

SDSoC Environment User Guide

UG1027 (v2017.4) January 26, 2018

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/27/2018	2017.4	Updated the following: <ul style="list-style-type: none"> Updated Elements of SDC to show how to open the Vivado® tool. Added links to Working with SDx in Chapter 1: The SDC Environment Added reference to SDC Environment Profiling and Optimization Guide (UG1235) in Selecting Functions for Hardware Acceleration Added information about <code>zero_copy</code> buses and Data Motion Clocks to Selecting Clock Frequencies Added content to Running System Emulation Updated Chapter 3: Profiling and Optimization Updated Chapter 4: Debugging an Application Added content to Debugging Performance Tips Added link to SDx Pragma Reference Guide (UG1253) to Chapter 5: Hardware/Software Event Tracing Updated commands in Appendix C: SDC/SD++ Compiler Commands and Options Added explanation of <code>-sds -hw</code> and <code>-sds -end</code> to Hardware Function Options. Updated Appendix E: Using C-Callable IP Libraries, Packaging RTL IP for SDx, and C-Callable Library Examples.
12/20/2017	2017.4	Major reorganization and rewrite of the content in this guide, including: <ul style="list-style-type: none"> Relocated Profiling and Optimization content to SDC Environment Profiling and Optimization Guide (UG1235). Rewrite of Chapter 4: Debugging an Application. Changed coding examples from <code>sdslib</code> to <code>sd_x_pack</code> Added a parameterized option to Appendix C: SDC/SD++ Compiler Commands and Options
08/16/2017	2017.2	<ul style="list-style-type: none"> Updated for SDx™ IDE 2017.2. Added Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc).
06/20/2017	2017.1	<ul style="list-style-type: none"> Updated for SDx IDE 2017.1. Added Appendix A: Getting Started with Examples.

Table of Contents

Revision History.....	3
Chapter 1: The SDSoC Environment.....	9
Getting Started.....	10
Elements of SDSoC.....	10
User Design Flows.....	11
Working with SDx.....	13
Chapter 2: Creating an SDSoC Project.....	17
Launching SDx.....	17
Creating an SDSoC Application Project.....	19
Importing Sources.....	23
Selecting Functions for Hardware Acceleration.....	25
Selecting Clock Frequencies.....	28
Running System Emulation.....	30
Chapter 3: Profiling and Optimization.....	35
Chapter 4: Debugging an Application.....	41
Debugging Linux Applications in the SDSoC IDE.....	41
Debugging Standalone Applications in the SDSoC IDE.....	41
Debugging FreeRTOS Applications.....	42
Peeking and Poking IP Registers.....	42
Debugging Performance Tips.....	42
Chapter 5: Hardware/Software Event Tracing.....	45
Hardware/Software System Runtime Operation.....	46
Software Tracing.....	47
Hardware Tracing.....	48
Implementation Flow.....	49
Runtime Trace Collection.....	50
Trace Visualization.....	51
Performance Measurement Using the AXI Performance Monitor.....	53

Troubleshooting.....	60
Chapter 6: Makefile/Command-Line Flow.....	63
Makefile Guidelines.....	66
Guidelines for Invoking SDSCC/SDS++.....	67
Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc).....	68
Appendix A: Getting Started with Examples.....	79
Installing Examples.....	79
Using Local Copies.....	81
Synthesizeable FIR Filter.....	82
Matrix Multiplication.....	83
Using a C-Callable RTL Library.....	83
C++ Design Libraries.....	83
Appendix B: Managing Platforms and Repositories.....	85
Appendix C: SDSCC/SDS++ Compiler Commands and Options.....	87
Command Synopsis.....	87
General Options.....	89
Hardware Function Options.....	91
Compiler Macros.....	93
System Options.....	94
Compiler Toolchain Support.....	99
Appendix D: Coding Guidelines.....	103
General C/C++ Guidelines.....	103
Hardware Function Argument Types.....	104
Hardware Function Call Guidelines.....	105
Appendix E: Using C-Callable IP Libraries.....	107
C-Callable Libraries.....	107
Appendix F: Exporting a Library for GCC.....	117
Building a Shared Library.....	117
Compiling and Linking Against a Library.....	120
Exporting a Shared Library.....	121
Appendix G: Compiling and Running Applications on an ARM Processor.....	123

Compiling and Running Applications on a MicroBlaze Processor.....	124
Appendix H: Using External I/O.....	125
Accessing External I/O using Memory Buffers.....	125
Accessing External I/O using Direct Hardware Connections.....	127
Appendix I: SDSoC Environment Troubleshooting.....	131
Troubleshooting Compile and Link Time Errors.....	131
Improving Hardware Function Parallelism.....	132
Troubleshooting System Hangs and Runtime Errors.....	133
Troubleshooting Performance Issues.....	134
Debugging an Application.....	135
Appendix J: SDSoC Environment API.....	137
Appendix K: Additional Resources and Legal Notices.....	139
References.....	139
Please Read: Important Legal Notices.....	140

The SDSoC Environment

The SDSoC™ (software-defined system-on-chip) Environment is a tool suite that includes an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems. SDSoC supports ARM® Cortex-based applications using the Zynq®-7000 All Programmable SoCs and Zynq UltraScale+™ MPSoCs, as well as MicroBlaze™ processor-based applications on all Xilinx® SoCs and FPGAs. The SDSoC Environment also includes system compilers that transform OpenCL™ programs into complete hardware/software systems with select functions compiled into programmable logic.

The SDSoC system compilers analyze a program to determine the data flow between software and hardware functions, and generate an application specific system-on-chip to realize the program. To achieve high performance, each hardware function runs as an independent thread; the system compilers generate 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 SDSoC IDE supports software development workflows including profiling, compilation, linking, system performance analysis, and debugging. In addition, the SDSoC environment provides a fast performance estimation capability to enable "what if" exploration of the hardware/software interface before committing to a full hardware compile.

The SDSoC system compilers target a base platform and invoke the Vivado® High-Level Synthesis (HLS) tool to compile synthesizable C/C++ functions into programmable logic. They then generate a complete hardware system, including DMAs, interconnects, hardware buffers, other IP, and the Field Programmable Gate Array (FPGA) bitstream by invoking the Vivado Design Suite tools. To ensure all hardware function calls preserve their original behavior, the SDSoC system compilers generate system-specific software stubs and configuration data. The program includes 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 "single source," the system compilers let you iterate over design and architecture changes by refactoring at the program level, reducing the time needed to achieve working programs running on the target platform.

Getting Started

Download and install the SDSoC™ Environment according to the directions provided in *SDx Environments Release Notes, Installation, and Licensing Guide* ([UG1238](#)). This guide provides detailed instructions and hands-on tutorials to introduce the primary work flows for project creation, specifying functions to run in programmable logic, system compilation, debugging, and performance estimation. Working through these tutorials is the best way to get an overview of the SDSoC environment, and should be considered prerequisite to application development.

Note: The SDSoC Environment includes the entire tools stack to create a bitstream, object code, and executables. If you have installed the Xilinx® Vivado® Design Suite and the Software Development Kit (SDK) tools independently, you should not attempt to combine these installations with the SDSoC Environment.

Elements of SDSoC

The SDSoC™ Environment inherits many of the tools in the Xilinx® Software Development Kit (SDK), including GNU toolchains and standard libraries (for example, `glibc`) as well as the Target Communication Framework (TCF) and GDB interactive debuggers, a performance analysis perspective within the Eclipse/CDT-based GUI, and command-line tools.

The SDSoC Environment includes a system compiler (`sdscc/sds++`) that generates complete hardware/software systems, an Eclipse-based user interface to create and manage projects and workflows, and a system performance estimation capability to explore different "what if" scenarios for the hardware/software interface.

The SDSoC system compiler employs underlying tools from the Vivado Design Suite (System Edition), including Vivado® HLS, IP integrator, IP libraries for data movement and interconnect, and the RTL synthesis, placement, routing, and bitstream generation tools.



TIP: After you have done a build in SDx, you can launch Vivado. From the SDx menu, select **Xilinx** → **Vivado Integration** → **Open Vivado Project**.

The principle of design reuse underlies workflows you employ with the SDSoC environment, using well established platform-based design methodologies. The SDSoC system compiler generates an application-specific system on chip by customizing a target platform. The SDSoC Environment includes a number of platforms for application development, and others can be provided by Xilinx partners, or custom developed by FPGA design teams. The *SDSoC Environment Platform Development Guide* ([UG1146](#)) describes how to create a design using the Vivado Design Suite, specify platform properties to define and configure Platform Interfaces, and define the corresponding software run-time environment to build a platform for use in the SDSoC environment.

An SDSoC platform defines a base hardware and software architecture and application context, including processing system, external memory interfaces, custom input/output, and software run time including operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root file system. Every project you create within the SDSoC environment targets a specific platform, and you employ the tools within the SDx IDE to customize the platform with application-specific hardware accelerators and data motion networks connecting accelerators to the platform. In this way, you can easily create highly tailored application-specific systems-on-chip for different base platforms, and can reuse base platforms for many different application-specific systems-on-chip.

See the *SDx Environments Release Notes, Installation, and Licensing Guide* ([UG1238](#)) for the most up-to-date list of supported devices.

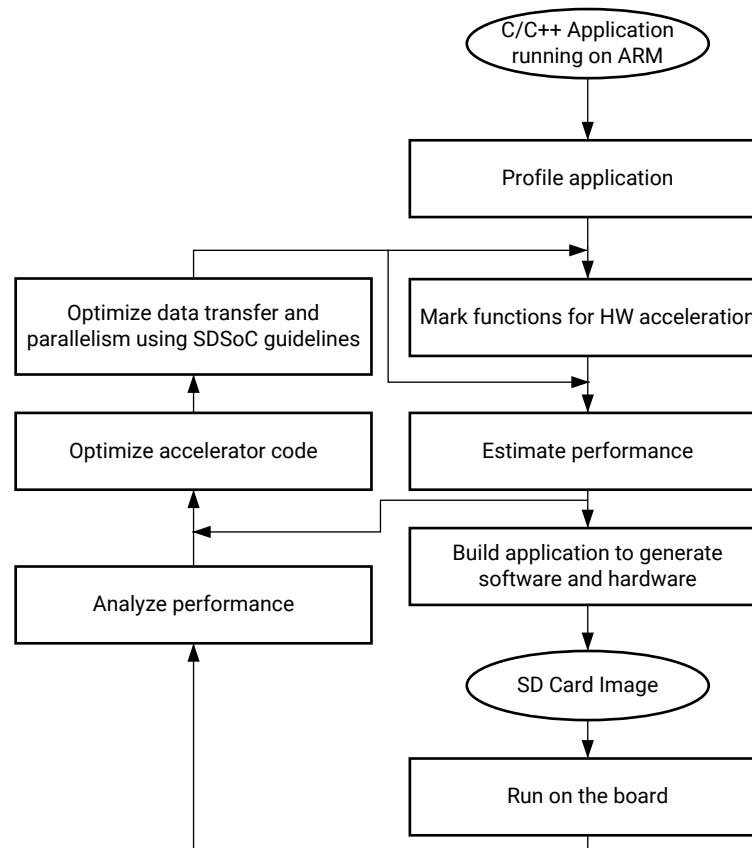
User Design Flows

The SDSoC™ Environment is a tool suite for building efficient application-specific systems-on-chip, starting from a platform SoC that provides a base hardware and target software architecture including boot options.

The following figure shows a representative top-level user visible design flow that involves key components of the tool suite. For the purposes of exposition, the design flow proceeds linearly from one step to the next, but in practice you are free to choose other work flows with different entry and exit points.

Starting with a software-only version of the application that has been cross-compiled for ARM® CPUs, the primary goal is to identify portions of the program to move into programmable logic and to implement the application in hardware and software built upon a base platform.

Figure 1: User Design Flow



X14740-102417

The steps are:

- Select a development platform, cross-compile the application, and ensure it runs properly on the platform.
- Identify compute-intensive hot spots to migrate into programmable logic to improve system performance, and isolate them into functions that can be compiled into hardware.
- Invoke the SDSoC system compiler to generate a complete system-on-chip and SD card image for your application.

You can instrument your code to analyze performance, and if necessary, optimize your system and hardware functions using a set of directives and tools within the SDSoC Environment.

The system generation process is orchestrated by the `sdscc/sds++` system compilers through the SDx IDE or in an SDx terminal shell using the command line and makefiles. Using the SDx IDE or `sdscc` command line options, you select functions to run in hardware, specify accelerator and system clocks, and set properties on data transfers (for example, interrupt vs. polling for DMA transfers). You can insert pragmas into application source code to control the system mapping and generation flows, providing directives to the system compiler for implementing the accelerators and data motion networks.

Because a complete system compile can be time-consuming compared with an "object code" compile for a CPU, the SDSoC Environment provides a faster performance estimation capability. The estimate allows you to approximate the expected speed-up over a software-only implementation for a given choice of hardware functions and can be functionally verified and analyzed through system emulation. The system emulation feature uses a QEMU model executing the software and RTL model of the hardware functions to enable fast and accurate analysis of the system.

As shown in the preceding figure (User Design Flow), the overall design process involves iterating the steps until the generated system achieves your performance and cost objectives.

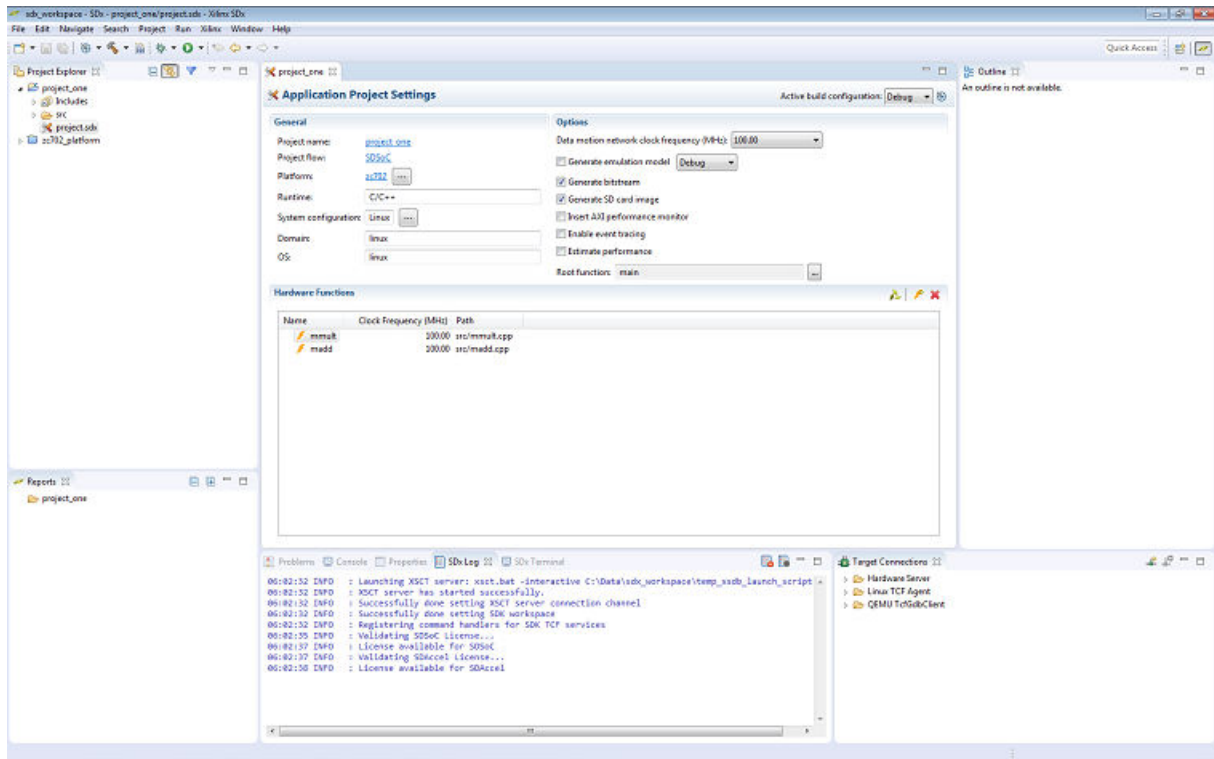
It is assumed that you have already worked through the introductory tutorials and are familiar with project creation, hardware function selection, compilation, and running a generated application on the target platform. If you have not already done so, see *SDSoC Environment Tutorial* ([UG1028](#)).

Working with SDx

When a project is opened in the SDx IDE, the project is arranged in a series of different window and editor views, also known as a perspective in the IDE. The tool opens with the SDx (default) perspective.

See [Launching SDx](#) for more information about opening the software.

Figure 2: SDSoC - Default Perspective



As shown, the default perspective has an arrangement of Project Explorer view, Project Editor window, and the Outline view across the top, and the Report view, the Console view, and Target Connections view across the bottom. A brief description of these are:

- **Project Explorer:** Displays a tree view of the project folders and the source files, build files, and reports generated by the tool.
- **Project Editor:** This is the primary window for interacting with the project in the SDx IDE. It displays project settings, context sensitive code editors, and provides access to many commands for working with the project.
- **Outline:** Displays an outline of the current file opened in the Project Editor.
- **Report:** Displays the SDx reports of performance estimation, profile summaries, and build results.
- **Console:** Presents multiple views including the command console, problem reports, project properties, and logs and terminal views.
- **Target Connections:** Provides status for different targets connected to the SDx tool, such as the Vivado hardware server, Target Communication Framework (TCF), and QEMU networking.

You can open and close windows, using the **Window** → **Show View** command, and arrange them to suit your needs by dragging and dropping them into new locations in the IDE. Save the window arrangement as a perspective using the **Window** → **Perspective** → **Save Perspective As** command. This lets you define different perspectives for initial project editing, report analysis, and debug for example.

You can open different perspectives using the **Window** → **Perspective** → **Open Perspective** command. You can restore the default window arrangement by opening the SDx (default) perspective.

Command-Line Flow

In addition to the SDx IDE, the SDSoC environment provides a command line interface to support a scripted Makefile flow, as described in [Chapter 6: Makefile/Command-Line Flow](#).

- For C-based projects, the command line is invoked using `sdscc` command. See [Appendix C: SDSCC/SDS++ Compiler Commands and Options](#).
- For C++ projects, the command line is invoked using the `sds++` command.
- For OpenCL projects, the command line is invoked using the `xocc` command. See [Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler \(xocc\)](#).

Note: In SDSoC, the `clCreateBuffer` flag option `CL_MEM_USE_HOST_PTR` is not supported. OpenCL™ is only supported in the Linux environment.

