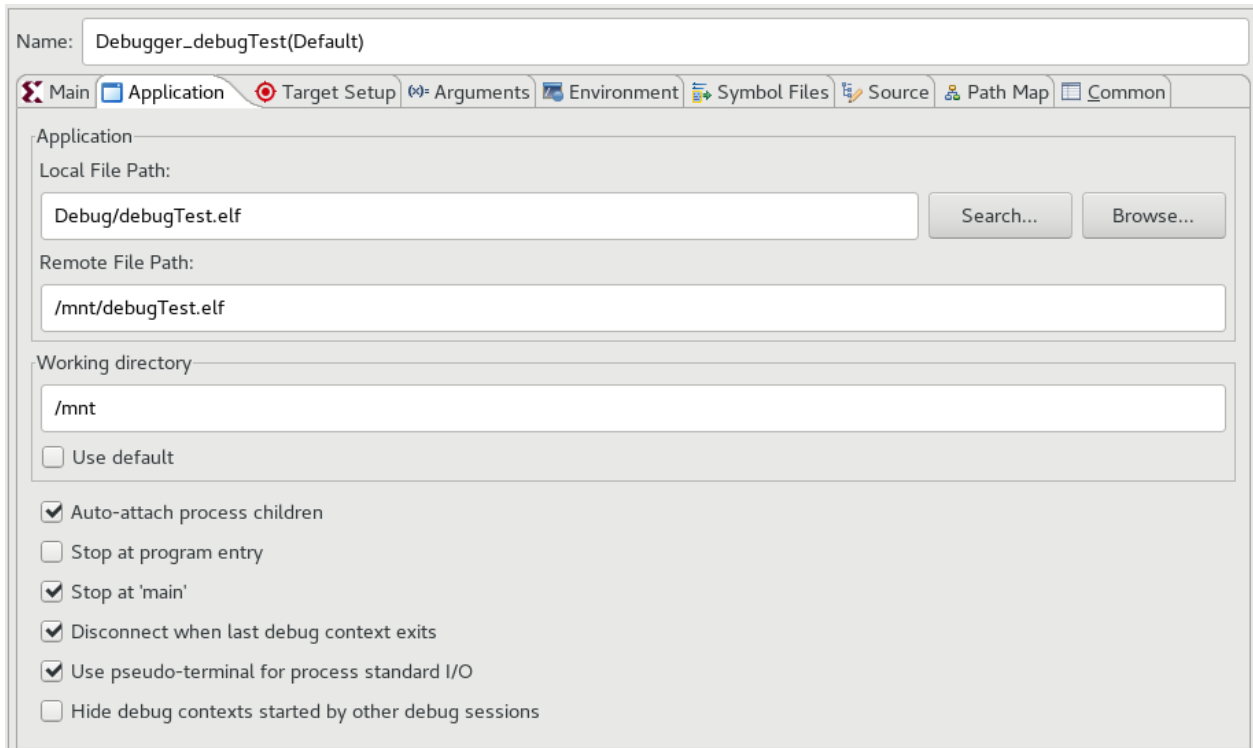
Figure 8: Debug Configurations - Main Tab



## Application Tab

The Application tab displays the compiled application .ELF file that is being downloaded to be run on the processor.

Figure 9: Debug Configuration - Application Tab

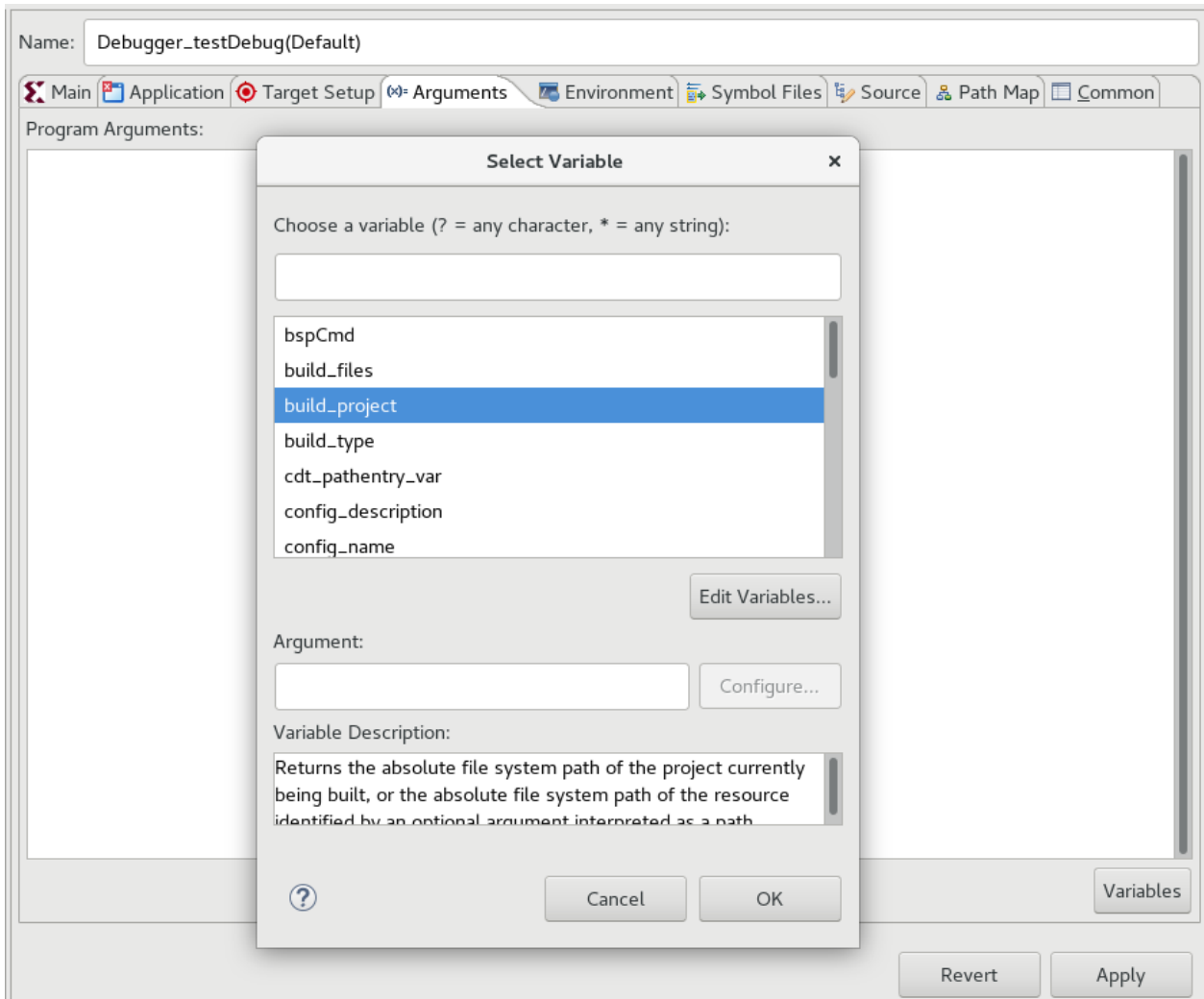


## Target Setup Tab

For Linux applications, the Target Setup tab is blank. For standalone applications, the tab lets you specify the hardware platform, and whether to use the first stage boot loader (FSBL) flow for initialization (if you need to initialize devices on the platform).

## Arguments Tab

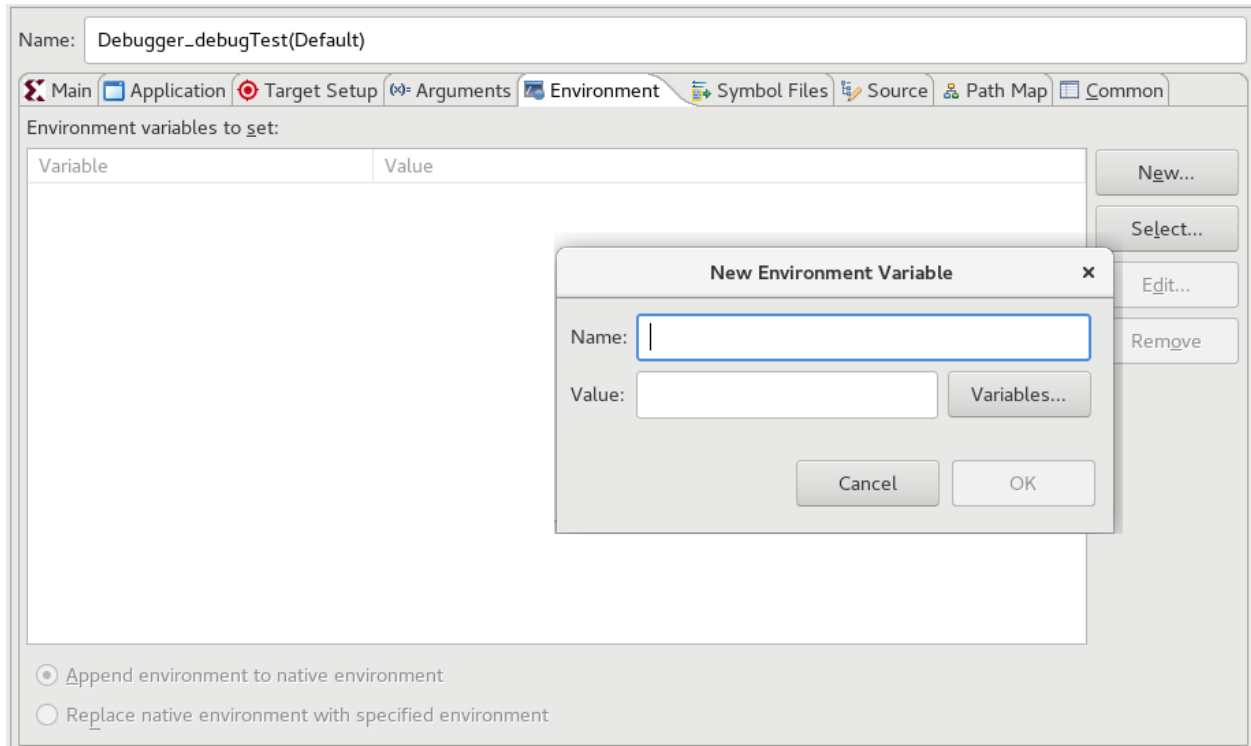
In the Arguments tab, you can specify any variables that are needed for launching the debug session. Click **Variables** to display the Select Variable dialog box.



## Environment Tab

In the Environment tab, you can set any environment variables for the debug configurations.

Figure 10: Add, Set, and Edit Environment Variables



Click **New** to create and define a value for a new environment variable to add to the debug configuration. Click **Select** to display a list of existing environment variables that can be added to the debug configuration. These can be edited and set to specific values.

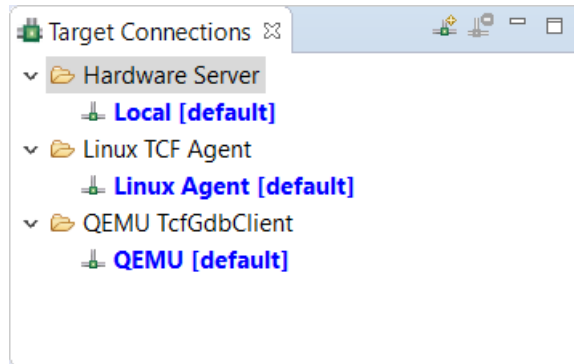
### Remaining Debug Configuration Tabs

The Symbol Files, Source, Path Map, and Common tabs are for advanced debugging of application-specific functions that do not apply to XSDB, and can be safely ignored.

### Target Connections

In the **Target Connections** view, you can configure multiple remote targets. It displays connected targets, and you can add or delete target connections. The SDx environment establishes target connections through the Hardware Server agent. In order to connect to remote targets, the hardware server agent must be running on the remote host, which is connected to the target.

Use the **Hardware Server** when the application is for standalone. The Hardware Server only requires a JTAG connection to the board. Use the **Linux TCF Agent** for when the application is compiled to run on Linux for the SoC. The **Linux TCF Agent** requires an Ethernet connection from the machine to the board.




For more information, refer to [Connecting to the Hardware](#).

## Debug Linux Applications in the SDx IDE

In the SDx IDE, use the following procedure to debug your application:


Ensure the board is connected to your host computer using the JTAG Debug connector, and that there is an Ethernet connection between the board and host PC.

1. Set the platform to boot from the SD card, as specified in the User Guide for the selected SDSoC platform.
2. In the SDx Application Project Settings window, set the Target to **Hardware**, and enable the **Generate SD card image** checkbox.
3. In the Assistant view, right-click the **Debug** build configuration, and select the **Set Active** command.
4. Click the **Build** () button, in the Assistant view or the main menu, to build the Debug configuration.
5. From a file browser, or command shell, copy the contents of the `Debug/sd_card` folder to an SD card.
6. Insert the SD card into the card reader of the platform, and boot the card.
7. Make sure the board is connected to the network, and note its IP address, for example, by executing `ifconfig eth0` on the board at the command prompt using a terminal communicating with the board over UART.
8. In the Assistant view, right-click the **Debug** build configuration, and select **Debug → Debug Configurations** to create a new debug configuration.
9. Double click or right-click and select **New** on the **Xilinx SDx Application Debugger**.
10. In the new configuration, click the **New** button next to **Connection: Linux Agent**.
11. In the **Target Connection Details** specify the target name and enter the IP address of the board. It is highly suggested to test the connection by click the **Test Connection** button to make sure it can connect to the board.
12. Click **Apply** to save the changes and click **Debug**.

13. Switch to the SDSoC environment debug perspective, where you can start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

## Debugging Standalone or FreeRTOS Applications in the SDx IDE

To debug applications running on a standalone (bare-metal) or FreeRTOS OS, ensure the board is connected to your host computer using the JTAG debug connector, and then set the board to boot from JTAG.

1. In the Assistant view, right-click the **Debug** build configuration, and select the **Set Active** command.
2. Click the **Build** (  ) button, in the Assistant view or the main menu, to build the Debug configuration.
3. In the Assistant view, right-click the **Debug** build configuration, and select **Debug → Debug Configurations** to create a new debug configuration.
4. Optional: Switch to the SDSoC environment Debug Perspective, where you can start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.
5. Optional: In the SDx IDE toolbar, click **Debug**, which provides a shortcut to the procedure described above.

## Xilinx Software Command-Line Tool (XSCT)

Graphical development environments such as the SDx environment are useful for improving development for a new processor architecture. It helps to abstract away and group most of the common functions into logical wizards that even a novice can use. However, the scriptability of a tool is also essential for providing the flexibility to extend what is done with that tool. It is particularly useful when developing regression tests that are run nightly, or for running a set of commands that are used often by the developer.

Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command line interface to the SDx environment. As with other Xilinx tools, the scripting language for XSCT is based on Tool Command Language (Tcl). You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions:

- Create hardware, board support packages (BSPs), and application projects.
- Manage repositories.
- Set toolchain preferences.
- Configure and build BSPs/applications.
- Download and run applications on hardware targets.
- Create and flash boot images by running `Bootgen` and `program_flash` tools.



For information on XSCT commands, see the *Xilinx Software Command-Line Tool (XSCT) Reference Guide* (UG1208).