- The command line executables are located in the installation directory at `<sdx_root>/bin`.

In the SDSoC environment, you control the system generation process by structuring hardware functions and calls to hardware functions to balance communication and computation, and by inserting pragmas into your source code to guide the `sdscc` system compiler.

The hardware/software interface is defined implicitly in your application source code after you select a platform and a set of functions in the program to be implemented in hardware. The `sdscc/sds++` system compilers analyze the program data flow involving hardware functions, schedule each such function call, and generate a hardware accelerator and data motion network realizing the hardware functions in programmable logic. They do so not by implementing each function call on the stack through the standard ARM® application binary interface, but instead by redefining hardware function calls as calls to function stubs having the same interface as the original hardware function. These stubs are implemented with low-level function calls to a `send/receive` middleware layer that efficiently transfers data between the platform memory and CPU and hardware accelerators, interfacing as needed to underlying kernel drivers.

The `send/receive` calls are implemented in hardware with data mover IP cores based on program properties like memory allocation of array arguments, payload size, the corresponding hardware interface for a function argument, as well as function properties such as memory access patterns and latency of the hardware function.

Creating an SDSoC Project

Launching SDx

Note: Although SDSoC™ supports Linux application development on Windows hosts, a Linux host is strongly recommended for SDSoC platform development, and required for creating a platform supporting a target Linux OS.

You can launch the SDx IDE directly from the desktop icon, or from the command line by one of the following methods:

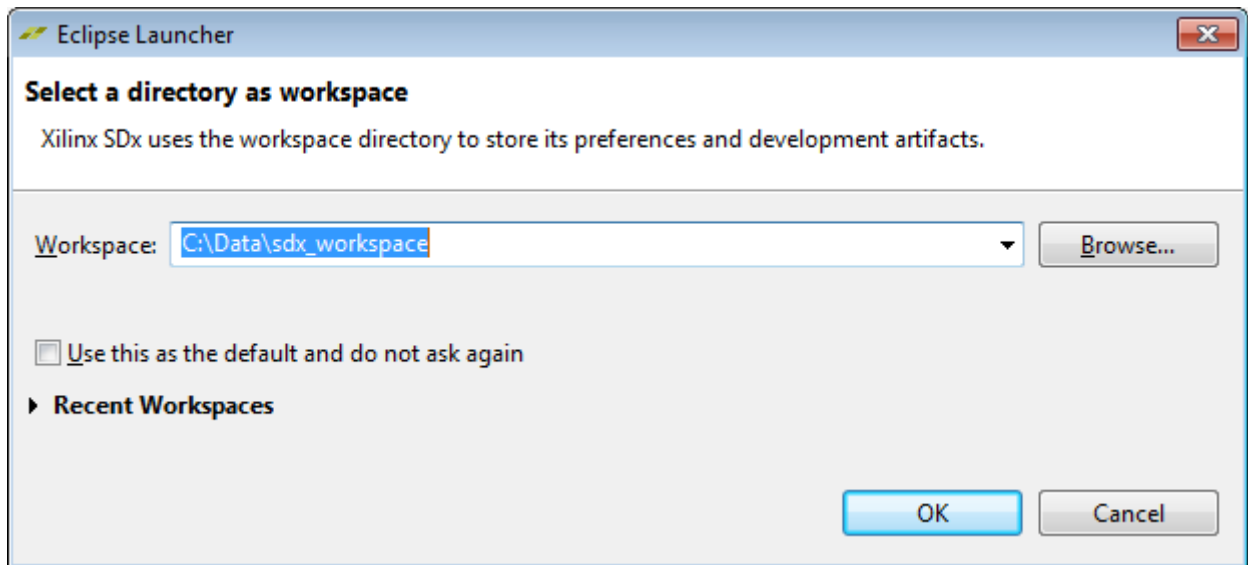
- Double-clicking the SDx icon to start the application
- Launching it from the Start menu in the Windows operating system
- Using the following command from the command prompt: `sdx`



TIP: Launching the SDx tool from the command line requires that the command shell is configured to run the application, or is an SDx Terminal window launched from the Start menu. To configure a command shell, run the `settings64.bat` file on Windows, or source the `settings64.sh` or `settings64.csh` file on Linux, from the `<install_dir>/SDx/2017.4` directory. Where `<install_dir>` is the installation folder of the SDx software.

The SDx IDE opens, and prompts you to select a workspace when you first open the tool.

Figure 3: Specify SDSoc Workspace



The SDx workspace is the SDx folder that stores your projects, source files, and results while working in the tool. You can define separate workspaces for each project, or have workspaces for different types of projects, such as the SDSoc project you can create with the following instructions.

1. Use the **Browse** button to navigate to and specify the workspace, or type the appropriate path in the **Workspace** field.
2. Select the **Use this as the default and do not ask again** checkbox to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.



TIP: You can always change the current workspace from within the SDx IDE using the **File → Switch Workspace** command.

3. When you click **OK**, if this is the first time the SDx IDE has been launched, the **SDx Welcome** screen opens to let you specify an action. Select either **Create SDx Project**, or **File → New → SDx Project**.

Creating an SDSoC Application Project



TIP: Example designs are provided with the SDSoC tool installation, and also on the Xilinx® GitHub repository. See [Appendix A: Getting Started with Examples](#) for more information.

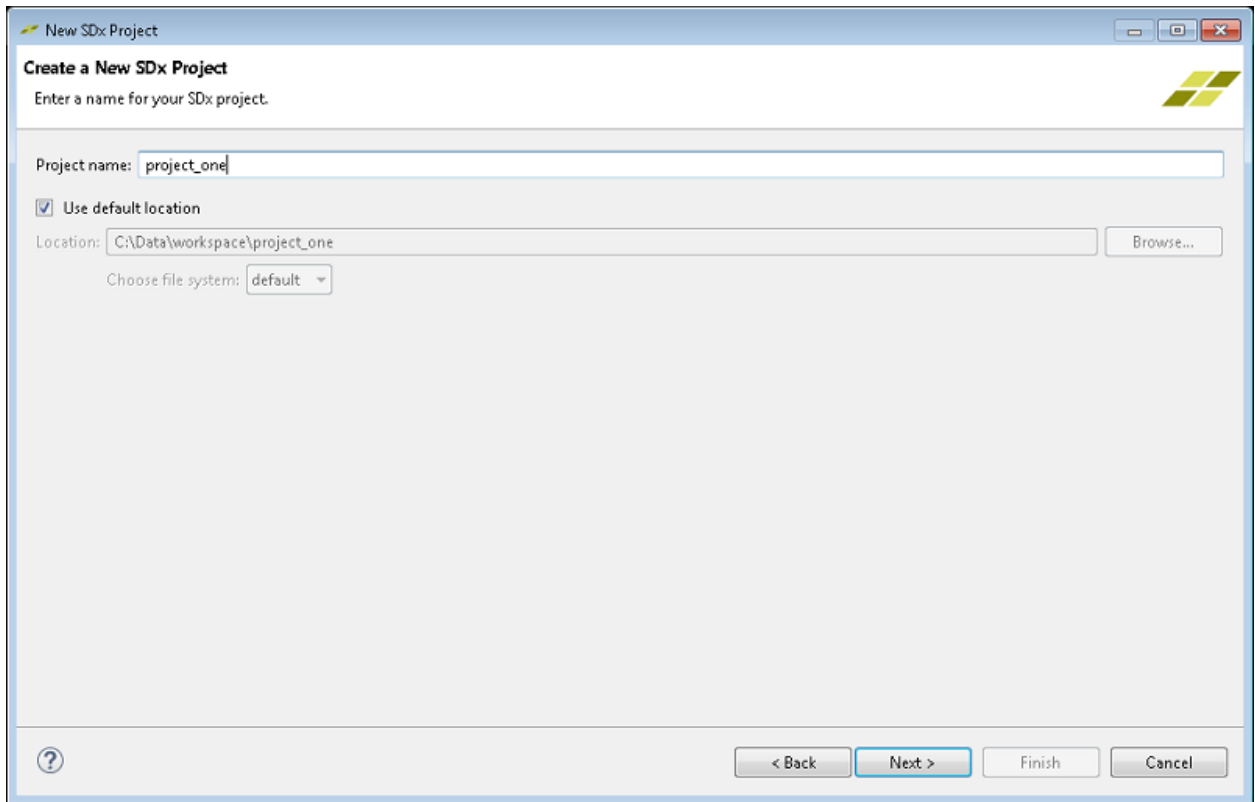
After launching the SDx IDE, you can create a new project to define an SDSoC project. The **New SDx Project** wizard opens with the Project Type dialog box to let you specify the type of project to create. SDx supports three project types:

- **Application Project:** A software application with portions of the application accelerated onto a single hardware kernel on an SDAccel or the SDSoC platform.
- **System Project:** Contains multiple application projects to support different applications and hardware kernels running on the platform.
- **Platform Project:** Defines the hardware and software platform for use in SDSoC projects.

Select **Application Project** and click **Next**.

SDx opens the Create a New SDx Project dialog box to let you specify the name for your project, as shown in the following figure.

Figure 4: Create New SDSoc Project



Specify the **Project name**.

You can enable **Use default location** to have the platform project created in your SDx workspace, or disable this checkbox to specify a **Location**. If you specify the location, you can use **Choose file system** to select the **default** file system, or enable the Eclipse Remote File System Explorer, **RSE**.

Select **Next** and the **New SDx Project** wizard prompts you to select an SDSoC compatible platform for your project, as shown below.

The platform is composed of a Device Support Archive (DSA), which describes the hardware platform design, interfaces, and a software platform, which includes the OS boot files and runtime files.



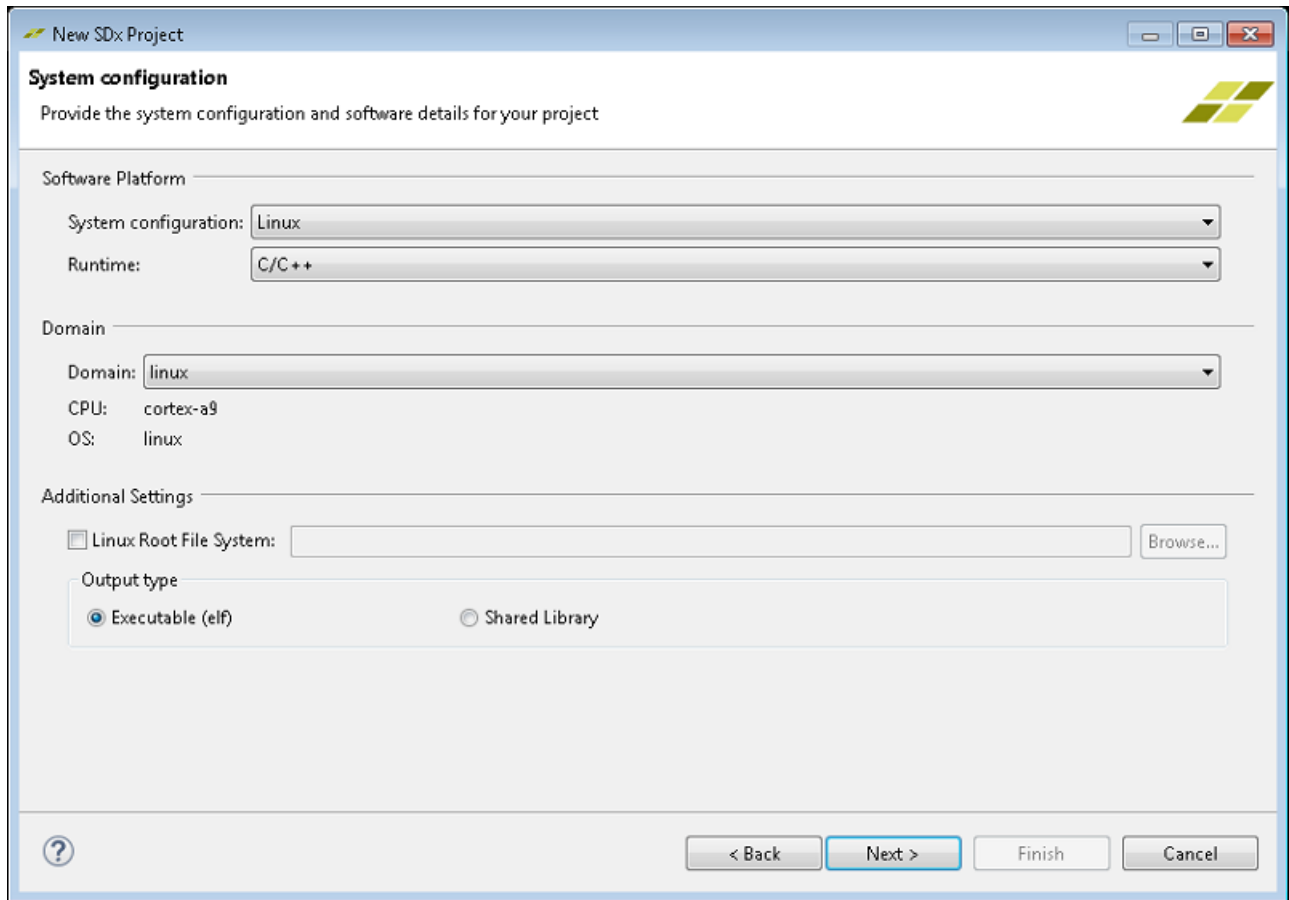
IMPORTANT!: *Your selection of a platform determines if you are working in an SDAccel project or an SDSoC project. Be sure to select the right platform for your project, as subsequent processes are driven by this choice.*

As you can see from the figure above, you can add new platforms into a custom repository, or even develop a platform to add to projects. See [Appendix B: Managing Platforms and Repositories](#) for more information.

You can select the target platform for your project from the listed platforms. You can also filter the listed platform by clicking the **Filter** link, which opens the Filter dialog box so you can set filters to specify the **Flow**, **Family**, or **Vendor** for your target platform.

After selecting the target platform and clicking **Next**, the System Configuration dialog box opens to let you select a **System configuration** and a **Runtime** from a list of those defined for the currently selected platform. The System Configuration defines the software environment that runs on the hardware platform. It specifies the operating system and the available run-time settings for the processors in the hardware platform, and provide software-configurable hardware parameters.

Figure 5: Specify System Configuration



New SDx Project

System configuration
Provide the system configuration and software details for your project

Software Platform

System configuration: Linux

Runtime: C/C++

Domain

Domain: linux

CPU: cortex-a9

OS: linux

Additional Settings

☐ Linux Root File System: [] Browse...

Output type

☒ Executable (elf) ☐ Shared Library

< Back Next > Finish Cancel

For SDSoc you can also select a **Processor Domain**, specifying the processor group and OS, Linux Root File System, and Output Type.



TIP: For OpenCL™ support, the System Configuration must be set to **A53 OpenCL Linux: the Runtime selection automatically updates to OpenCL. C/C++ is supported for all platforms.**

After selecting the System Configuration and clicking **Next**, the Templates dialog box displays to let you specify an application template for your new project. Initially the Template dialog box is empty, but you can download many SDSoc examples from GitHub as discussed in [Appendix A: Getting Started with Examples](#).

You can use the template projects as examples to learn about the SDx tool and acceleration kernels, and use a template as a foundation for your new project. You must select a template, even if you select **Empty Application** to create a blank project into which you can import files and build your project from scratch.

Click **Finish** to close the **New SDx Project** wizard and open the project.

Importing Sources

With the project open in the SDx IDE, you can import source files to add them to the project. To add files, right-click the `src` folder in the Project Explorer view, and select the **Import** command.

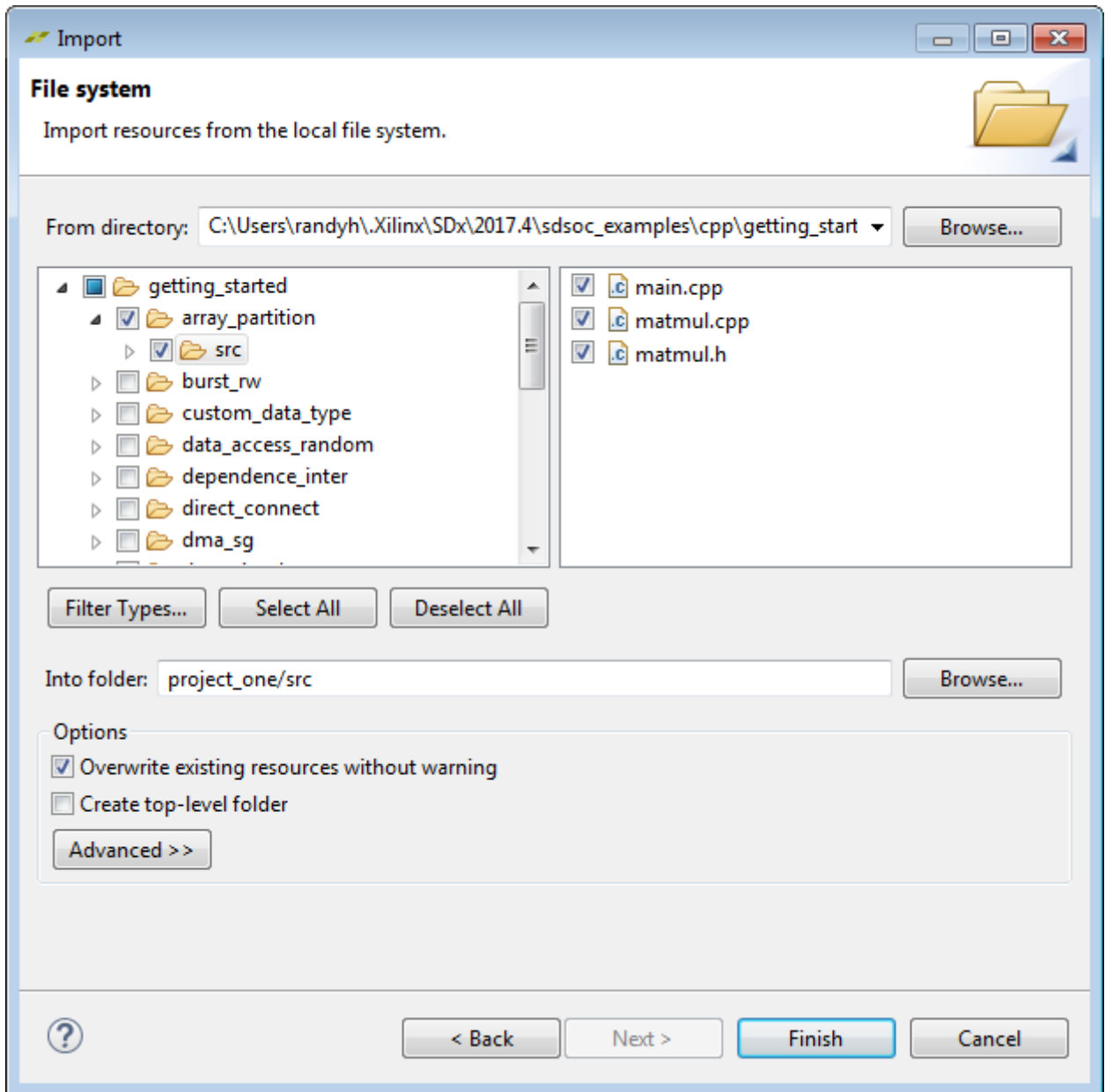


TIP: You can also access this through the **File** → **Import** menu command.