---

## System Emulation

System emulation can be run on System Debugger using the Target Communications Framework (TCF) server.

**Note:** Currently, emulation is not supported for custom platforms. Only the base platforms provided by Xilinx support emulation.

### Running System Emulation from the IDE

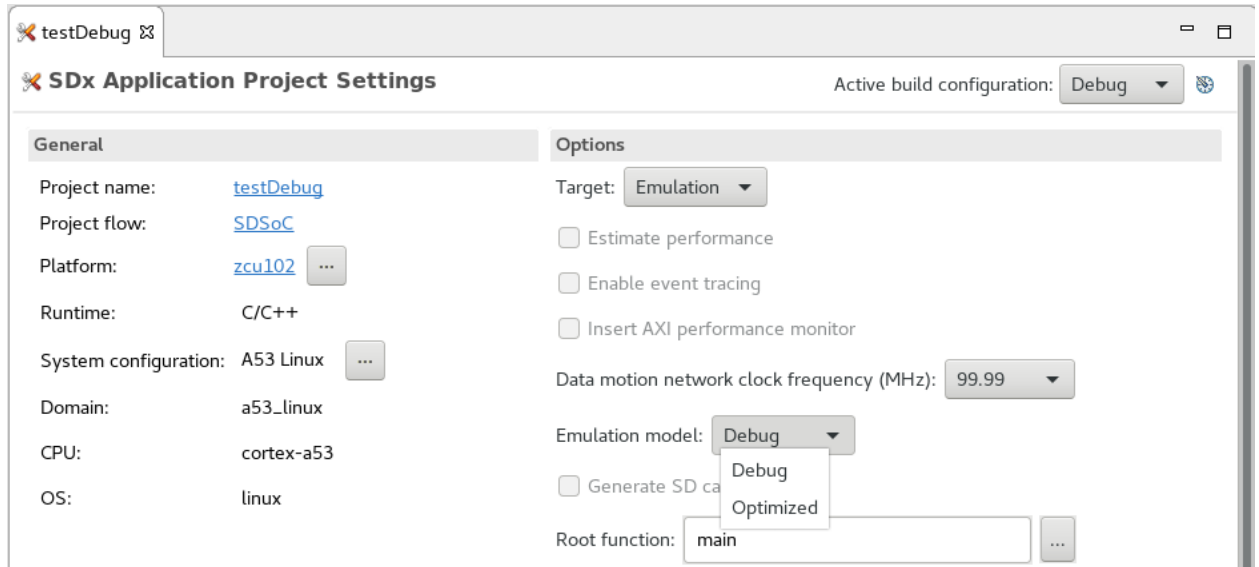
System emulation provides the same level of accuracy as the final implementation without the need to compile the system into a bitstream and program the device on the board. System emulation can be used for debugging applications without involving the actual hardware. It can also be used for identifying any bottlenecks in performance.

#### *Enable System Emulation*


To enable system emulation within the Application Project Settings window, take the following steps:

1. Set the Active build configuration to **Debug**.
2. Set the Target to **Emulation**.
3. Set the emulation model. There are two emulation model modes:
  - **Debug:** Builds the system through RTL generation, and the IP integrator block design containing the hardware function, elaborates the hardware design, and runs behavioral simulation on the design, with a waveform viewer to help you analyze the results. You interact with the Vivado simulator within the Vivado Design Suite to analyze the waveforms.
  - **Optimized:** Runs the behavioral simulation in batch mode, returning the results without the waveform data. While the Optimized model can be faster, it returns less information than the Debug model.

For faster emulation without capturing this hardware debug information, select **Optimized**. For example, to debug system hang issues, use the **Debug** mode and look at the state of different signals in the Waveform viewer within the Vivado simulator. Alternatively, if you are debugging the application only, you can use the **Optimized** emulation model.



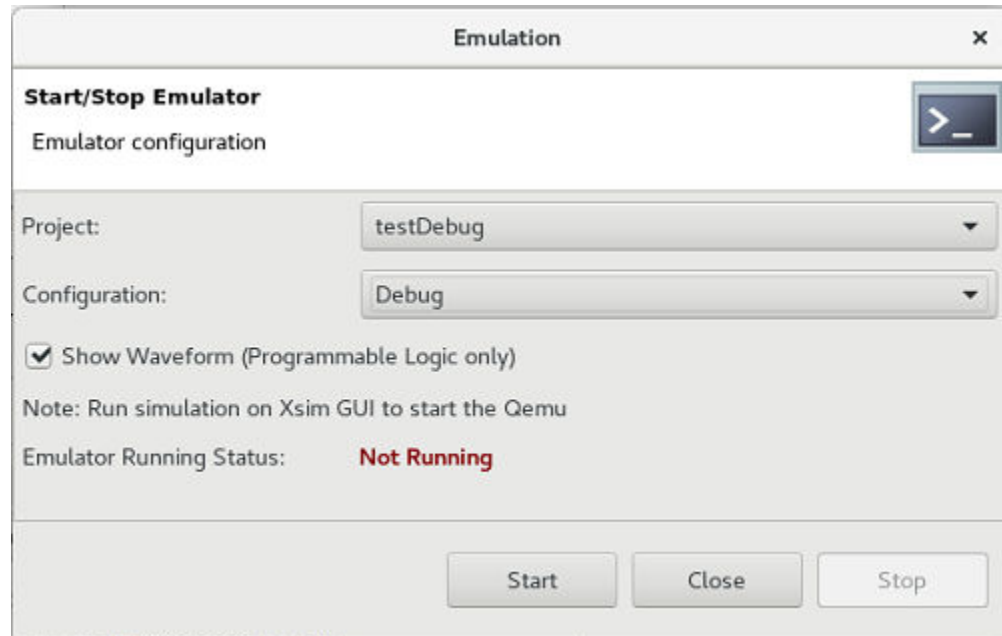
Because emulation does not require a full system compile, the tool disables the generation of the bitstream and the Generate SD card image option to improve runtime and iteration time. Using system emulation allows you to verify and debug the system with the same level of accuracy as a full bitstream compilation.

4. After specifying the emulation model, click the **Build** button (  ) to compile the system for emulation.

The duration of the build process depends on your application code, the size of your hardware functions, and the options you have selected. To compile the hardware functions, the tool stack includes the SDx environment, and Vivado High-Level Synthesis (HLS) tool, and the Vivado Design Suite.

## Run the System Emulator

1. After building the emulation target, you can run the system emulator using **Xilinx → Start/Stop Emulator**. Alternatively, you can also select the application in the **Assistant** panel, by right-clicking, and then selecting **Start/Stop Emulator**.
2. When the Start/Stop Emulator dialog box opens, the emulation mode is specified:
  - If the emulation mode is Debug, you can choose to run the emulation with or without waveforms.
  - If the emulation mode is Optimized, the Show Waveform check box is disabled, and cannot be changed.



The Start/Stop Emulator dialog box displays the Project name, the build Configuration, and has the Show Waveform option. Disabling the **Show Waveform** option lets you run emulation with the output directed solely at the Emulation Console view, which shows all system messages including the results of any print statements in the source code. Some of these statements might include the values transferred to and from the hardware functions, or a statement that the application has completed successfully, which would verify that the source code running on the PS and the compiled hardware functions running in the PL are functionally correct. Enabling the **Show Waveform** option provides the same functionality in the Console window, plus the behavioral simulation of the register transfer level (RTL), with a waveform window. The RTL waveform window allows you to see the value of any signal in the hardware functions over time. When using **Show Waveform**, you must manually add signals to the waveform window before starting the emulation.

3. Use the Scopes pane to navigate the design hierarchy.
4. Select the signals to monitor in the Object pane, and then right-click to add the signals to the waveform pane.
5. Click the **Run All** toolbar button to start updates to the waveform window. For more information about working with the Vivado simulator waveform window, refer to *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

**Note:** Running with RTL waveforms results in a slower runtime, but enables detailed analysis into the operation of the hardware functions.



**TIP:**

You can also start the system emulation by selecting the active project in the Project Explorer view, and then right-clicking to select one of the following menu commands:

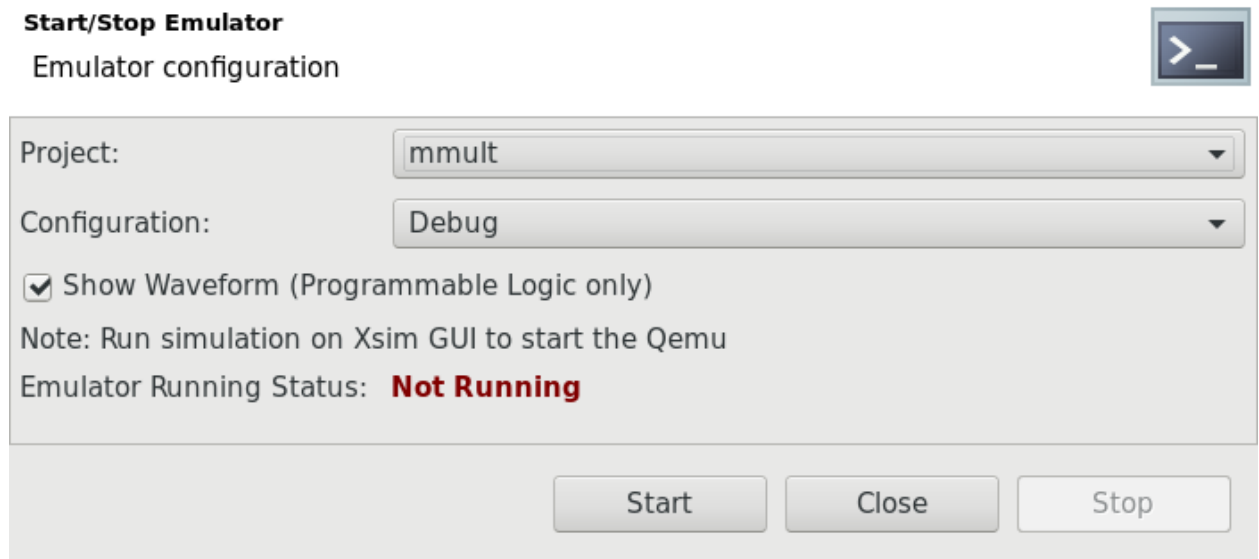
- **Run As** → **Launch on Emulator**

- **Debug As → Launch on Emulator**

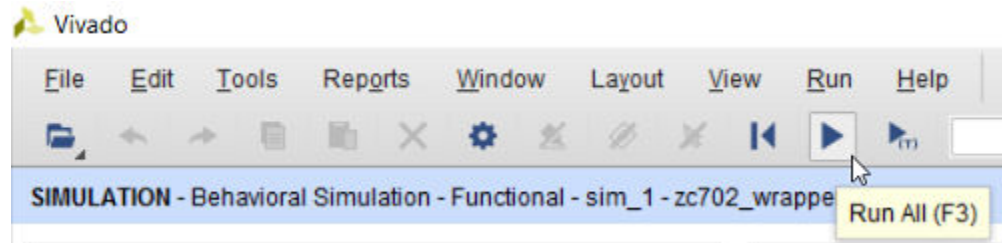
Launching the emulator from the Debug As menu causes the perspective change to the debug perspective to arrange the windows and views to facilitate debugging the project.

### View Emulation Output

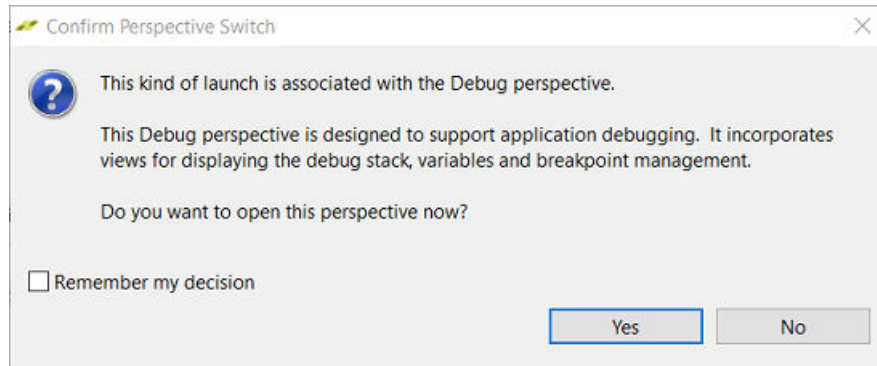
1. After you run the system emulator, you can see the program output in the console tab, and if the **Show Waveform** option was selected, the Vivado IDE is launched with the simulator running.



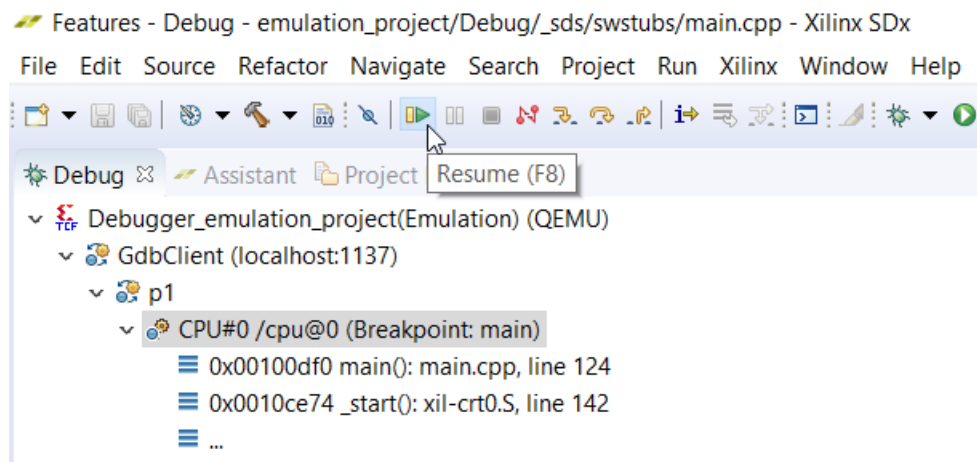
Add waveforms to the Waveforms window as desired. To start the simulation, click the **Run All** button.



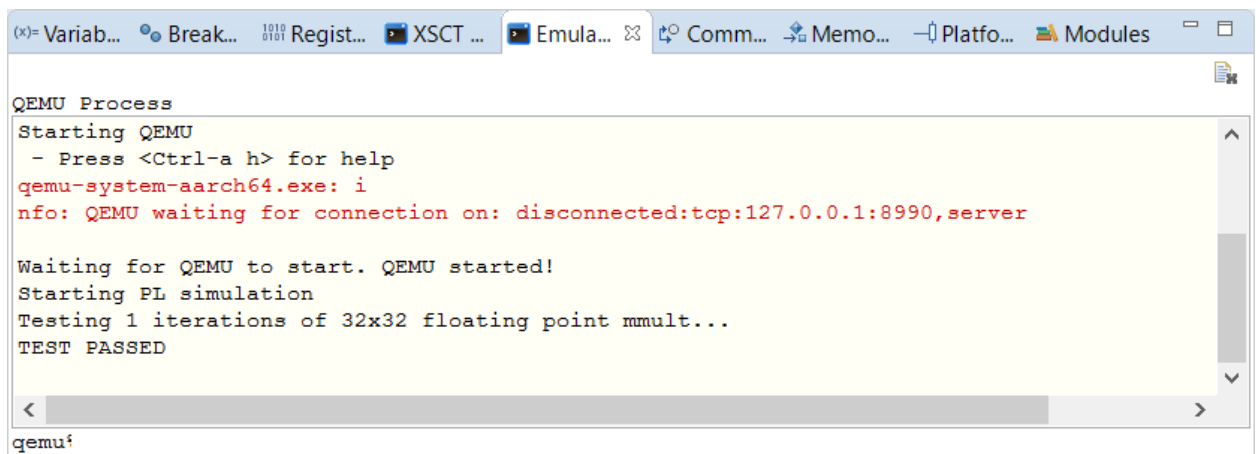
2. To start a debug session with the emulator running, in the Assistant view right-click on the Debug build configuration and select **Debug → Launch on Emulator (SDx Application Debugger)**.
3. The Confirm Perspective Switch dialog box is displayed. Click **Yes** to switch to the Debug perspective.