This displays the Import dialog box, which lets you specify the source of files you from which you are importing. The different sources include importing from archives, from existing projects, from the file system, and from a Git repository. Select the source of files you will be importing, and click **Next**.

The dialog box that displays depends on the source of files you selected in the prior step. In the following figure you can see the File System dialog box that is displayed as result of choosing to import sources from the file system.

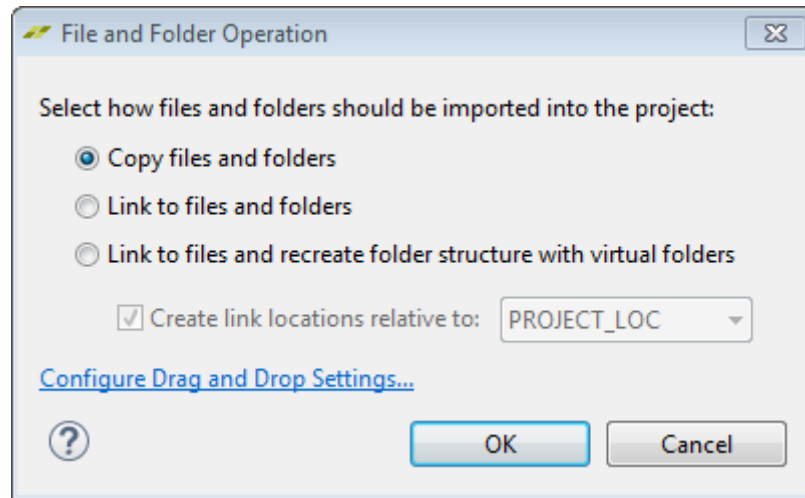
Figure 6: Import File System Sources



The File System dialog box lets you navigate to a folder in the system, and select files to import into your project. You can specify files from multiple folders, and specify the folder to import files into. You can enable the **Overwrite existing resource without warning** to simply overwrite any existing files, and enable **Create top-level folder** to have the files imported into a directory structure that matches the source file structure. If this checkbox is not enabled, which is the default, then the files will simply be imported into the specified **Into folder**.

On the Windows operating system, you can add files to your project by dragging and dropping them from the Windows Explorer. Select files or folders in the Explorer and drop them into the `src` folder, or another appropriate folder in the Project Explorer in the SDx IDE. When you do this, the tool prompts you to specify how to add the files to your project, as shown in the following figure.

Figure 7: **File and Folder Operation**



You can copy files and folders into your project, add links to the files, or link to the files in virtual folders to preserve the original file structure. There is also a link to **Configure Drag and Drop Settings**, which lets you specify how the tool should handle these types of drag and drop operations by default. You can also access these settings through the **Window** → **Preferences** menu command.

After adding source files to your project, you are ready to begin configuring, compiling, and running the application.

Selecting Functions for Hardware Acceleration

The first major task in creating a hardware accelerated SoC is to identify portions of application code that are suitable for implementation in hardware, and that significantly improve the overall application performance when run in hardware. Every platform included in the SDSoc™ Environment includes a pre-built SD card image from which you can boot and run your application code if you don't have any functions selected for acceleration on the hardware platform. Running the application this way lets you profile the original application code to identify candidates for acceleration.

Program hot-spots that are compute-intensive are good candidates for hardware acceleration, especially when it is possible to stream data between hardware and the CPU and memory to overlap the computation with the communication. Software profiling is a standard way to identify the most CPU-intensive portions of your program. See [Chapter 3: Profiling and Optimization](#) for more information on profiling your application.

After determining the function or functions to move into hardware, with a project opened in the SDx IDE, you can select the function from within the Project Editor window.



TIP: If the Project Editor window is not opened, you can double-click the `<project>.sdx` file in the **Project Explorer** to open it.


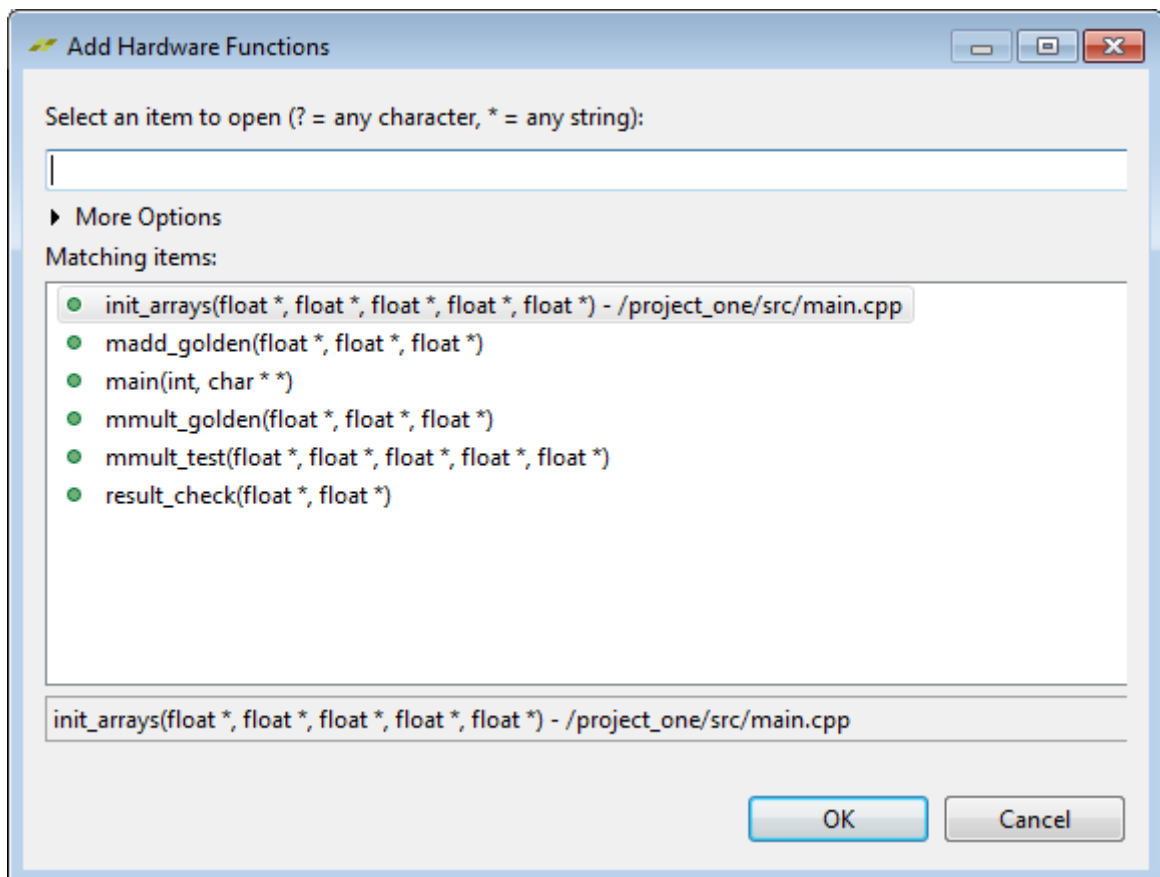

Click the  symbol in the **Hardware Functions** panel of the Project Editor window to display the list of candidate functions within your program. This displays the Add Hardware Functions dialog box, which lets you select one or more functions from a list of functions in the call graph of the `main` function by default.

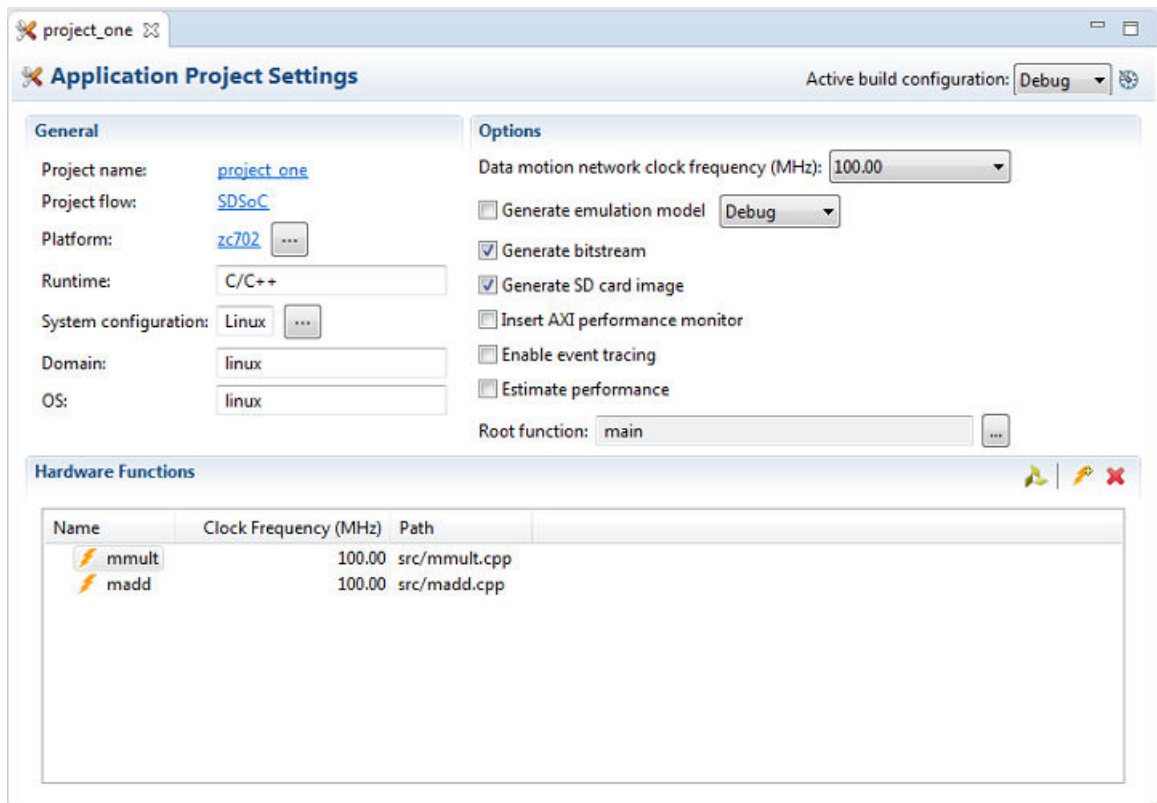
Figure 8: **Add Hardware Functions Dialog Box**



The list of functions starts at the **Root function** as defined in the **Options** panel of the Project Editor window, and is set to `main` by default. You can change the **Root function** by clicking the **Browse** () command and selecting an alternate root.

From within the **Add Hardware Function** dialog box, you can select one or more functions for hardware acceleration and click **OK**. The functions appear in the **Hardware Function** panel of the Project Editor as shown below.

Figure 9: Hardware Function Panel



TIP: The Eclipse CDT indexing mechanism is not foolproof, and you might need to close and reopen the **Add Hardware Function** dialog box to view the available functions. If a function does not appear in the list, you can navigate to its source file in the **Project Explorer** window, expand the outline of the source, right-click the function and select **Toggle HW/SW**.

When moving a function optimized for CPU execution into programmable logic, you can usually revise the code to improve the overall performance. See [Coding Guidelines](#).

For xFAST libraries, right-click and select **Toggle Hardware** from the associated header files in the project includes in project explorer.

See the *Xilinx OpenCV User Guide* ([UG1233](#)) for more information.

Selecting Clock Frequencies


After selecting hardware functions, it could be necessary to select the clock frequency for running the function or the data motion network clock.

Every platform supports one or more clock sources, which is defined by the platform developer as described in *SDSoC Environment Platform Development Guide* (UG1146). The platform clocks and default clock are used for the data motion network (data mover IPs and control buses) generated during system generation. You can select the data motion clock. It is not bound to any default.

Although the control buses and data mover IP within the data motion network run off of a single clock, it is possible to run hardware functions and `zero_copy` data buses at a different clock frequency than the data motion network to achieve higher performance. However, the `zero_copy` data buses are `m_axi` interfaces integrated into the IP core implementing the target function. Therefore, it is not possible to run the hardware function and the `zero_copy` data buses at different frequencies.

You can view the available platform clocks by selecting the **Platform** link in the **General** panel of the **Project Settings** window. This displays details of the platform, including the available clock frequencies.

Figure 10: SDSoc IDE - General



The screenshot shows the 'General' panel of the SDSoc IDE. It contains the following fields and values:

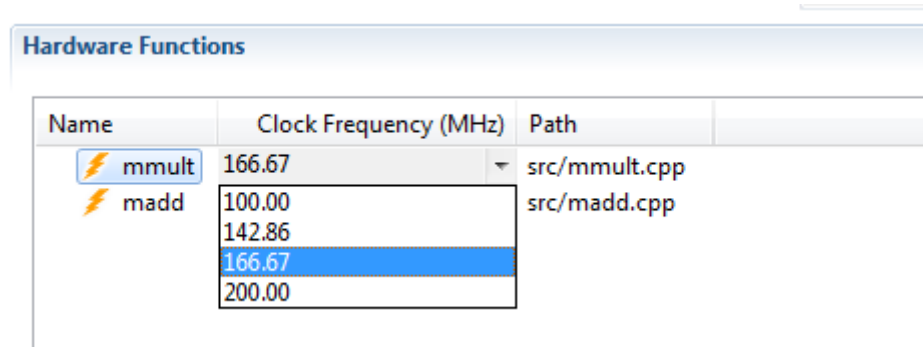
- Project name:** `project one`
- Project flow:** `SDSoC`
- Platform:** `zc702` (with a dropdown arrow)
- Runtime:** `C/C++`
- System configuration:** `Linux` (with a dropdown arrow)
- Domain:** `linux`
- OS:** `linux`



IMPORTANT!: Be aware that it might not be possible to implement the hardware system with some clock selections.

The function clock displays in the SDx Project Editor window, in the **Hardware Functions** panel. Select a function from the list, like `mmult` in the figure below, and click in the **Clock Frequency** column to access the pull-down menu to specify the clock frequency for the function.

Figure 11: Select Function Clock Frequency

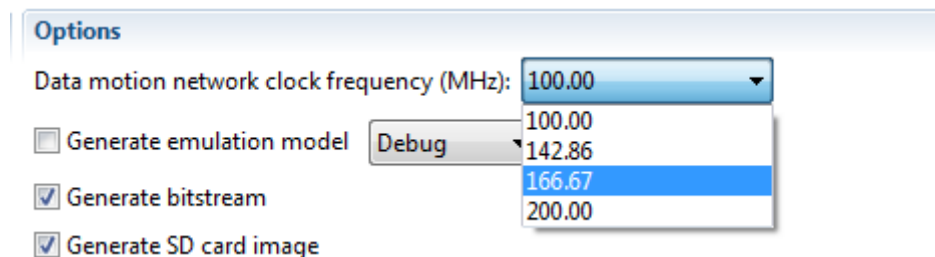


You can also set the clock frequency for a function from the command line, by specifying the Clock ID (see [Makefile Guidelines](#)):

```
$ -sds-hw foo foo_src.c -clkid 1 -sds-end
```

To specify the data motion clock frequency, select the **Data motion network clock frequency** pull-down menu in the **Options** panel of the SDx Project Editor window. The Data motion network clock frequency menu is populated by the available clocks on the platform.

Figure 12: Data Motion Network Clock Frequency



You can also select a Data Motion clock frequency from the command line with the `-dmclkid` option. For example:

```
$ sdsc -sds-pf zc702 -dmclkid 1
```

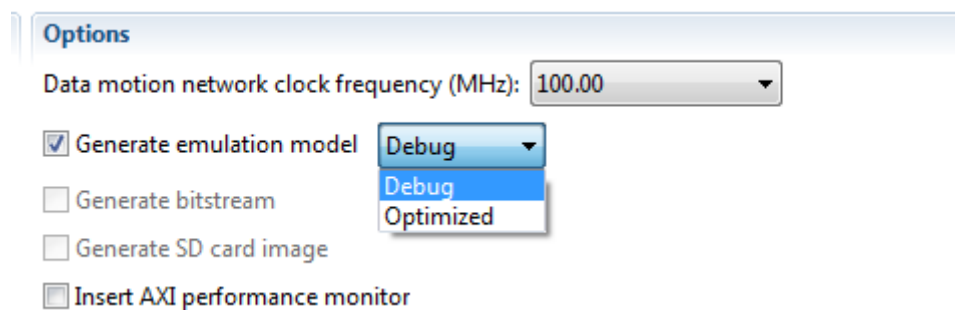
You can use the `sdsc -sds-pf -info` option to see the available clocks for a platform, and determine the clock ID. See [General Options](#) of the [Appendix C: SDSC/SDS++ Compiler Commands and Options](#) for more information.

Running System Emulation

After the hardware functions are identified, the logic can be compiled and the entire system (PS and PL) verified using emulation. This 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.

Within the Project Editor window, select **Generate emulation model** to enable system emulation.

Figure 13: **Generate Emulation Model**




Because emulation does not require a full system compile, the tool disables **Generate bitstream** and **Generate SD card image** to improve run time. The bitstream generation takes a significant amount of time, and disabling it can reduce your design iteration time. System emulation allows you to verify and debug the system with the same level of accuracy as a full bitstream compilation.

Generate emulation model offers two modes: **Debug** and **Optimized**.

- **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.
- **Optimized:** runs the behavioral simulation in batch mode, returning the results without the waveform data. While **Optimized** can be faster, it returns less information than **Debug** mode.

To capture waveform data from the PL hardware emulation for viewing and debugging, select the **Debug** pull-down menu option. For faster emulation without capturing this hardware debug information, select **Optimized**.