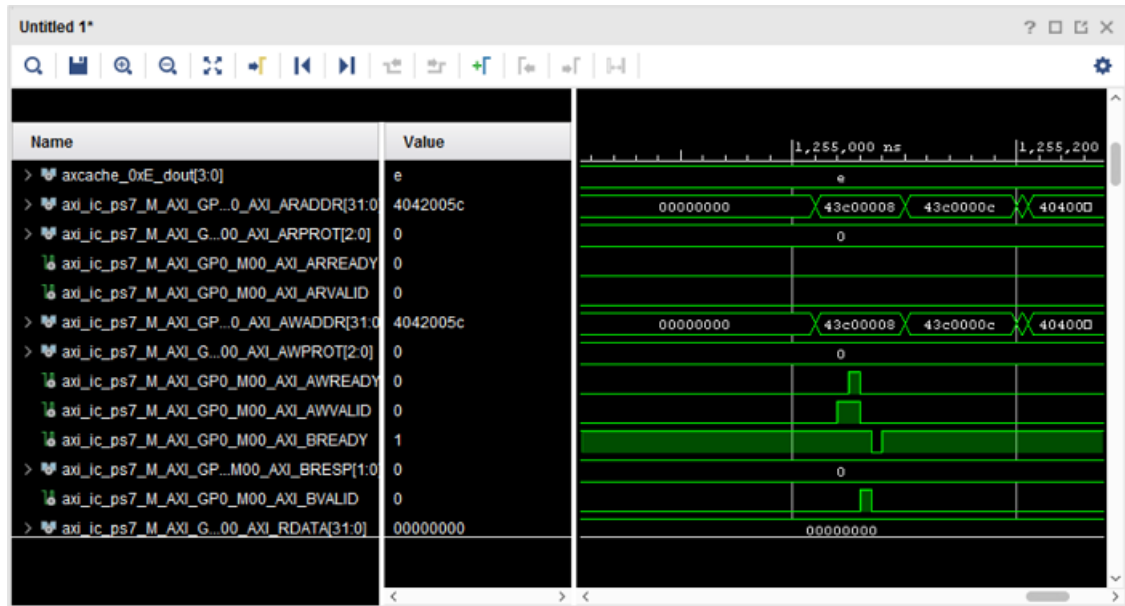
- The application is started in the Debug perspective and the program execution is stopped at the main function. To resume the execution of the application code, click **Resume**.



This starts execution of the application code. The output of the application code can be seen in the Emulation Console, as shown in the following figure:



The status of different signals is displayed in the Vivado Waveform window. You also see any appropriate response in the hardware functions in the register transfer level (RTL) waveform. During any pause in the execution of the code, the RTL waveform window continues to execute and update, just like an FPGA running on the board.



5. You can stop the emulation at any time using the menu option **Xilinx → Start/Stop Emulator**, and then selecting **Stop**.



**TIP:** For an example project to demonstrate emulation, create a new SDx environment project using the Emulation Example template. The `README.txt` file in the project has a step-by-step guide for doing emulation on both the SDx IDE and the command line.

## Running System Emulation from the Command Line

You can create a design outside of the SDx IDE in a general command-line flow, using individual SDx commands to build and compile the project, or with a Makefile flow. In the following sample script, the `TARGET` flag defines that the compilation should be done for emulation.

```
# FPGA Board Platform (Default ~ zcu102)
PLATFORM := zcu102

# Run Target:
# hw - Compile for hardware
# emu - Compile for emulation (Default)
TARGET := emu
```

The emulation mode, as shown in the sample script below, can be specified with one of two options:

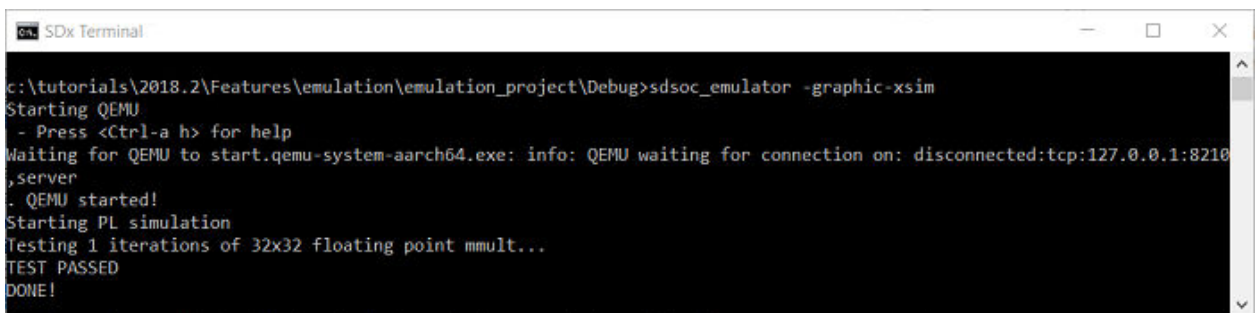
- `debug`: Captures waveform data from the PL hardware emulation for viewing and debugging.

- **optimized:** Provides faster emulation without capturing hardware debug information.

```
# Target OS:
#   linux (Default), standalone
TARGET_OS := linux

# Emulation Mode:
#   debug      - Include debug data
#   optimized  - Exclude debug data (Default)
EMU_MODE := optimized
```

Type `make` to build the program at the command prompt. If you want to view the waveform in the simulator, change directory to the level where you have the `_sds` directory, then type `sdsoc_emulator -graphic-xsim`. This starts the Vivado Simulator, as shown below.



```
SDx Terminal
c:\tutorials\2018.2\Features\emulation\emulation_project\Debug>sdsoc_emulator -graphic-xsim
Starting QEMU
- Press <Ctrl-a h> for help
Waiting for QEMU to start.qemu-system-aarch64.exe: info: QEMU waiting for connection on: disconnected:tcp:127.0.0.1:8210
server
. QEMU started!
Starting PL simulation
Testing 1 iterations of 32x32 floating point mmult...
TEST PASSED
DONE!
```

## Hardware Execution Features Available to All Platforms

Although system emulation is only available for application projects running on Xilinx base platforms, the hardware execution flow is available to run on any platform that is the target of an SDSoc project. The hardware execution flow is the embedded processor operating system, the application code, and the hardware functions running in concert, as designed, on the hardware platform. The types of debugging you can perform on the hardware include the following:

- Full software debug using the [Xilinx System Debugger \(XSDB\)](#)
- Co-debug of hardware and software using the [Xilinx System Debugger \(XSDB\)](#)
- [Event Tracing](#)

**Note:** Hardware debug includes instrumenting the hardware for analyzing signals in the Vivado hardware manager feature. The application needs to be built with special instructions to instrument the hardware for this.

## Hardware Debugging in SDSoC Using ChipScope

After the final system image is generated and executed in the SDx environment, the entire system (including the embedded processor OS, the application code, and the accelerated hardware functions) can be validated to be executing correctly on the actual hardware, and any necessary debug activity can be performed. The ChipScope™ feature is used to debug designs in hardware using the Vivado IDE. Cross-probing hardware and software requires an advanced understanding of the SDx environment and the Vivado tool suite.

This debugging step can reveal issues relating to connecting to the target platform, booting the processor, and programming the hardware with the system image. It might also highlight problems with interactions between the application code and the hardware functions in the form of protocol violations, and with validating multiple hardware functions with the application code.

This step could also reveal system performance metrics that could shift your focus from debug to performance tuning. In the SDx environment, you can instrument the hardware to analyze transactions on the interfaces of the hardware accelerators and adapters. You can also debug the hardware portion of the design.

### ***Using --dk to Enable Debugging the Accelerated Function***

Visibility into a running design is crucial for debugging difficult situations, like when the application hangs. The System ILA debug core provides transaction-level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core.

The System ILA core can be instantiated in the overall hardware of an existing SDx environment design to enable debugging features within that design, or it can be inserted automatically by the compiler. The `sds++` compiler provides the `--dk` switch to attach System ILA cores at the interfaces to the hardware functions for debugging and performance monitoring purposes. Use the `--dk` option to enable System ILA core insertion:

```
--dk arg <[protocol|chipscope|
list_ports]:<compute_unit_name>:<interface_name>>
```

The following is an example of the `--dk` option in use:

```
sds++ -c --dk chipscope:vadd_cu0:s_axi_control --dk
chipscope:vadd_cu0:m_axi_gmem
```

The following is an example of a Makefile to insert debug cores:

```
APPSOURCES = main.cpp mmult.cpp madd.cpp
EXECUTABLE = mmultadd.elf

PLATFORM = zc702
CLKID =
DMCLKID =
SDSFLAGS = -sds-pf ${PLATFORM} ${DMCLKID} \
```



```

-sds-hw mmult mmult.cpp ${CLKID} -sds-end \
-sds-hw madd madd.cpp ${CLKID} -sds-end \
-debug-port mmult:A \
    -debug-port madd:C \
    --dk chipscope:madd_1:A \
    --dk chipscope:madd_1_if:ap_ctrl

CC = sds++ ${SDSFLAGS}

CFLAGS = -O3 -c
CFLAGS += -MMD -MP -MF"$(@:%.o=%.d)"
LFLAGS = -O3

OBJECTS := $(APPSOURCES:.cpp=.o)
DEPS := $(OBJECTS:.o=.d)

.PHONY: all clean ultraclean

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
    ${CC} ${LFLAGS} $^ -o $@

-include ${DEPS}

%.o: %.cpp
    ${CC} ${CFLAGS} $^ -o $@

clean:
    ${RM} ${EXECUTABLE} ${OBJECTS} ${DEPS}

ultraclean: clean
    ${RM} ${EXECUTABLE}.bit
    ${RM} -rf _sds sd_card
    
```

The `-debug-port` option specifies a function name and argument name to insert a System ILA for accelerators. The lower level `--dk` option specifies the tool command language (Tcl) file used to recreate the block design instance and port name, such as in the following example.

- `-debug-port mmult:A` is equivalent to `--dk chipscope:mmult_1:A`, but the `sds++` command determines what the instance and port names are in the Tcl file used to recreate the block design.

**Note:** A Tcl file is used by the SDx environment to recreate a block design in the hardware platform including the accelerators in the Vivado Design Suite.

- `--xp param:compiler.userPostSysLinkTcl=<user_tcl_file>`, where `<user_tcl_file>` contains IP integrator Tcl commands for advanced users who need to perform post-processing of the System ILA in the block diagram after system linking and before synthesis.

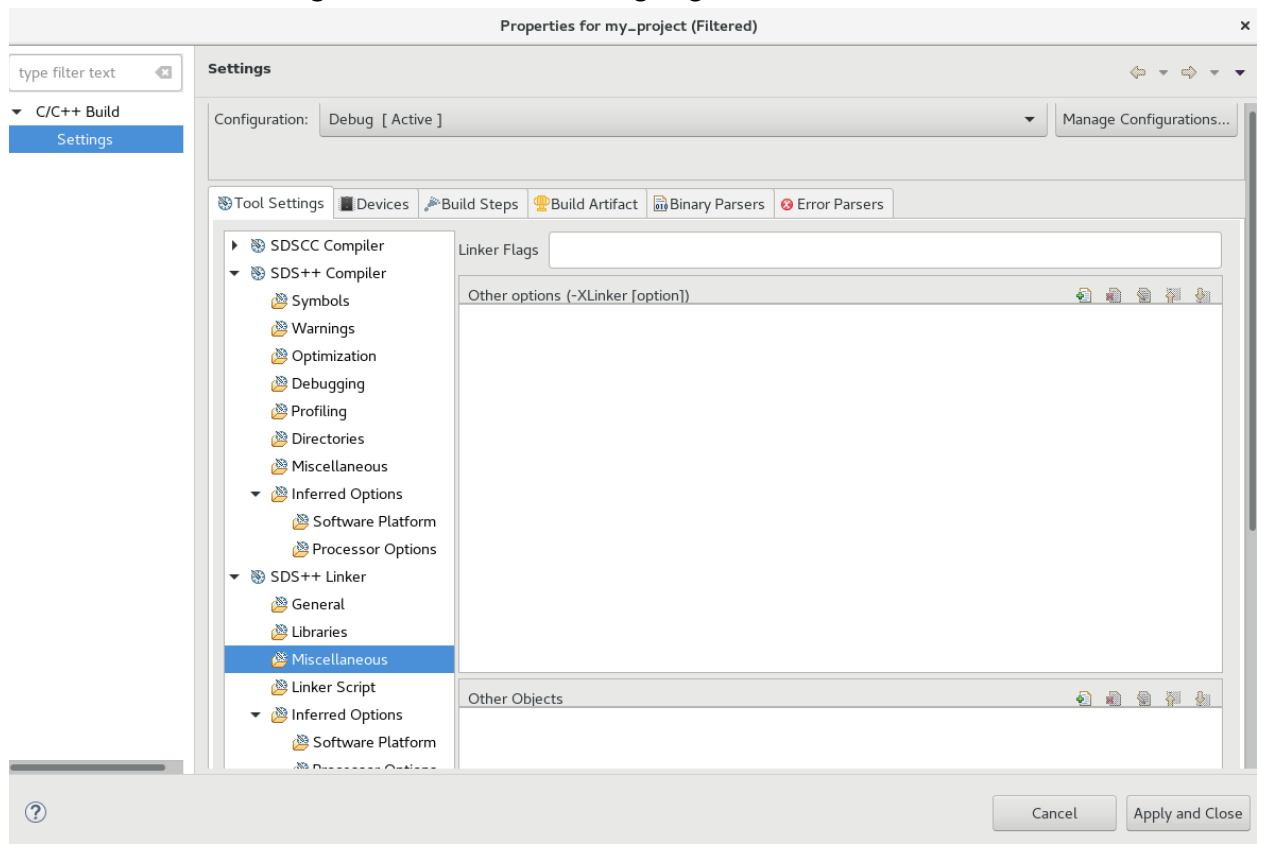
**Note:** Advanced users can change ILA settings using Tcl commands. It is often possible to enable additional probes and interfaces in an ILA and debug other signals in the same clock domain as needed. Doing this can save logic resources in the FPGA. You can also cross-trigger a chain of ILAs using this feature.