After specifying the **Generate emulation model** option, use the **Build** () command to compile the system for emulation. The **Build** command invokes the system compilers to build your application project. Build offers two modes as well: **Debug**, and **Release**.

- **Debug:** This compiles the project use in software debug. The compiler produces extra information for use by a debugger to facilitate debug and let you step through the code.

- **Release:** The compiler tries to reduce code size and execution time of the application. This strips out the debug code so that you really cannot do debug with this version.



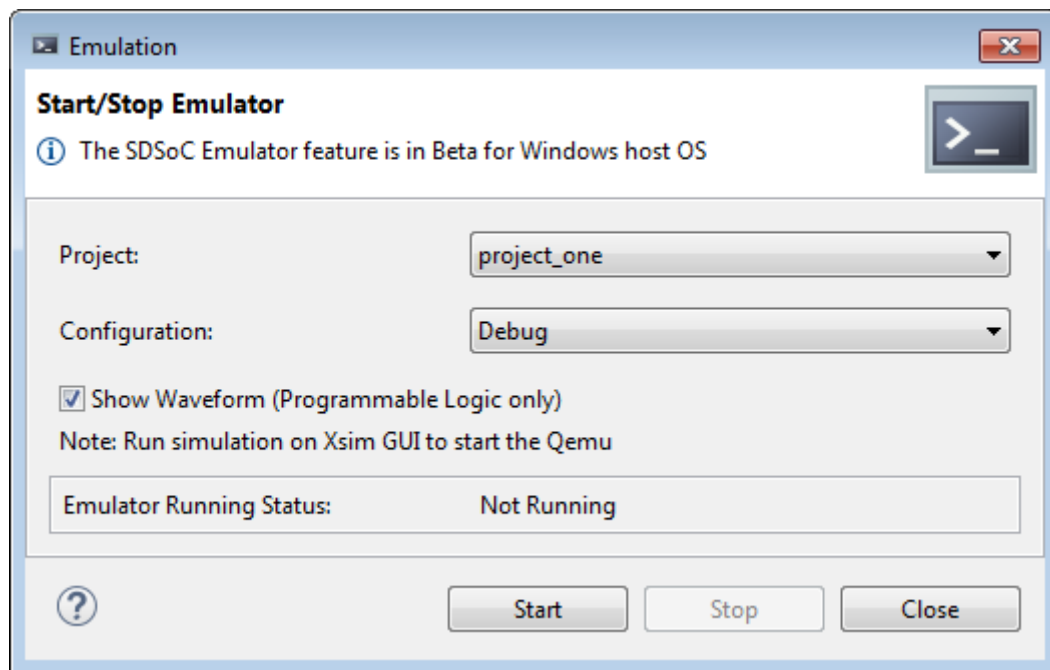
TIP: *Debug and Release modes describe how the software code is compiled; it does not affect the compilation and implementation of the hardware functions.*

If you add a new Build Configuration, it shows in this drop-down list.

The Build process can take some time, depending on your application code, the size of your hardware functions, and the various options you have selected. To compile the hardware functions, the tool stack includes SDx, Vivado® HLS, and Vivado Design Suite.

After the system is compiled for emulation, you can invoke the system emulator using the **Xilinx** → **Start/Stop Emulator** menu command. When the Start/Stop Emulator dialog box opens, if the specified emulation mode is **Debug**, you can choose to run the emulation with or without waveforms. If the emulation mode is **Optimized**, the **Show waveforms** checkbox is disabled, and cannot be changed.

Figure 14: **Start/Stop Emulator**



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 PL and the compiled hardware functions running in the PS are functionally correct.

Enabling the **Show Waveform** option provides the same functionality in the console window, plus the behavioral simulation of the 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. Use the **Scopes** pane to navigate the design hierarchy, then select the signals to monitor in the **Object** pane, and right-click to add the signals to the waveform pane. Press the **Run All** toolbar button to start updates to the waveform window. For more information on working with the Vivado simulator waveform window refer to the *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

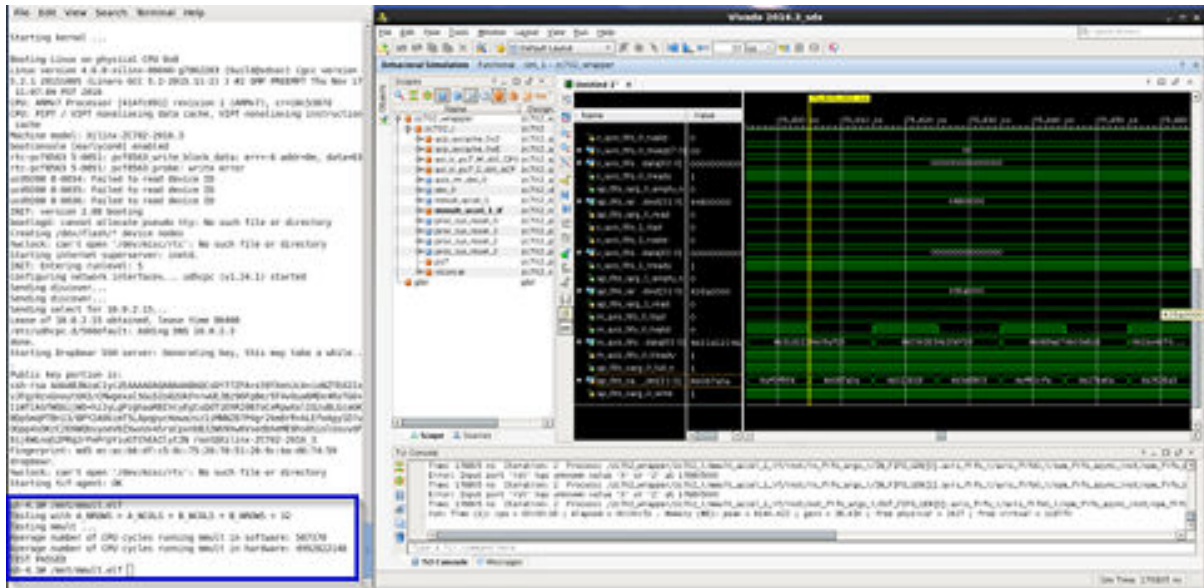
Note: Running with RTL waveforms results in a slower run time, but enables detailed analysis into the operation of the hardware functions.

The system emulation can also be started by selecting the active project in the Project Explorer view and right-clicking to select the **Run As → Launch on Emulator** menu command, or the **Debug As → Launch on Emulator** menu command. 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. See [Working with SDx](#) for more information on changing perspectives.

You see the program output in the console tab, and if the **Show Waveform** option was selected, you also see any appropriate response in the hardware functions in the 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.

The emulation can be stopped at any time using the menu option **Xilinx → Start/Stop Emulator** and selecting **Stop**.

A system emulation session run from the command line is shown in the following figure, with the QEMU console shown at left and the PL waveform shown on the right.



TIP: For an example project to demonstrate emulation, create a new Sdx 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 GUI and the command line.

Profiling and Optimization

There are two distinct areas to be considered when performing algorithm optimization in the SDSoC™ Environment:

- Application code optimization
- Hardware function optimization

Most application developers are familiar with optimizing software targeted to a CPU. This usually requires programmers to analyze algorithmic complexities, overall system performance, and data locality. There are many methodology guides and software tools to guide the developer identifying performance bottlenecks. These same techniques can be applied to the functions targeting hardware acceleration in the SDSoC Environment.

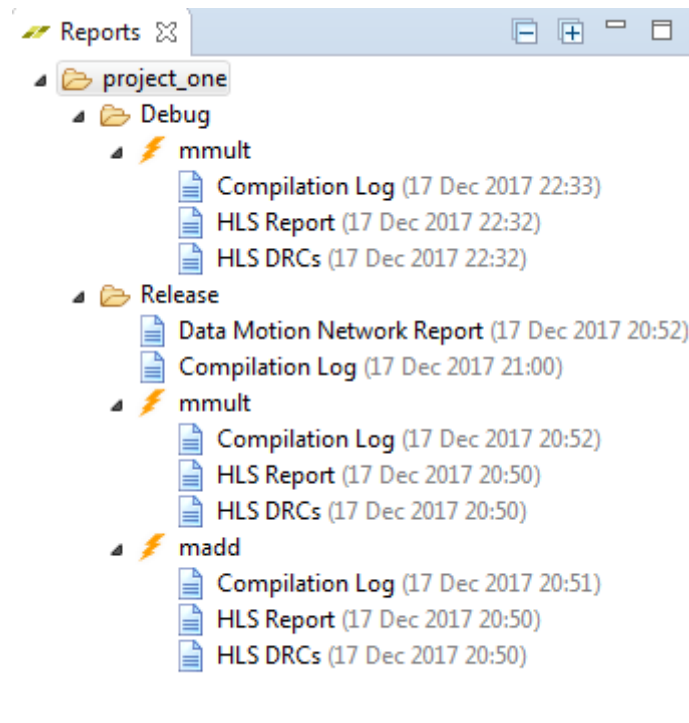
As a first step, programmers should optimize their overall program performance independently of the final target. The main difference between SDSoC and general purpose software is that, in SDSoC projects part of the core compute algorithms are pushed onto the FPGA. This implies that the developer must also be aware of algorithm concurrency, data transfers, memory usage/consumption, and the fact that programmable logic is targeted.

Generally, the programmer must identify the section of the algorithm to be accelerated and how best to keep the hardware accelerator busy while transferring data to and from the accelerator. The primary objective is to reduce the overall computation time taken by the combined hardware accelerator and data motion network versus the CPU software only approach.

Software running on the CPU must efficiently manage the hardware function(s), optimize its data transfers, and perform any necessary pre- or post- processing steps.

The SDSoC Environment is designed to support your efforts to optimize these areas, by generating reports that help you analyze the application and the hardware functions in some detail. The reports are automatically generated when you build the project, and listed in the **Reports** view of the SDx IDE, as shown in the following figure. Double-click on a listed report to open it.

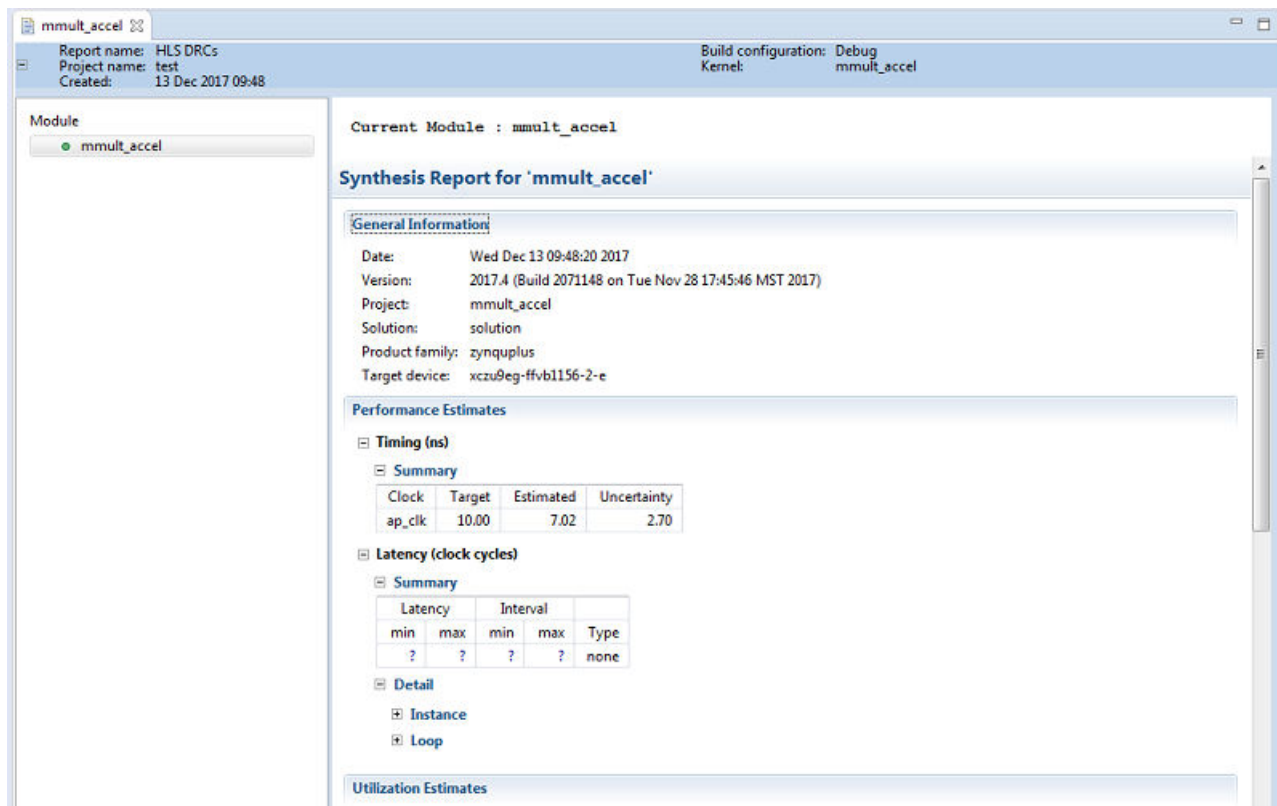
Figure 15: Report View



The following figures show the two main reports: the HLS Report, and Data Motion Network Report.

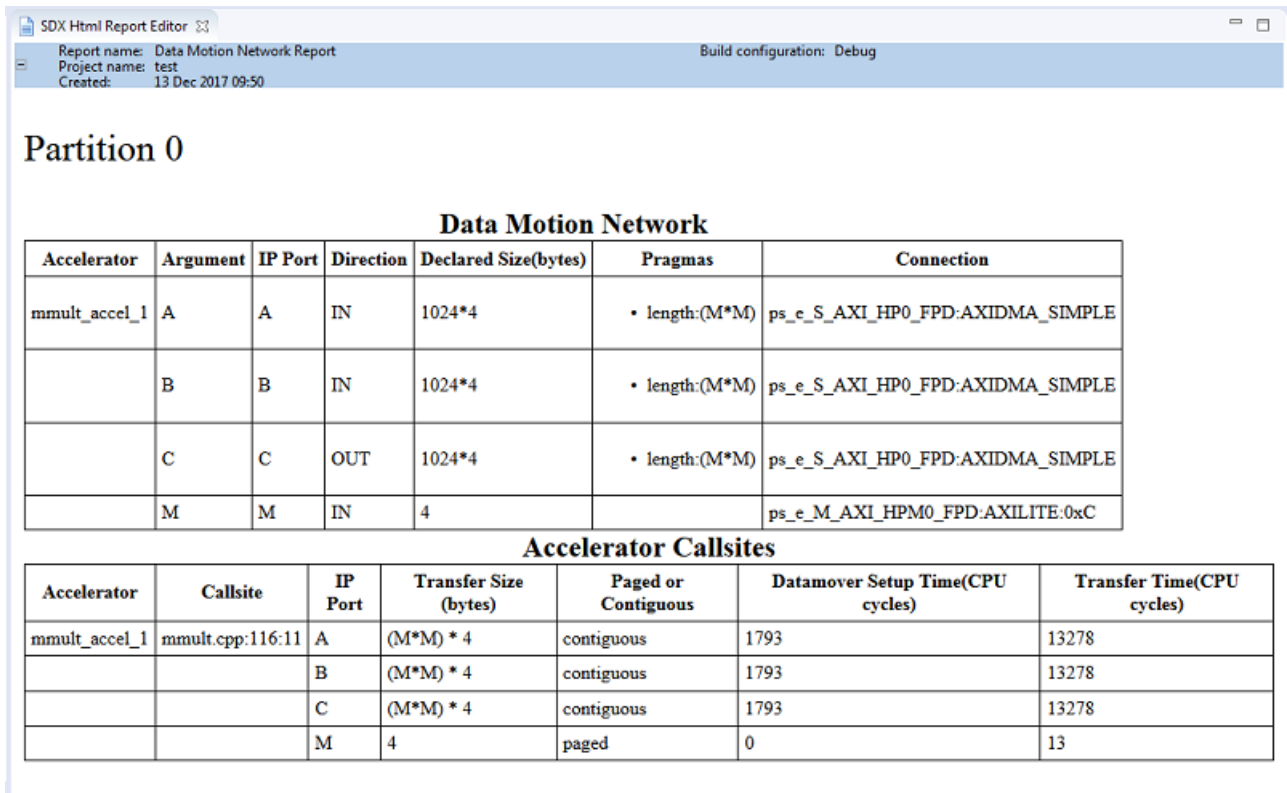
To access these reports from the GUI, ensure the **Reports** view is visible. This view is typically below the **Project Explorer** view. You can use the **Window** → **Show View** → **Other** menu command to display the Reports view if it is not displayed. See [Working with SDx](#) for more information.

Figure 16: HLS Report Window



The HLS Report provides details about the High-Level Synthesis process (HLS). This task translates the C/C++ model into a hardware description language responsible for implementing the functionality on the FPGA. This lets you see the impact of the design on the hardware implementation. You then optimize the hardware function(s) based on the information.

Figure 17: Data Motion Network Report



The Data Motion Network Report describes the hardware/software connectivity for each hardware function. The **Data Motion Network** table shows (from the right most column to the left most) what sort of datamover is used for transport of each hardware function argument, and to which system port that datamover is attached. The **Pragmas** shows any SDS based pragmas used for the hardware function.

The **Accelerator Callsites** table shows the following:

- The Accelerator instance name.
- The Accelerator argument.
- The name of the port on the IP that pertains to said argument [typically the same as the previous, except when bundling].
- The direction of transfer
- The size, in bytes, of data to be transfered, to the degree in which the compiler can deduce that size. If the transfer size is runtime determined, this is zero.
- List of all pragmas related to this argument
- `<system port>:<datamover>`, if applicable. Indicates which platform port and which datamover will be used for transport of this argument.

- What accelerator(s) are used, the inferred compiler as being used, and the CPU cycles used for setup and transfer of the memory

Generally, the Data Motion report page indicates first:

- What characteristics are specified in pragmas.
- In the absense of a pragma, what the compiler was able to infer.

The distinction is that the compiler might not be able to deduce certain program properties. In particular, the most important distinction here is cacheability. If the DM report indicates cacheable, and the data is in fact uncacheable [or vice versa], correct cache behavior would occur at runtime; it is not necessary to structure your program such that the compiler can identify data as being uncacheable to remove flushes.

Additional details for each report, as well as a profiling and optimization methodology, and coding guidelines can be found in the *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#)) .

Debugging an Application

The SDSoC™ Environment lets you create and debug projects using the SDx IDE. Projects can also be created outside the SDx IDE, using makefiles for instance, and debugged either on the command line or using the SDx IDE.

See the *SDSoC Environment Tutorial* ([UG1028](#)) for information on using the interactive debuggers in the SDx IDE.

Debugging Linux Applications in the SDSoC IDE

Within the SDSoC™ IDE, use the following procedure to debug your application:

1. Set the board to boot from the SD card, per the relevant evaluation board user guide.
2. Select the **Debug** as the active build configuration and build the project.
3. Copy the generated `Debug/sd_card` image to an SD card, and boot the board.
4. 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.
5. Select the **Debug As** option to create a new debug-configuration, and enter the IP address for the board.
6. You now switch to the SDSoC Environment debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

Debugging Standalone Applications in the SDSoC IDE

Use the following procedure to debug a standalone (bare-metal) application project using the SDSoC™ IDE.

1. Make sure the board is connected to your host computer using the JTAG Debug connector, and set the board to boot from JTAG.
2. Select **Debug** as the active build configuration, and build the project.
3. Select the **Debug As** option to create a new debug-configuration.
 You now switch to the SDSoC Environment debug perspective, which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.
 In the SDSoC IDE toolbar, click the **Debug** option, which provides a shortcut to the procedure described above.

Debugging FreeRTOS Applications

If you create a FreeRTOS application project using the SDSoC™ Environment, you can debug your application using the same steps as a standalone (bare-metal) application project.

Peeking and Poking IP Registers

Two small executables called `mrd` and `mwr` are available to peek and poke registers in memory-mapped programmable logic. These executables are invoked with the physical address to be accessed.

For example: `mrd 0x80000000 10` reads ten 4-byte values starting at physical address 0x80000000 and prints them to standard output, while `mwr 0x80000000 20` writes the value 20 to the address 0x80000000.

These executables can be used to monitor and change the state of memory-mapped registers in hardware functions and in other IP generated by the SDSoC™ environment.



CAUTION!: *Trying to access an address that is not mapped reports a BUS ERROR; addresses that are mapped but lack proper backing results in a system hang.*

Debugging Performance Tips

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

- The `sds_clock_counter()` function. Use this function to determine how much time different code sections, such as the accelerated code and the non-accelerated code, take to execute.

- The `sds_clock_frequency()` function. 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 HLS report files (`_sds/vhls/.../*.rpt`) or in the GUI under **Reports > HLS Report**. Latency of X accelerator clock cycles = $X * (\text{processor_clock_freq} / \text{accelerator_clock_freq})$ 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 `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 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 `sdscc -sds-pf-info <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, as it is more efficient to burst transfers than to make a series of unrelated random accesses.
- Assure that data transfers make use of system ports that are appropriate for the cacheability of the data being transferred. Cache flushing can be an expensive procedure, and using coherent ports to access coherent data, and noncoherent ports to access non-coherent ports makes a big difference.

Use `sds_alloc()` instead of `malloc`, where possible. The memory that `sds_alloc()` issues is physically contiguous, and enables the use of datamovers 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.

Hardware/Software Event Tracing

The systems produced by the SDSoC™ Environment are high-performance and complex, hardware and software. 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 occurring. Through the use of *event tracing*, the SDSoC Environment tracing feature provides to you a detailed view of what is happening in the system during the execution of an application.

This detailed view can help you understand the performance of your application given the workload, hardware/software partitioning, and system design choices. Such information helps you to optimize and improve system implementation. This view enables event tracing of software running on the processor, as well as hardware accelerators and data transfer links in the system. Trace events are produced and gathered into a timeline view, showing you a detailed perspective unavailable anywhere else about how their application executes.

Tracing an application produces a log that records information about system execution. Compared to event logging, event tracing provides correlation between events for a duration of time (where events have a duration, 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. Tracing shows the performance of execution with more granularity than overall runtime.

All possible trace points are included automatically, including the standard HLS-produced hardware accelerators, AXI4-Stream interfaces that serve data to or from an accelerator core, and the accelerator control code in software (stub code). See the *SDx Pragma Reference Guide* ([UG1253](#)) for more information.

As with application debugging, for event tracing, you must connect a board to the host PC using JTAG.

Also, execute the application using the SDSoC GUI from the host using a debug or run configuration.

Hardware/Software System Runtime Operation

The SDSoc™ Environment compilers implement hardware functions either by cross-compiling them into IP using the Vivado® HLS tool, or by linking them as C-Callable IP as described in the *SDSoC Environment Platform Development Guide* ([UG1146](#)).

Each hardware function callsite 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 18: 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. If, for example, 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 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(0);
    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 set up and initiation
- Data transfer setup
- Hardware/software synchronization barriers ("wait for event")