- `--dk` can be used to insert the System ILA for accelerator and adapter ports. You need to use this option to observe the adapter ports. Once the design is built, you can debug the design using the Vivado hardware manager features, as described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

## Add Flags to Build Settings

If you are working in the SDx IDE, the system options shown in the previous section can be specified in the build settings as shown below:

1. From the Assistant view, right-click the Debug or Release build configuration, and select the **Settings** command.
2. In the Build Configuration Settings dialog box, click the **Edit Toolchain Settings** link.
3. In the Tool Settings tab of the Properties for <Application\_Name> dialog box, select **SDS++ Linker → Miscellaneous**.
4. Click in the **Linker Flags** field and add the debug flags as needed.



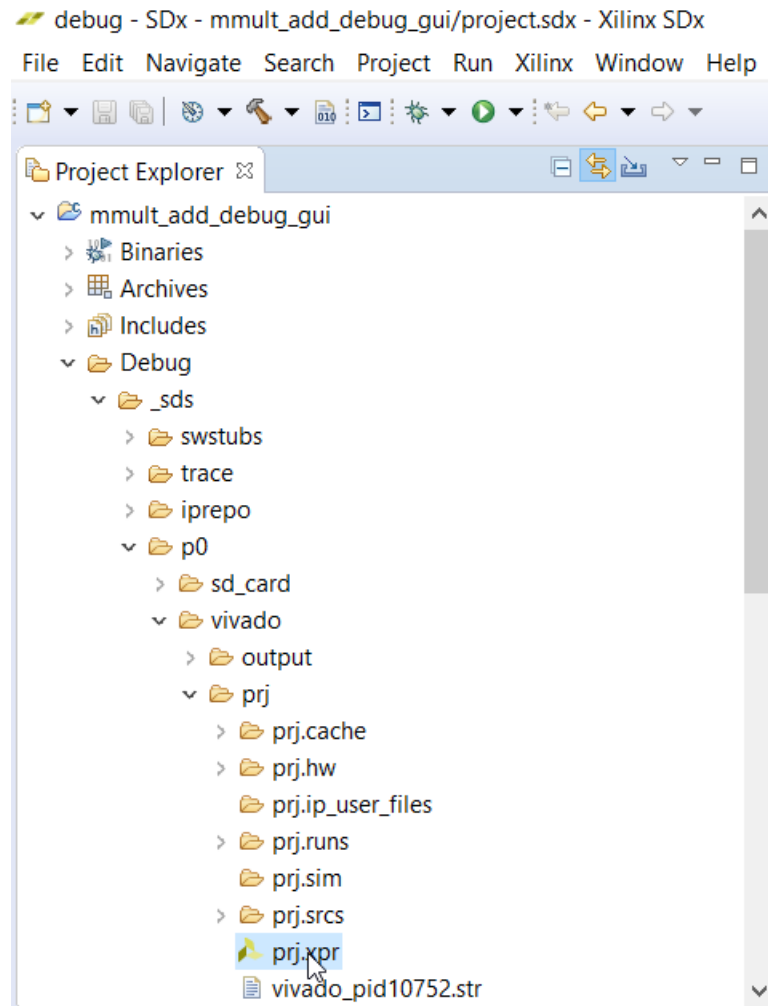
**TIP:** At the top of the Tool Settings tab, there is a Configuration field that lets you select the build to apply the settings to the Debug build, the Release build, or All builds.

See the *SDx Command and Utility Reference Guide* (UG1279) for more information on compiler and linker options.

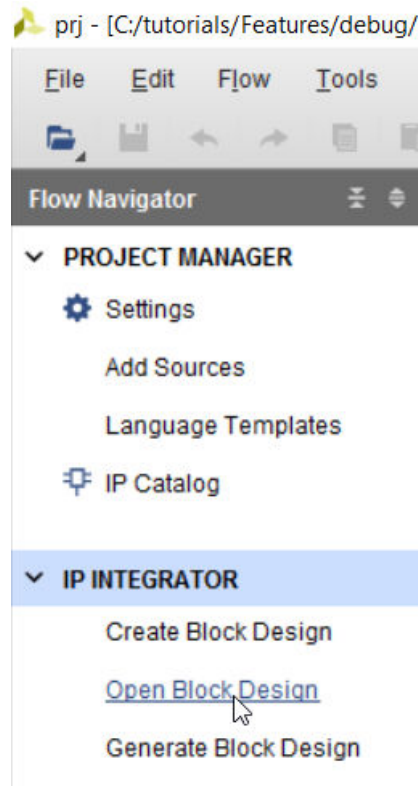
## Analyzing the Hardware Design

When the design has been built with appropriate System ILA instances, you can open and analyze the Vivado design by performing the following steps:

1. To confirm which signals can be debugged, navigate to **Debug/Release** → **\_sds** → **p0** → **vivado** → **prj** folder in the **Project Explorer**.



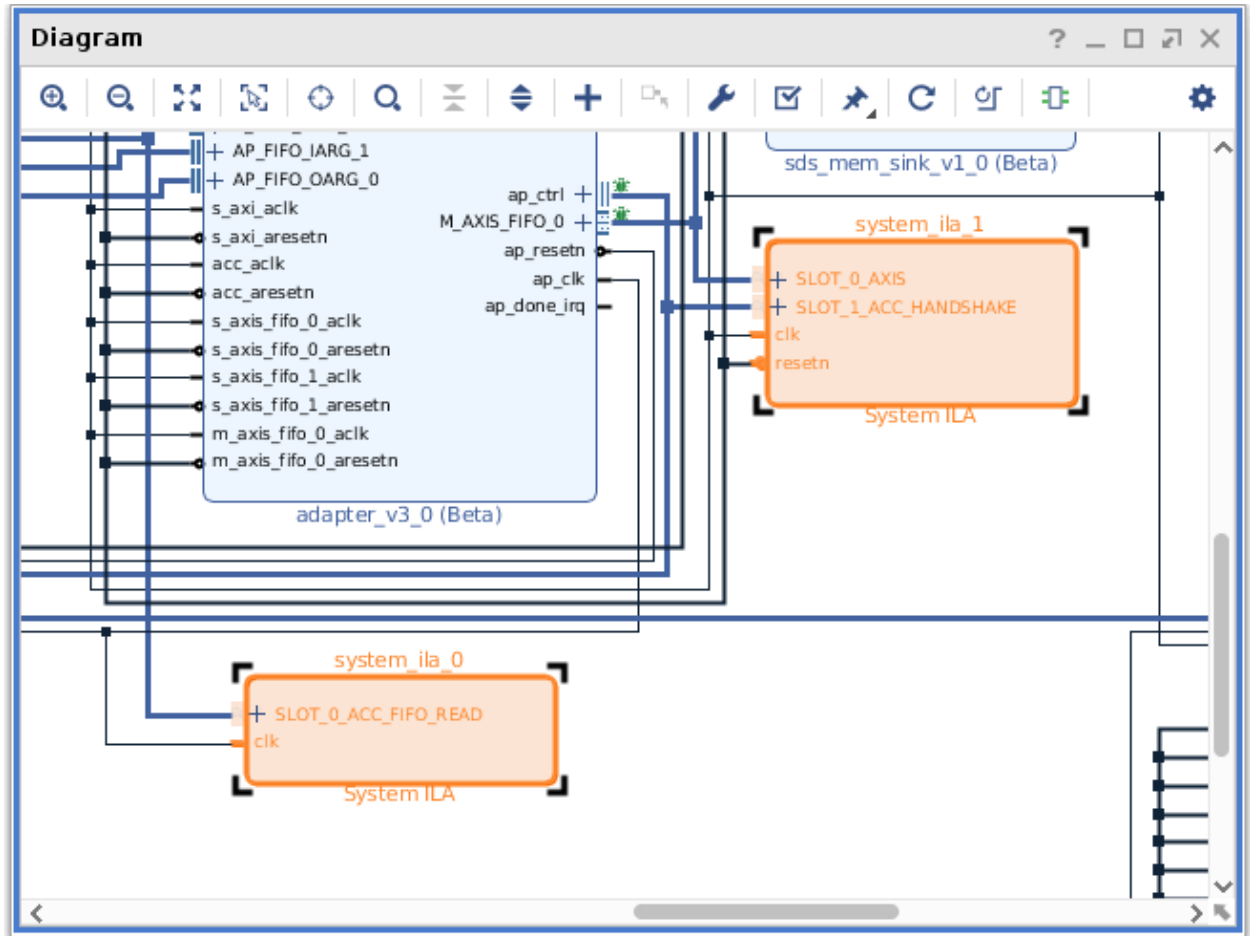
2. Double-click **prj.xpr**, which opens the design in the Vivado IDE.
3. In the Vivado IDE, click **Open Block Design** in the Flow Navigator under IP integrator.



4. In the Designs window, look for the instances of `system_ila_x`.



5. Select the System ILA instance(s) in the **Design** window to highlight the instances in the block design.



6. Select the interface nets connected to the System ILA and ensure that they have been connected to the interfaces specified in the SDx IDE.

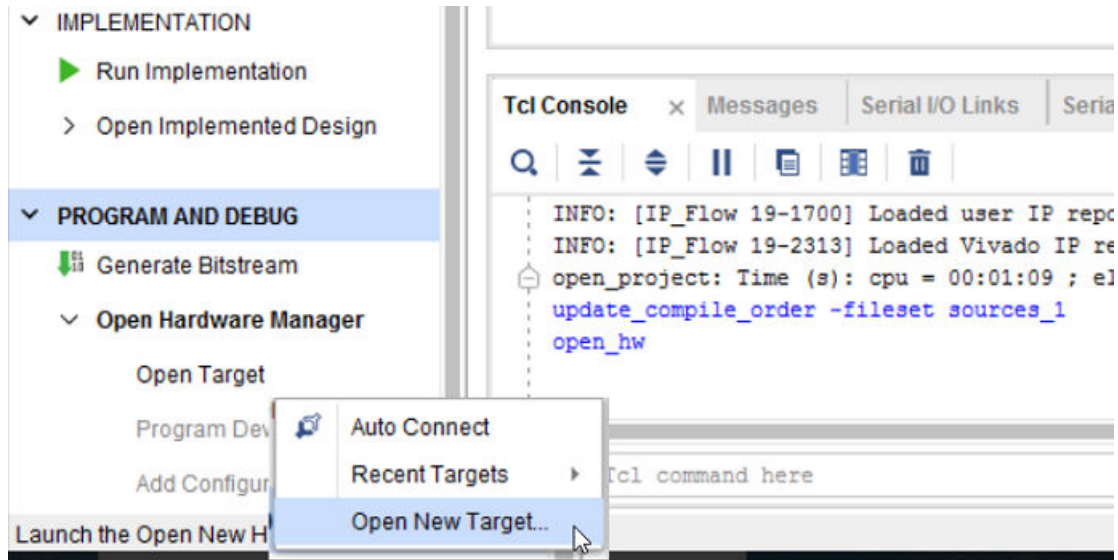
## Debugging Designs Using Vivado Hardware Manager

After you instrument the SDx environment application to insert debug cores, the next step is to connect to the Vivado hardware manager feature and look at Integrated Logic Analyzer (ILA) core transactions. To connect to the target board using the hardware manager, perform the following steps:

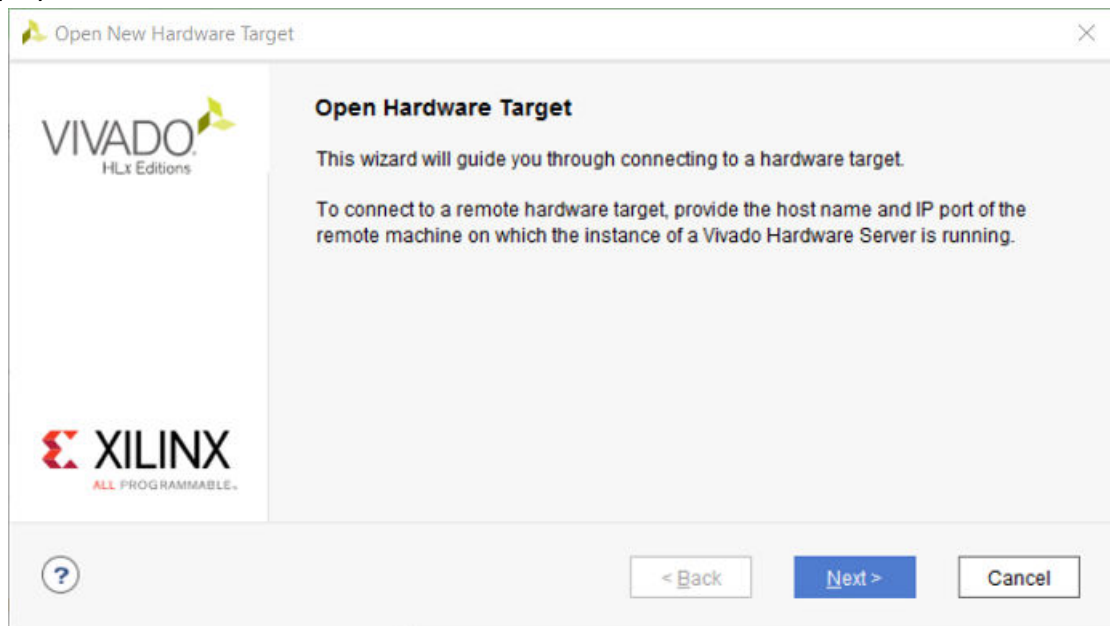
1. Launch the Vivado Design Suite.
2. Select **Open Hardware Manager** from the **Tasks** menu. An alternate method is to open the Vivado project from the SDx IDE:

```
<application_project_name>/Debug/_sds/p0/vivado/prj/prj.xpr
```

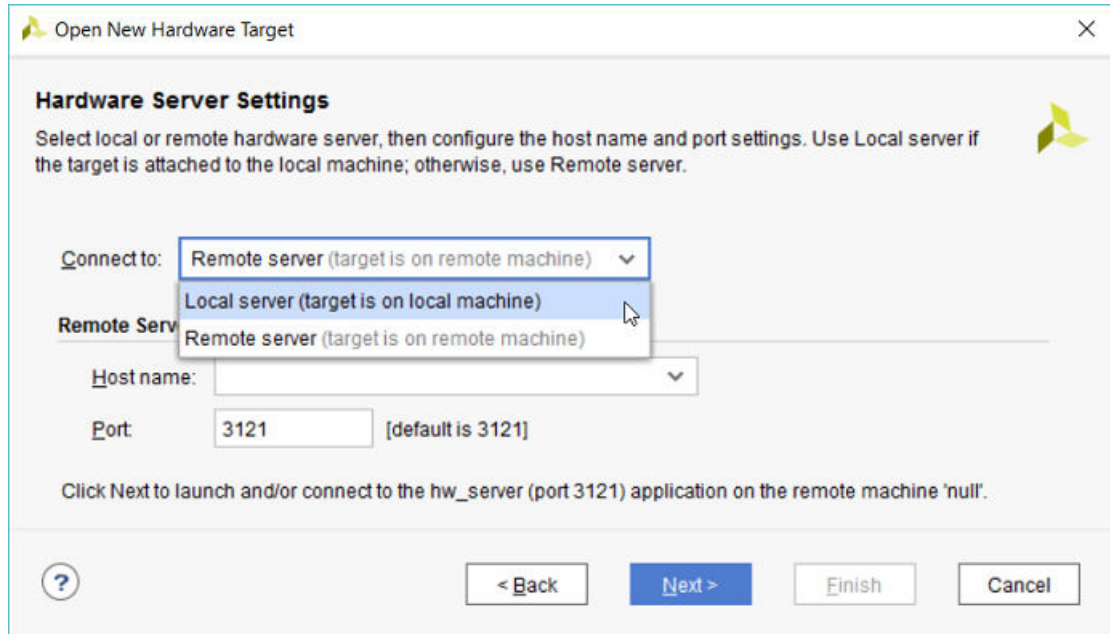
Then, from the Vivado Flow Navigator, click **Program and Debug** → **Open Hardware Manager** → **Open Target** → **Open New Target**, as shown below.