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

Hardware Tracing

The SDSoC Environment supports hardware event tracing of accelerators cross-compiled using Vivado HLS, and data transfers over AXI4-Stream connections. When the `sdscc/++` linker is invoked with the `-trace` option, it automatically inserts hardware monitor IP cores into the generated system to log these 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 timestamp 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
```

To give you an idea of the approximate resource utilization of these hardware monitor cores, the following table shows the resource utilization of these cores for a Zynq™-7000 (xc7z020-1clg400) device:

Core Name	LUTs	FFs	BRAMs	DSPs
Accelerator	79	18	0	0
AXI4-Stream (basic)	79	14	0	0
AXI4-Stream (statistics)	132	183	0	0

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. The following table shows the resource utilization of this core for a Zynq-7000 (xc7z020-1clg400) device:

Number of Ports	LUTs	FFs	BRAMs	DSPs
1	241	404	0	0
2	307	459	0	0
3	366	526	0	0
4	407	633	0	0
6	516	686	0	0
8	644	912	0	0
16	1243	1409	0	0
32	2190	2338	0	0
63	3830	3812	0	0

Depending on the number of ports (i.e., monitor cores), the integration core will use on average 110 flip-flops (FFs) and 160 look-up tables (LUTs). At the system level for example, the resource utilization for the matrix multiplication template application on the ZC702 platform (using the same xc7z020-1clg400 part) is shown in the table below:

System	LUTs	FFs	BRAMs	DSPs
Base (no trace)	16,433	21,426	46	160
Event trace enabled	17,612	22,829	48	160

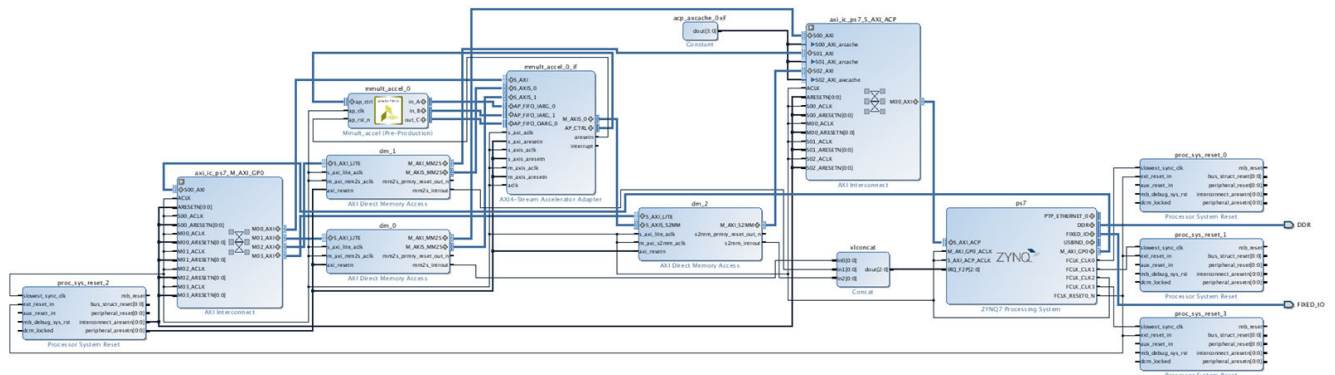
Based on the results above, the difference in designs is approximately 1,000 LUTs, 1,200 FFs, and two BRAMs. This design has a single accelerator with three AXI4-Stream ports (two inputs and one output). When event trace is enabled, four monitors are inserted into the system (one accelerator and three AXI4-Stream monitors), in addition to a single integration core and other associated read-out logic. Given the resource estimations above, 720 LUTs and 700 FFs are from the actual trace monitoring hardware (monitors and integration core). The remaining 280 LUTs, 500 FFs and two BRAMs are from the read-out logic which converts the AXI4-Stream output trace data stream to JTAG. The resource utilization for this read-out logic is static and does not vary based on the design.

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.

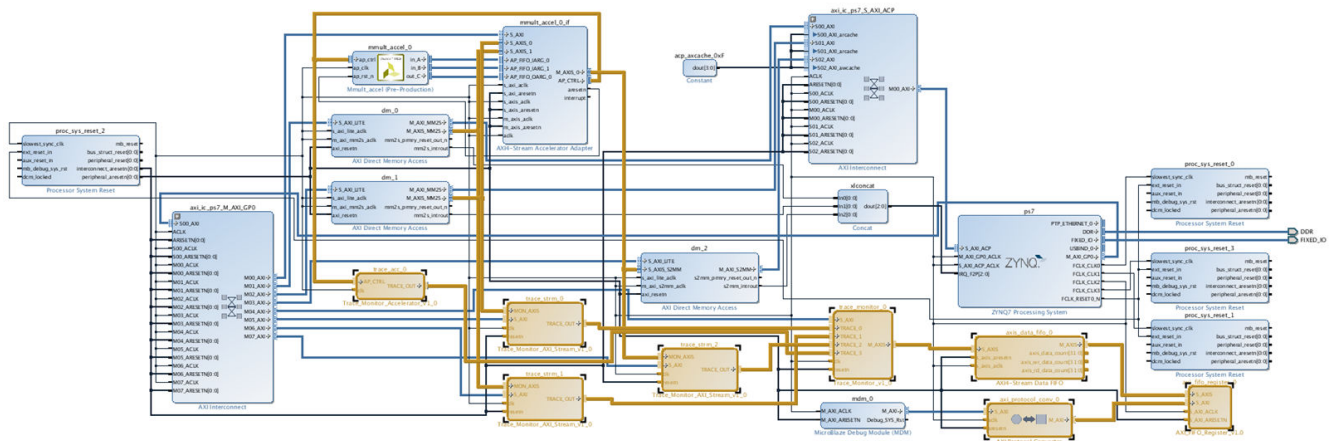
Hardware and software traces are timestamped in hardware and collected into a single trace stream that is buffered up in the programmable logic.

Figure 19: Matrix Multiplication Example Vivado IP Integrator Design Without Tracing Hardware



X16741-040516

Figure 20: Matrix Multiplication Example Vivado IP Integrator Design With Tracing Hardware (Shown in Orange)



X16741-040516

Runtime Trace Collection

Software traces are inserted into the same storage path as the hardware traces and receive a timestamp 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.

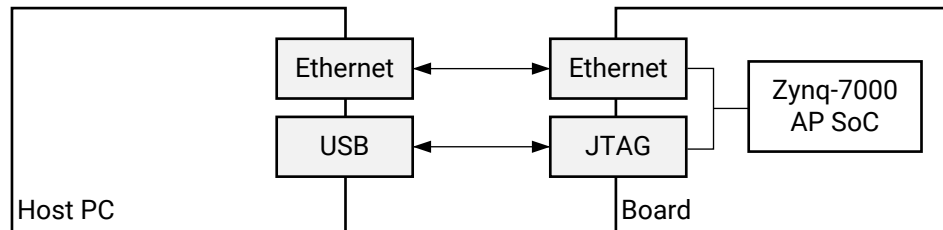
In the SDSoC environment, traces are read back constantly as the program executes attempting to empty the hardware buffer as quickly as possible and prevent buffer overflow; however, trace data only displays when the application is finished.

The board connection requirements are slightly different depending on the operating system (standalone, FreeRTOS, or Linux). For standalone and FreeRTOS, you must download the executable link file (ELF) to the board using the USB/JTAG interface. Trace data is read out over the same USB/JTAG interface as well.

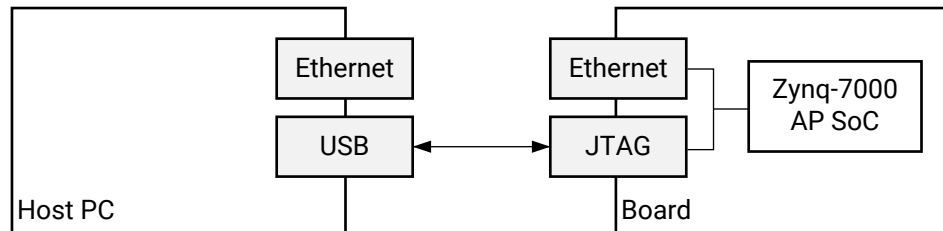
For Linux, the SDSoC Environment assumes the OS boots from the SD card. SDX then copies the ELF and runs it using the TCP/TCF agent running in Linux over the Ethernet connection between the board and host PC. The trace data is read out over the USB/JTAG interface. Both USB/JTAG and TCP/TCF agent interfaces are needed for tracing Linux applications. The figure below shows the connections required.

Figure 21: Connections Required When Using Trace with Different Operating Systems

Linux



Standalone/FreeRTOS

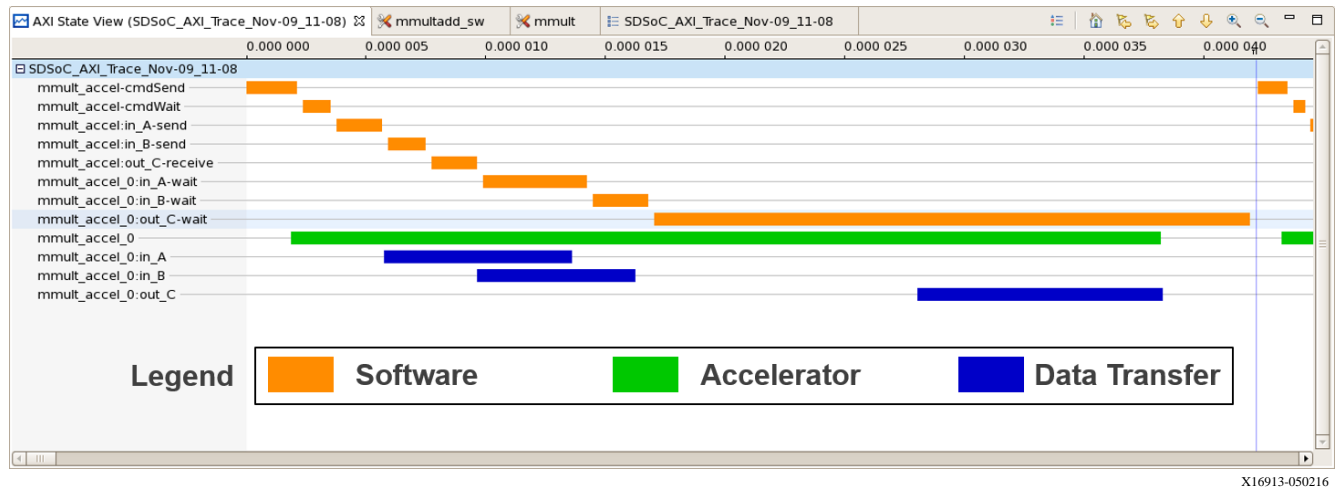


X16744-121417

Trace Visualization

The SDSoC environment GUI provides 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/swimlane 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 22: Example Trace Visualization Highlighting the Different Types of Events



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

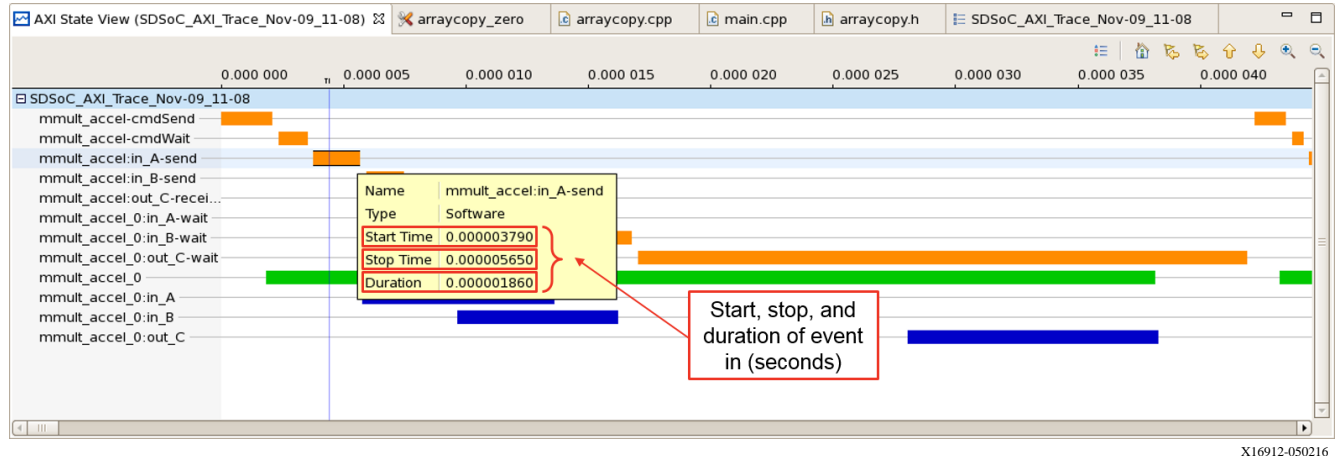
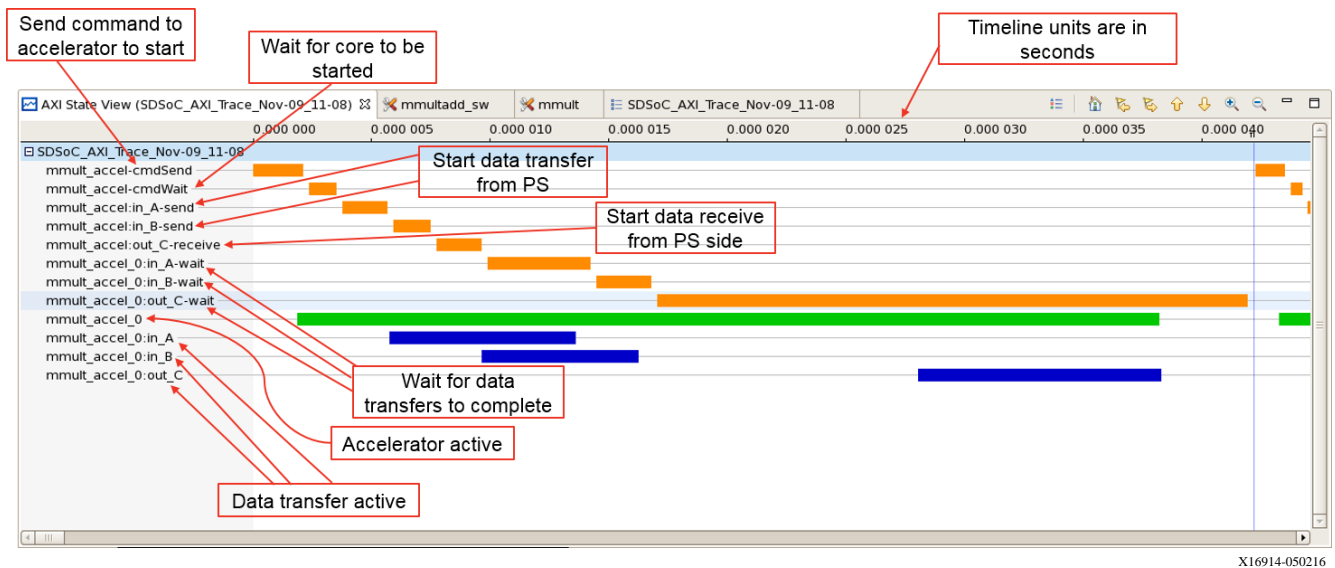


Figure 24: Example Trace Visualization Highlighting the Event Names and Correlation to the User Program



Performance Measurement Using the AXI Performance Monitor

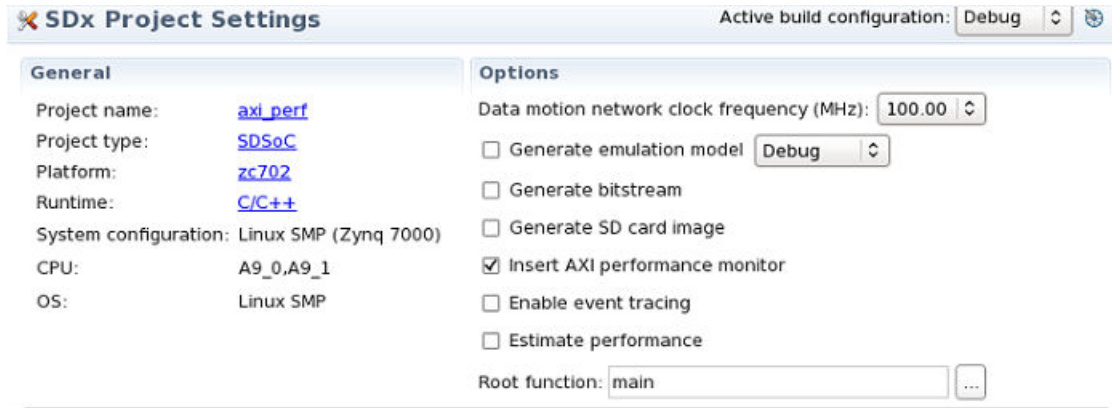
The AXI Performance Monitor (APM) module is used to monitor basic information about data transfers between the processing system (PS) ARM cores and the hardware in the programmable logic (PL). It captures statistics such as number of read/write transactions, throughput, and latency for the AXI transactions on the busses in the system.

In this section we will show how to insert an APM core into the system, monitor the instrumented system, and view the performance data produced.

Creating a Standalone Project and Implementing APM

Open the SDSoc environment and create a new SDSoc Project using any platform or operating system selection. Choose the **Matrix Multiplication and Addition Template**.

In the **SDx Project Settings**, check the option **Insert AXI Performance Monitor**. Enabling this option and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. SDSoc connects the APM to the hardware/software interface ports, which are the Accelerator Coherency Port (ACP), General Purpose Ports (GP) and High Performance Ports (HP).

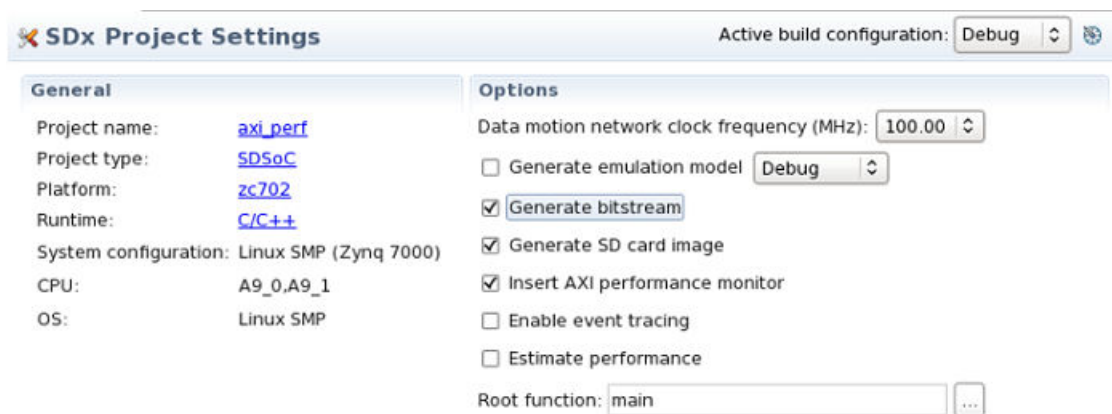


Select the `mmult` and `madd` functions to be implemented in hardware. Clean and build the project using the **Debug** configuration, which is selected by default.

Creating a Linux Project and Implementing APM

Open the SDSoC environment and create a new SDSoC Project using any platform or operating system selection. Choose the **Matrix Multiplication and Addition Template**.

In the **SDx Project Settings**, check the option **Insert AXI Performance Monitor**. Enabling this option and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. SDSoC connects the APM to the hardware/software interface ports, which are the Accelerator Coherency Port (ACP), General Purpose Ports (GP) and High Performance Ports (HP).



Select the `mmult` and `madd` functions to be implemented in hardware. Clean and build the project using the **Debug** configuration, which is selected by default.

Monitoring the Standalone Instrumented System

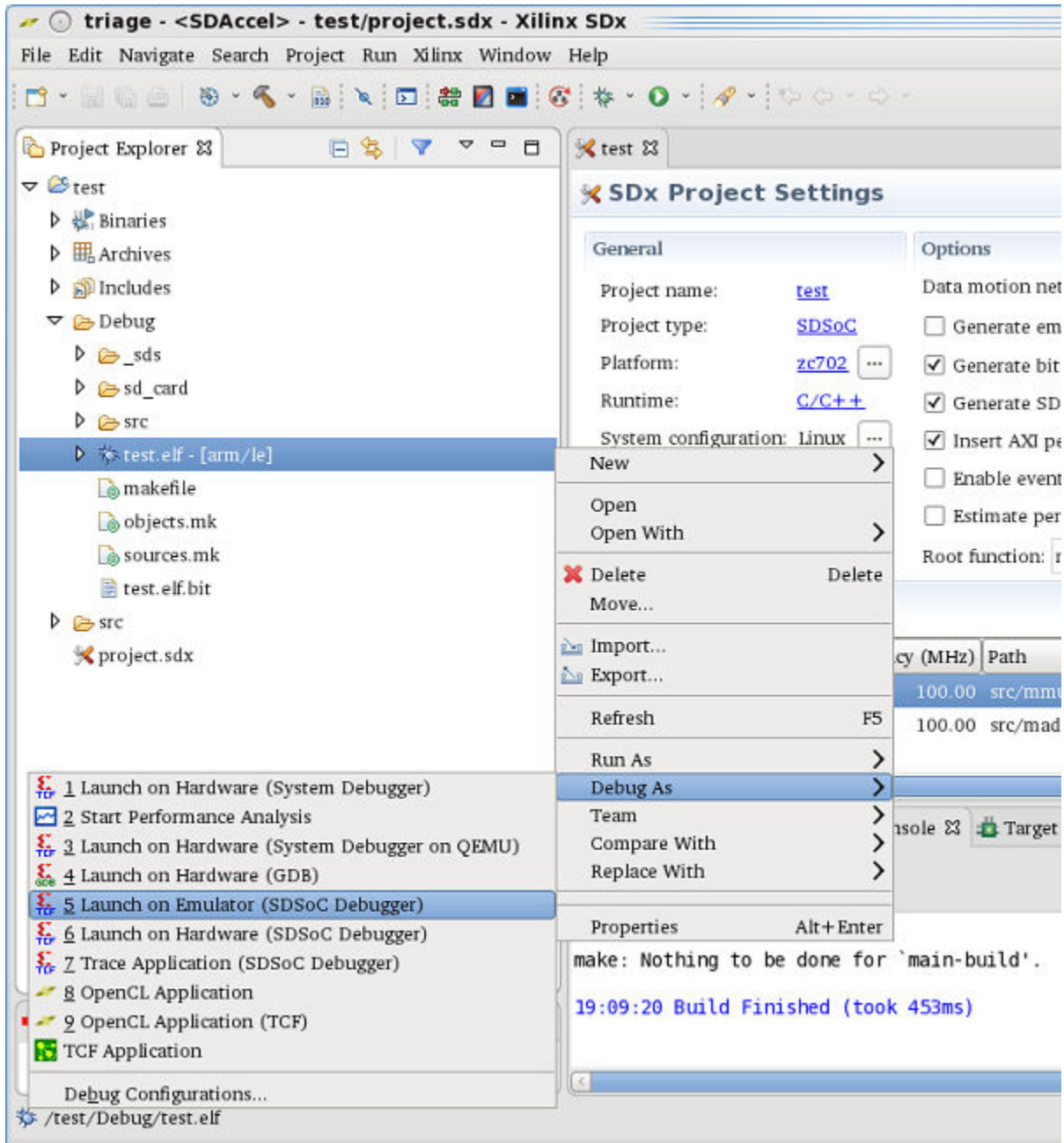
After the build completes, connect the board to your computer and power up the board. Click the **Debug** button to launch the application on the target. Switch to the **Debug** perspective. After programming the PL and launching the ELF, the program halts in main. Click **Window** → **Perspective**.

Select **Performance Analysis** in the **Open Perspective** dialog and click **OK**.

Switch back to the **SDx** perspective.

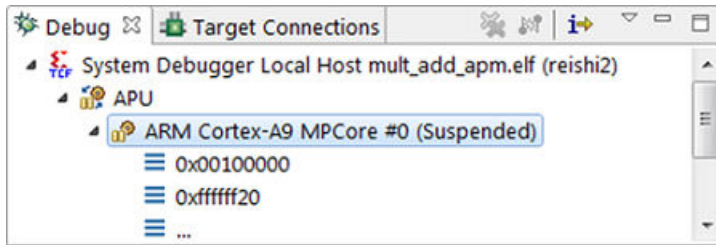
Expand the **Debug** folder in the **Project Explorer** view.

Right-click the ELF executable and select **Debug As** → **Launch on Hardware (SDSoC Debugger)**. If you are prompted to relaunch the application, click **OK**.

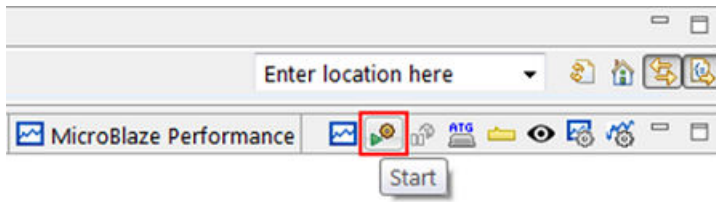


Click **Yes** to switch to the **Debug** perspective. After the application launches and halts at a breakpoint in the main function, switch back to the **Performance Analysis** perspective.

In the **Debug** view in the top left of the perspective, click **ARM Cortex-A9 MPCore #0**.



Next, click the **Start Analysis** button, which opens the **Performance Analysis Input** dialog box.



Check the box to **Enable APM Counters**.

Click the **Edit** button to set up **APM Hardware Information**.

Click the **Load** button in the **APM Hardware Information** dialog. Navigate to `workspace_path/project/Debug/_sds/p0/vp1` and select the `zc702.hdf` file (zc702 is the platform name used in this example - use your platform instead).

Click **Open**, then click **OK** in the **APM Hardware Information** dialog.

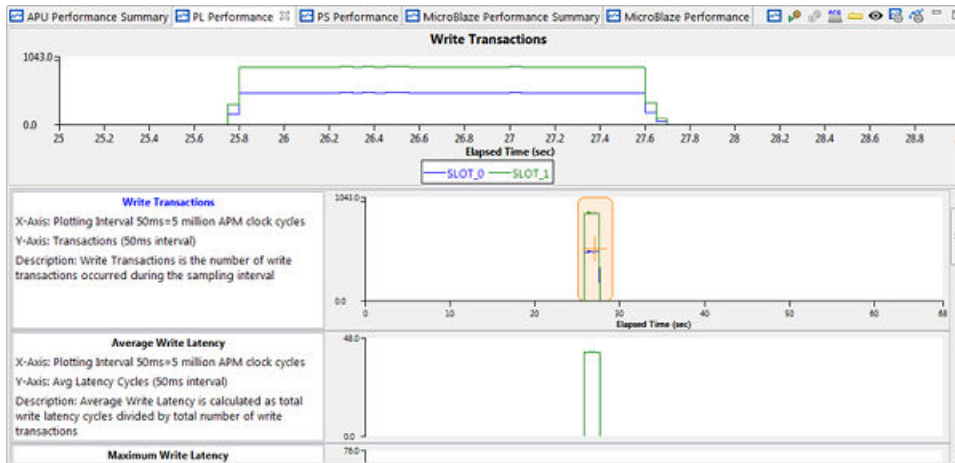
Finally, click **OK** in the **Performance Analysis Input** dialog.

The **Analysis** views open in the **PL Performance** tab. Click the **Resume** button to run the application.

After your program completes execution, click the **Stop Analysis** button. If prompted by the **Confirm Perspective Switch** dialog to stay in the **Performance Analysis** perspective, click **No**.



Scroll through the analysis plots in the lower portion of the perspective to view different performance statistics. Click in any plot area to show a bigger version in the middle of the perspective. The orange box below allows you to focus on a particular time slice of data.



Monitoring the Linux Instrumented System

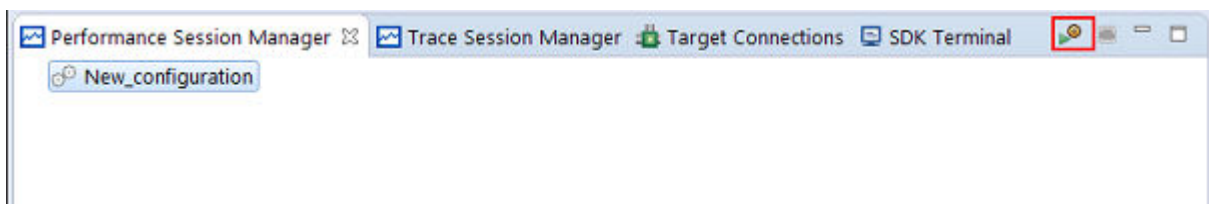
After the build completes, copy the contents of the `sd_card` directory onto an SD card, and boot Linux on the board. Connect the board to your computer (both UART and JTAG cables). Set up the Linux TCF agent target connection with the IP address of the board. Click the **Debug** button to launch the application on the target. Switch to the **Debug** perspective. After launching the ELF, the program halts in main.

Create a new Run Configuration by selecting **Run** → **Run Configuration** and double-clicking on **Xilinx C/C++ application (System Debugger)**. Ensure that the **Debug Type** is set to **Attach to running target**, then click **Run** to close the Run Configurations window. Click **Yes** in the Conflict dialog box that says "Existing launch configuration 'System Debugger on Local <your project>.elf' conflicts with the newly launched configuration...".

Switch to the Performance Analysis perspective by clicking on **Window** → **Open Perspective** → **Other**

Select **Performance Analysis** in the **Open Perspective** dialog and click **OK**.

Next, click on the **Start Analysis** button, which opens the **Performance Analysis Input** dialog.

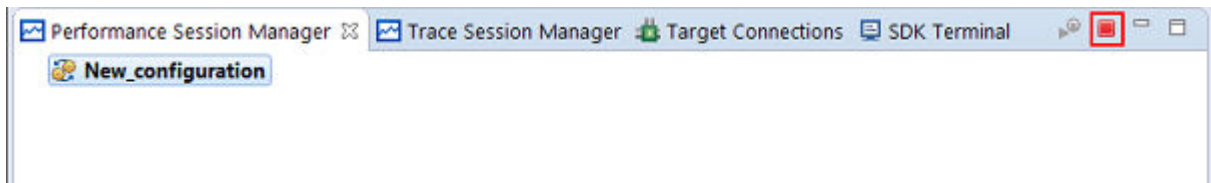


Check the box to **Enable APM Counters**. Click the **Edit** button to set up **APM Hardware Information**.

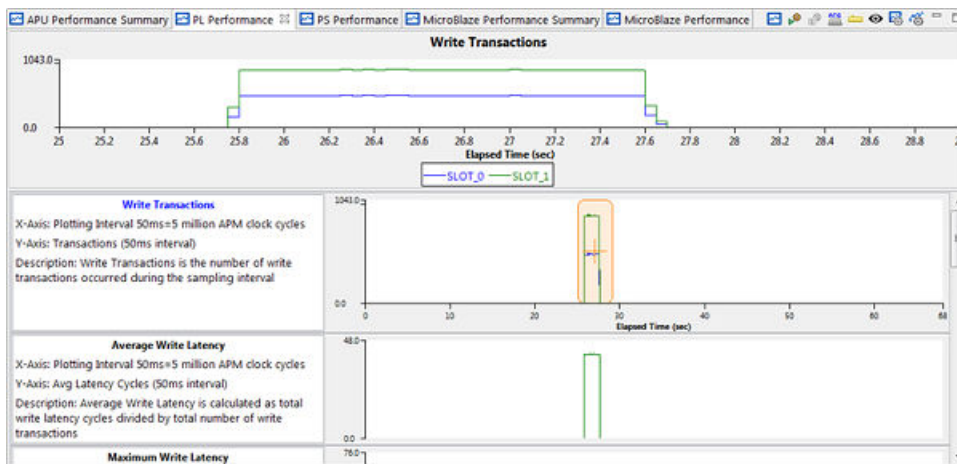
Click the **Load** button in the **APM Hardware Information** dialog. Navigate to `workspace_path/project/Debug/_sds/p0/vp1` and select the `zc702.hdf` file (zc702 is the platform name used in this example - use your platform instead). Click **Open**, then click **OK** in the **APM Hardware Information** dialog. Finally, click **OK** in the **Performance Analysis Input** dialog.

The **Analysis** views open in the **PL Performance** tab. Click the **Resume** button to run the application.

After your program completes execution, click the **Stop Analysis** button. If prompted by the **Confirm Perspective Switch** dialog to stay in the **Performance Analysis** perspective, click **No**.



Scroll through the analysis plots in the lower portion of the perspective to view different performance statistics. Click in any plot area to show a bigger version in the middle of the perspective. The orange box below allows you to focus on a particular time slice of data.



Analyzing the Performance

In this system, the APM is connected to the two ports in use between the PS and the PL: the Accelerator Coherency Port (ACP) and the general purpose AXI port (GP). The multiplier and adder accelerator cores are both connected to the ACP for data input and output. The GP port is used to issue control commands and get the status of the accelerator cores only, not for data transfer. The blue Slot 0 is connected to the GP port, and the green Slot 1 is connect to the ACP.

The APM is configured in Profile mode with two monitoring slots, one for each: ACP and GP ports. Profile mode provides event counting functionality for each slot. The type of statistics computed by the APM for both reading and writing include:

- Transaction Count - Total number of requests that occur on the bus
- Byte Counter - Total number of bytes sent (used for write throughput calculation)
- Latency - Time from the start of the address issuance to the last element sent

The latency and byte counter statistics are used by the APM to automatically compute the throughput (in mega-bytes per second: MB/sec). The latency and throughput values shown are for a 50 millisecond (ms) time interval. Also, minimum, maximum, and averages are also displayed for latency and throughput statistics.

Troubleshooting

1. 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, be sure to 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.
2. Programming and bitstream: The trace functionality is a "one-shot" type of analysis. The timer used for timestamping events is not started until the first event occurs and runs forever afterwards. If you run your software application once after programming the bitstream, the timer will be in an unknown state after your program is finished running. Running your software for a second time will result 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.
3. 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 incorrectly.

4. 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 is cognizant of this and will set 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, etc.) prior to running the application and collecting trace data again. The only way to reset the trace hardware in the design is with reprogramming.

Makefile/Command-Line Flow

You can create a design outside of the SDx IDE. This can be done in a general command-line flow, using individual SDx commands to build and compile the project, or be done with a Makefile flow. An example of the Makefile flow follows.

Note: The Makefile discussed here is from the `cpp/getting_started/hello_vadd` example in the **SDSoC_Examples** repository. See [Appendix A: Getting Started with Examples](#) for information on the example projects.

If you are using the command line interface and writing makefiles outside of the SDSoC™ IDE, you must include the platform using the `-sds-pf` command line option on every call to `sdscc`. You can also specify the software platform, which includes the operating system that runs on the target CPU, using the `-sds-sys-config <system_configuration>` command line option.