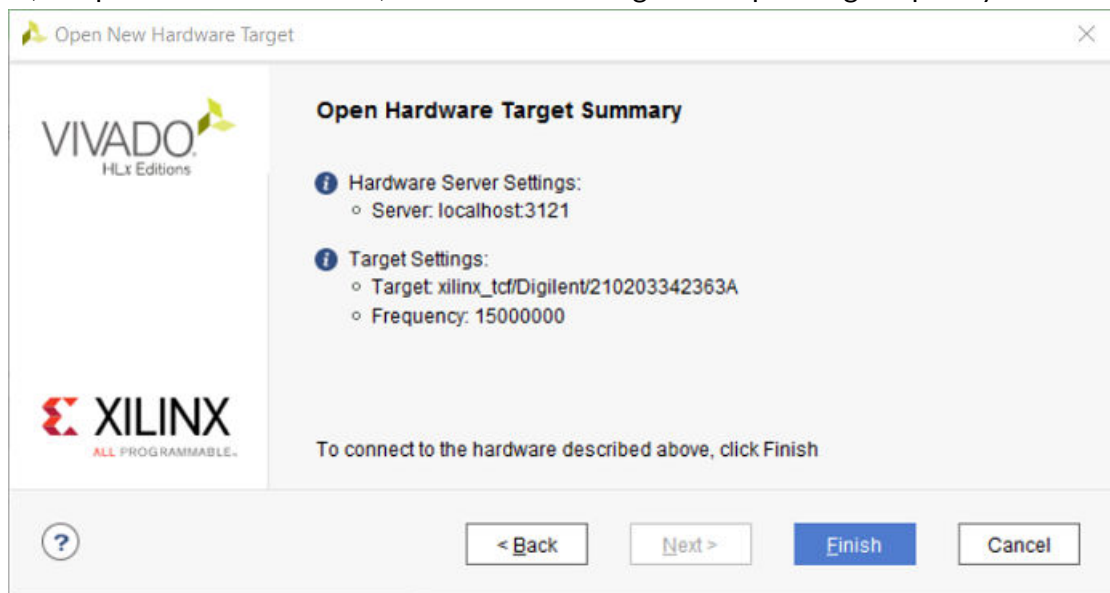
- For either method you used to open the project, the Open New Hardware Target wizard is displayed as shown below. Click **Next**.



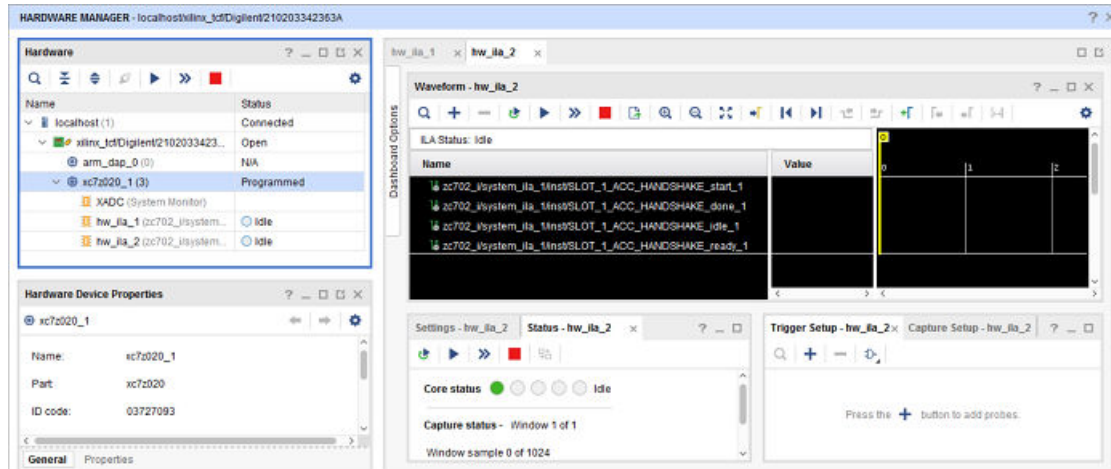
- In Hardware Server Settings, connect to the correct target by clicking **Connect to**, and then selecting either **Remote Server** or **Local Server**. If you select **Remote Server**, you need to add a **Host name** and the correct **Port** number. The following example assumes that you are connected locally:



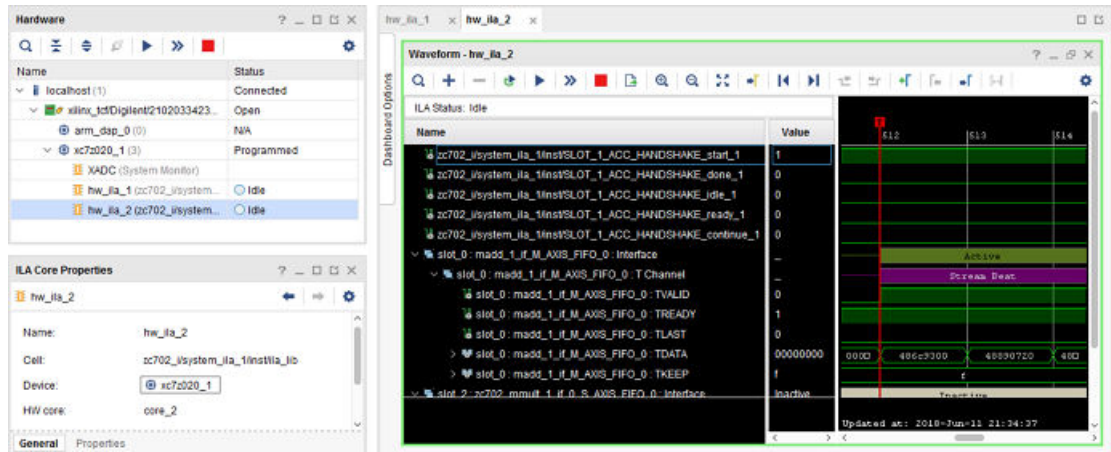
5. Click **Next**. The Select Hardware Target page opens which identifies the target(s) present on the board.
6. Click **Next**. The Open Hardware Target Summary page opens which summarizes the server name, the port it is connected to, and the correct target and operating frequency.



7. Click **Finish**. The Hardware Manager window opens as shown below.



- The Vivado hardware manager can now be used to connect to the ILA that is running on your design. Refer to the *Vivado Design Suite User Guide: Programming and Debugging (UG908)* for more information on working with the tool.



## Hardware/Software Event Tracing

Event tracing provides visibility into each phase of the hardware function execution, including the software setup for the accelerators and data transfers, as well as the hardware execution of the accelerators and data transfers. Tracing an application produces a log that records correlation between events for a duration of time. The goal of tracing is to help debug execution by observing what happened when, and how long events took.

Software event tracing automatically instruments the stub of the hardware function to capture software control events associated with a hardware function call. The event types that are recorded include the setup and initialization of the hardware accelerator, data transfers, and hardware-software synchronization events.



Hardware event tracing of accelerators with data transfers over AXI4-Stream connections can also be enabled through the use of the `-trace` option of the `sds++` system compiler. When the linker is invoked with the `-trace` option, it inserts hardware monitor IP cores into the RTL implementation of the hardware function to track the accelerator start and stop, and the duration of data transfers.

As with hardware debugging, event tracing requires you to connect the SDSoC environment platform to a host computer as described in [Connecting to the Hardware](#). To run event tracing, execute the application using the SDx IDE from the host using a debug or release build configuration.

## Hardware/Software System Runtime Operation

The system compiler implements hardware functions either by cross-compiling them into IP using the Vivado High-Level Synthesis (HLS) tool, or by linking them as C-callable IP, as described in the *SDSoC Environment Platform Development Guide* ([UG1146](#)).

Each hardware function call site is rewritten to call a stub function that manages the execution of the hardware accelerator. The figure below shows an example of hardware function rewriting. The original user code is shown on the left. The code section on the right shows the hardware function calls rewritten with new function names.

**Figure 11: Hardware Function Call Site Rewriting**

<pre>int main(int argc, char* argv[]) {     float *A, *B, *C, *D, tmp1;     init(A, B, C, D);     mmult(A, B, tmp1);     madd(tmp1, C, D);     check(D); }</pre>	<pre>int main(int argc, char* argv[]) {     float *A, *B, *C, *D, tmp1;     init(A, B, C, D);     p0_mmult_0(A, B, tmp1);     p0_madd_0(tmp1, C, D);     check(D); }</pre>
--	--

X16743-040516

The stub function initializes the hardware accelerator, initiates any required data transfers for the function arguments, and then synchronizes hardware and software by waiting at an appropriate point in the program for the accelerator and all associated data transfers to complete. For example, if the hardware function `foo()` is defined in `foo.cpp`, you can view the generated rewritten code in `_sds/swstubs/foo.cpp` for the project build configuration. As an example, the stub code shown below replaces a user function marked for hardware. This function starts the accelerator, starts data transfers to and from the accelerator, and waits for those transfers to complete.

```
void _p0_mmult0(float *A, float *B, float *C) {
    switch_to_next_partition();
    int start_seq[3];
    start_seq[0] = 0x00000f00;
    start_seq[1] = 0x00010100;
    start_seq[2] = 0x00020000;
    cf_send_i(cmd_addr, start_seq, cmd_handle);
}
```

```

cf_wait(cmd_handle);
cf_send_i(A_addr, A, A_handle);
cf_send_i(B_addr, B, B_handle);
cf_receive_i(C_addr, C, C_handle);
cf_wait(A_handle);
cf_wait(B_handle);
cf_wait(C_handle);
    
```

Event tracing provides visibility into each phase of the hardware function execution, including the software setup for the accelerators and data transfers, as well as the hardware execution of the accelerators and data transfers. For example, the stub code below is instrumented for trace. Each command that starts the accelerator, starts a transfer, or waits for a transfer to complete is instrumented.

```

void_p0_mmult_0(float *A, float *B, float *C) {
    switch_to_next_partition(0);
    int start_seq[3];
    start_seq[0] = 0x00000f00;
    start_seq[1] = 0x00010100;
    start_seq[2] = 0x00020000;
    sds_trace(EVENT_START);
    cf_send_i(cmd_addr, start_seq, cmd_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(cmd_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(A_addr, A, A_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(B_addr, B, B_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_receive_i(C_addr, C, C_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(A_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(B_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(C_handle);
    sds_trace(EVENT_STOP);
}
    
```

## Software Tracing

Event tracing automatically instruments the stub function to capture software control events associated with the implementation of a hardware function call. The event types include the following:

- Accelerator setup and initiation
- Data transfer setup
- Hardware/software synchronization barriers (“wait for event”)

See *SDSoC Environment Programmers Guide* ([UG1278](#)) for more detail on these topics.

Each of these events is independently traced and results in a single AXI4-Lite write into the programmable logic, where it receives a time stamp from the same global timer as hardware events.

## Hardware Tracing

The SDSoC environment supports hardware event tracing of accelerators cross-compiled using Vivado High-Level Synthesis (HLS) tool, and data transfers over AXI4-Stream connections. When `sds++` is invoked with the `-trace` option, it automatically inserts hardware monitor IP cores into the generated system to log the following event types:

- Accelerator start and stop, defined by `ap_start` and `ap_done` signals.
- Data transfer start and stop, defined by AXI4-Stream handshake and `TLAST` signals.

Each of these events is independently monitored and receives a time stamp from the same global timer used for software events. If the hardware function explicitly declares an AXI4-Lite control interface using the following pragma, it cannot be traced because its `ap_start` and `ap_done` signals are not part of the IP interface:

```
#pragma HLS interface s_axilite port=foo
```

These debug cores use some hardware resources; less than 0.1% of the hardware resources available on a ZC706 board.

The AXI4-Stream monitor core has two modes: basic and statistics. The basic mode does just the start/stop trace event generation. The statistics mode enables an AXI4-Lite interface to two 32-bit registers. The register at offset 0x0 presents the word count of the current, on-going transfer. The register at offset 0x4 presents the word count of the previous transfer. As soon as a transfer is complete, the current count is moved to the previous register. By default, the AXI4-Stream core is configured in the basic mode.

In addition to the hardware trace monitor cores, the output trace event signals are combined by a single integration core. This core has a parameterizable number of ports (from 1–63), and can thus support up to 63 individual monitor cores (either accelerator or AXI4-Stream). The resource utilization of this core depends on the number of ports enabled, and thus the number of monitor cores inserted.

On a ZC706 platform, this can use between roughly 0.1-1.0 percent of the available hardware resources, and up to approximately 10% of the memories with the integration logic.

## Implementation Flow

During the implementation flow, when tracing is enabled, tracing instrumentation is inserted into the software code and hardware monitors are inserted into the hardware system automatically. The hardware system (including the monitor cores) is then synthesized and implemented, producing the bitstream. The software tracing is compiled into the regular user program.

Hardware and software traces are time-stamped in hardware and collected into a single trace stream that is buffered up in the programmable logic.

# Debug Techniques

This chapter describes the different styles of debugging techniques applicable to SDSoC™ applications. It highlights different approaches for software-based debugging and hardware-oriented techniques. In the software-based approaches, a full understanding of the implementation of the design in the FPGA is not required. However, this concept can only be extended to a certain degree, at which point a hardware-based detailed analysis should be performed. Highlighting pure software debugging techniques is not the intent of this document.

When debugging SDSoC applications, you can use the same methods and techniques as applications used for debugging standard C/C++. Most SDSoC applications consist of specific functions tagged for hardware acceleration and surrounded by standard C/C++ code.

When debugging an SDSoC application with a board attached to the debug host machine, you can right-click on a build configuration in the Assistant view, and select the **Debug → Launch on Hardware** option to begin a debug session.

You can select options other than the default settings by using the **Debug → Debug Configurations** command to create a new custom debug configuration. As the debug environment is initialized, Xilinx recommends that you switch to the Debug perspective when prompted. The debug perspective view provides the ability to debug the standard C/C++ portions of the application by single-stepping code, setting and removing breakpoints, displaying variables, dumping registers, viewing memory, and controlling the code flow with “run until” and “jump to” debugging directives. Inputs and outputs can be observed before and after the function call to determine the correct behavior.

You can determine if a hardware accelerated application meets its real-time requirements by placing debug statements to start and stop a counter just before and just after a hardware accelerated function. The SDx™ environment provides the `sds_clock_counter()` function, which is typically used to calculate the elapsed time for a hardware accelerated function.

You can also perform debugging without a target board connected to the debug host by building the SDx project for emulation. During emulation, you can control and observe the software and data just as before through the debug perspective view, but you can also view the hardware accelerated functions through a Vivado® simulator waveform viewer. You can observe accelerator signaling for conditions such as accelerator start and accelerator done, and you can monitor data buses for inputs and outputs. Building a project for emulation also avoids a possibly long Vivado implementation step to generate an FPGA bitstream.

See the *SDSoC Environment Debugging Guide (UG1282)* for information on using the interactive debuggers in the SDx IDE.

## Debugging System Hangs and Runtime Errors

Programs compiled using `sds++` can be debugged using the standard debuggers supplied with the SDx environment or Vivado. Typical runtime errors are incorrect results, premature program exits, and program hangs. The first two kinds of error are familiar to C/C++ programmers, and can be debugged by stepping through the code using a debugger.

**Note:** Applications might hang when you are running on the board. Hangs commonly happen due to a mismatch of data size between the producer and the consumer.

A program hang is a runtime error caused by specifying an incorrect amount of data to be transferred across a streaming connection created using `#pragma SDS data access_pattern(A:SEQUENTIAL)`, by specifying a streaming interface in a synthesizable function within the Vivado High-Level Synthesis (HLS) tool, or by a C-callable hardware function in a pre-built library that has streaming hardware interfaces. A program hangs when the consumer of a stream is waiting for more data from the producer, but the producer has stopped sending data. Consider the following code fragment that results in streaming input/output from a hardware function:

```
#pragma SDS data access_pattern(in_a:SEQUENTIAL, out_b:SEQUENTIAL)
void f1(int in_a[20], int out_b[20]); // declaration

void f1(int in_a[20], int out_b[20]) { // definition
    int i;
    for (i=0; i < 19; i++) {
        out_b[i] = in_a[i];
    }
}
```

`In_a[]` has 20 elements, but the loop only reads 19 of them. Anything calling `f1` would appear to hang, waiting indefinitely for `f1` to consume the final element. Program errors that lead to hangs can be detected by using system emulation to ascertain whether the data signals are static by reviewing the associated protocol signals such as `TLAST`, `ap_ready`, `ap_done`, and `TREADY`. Program errors causing hangs can also be detected by instrumenting the code to flag streaming access errors such as non-sequential access or incorrect access counts within a function and running in software. Streaming access issues are typically flagged as `improper streaming access` warnings in the log file, and you can determine if these are actual errors. Running your application on the SDSoC emulator is a good way to gain visibility of data transfers with a debugger. You can see where in software the system is hanging (often within a `cf_wait()` call), and can then inspect associated data transfers in the simulation waveform view, which gives you access to signals on the hardware blocks associated with the data transfer.

## System Hang Debugging Example

As another example, consider the following code that results in streaming input/output from the hardware function:

```
#pragma SDS data access_pattern(in:SEQUENTIAL, out:SEQUENTIAL)
#pragma SDS data copy(in[0:large], out[0:small])
void too_large_copy(int* in, int* out, int small, int large)
{
    for(int i = 0; i < small; i++) {out[i] = in[i];}
}

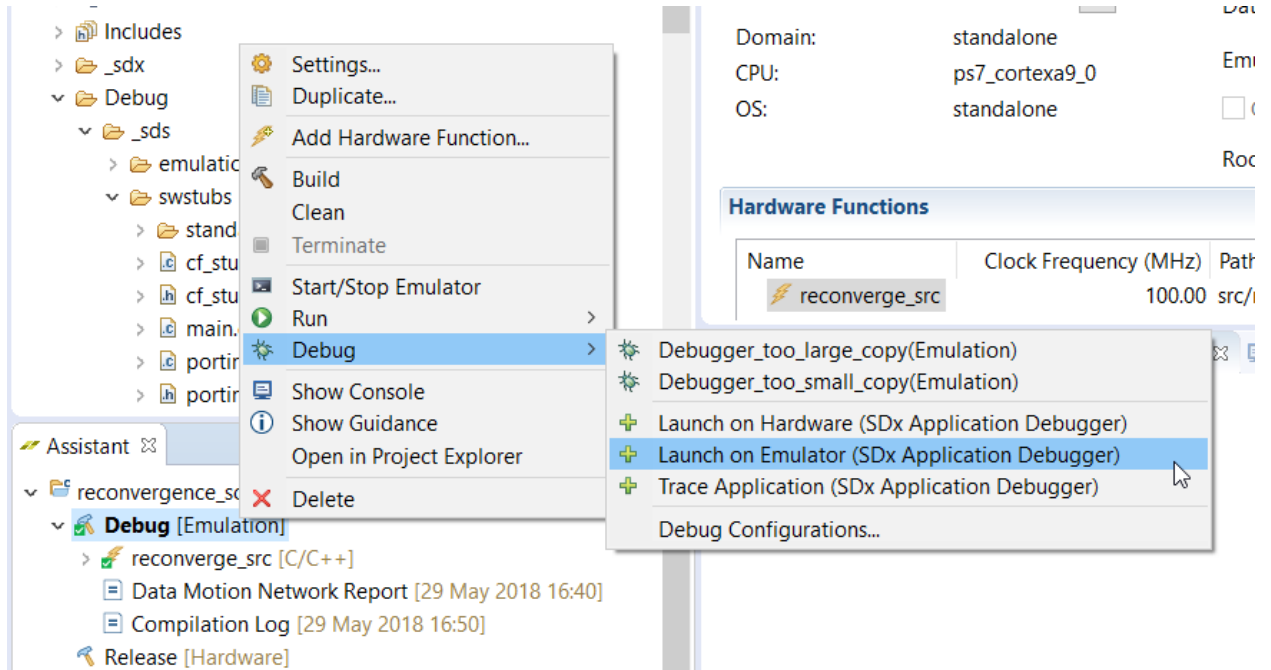
int main()
{
    int* temp_var1 = new int[1024 * 1024];
    int* temp_var2 = new int[1024 * 1024];

    too_large_copy(temp_var1, temp_var2, 1024, 1024 * 1024); //hangs
    because the input DMA continues to try to feed data to a halted HLS core
}

```

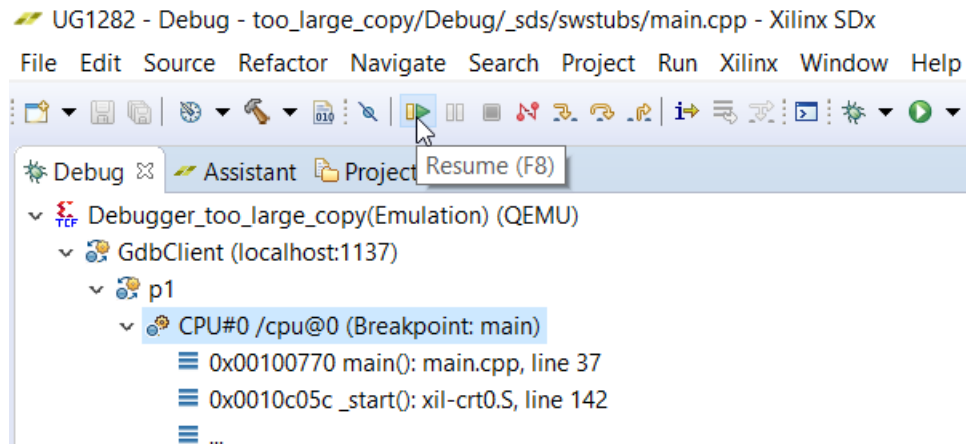
In this case, the direct memory access (DMA) continues to try to send data to the hardware function, whereas the hardware function is already done and is not accepting any data. This results in a system hang.

1. To debug this type of issue, build the code for emulation on the base platform. When the application is compiled, start the emulator by selecting **Xilinx → Start/Stop Emulator**. Alternatively, you can start the emulator from the **Assistant** window as shown below. Right-click the **Active build configuration** for the application and select **Start/Stop Emulator**.
2. In the Emulation dialog box, ensure that the **Show Waveform (Programmable Logic only)** check box is checked. This brings up the Vivado Simulator where the state of different interfaces can be viewed in the Waveform window. To monitor the interfaces of the hardware function, right-click on the function and select **Add to Wave window**. This adds all the I/O ports of the selected function to the Waveform window.
3. Start the simulator by clicking the **Run All** icon in the toolbar.
4. Go back to the SDx IDE, and then launch the application on the debugger. To do this, select the application to be debugged, right-click, and then select **Launch on Emulator (SDx Application Debugger)**.



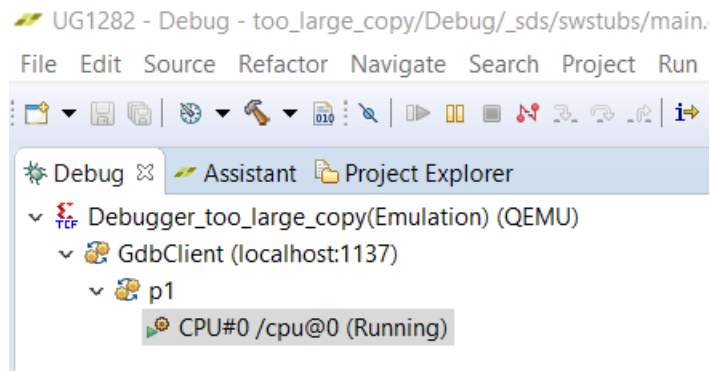
In the Confirm Perspective Switch dialog box, click **Yes**. The Debug Perspective opens with the application running on the hardware. The code execution stops at the main program entry.

5. Click the **Resume** button on the toolbar to execute the application.

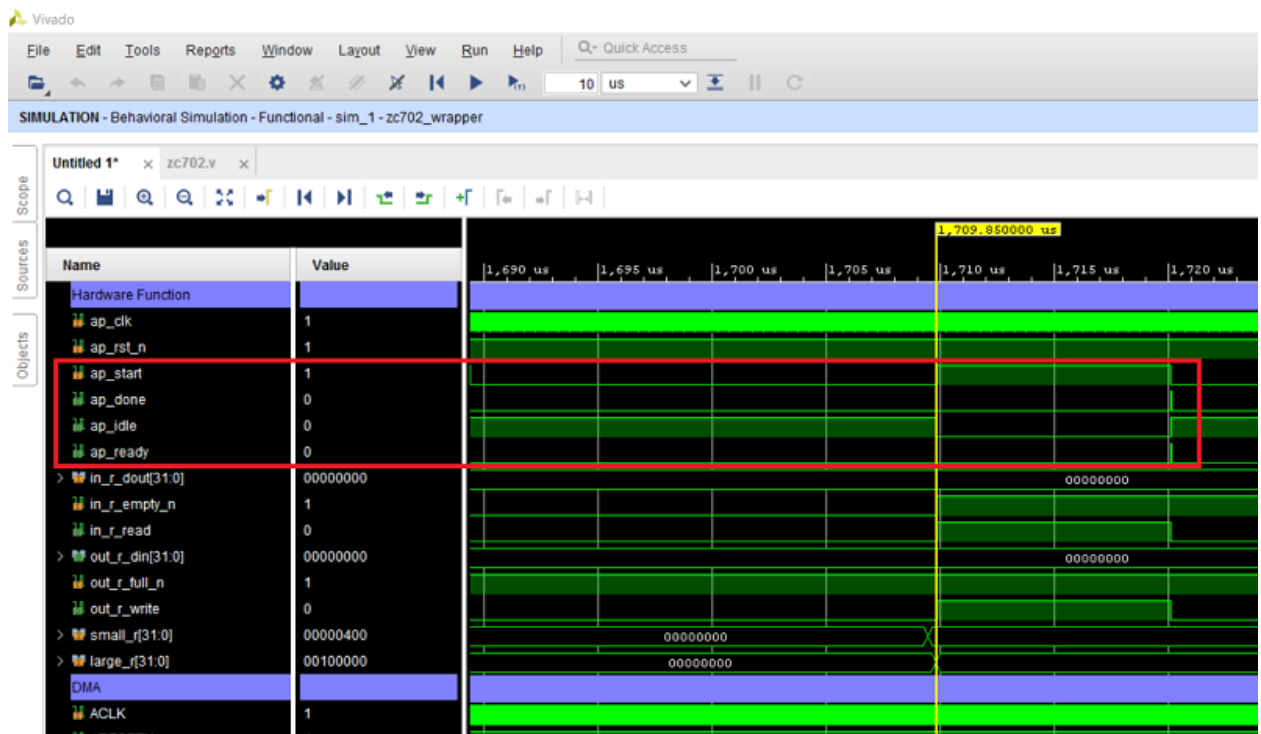


The application is now stuck: a system hang has been encountered.

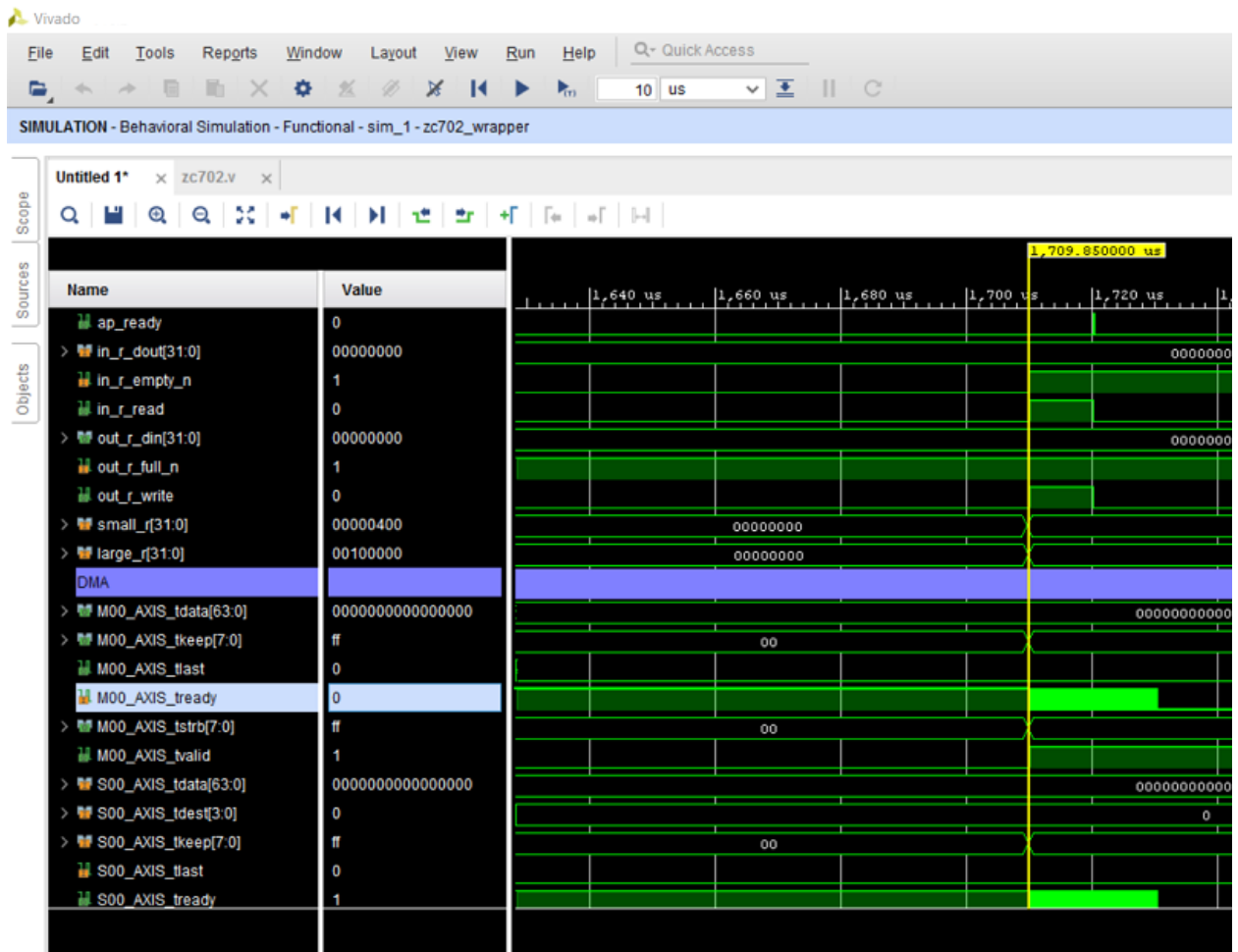




- To determine the cause of the system hang, go back to Vivado Design Suite. Look at the state of the `ap_done`, `ap_start`, `ap_idle` and `ap_ready` signals for the function. The state of these signals indicates that a transaction was started at the instance when the `ap_start` signal went High, followed by the transaction ending when the `ap_done` signal went Low. The `ap_ready` and `ap_idle` signals likewise indicate the state of the function.