```
sdscc -sds-pf <platform path name>
```

Here, the platform is either a file path or a named platform within the `<sdsoc_root>/platforms` directory. To view the available base platforms from the command line, run the following command.

```
$ sdscc -sds -pf -list
```

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

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

# Current Directory
pwd := $(CURDIR)

# Points to Utility Directory
```

```
COMMON_REPO = ../../../../
ABS_COMMON_REPO = $(shell readlink -f $(COMMON_REPO))

# Include Libraries
include $(ABS_COMMON_REPO)/libs/sds_utils/sds_utils.mk
```

At the beginning of the file, as shown above, you will need to define the inputs and any include files needed to build the design. In this design, the default platform of ZCU102 is used, as well as building for emulation. The include statement for the `sds_utils.mk` targets a header file that is used for performance timing (start/stop/pause).

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

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

# Additional sds++ flags - this should be reserved for sds++ flags defined
# at run-time. Other sds++ options should be defined in the makefile data
# section below
ADDL_FLAGS :=
```

The preceding section determines the target OS. If the platform is going to be running Linux, specify Linux, which is the default. To run the application on the bare metal of the processor, then select the `standalone` option. The emulation mode is also specified as either `debug`, to capture waveform data from the PL hardware emulation for viewing and debugging, or `optimized`, for faster emulation without capturing this hardware debug information.

```
# Set to 1 (number one) to enable sds++ verbose output
VERBOSE :=
# Build Executable
EXECUTABLE := run.elf
# Build Directory
BUILD_DIR := build/$(PLATFORM)_$(TARGET_OS)_$(TARGET)
```

The executable file is defined, and the output directory for the build. For this project, the build will be located in `build/zcu102_linux_emu` based on the options of the Makefile. Specifying the `BUILD_DIR` this way will let you compare and contrast results between different builds, such as an emulation build and a hardware build.

```
# Source Files
SRC_DIR := src
OBJECTS += \
$(pwd)/$(BUILD_DIR)/main.o \
$(pwd)/$(BUILD_DIR)/vector_addition.o

# SDS Options
HW_FLAGS :=
HW_FLAGS += -sds-hw vadd_accel vector_addition.cpp -sds-end
```



```
EMU_FLAGS :=
ifeq ($(TARGET), emu)
    EMU_FLAGS := -mno-bitstream -mno-boot-files -emulation $(EMU_MODE)
endif
```

With the project and platform defined, you need to define the source files, and specify which function or file will be accelerated on the hardware. The `-sds-hw` flag specifies a function name to be compiled for hardware, and the source file it is found in. The `EMU_FLAGS` disable bitstream generation and bootfile generation for `sds++/sdsc`, and specify the emulation mode.

Remember, `EMU_MODE` was previously set to be `optimized`.

Note: You can define multiple hardware functions in a design by replicating the `HW_FLAGS += * line` with additional flags.

```
# Compilation and Link Flags
IFLAGS := -I.
CFLAGS = -Wall -O3 -c
CFLAGS += -MT"$@" -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@)"
CFLAGS += -I$(sds_utils_HDRS)
CFLAGS += $(ADDL_FLAGS)
LFLAGS = "$@" "$<"
#-----

SDSFLAGS := -sds -pf $(PLATFORM) \
            -target-os $(TARGET_OS)

# SDS Compiler
CC := sds++ $(SDSFLAGS)
```

Like GCC, the `sds++/sdsc` compiler takes compiler flags. For this project, it will be producing warnings, perform a level 3 optimization, and compile only. In general, most of the standard GCC flags can be used as shown here. Next, the platform and target OS flags are set as defined earlier. Finally the compiler is specified as `sds++` or `sdsc`, and the compiler flags applied.

Note: `-MT`, `-MMD`, `-MP`, `-MF`, etc are all `gcc/g++` flags. Refer to the [GCC documentation](#) to understand what these specific flags will do.

```
all: $(BUILD_DIR)/$(EXECUTABLE)

$(BUILD_DIR)/$(EXECUTABLE): $(OBJECTS)
    mkdir -p $(BUILD_DIR)
    @echo 'Building Target: $@'
    @echo 'Triggerring: SDS++ Linker'
    cd $(BUILD_DIR) ; $(CC) -o $(EXECUTABLE) $(OBJECTS) $(EMU_FLAGS)
    @echo 'SDx Completed Building Target: $@'
    @echo ' '

$(pwd)/$(BUILD_DIR)/%.o: $(pwd)/$(SRC_DIR)/%.cpp
    @echo 'Building file: $<'
    @echo 'Invoking: SDS++ Compiler'
    mkdir -p $(BUILD_DIR)
    cd $(BUILD_DIR) ; $(CC) $(CFLAGS) -o $(LFLAGS) $(HW_FLAGS)
    @echo 'Finished building: $<'
    @echo ' '
```

In this part of the Makefile, notice that the `$(pwd)/$(BUILD_DIR)/%.o` section is building all the individual C/C++ files to their respective object files, and the section above it, `$(Build_DIR)/$(EXECUTABLE)` links these object files together for the final generation of the ELF executable.

The Makefile effectively generates the following sds++ compiler command lines:

```
sds++ -sds-pf zcu102 -target-os linux -sds-sys-config a53_linux -Wall -O3 \
-c -MT"./main.o" -MMD -MP -MF"./main.d" -I.././../libs/sds_utils/ \
-o "./main.o" "./src/main.cpp" -sds-hw vadd_accel vector_addition.cpp -sds-
end
```

```
sds++ -sds-pf zcu102 -target-os linux -sds-sys-config a53_linux -Wall -O3 \
-c -MT"./vector_addition.o" -MMD -MP -MF"./vector_addition.d" -I.././../
libs/sds_utils/ \
-o "./vector_addition.o" "./src/vector_addition.cpp" -sds-hw vadd_accel
vector_addition.cpp -sds-end
```

```
sds++ -sds-pf zcu102 -target-os linux -sds-sys-config a53_linux \
-o run.elf main.o vector_addition.o -mno-bitstream -mno-boot-files -
emulation optimized
```

These three lines do the following things from the Makefile:

1. Compile the `main.cpp` to `main.o` and `vector_addition.cpp` to `vector_addition.o` separately with the system configuration for Linux emulation

Note: To build this design for hardware, the last command would need to generate a bitstream, boot files, and not be set to emulation. This can be done by removing the `-mno-bitstream`, `-mno-boot-files`, and `-emulation optimized` flags.

2. Specify that the function `vadd_accel` is to be accelerated from the `vector_addition.cpp`
3. Link the `main.o` and the `vector_addition.o` together in the executable file of `run.elf`

Makefile Guidelines

The makefiles provided with the designs in `<sdsoc_installation>/samples` folder consolidate all `sdsoc` hardware function options into a single command line. The use of makefiles is not required, but has the benefit of preserving the overall control structure and dependencies for files containing a hardware function.

- You can define make variables to capture the entire SDSoC™ Environment command line, for example: `CC = sds++ ${SDSFLAGS}` for C++ files, invoking `sdsoc` for C files. In this way, all SDSoC Environment options are consolidated in the `${CC}` variable. Define the platform and target OS one time in this variable.

- There must be a separate `-sds-hw/-sds-end` clause in the command line for each file that contains a hardware function. For example:

```
-sds-hw foo foo.cpp -clkid 1 -sds-end
```

For the list of the SDSoc compiler and linker options, see [SDSCC/SDS++ Compiler Commands and Options](#) or use `sdscc --help`.

- If `foo` invokes sub-functions contained in files `foo_sub0.c` and `foo_sub1.c`, use the `-files` option.

```
-sds-hw foo foo_src.c -files foo_sub0.c,foo_sub1.c -sds-end
```

- It is possible to run hardware functions and `zero_copy` data buses at different clock rates to achieve higher performance. To set a clock on the command line, determine the corresponding clock id using `sdscc -sds-pf-info <platform>` and use the `-clkid` option.

```
-sds-hw foo foo_src.c -clkid 1 -sds-end
```

Note: Be aware that it might not be possible to implement the hardware system with some clock selections.

The SDSoc Environment includes the standard SDK toolchain for MicroBlaze™ processors, including `microblaze-xilinx-elf` for developing standalone ("bare-metal") and FreeRTOS applications. A MicroBlaze platform in SDSoc is a standard MicroBlaze processor system built using the Vivado® tools and SDK that must be a self-contained system with a local memory bus (LMB) memory, MicroBlaze Debug Module (MDM), UART, and AXI timer.

By default, the SDSoc system compilers do not generate an SD card image for projects targeting a MicroBlaze platform. A user can package the bitstream and corresponding ELF executable as needed for their application.

To run an application, the bitstream must be programmed onto the device before the ELF can be downloaded to a MicroBlaze core. The SDSoc Environment includes Vivado tools and SDK facilities to create MCS files, insert an ELF file into a bitstream, and boot the system from an SD card.

Guidelines for Invoking SDSCC/SDS++

The SDSoc IDE automatically generates makefiles that invoke `sds++` for all C++ files and `sdscc` for all C files, but the only source files that must be compiled with `sdscc/sds++` are those containing code that:

- Define a hardware accelerated function.
- Call a hardware accelerated function.

- Use `sds_lib` functions, for example, to allocate or memory map buffers that are sent to hardware functions.
- Files that contain functions in the transitive closure of the downward call graph of the above.

A large software project may include many files and libraries that are unrelated to the hardware accelerator and data motion networks generated by `sdscc`. All other source files can safely be compiled with the ARM GNU toolchain.

If the `sdscc` compiler issues errors on source files unrelated to the generated hardware system (for example, from an OpenCV library), you can compile these files through `gcc` instead of `sdscc`.

Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc)

The Xilinx® OpenCL™ Compiler (xocc) is a standalone command line utility for compiling an OpenCL kernel supporting all flows in the SDSoc™ environment. It provides a mechanism for command line users to compile their kernels, which is ideal for compiling host applications and kernels using a makefile.

Following are details of xocc command line format and options.

Syntax:

```
xocc [options] <input_file>
```

Table 1: XOCC Options

Option	Valid Values	Description
<code>--platform <arg></code>	Supported acceleration platforms by Xilinx and third-party board partners	Required Set target Xilinx device. See the <i>SDx Environments Release Notes, Installation, and Licensing Guide (UG1238)</i> for a list of supported devices.
<code>--list_xdevices</code>	N/A	Lists the supported devices.
<code>--target <arg></code>	[sw_emu hw_emu hw]	Specify a compile target. <ul style="list-style-type: none"> sw_emu: CPU emulation hw_emu: Hardware emulation hw: Hardware Default: hw Note: Without the <code>-c</code> or <code>-l</code> option, xocc is run in build mode, an <code>.xclbin</code> file is generated.

Table 1: XOCC Options (cont'd)

Option	Valid Values	Description
<code>--compile</code>	N/A	Optional Run xocc in compile mode, generate .xo file.
<code>--link</code>	N/A	Optional Run xocc in link mode, link .xo input files, generate .xclbin file.
<code>--kernel <arg></code>	Kernel to be compiled from the input .cl or .c / .cpp kernel source code	Required for C/C++ kernels Optional for OpenCL kernels Compile/build only the specified kernel from the input file. Only one -k option is allowed per command. Note: When an OpenCL kernel is compiled without the -k option, all the kernels in the input file are compiled.
<code>--output <arg></code>	File name with .xo or .xclbin extension depending on mode	Optional Set output file name. Default: a .xo for compile mode a .xclbin for link and build mode
<code>--version</code>	N/A	Prints the version and build information.
<code>--help</code>	N/A	Print help.
<code>--define <arg></code>	Valid macro name and definition pair <name>=<definition>	Predefine name as a macro with definition. This option is passed to the openCL preprocessor.
<code>--include <arg></code>	Directory name that includes required header files	Add the directory to the list of directories to be searched for header files. This option is passed to the SDSoc compiler preprocessor.
<code>--kernel-frequency</code>	Frequency (MHz) of the kernel.	Sets a user defined clock frequency in MHz for a the kernel overriding a default value from the DSA.
<code>--nk <arg></code>	<code><kernel_name>: <compute_units></code> (for example, foo:2)	N/A in compile mode Optional in link mode Instantiate the specified number of compute units for the given kernel in the .xclbin file. Default: One compute unit per kernel.
<code>--pk <arg></code>	[kernel_name all]:[none stream pipe memory]	Optional Set a stall profile type for the given kernel(s) Default: none
<code>--max-memory-ports <arg></code>	[all <kernel_name>]	Optional Set the maximum memory port property for all kernels or a given kernel.
<code>--memory-port-data-width <arg></code>	[all <kernel_name>]:<width>	Set the specified memory port data width for all kernels or a given kernel. Valid width values are 32, 64, 128, 256, and 512.

Table 1: XOCC Options (cont'd)

Option	Valid Values	Description
<code>--optimize<arg></code>	Valid optimization levels: 0, 1, 2, 3, s, quick example: <code>--optimize2</code>	<p>These options control the default optimizations performed by the Vivado® hardware synthesis engine.</p> <p>Note: Familiarity with the Vivado tool suite is recommended in order to make the most use of these settings.</p> <ul style="list-style-type: none"> • 0: Default optimization. Reduce compilation time and make debugging produce the expected results. • 1: Optimize to reduce power consumption. This takes more time to compile the design. • 2: Optimize to increase kernel speed. This option increases both compilation time and the performance of the generated code. • 3: This is the highest level of optimization. This option provides the highest level performance in the generated code, but compilation time may increase considerably. • s: Optimize for size. This reduces the logic resources for the kernel • quick: Quick compilation for fast run time. This may result in reduced performance and a greater use of resources in the hardware implementation.
<code>--xp</code>	Refer to the following table, XP Parameters.	<p>Specify detailed parameter and property settings in the Vivado tool suite used to implement the FPGA hardware.</p> <p>Note: Familiarity with the Vivado tool suite is recommended in order to make the most use of these parameters.</p>
<code>--debug</code>	N/A	Generate code for debugging.
<code>--log</code>	N/A	Creates a log for in the current working directory.
<code>--message-rules <arg></code>	Message rule file name	Optional - Specify a message rule file with message controlling rules. See <i>Using the Message Rule File</i> chapter for more details.
<code>--report <arg></code>	Generate [estimate system] reports	<p>Generate a report type specified by <code><arg></code>.</p> <p>estimate: Generate estimate report in <code>report_estimate.txt</code></p> <p>system: Generate the estimate report and detailed hardware reports in report directory</p>
<code>--save-temps</code>	N/A	Save intermediate files/directories created during the compilation and build process.
<code>--report_dir <arg></code>	Directory	Specify a report directory. If the <code>--report</code> option is specified, the default is to generate all reports in the current working directory (cwd).
<code>--log_dir <arg></code>	Directory	Specify a log directory. If the <code>--log</code> option is specified, the default is to generate the log file in the current working directory (cwd).
<code>--temp_dir <arg></code>	Directory	Specify a log directory. If the <code>--save-temps</code> option is specified, the default is to create the temporary compilation and build files in the current working directory (cwd).

Table 1: XOCC Options (cont'd)

Option	Valid Values	Description
--export_script	N/A	<p>This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware.</p> <p>Note: Familiarity with the Vivado tool suite is recommended in order to make the most use of the Tcl file generated by this option.</p> <p>Generates the Tcl script used to execute Vivado HLS <code><kernel_name>.tcl</code> but halts before Vivado HLS starts. The expectation is for the script to be modified and used with the --custom_script option.</p> <p>Not supported for -t sw_emu with OpenCL kernels.</p>
--custom_script	<kernel_name>:<path to kernel Tcl file>	<p>Intended for use with the <code><kernel_name>.tcl</code> file generated with option --export_script.</p> <p>This option allows you to customize the Tcl file used to create the kernel and execute using the customize version of the script.</p>
--jobs <arg>	Number of parallel jobs	<p>Optional</p> <p>This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware.</p> <p>Note: Familiarity with the Vivado tool suite is recommended in order to make the most use of the Tcl file generated by this option.</p> <p>Specify the number of parallel jobs to be passed to the Vivado tool suite for implementation. Increasing the number of jobs allows the hardware implementation step to spawn more parallel processes and complete faster.</p>
--lsf <arg>	bsub command line to pass to LSF cluster Note: This argument is required.	<p>Optional</p> <p>Use IBM Platform Load Sharing Facility (LSF) for Vivado implementation.</p>
input file	OpenCL or C/C++ kernel source file	Compile kernels into a .xo or .xclbin file depending on the xocc mode.
--sp	<kernel_compute_unit_name>. <kernel_port>:<system_port> (for example, k1.M_AXI_GMEM:bank0)	Supported for unified platform. System port mapping. This will replace map_connect for unified platform.
--clkid	index number	Supported for unified platform. Passes the index number to sdx_link. Each index available from selected platform has a different default clock frequency.
--remote_ip_cache	directory	Supported for unified platform. Specify a location for a remote IP cache. Passed to vpl.
--no_ip_cache	X	Display verbose/debug information (including output from Vivado runs).



IMPORTANT!: All examples in the SDSoc installation use Makefile to compile OpenCL applications with gcc and xocc commands, which can be used as references for compiling user applications using xocc.

XP Parameters

Use the `--xp` switch to specify parameter values in SDSoC™. These parameters allow fine grain control over the hardware generated by SDSoC and the hardware emulation process.



IMPORTANT! *Familiarity with the Vivado™ tool suite is recommended in order to make the most use of these parameters.*

Parameters are specified as `parm:<parameter>=<value>`. For example:

```
xocc -xp param:compiler.enableDSAIntegrityCheck=true
-xp param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

The `-xp` command option may be specified multiple times in a single `xocc` invocation, or the value(s) may be specified in a `xocc.ini` file with each option specified on a separate line (without `--xp` switch).

```
param:prop:solution.device_repo_paths=../dsa
param:compiler.preserveHlsOutput=1
```

Upon invocation, `xocc` first looks for an `xocc.ini` file in the `$HOME/.Xilinx/sdx` directory. If the file does not exist there, `xocc` will then look for it in the current working directory. If the same `--xp` parameter value is specified in both the command line and `xocc.ini` file, the command line value will be used.