Analyzing the state of the DMA at the same point of time, you can see that while the hardware function has finished accepting data, the DMA is still writing to it, as indicated by the `M00_AXIS_tready` and the `M00_AXIS_tvalid` signals.



Now that you know the cause of the system hang, you can go back to the hardware function code and fix any outstanding issues.

## Causes of System Hangs

There are other situations where a system hang can occur as listed below:

1. If you can **Ctrl+C** out of the application, there was probably not enough data from the accelerator. The Arm® processor is expecting more data than the accelerator is sending. Review latencies if there is more than one path from a producer to a consumer. Designs where there are multiple paths with equal latencies between two accelerators (for example, A -> B ... -> Z, while there is also A -> Z direct) need to be fixed at the design level equalizing the branches.
2. If **Ctrl+C** does not work, but you can `ping` or `ssh` into the board, there is not enough data in a Scatter Gather DMA (SGDMA) operation. Review the data movers (copy or zero-copy) and the access pattern.
3. If you cannot `ping` the board and it has hard locked, only coming back to life after a power cycle, common causes are interaction between the following:

- a. The SDSoC environment design and IP on the platform. Debug with the ChipScope™ feature and peeking and poking of registers; see [Hardware Debugging in SDSoC Using ChipScope](#) and [Peeking and Poking IP Registers](#).
- b. The SDSoC environment design and C-callable IP libraries. Debug with the ChipScope feature and peeking and poking of registers; see [Hardware Debugging in SDSoC Using ChipScope](#) and [Peeking and Poking IP Registers](#).
- c. The RTL or the SW driver generated in the SDSoC flow. If you have enough Vivado Design Suite or C driver experience you might be able to debug this; otherwise, contact the [Xilinx forums](#).

## Causes of Runtime Errors

The following list shows other sources of runtime errors:

- Improper placement of `wait()` statements could result in the following issues:
  - The software might read invalid data before a hardware accelerator has written the correct value.
  - A blocking `wait()` might be called before a related accelerator is started, resulting in a system hang.
- Inconsistent use of the memory consistency `SDS data mem_attribute` pragma can result in incorrect results.

### *Unexpected Data Values*

When the application is running, it is possible to get unexpected data. The hardware function might not be returning the expected data, or it might be returning expected data at the wrong time. This can be caused by hardware and/or software issues. If hardware is the suspected root cause, check data inputs to your board using the ChipScope feature if needed. If software is the suspected root cause, perform the following steps:

1. Go back to software debug and confirm that your software is good.
2. If the software debug is good, you need to visually inspect the code. Two common causes for unexpected data are from the use of the `#SDS data` or the `#SDS zero copy` pragmas.
3. If you are using `#SDS data` pragmas, the tools trust what you write. Confirm that the data access pattern in the code matches the data access pattern specified by the pragma.
4. An incorrectly sized (normally too large) `#SDS zero copy` can pull invalid data from cache. This is seen in hardware. Emulation is likely to pass as there is no cache controller in software.

# Peeking and Poking IP Registers

With the Xilinx® System Debugger tool (XSDB), you can understand what is happening with the IP blocks included with the platform or the various C-callable IP blocks. From the Xilinx Software Command Line Tool (XCST) console, you can read and write registers within various IP blocks in the integrated design. Registers can be read by typing the memory read command, `mrd`. Likewise, a writable register in any IP in the design can be written to by typing the `mwr` command in the XCST console. For help with commands, type `<command> -help`.

You need to be familiar with the memory map of the various IP blocks within the design to be able to perform reads and writes to the registers. You can access this information by opening the Vivado project and looking at the address editor. The Vivado project can be found at `<project_name>/<Debug or Release>/_sds/p0/vivado/prj/prj.xpr`. Double-clicking `prj.xpr` opens up the project in Vivado. In the Vivado IDE, click on **IP Integrator** → **Open Block Design** under **Flow Navigator**. Click on the **Address Editor** tab to view the memory map information.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
Data (40 address bits : 0x00A0000000 [ 256M ] , 0x0400000000 [ 4G ] , 0x1000000000 [ 224G ])					
dm_0	S_AXI_LITE	Reg	0x00_A000_0000	4K	0x00_A000_0FFF
dm_1	S_AXI_LITE	Reg	0x00_A000_1000	4K	0x00_A000_1FFF
dm_2	S_AXI_LITE	Reg	0x00_A000_2000	4K	0x00_A000_2FFF
dm_3	S_AXI_LITE	Reg	0x00_A000_3000	4K	0x00_A000_3FFF
madd_1_if	S_AXI	reg0	0x00_A001_0000	64K	0x00_A001_FFFF
mmult_1_if	S_AXI	reg0	0x00_A002_0000	64K	0x00_A002_FFFF
dm_0					
Data_MM2S (36 address bits : 64G)					
ps_e	S_AXI_HP2_FPD	HP2_DDR_HIGH	0x8_0000_0000	32G	0xF_FFFF_FFFF
ps_e	S_AXI_HP2_FPD	HP2_DDR_LOW	0x0_0000_0000	2G	0x0_7FFF_FFFF
ps_e	S_AXI_HP2_FPD	HP2_PCIE_LOW	0x0_E000_0000	256M	0x0_EFFF_FFFF
ps_e	S_AXI_HP2_FPD	HP2_QSPI	0x0_C000_0000	512M	0x0_DFFF_FFFF
Excluded Address Segments (1)					
ps_e	S_AXI_HP2_FPD	HP2_LPS_OCM	0x0_FF00_0000	16M	0x0_FFFF_FFFF
dm_1					
Data_MM2S (36 address bits : 64G)					
ps_e	S_AXI_HP1_FPD	HP1_DDR_HIGH	0x8_0000_0000	32G	0xF_FFFF_FFFF
ps_e	S_AXI_HP1_FPD	HP1_DDR_LOW	0x0_0000_0000	2G	0x0_7FFF_FFFF

For details on XSDB, refer to *SDK Online Help* ([UG782](#)).



**CAUTION!** Trying to access an address that is not mapped results in a **BUS ERROR**. Addresses that are mapped, but lack proper backing, result in a system hang.

# Event Tracing

This section describes how traces are collected and displayed in the SDSoC environment.

## Runtime Trace Collection

Software traces are inserted into the same storage path as the hardware traces and receive a time stamp using the same timer/counter as hardware traces. This single-trace data stream is buffered in the hardware system and accessed over JTAG by the host PC.

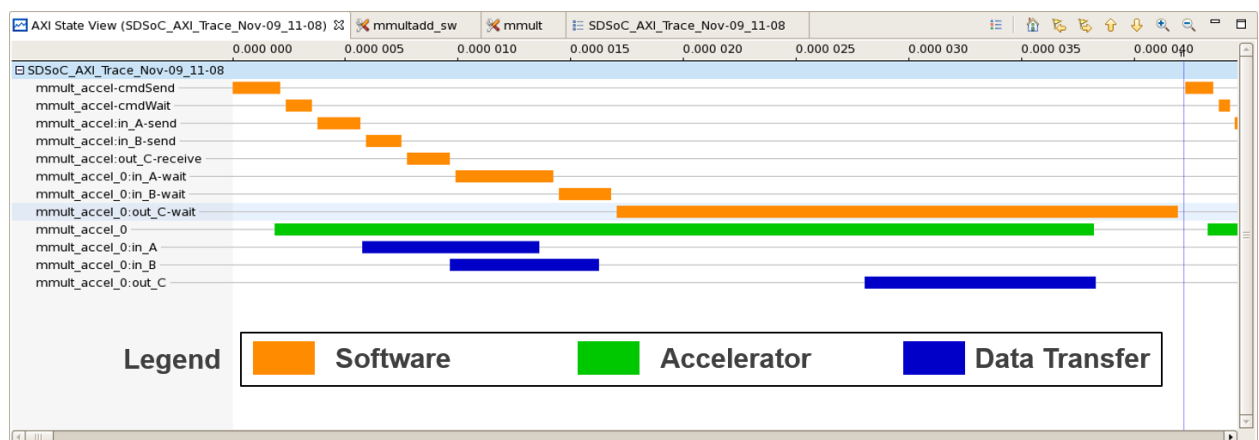
In the SDSoC environment, traces are read back constantly while the program executes attempting to empty the hardware buffer as quickly as possible and prevent buffer overflow. However, trace data only displays when the application is finished.

Trace data is collected in real time when you are running on the hardware. For information about connecting to the hardware, refer to [Connecting to the Hardware](#).

## Trace Visualization

The SDSoC environment displays a graphical rendering of the hardware and software trace stream. Each trace point in the user application is given a unique name, and its own axis on the timeline. In general, a trace point can create multiple trace events throughout the execution of the application, for example, if the same block of code is executed in a loop, or if an accelerator is invoked more than once.

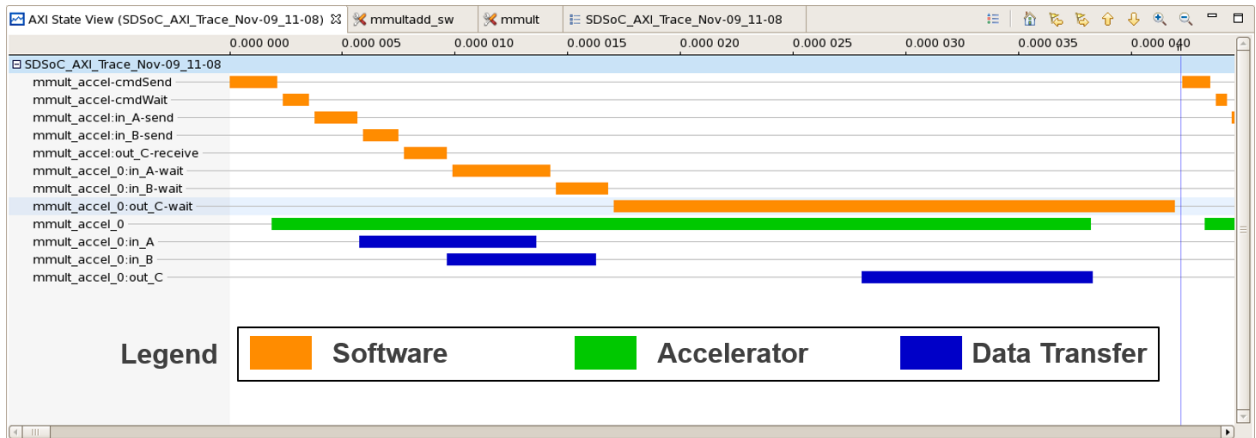
**Figure 12: Example Trace Visualization Highlighting the Different Types of Events**



X22041-112718

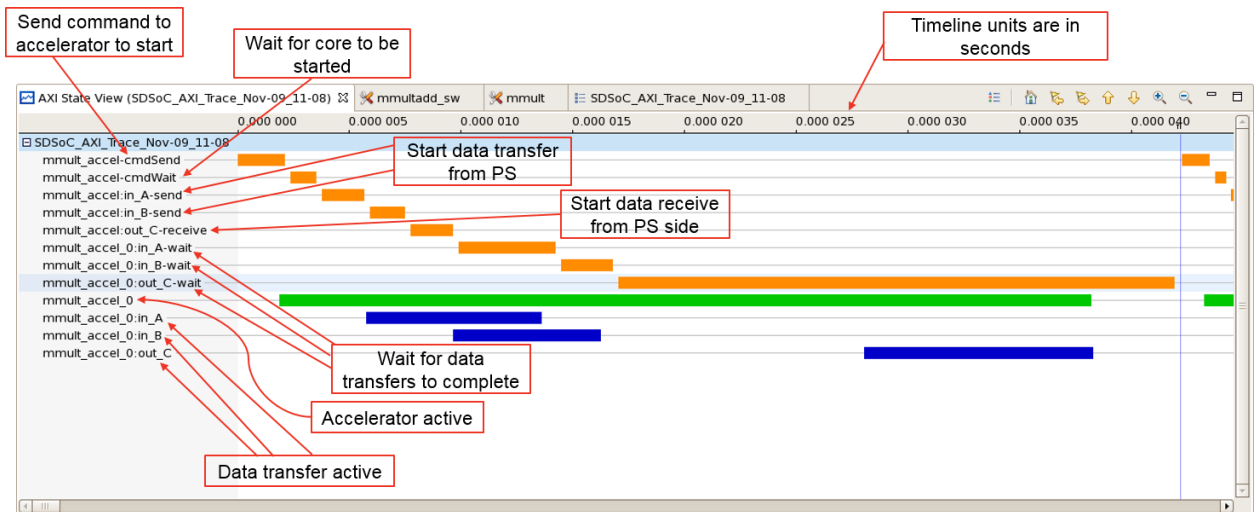
Each trace event has a few different attributes: name, type, start time, stop time, and duration. This data is shown as a tool-tip when the cursor hovers above one of the event rectangles in the view.

Figure 13: Example Trace Visualization Highlighting the Detailed Information Available for Each Event



X22041-112718

Figure 14: Example Trace Visualization Highlighting the Event Names and Correlation to the User Program



X22040-112718

## Troubleshooting

The following section provides general information on troubleshooting the different conditions encountered during event tracing.

- Incremental build flow:** The SDSoC environment does not support any incremental build flow using the trace feature. To ensure the correct build of your application and correct trace collection, do a project clean first, followed by a build after making any changes to your source code. Even if the source code you change does not relate to or impact any function marked for hardware, you can see incorrect results.

- **Programming and bitstream:** The trace functionality is a single-use type of analysis. The timer used for time-stamping events is not started until the first event occurs, and runs indefinitely afterward. If you run your software application once after programming the bitstream, the timer is in an unknown state after your program is finished running. Running your software for a second time results in incorrect timestamps for events. Be sure to program the bitstream first, followed by downloading your software application, each and every time you run your application to take advantage of the trace feature. Your application will run correctly a second time, but the trace data will not be correct. For Linux, you need to reboot because the bitstream is loaded during boot time by U-Boot.
- **Buffering up traces:** In the SDSoC environment, traces are buffered up and read out in real time as the application executes (although at a slower speed than they are created on the device), but are displayed after the application finishes in a post-processing fashion. This relies on having enough buffer space to store traces until they can be read out by the host PC. By default, there is enough buffer space for 1024 traces. After the buffer fills up, subsequent traces that are produced are dropped and lost. An error condition is set when the buffer overflows. Any traces created after the buffer overflows are not collected, and traces just prior to the overflow might be displayed.
- **Errors:** In the SDSoC environment, traces are buffered up in hardware before being read out over JTAG by the host PC. If traces are produced faster than they are consumed, a buffer overflow event might occur. The trace infrastructure recognizes this and sets an error flag that is detected during the collection on the host PC. After the error flag is parsed during trace data collection, collection is halted and the trace data that was read successfully is prepared for display. However, some data read successfully just prior to the buffer overflow might appear incorrectly in the visualization.

After an overflow occurs, an error file is created in the `<build_config>/_sds/trace` directory with the name in the following format: `archive_DAY_MON_DD_HH_MM_SS_GMT_YEAR_ERROR`. You must reprogram the device (reboot Linux and so on) prior to running the application and collecting trace data again. The only way to reset the trace hardware in the design is with reprogramming.

---

## Debugging with Software/Hardware Cross Probing

After an SDx environment application has been created and functions are marked for hardware acceleration, build the design with the appropriate settings. Then, connect to the target board (see [Connecting to the Hardware](#)).

### Setting Debug Configurations

1. In the Project Explorer view, click the ELF (`.elf`) file in the `Debug` folder in the project.

2. In the toolbar, click **Debug**, or use the **Debug** drop-down list to select **Debug As → Launch on Hardware (SDx Application Debugger)**.
3. Alternatively, right-click the project and select **Debug As → Launch on Hardware (SDx Application Debugger)**. The Confirm Perspective Switch dialog box appears.
4. Ensure that the board is switched on before debugging the project. Click **Yes** to switch to the debug perspective. You are now in the **Debug Perspective** of the SDx IDE.

**Note:** The debugger resets the system, programs and initializes the device, and then breaks at the main function. The source code is shown in the center panel, and local variables are shown in the top right corner panel. The SDx environment log at the bottom right panel shows the Debug Configuration log.

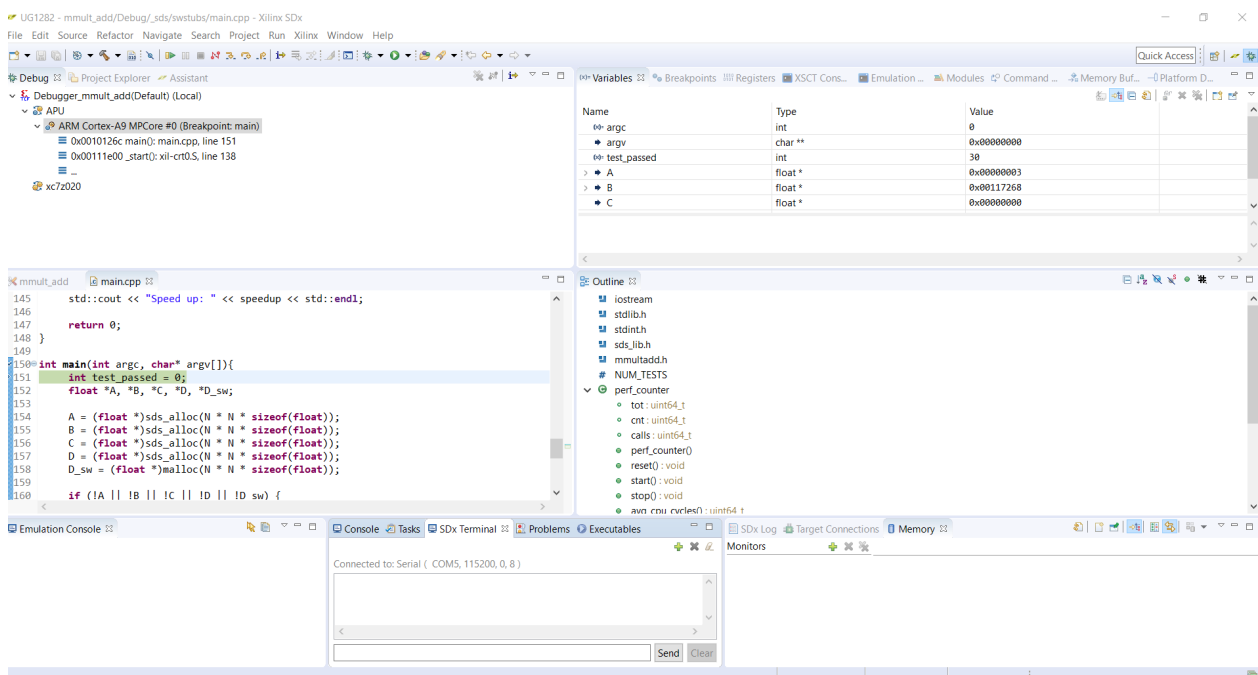
Before you run the application, connect a serial terminal to the board so that you can see the output from your program. As an example, the following settings can be used:

- **Connection Type:** Serial
- **Port:** COM<n>
- **Baud Rate:** 115200

## Running the Application

Click **Resume** to run your application and observe the output in the terminal window. The source code window shows the `_exit` function, and the **Terminal** tab shows the output from the application.

Figure 15: Debug Perspective





The code stops execution at the main function, as can be seen in the Debug tab. Additional breakpoints can be set in the code at specific points to stop the execution of the code at that specific point. Breakpoints can be enabled or disabled by double-clicking on the vertical blue bar adjacent to the line numbers in the code. Execution of the code can be resumed by clicking the **Resume** icon on the toolbar.

---

## Tips for Debugging Performance

The SDSoC environment provides some basic performance monitoring capabilities with the following functions:

- `sds_clock_counter()`: Use this function to determine how much time different code sections, such as the accelerated code and the non-accelerated code, take to execute.
- `sds_clock_frequency()`: This function returns the number of CPU cycles per second.

You can estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado Design Suite High-level Synthesis (HLS) tool report files (`_sds/vhls/.../*.rpt`) or in the IDE under **Reports** → **HLS Report**. The latency of X accelerator clock cycles equals  $X * (\text{processor\_clock\_freq}/\text{accelerator\_clock\_freq})$  processor clock cycles. Compare this with the time spent on the actual function call to determine the overhead of setup and data transfers.

For best performance improvement, the time required for executing the accelerated function must be much smaller than the time required for executing the original software function. If this is not true, try to run the accelerator at a higher frequency by selecting a different `clkid` on the `sds++` command line. If that does not work, try to determine whether the data transfer overhead is a significant part of the accelerated function execution time, and reduce the data transfer overhead. Note that the default `clkid` is 100 MHz for all platforms. More details about the `clkid` values for the given platform can be obtained by running `-sds-pf-info <path>/<platform_name>`.

If the data transfer overhead is large, the following changes might help:

- Move more code into the accelerated function so that the computation time increases, and the ratio of computation to data transfer time is improved.
- Reduce the amount of data to be transferred by modifying the code or using pragmas to transfer only the required data.
- Sequentialize the access pattern as observed from the accelerator code, because it is more efficient to burst transfers than to make a series of unrelated random accesses.

- Ensure that data transfers make use of system ports that are appropriate for the cache-ability of the data being transferred. Cache flushing can be a resource-intensive procedure, and using coherent ports to access coherent data, and non-coherent ports to access non-coherent ports makes a significant impact.

Use `sds_alloc()` instead of `malloc`, where possible. The memory that `sds_alloc()` issues is physically contiguous, and enables the use of data movers that are faster to configure that require physically contiguous memory. Also, pinning virtual pages, which is necessary when transferring data issue by `malloc()` data, is very costly.

---

## Troubleshooting Compile and Link Time Errors

Typical compile/link time errors are indicated by error messages issued when running `make`. To analyze further, look at the log files and `rpt` files in the `_sds/reports` sub-directory created by the SDSoC environment in the build directory. The most recently generated log file usually indicates the cause of the error, such as a syntax error in the corresponding input file, or an error generated by the tool chain while synthesizing accelerator hardware or the data motion network.

The following are tips and strategies to address errors specific to the SDSoC environment.

### Tool Errors Are Reported by Tools in the SDSoC Environment Chain

Try the following troubleshooting steps:

- Check whether the corresponding code adheres to the Coding Guidelines in *SDSoC Environment Programmers Guide* ([UG1278](#)).
- Check the syntax of pragmas. See the *SDx Pragma Reference Guide* ([UG1253](#)) for more details.
- Check for typos in pragmas that might prevent them from being applied as intended.

### Vivado Design Suite High-Level Synthesis (HLS) Cannot Meet Timing Requirement

Try the following troubleshooting steps:

- Select a slower clock frequency for the accelerator in the SDx IDE (or with the `sdscc/sds++` command line parameter).
- Modify the code structure to allow HLS to generate a faster implementation. See the Improving Hardware Function Parallelism section in *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#)) for more information on how to do this.

### Vivado Tools Cannot Meet Timing

Try the following troubleshooting steps:

- In the SDx IDE, select a slower clock frequency for the data motion network or accelerator, or both (from the command line, use `sdscc/sds++` command line parameters).
- Use the `-xp` option to specify a Vivado implementation strategy to improve results. For example:

```
-impl-strategy Performance_Explore
```

- Provide an example/resource to help the user synthesize the HLS block to a higher clock frequency so that the synthesis/implementation tools have a bigger margin.
- Modify the C/C++ code passed to HLS, or add more HLS directives to make the HLS block go faster.
- Reduce the size of the design in cases where the resource usage exceeds 80%. Refer to the Vivado tools reports in the `_sds` folder.

### The Design Is Too Large to Fit

Try the following troubleshooting steps:

- Reduce the number of accelerated functions.
- Change the coding style for an accelerator function to produce a more compact accelerator. You can reduce the amount of parallelism using the mechanisms described in the Improving Hardware Function Parallelism section in *SDSoC Environment Profiling and Optimization Guide (UG1235)*.
- Modify pragmas and coding styles (pipelining) that cause multiple instances of accelerators to be created.
- Use pragmas to select smaller data movers such as `AXIFIFO` instead of `AXIDMA_SG`.
- Rewrite hardware functions to have fewer input and output parameters/arguments, especially in cases where the inputs/outputs are continuous stream (sequential access array argument) types that prevent the sharing of data mover hardware.

---

## Troubleshooting Performance Issues

The SDSoC environment provides some basic performance monitoring capabilities in the form of the `sds_clock_counter()` function. Use this function to determine how much time different code sections, such as the accelerated and the non-accelerated code, take to execute.

To estimate the actual hardware acceleration time, you need to know the latency numbers from the Vivado HLS report, the clock frequency for the accelerator, and the Arm CPU clock frequency. To open the Vivado HLS report for the latency numbers, in the Assistant view, go to **<Project Name> → <Build Configuration> → <Accelerator Name> → HLS report**. To view the clock frequency for the accelerator, go to the Hardware Functions section of the Project Settings.

Click on the **Platform** link in the Project Overview to open the Platform Summary dialog. The CPU frequency is shown under Clock Frequencies. A latency of X accelerator clock cycles is equal to  $X * (\text{processor clock frequency} / \text{accelerator clock frequency})$  processor clock cycles. Compare this with the time spent on the actual function call to determine the data transfer overhead.

For best performance improvement, the time required for executing the accelerated function must be much smaller than the time required for executing the original software function. If this is not true, try to run the accelerator at a higher frequency by selecting a different `clkid` on the `sdscc/sds++` command line. If that does not work, try to determine whether the data transfer overhead is a significant part of the accelerated function execution time, and reduce the data transfer overhead.

**Note:** More details about the `clkid` values for a given platform can be obtained by running the following command:

```
sds++ -sds-pf-info
```

If the data transfer overhead is large, the following changes might help:

- Move more code into the accelerated function so that the computation time increases, and the ratio of computation to data transfer time is improved.
- Reduce the amount of data to be transferred by modifying the code or using pragmas to transfer only the required data.

# SDSoC Environment Troubleshooting

There are several common types of issues you might encounter using the SDSoC™ environment flow:

- Compile/link time errors might be the result of typical software syntax errors caught by software compilers, or errors specific to the SDSoC environment flow, such as the design being too large to fit on the target platform.
- Runtime errors might be the result of general software issues, such as null-pointer access, or issues specific to the SDSoC environment, such as incorrect data being transferred to/from accelerators.
- Performance issues are related to the choice of the algorithms used for acceleration, the time taken for transferring the data to/from the accelerator, and the actual speed at which the accelerators and the data motion network operate.
- Incorrect program behavior can be the result of logical errors in code that fails to implement algorithmic intent.

# Additional Resources and Legal Notices

---

## Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

**Note:** For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

---

## References

These documents provide supplemental material useful with this guide:

1. *SDSoC Environments Release Notes, Installation, and Licensing Guide* ([UG1294](#))
2. *SDSoC Environment User Guide* ([UG1027](#))
3. *SDSoC Environment Getting Started Tutorial* ([UG1028](#))
4. *SDSoC Environment Tutorial: Platform Creation* ([UG1236](#))
5. *SDSoC Environment Platform Development Guide* ([UG1146](#))

6. *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#))
7. *SDx Command and Utility Reference Guide* ([UG1279](#))
8. *SDSoC Environment Programmers Guide* ([UG1278](#))
9. *SDSoC Environment Debugging Guide* ([UG1282](#))
10. *SDx Pragma Reference Guide* ([UG1253](#))
11. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
12. *Zynq-7000 SoC Software Developers Guide* ([UG821](#))
13. *Zynq UltraScale+ MPSoC: Software Developers Guide* ([UG1137](#))
14. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide* ([UG850](#))
15. *ZCU102 Evaluation Board User Guide* ([UG1182](#))
16. *PetaLinux Tools Documentation: Reference Guide* ([UG1144](#))
17. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
18. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))
19. [SDSoC Development Environment web page](#)
20. [Vivado® Design Suite Documentation](#)

---

## Training Resources

1. [SDSoC Development Environment and Methodology](#)
2. [Advanced SDSoC Development Environment and Methodology](#)

---

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any

action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

### **AUTOMOTIVE APPLICATIONS DISCLAIMER**

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### **Copyright**

© Copyright 2018–2019 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. HDMI, HDMI logo, and High-Definition Multimedia Interface are trademarks of HDMI Licensing LLC. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.