The following table lists the `-xp` parameters and their values.

Table 2: XP Parameters

Parameter Name	Type	Default Value	Description
<code>param:compiler.enableDSAIntegrityCheck</code>	Boolean	False	Enables the DSA Integrity Check. If this value is set to True, and SDSoC detects a DSA which has been modified outside the of the Vivado® tool suite SDSoC halts operation.
<code>param:compiler.errorOnHoldViolation</code>	Boolean	True	Error out if there is hold violation.
<code>param:compiler.maxComputeUnits</code>	Int	-1	The maximum compute units allowed in the system. Any positive value will overwrite the <code>numComputeUnits</code> setting in the DSA.

Table 2: XP Parameters (cont'd)

Parameter Name	Type	Default Value	Description
<code>param:hw_em.debugLevel</code>	String	OFF	The debug level of the simulator. Option OFF is used for optimized run times, BATCH is for batch runs and GUI for use in GUI-mode
<code>param:hw_em.enableProtocolChecker</code>	Boolean	False	Enables the AXI protocol checker during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design.
<code>param:compiler.interfaceLatency</code>	Int	-1	This option specifies the expected latency on the kernel AXI bus, the number of clock cycles from when bus access is requested until it is granted.
<code>param:compiler.xclDataflowFifoDepth</code>	Int	-1	Specifies the depth of FIFOs used in kernel dataflow region.
<code>param:compiler.interfaceWrOutstanding</code>	Int Range	0	Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256.
<code>param:compiler.interfaceRdOutstanding</code>	Int Range	0	Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256.
<code>param:compiler.interfaceWrBurstLen</code>	Int Range	0	Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceWrOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256.
<code>param:compiler.interfaceRdBurstLen</code>	Int Range	0	Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceRdOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256.

Table 2: XP Parameters (cont'd)

Parameter Name	Type	Default Value	Description
<code>misc:map_connect=<type>. kernel.<kernel_name>. <kernel_AXI_interface> .core. OCL_REGION_0.<dest_port></code>	String	<empty>	Used to map AXI interfaces from a kernel to DDR memory banks. <ul style="list-style-type: none"> • <type> is add or remove. • <kernel_name> is the name of the kernel. • <dest_port> is DDR memory bank M00_AXI, M01_AXI, M02_AXI or M03_AXI.
<code>prop:kernel.<kernel_name>.kernel_flags</code>	String	<empty>	Sets specific compile flags on kernel <kernel_name>. e.g.
<code>prop:solution. device_repo_path</code>	String	<empty>	Specifies the path to the DSA repository.
<code>prop:solution. hls_pre_tcl</code>	String	<empty>	Specifies the path to a Vivado HLS Tcl file, which is executed before the C code is synthesized. This allows Vivado HLS configuration settings to be applied prior to synthesis.
<code>prop:solution. hls_post_tcl</code>	String	<empty>	Specifies the path to a Vivado HLS Tcl file, which is executed after the C code is synthesized.
<code>prop:solution. kernel_compiler_margin</code>	Float	12.5% of the kernel clock period.	The clock margin in ns for the kernel. This value is subtracted from the kernel clock period prior to synthesis to provide some margin for P&R delays.

Table 2: XP Parameters (cont'd)

Parameter Name	Type	Default Value	Description
<code>vivado_prop:<object_type>. <object_name>.<prop_name></code>	Various	Various	<p>This allows you to specify any property used in the Vivado hardware compilation flow.</p> <p>Object_type is run fileset file project</p> <p>The object_name and prop_name values are described in <i>Vivado Design Suite Properties Reference Guide</i>, (UG912)</p> <p>Examples:</p> <pre>vivado_prop:run.impl_1. {STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-fanout_opt}</pre> <pre>vivado_prop:fileset. current.top=foo</pre> <p>Note: For object_type file, current is not supported</p> <p>Note: For object type run the special value of <code>--KERNEL--</code> can be used to specify run optimization settings for ALL kernels, instead of having to specify them one by one</p>

Running Software and Hardware Emulation in XOCC Flow

In the XOCC/Makefile flow, users manage compilation and execution of host code and kernels outside the Xilinx® SDSoC™ development environment. Follow these steps to run software and hardware emulation:

1. Create the emulation configuration file.

For software or hardware emulation, the runtime library needs the information about the devices and how many to emulate. This information is provided to the runtime library by an emulation configuration file. SDSoC provides a utility, `emconfigutil` to automate creation of the emulation configuration file. The following are details of the `emconfigutil` command line format and options:

Option	Valid Values	Description
<code>--platform</code>	Target device	Required: Set target device. Set target Xilinx device. See the <i>SDx Environments Release Notes, Installation, and Licensing Guide (UG1238)</i> for a list of supported devices.
<code>--nd</code>	Any positive integer	Optional: Number of devices. Default is 1.
<code>--od</code>	Valid directory	Optional: Output directory, <code>emconfig.json</code> file must be in the same directory as the host executable.
<code>--xp</code>	Valid Xilinx parameters and properties	Optional: Specify additional parameters and properties. For example: <code>--xp prop:solution.device_repo_paths=my_dsa_path</code> Sets the search path for the device specified in <code>--xdevice</code> option.
<code>-h</code>	NA	Print help messages

The `emconfigutil` command creates the configuration file `emconfig.json` in the output directory.

The `emconfig.json` file must be in the same directory as the host executable.

The following example creates a configuration file targeting two `xilinx-adm-pcie-7v3_1ddr_3_0` devices.

```
$emconfigutil --platform xilinx-adm-pcie-7v3_1ddr_3_0 --nd 2
```

2. Set XILINX_SDX environment variable.

The `XILINX_SDX` environment needs to be set and pointed to the SDSoC installation path for the emulation to work. Below are examples assuming SDSoC is installed in `/opt/Xilinx/SDx/2017.4`

C Shell:

```
setenv XILINX_SDX /opt/Xilinx/SDx/2017.4
```

Bash:

```
export XILINX_SDX=/opt/Xilinx/SDx/2017.4
```

3. Set emulation mode.

Setting `XCL_EMULATION_MODE` environment variable to `sw_emu` or `hw_emu` changes the application execution to emulation mode (`sw_emu` for software emulation and `hw_emu` for hardware emulation) so that the runtime looks for the file `emconfig.json` in the same directory as the host executable and reads in the target configuration for the emulation runs.

C Shell:

```
setenv XCL_EMULATION_MODE sw_emu
```

Bash:

```
export XCL_EMULATION_MODE=sw_emu
```

Unsetting the `XCL_EMULATION_MODE` environment variable will turn off the emulation mode.

4. Run CPU and hardware emulation.

With the configuration file `emconfig.json` and `XCL_EMULATION_MODE` set to `true`, execute the host application with proper arguments to run CPU and hardware emulation:

```
$/host.exe kernel.xclbin
```


Getting Started with Examples

All Xilinx® SDx™ Environments are provided with examples designs. These examples can:

- Be a useful learning tool for both the SDx IDE and Compilation flows such as makefile flows.
- Help you quickly get started in creating a new Application project.
- Demonstrate useful coding styles.
- Highlight important optimization techniques.

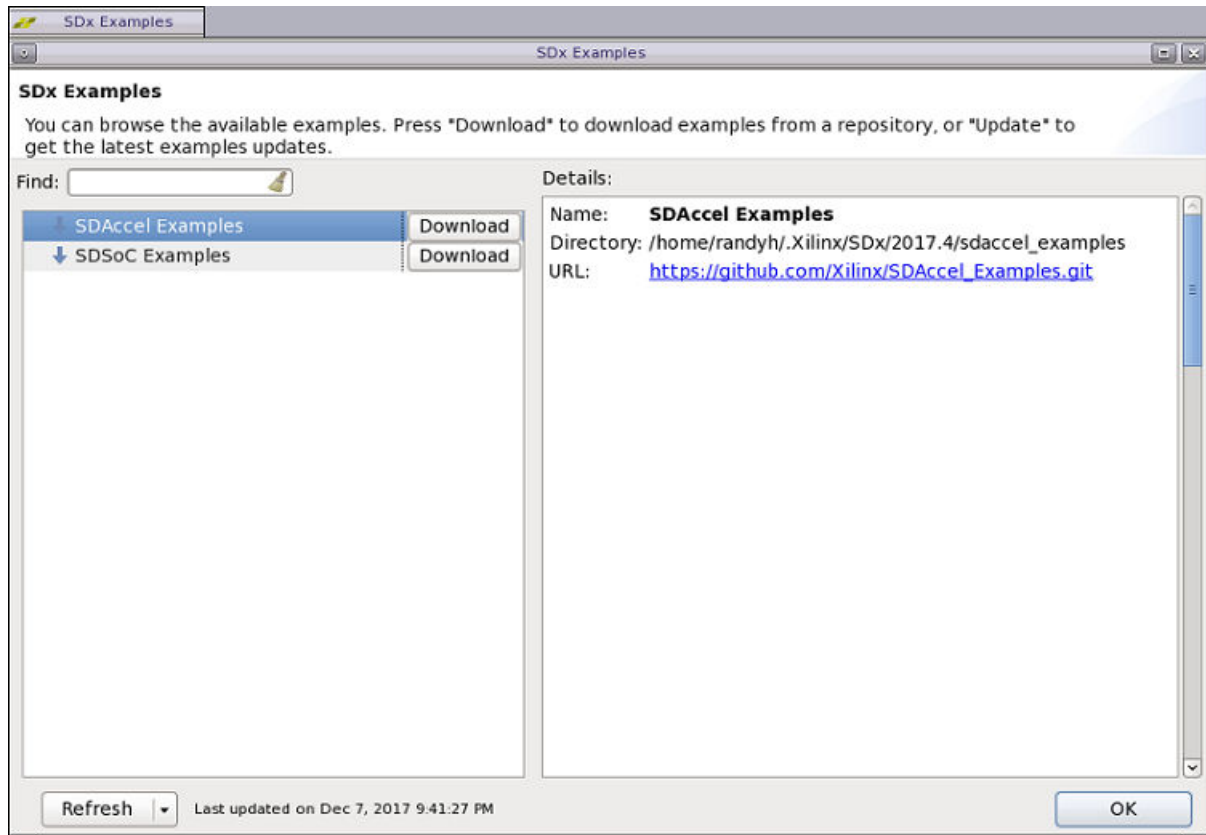
Many examples are available to be downloaded from the Xilinx GitHub repository, although a limited number are also included in the `samples` folder of the software installation.

Installing Examples

You will be asked to select a template for new projects when working through the **New SDx Project** wizard. You can also load template projects from within an existing project, through the **Xilinx → SDx Examples** command.

The first time the SDx Examples dialog box, or the Template page of the New SDx Project wizard is displayed, it will be empty if the SDSoC examples have not been downloaded. The empty dialog box is shown in the following figure. To add Templates to a new project, you must download the SDSoC Examples.

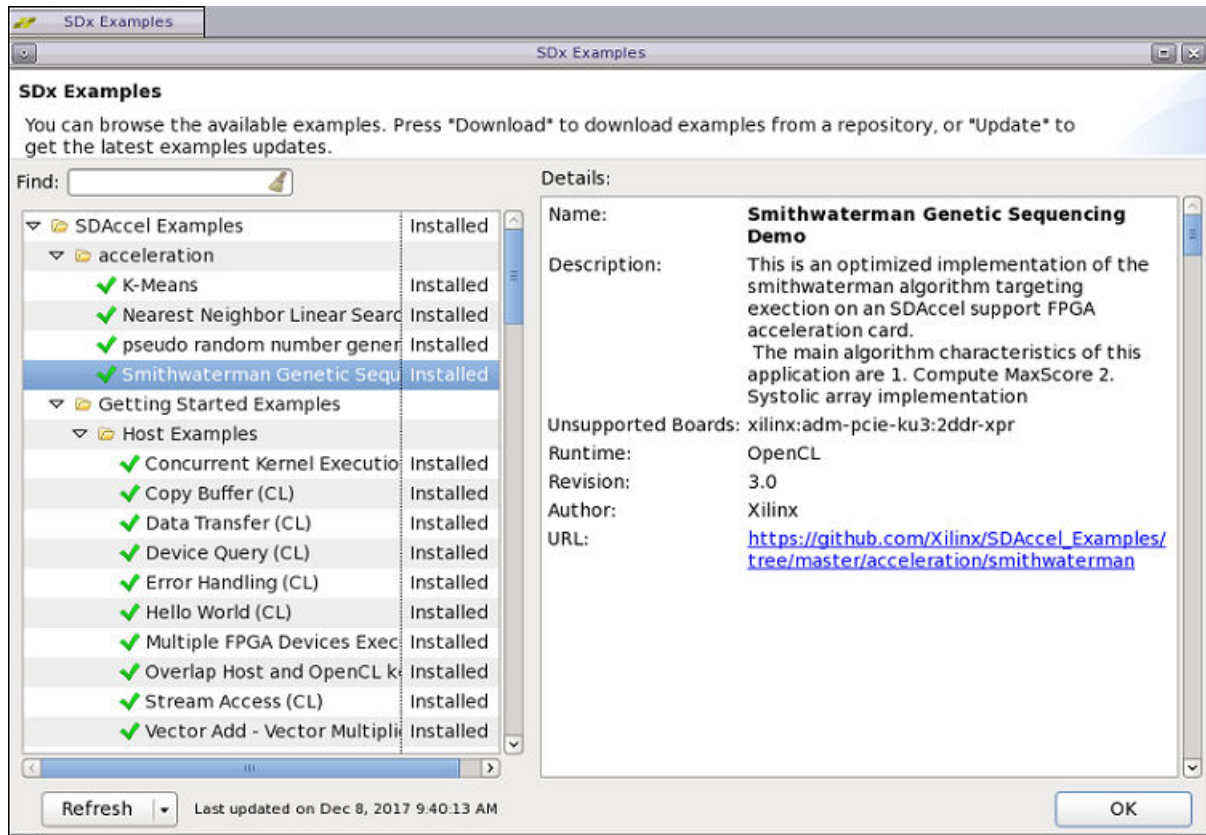
Figure 25: SDSoc Examples - Empty



The left side of the dialog box shows **SDAccel Examples** and **SDSoC Examples**, and has a download command for each category. The right side of the dialog box shows the directory where the examples will be downloaded to, and shows the URL where the examples will be downloaded from.

Click the **Download** button next to the **SDAccel Examples** to download the examples and populate the dialog. The examples are downloaded as shown in the following figure.

Figure 26: SDx Examples - Populated



The command menu at the bottom left of the SDx Examples dialog box provides two commands to manage the repository of examples:

- **Refresh:** Refreshes the list of downloaded examples to download any updates from the GitHub repository.
- **Reset:** Deletes the downloaded examples from the `.Xilinx` folder.

Using Local Copies

While you must download the examples to add Templates when you create new projects, the SDx IDE always downloads the examples into your local `.Xilinx` folder. The download directory cannot be changed from the SDx Examples dialog box. However, you may want to download the example files to a different location from the `.Xilinx` folder for a number of reasons.

You can use the `git` command from a command shell to specify a new destination folder for the downloaded examples:

```
git clone https://github.com/Xilinx/SDAccel_Examples
<workspace>/examples
```

When you clone the `SDAccel_Examples` using the `git` command as shown above, you can use the example files as a resource of application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the Makefiles of the various examples. These include files are automatically populated into the `src` folder of a project when the Template is added through the New SDx Project wizard. However, you will need to locate these files and make them local to your project manually.

You can find the needed files by searching for the file from the location of the cloned repository. For example, you could run the following command from the `examples` folder to find the `xcl2.hpp` file needed for the `vadd` example:

```
find -name xcl2.hpp
```

In addition, the Makefile in each of the example projects has a special command to localize any include files into the project. Use the following form of the `make` command in the example project:

```
make local-files
```

Synthesizeable FIR Filter

Many of the functions in the Vivado HLS source code libraries included in the SDSoC environment do not comply with the SDSoC environment [Coding Guidelines](#). To use these libraries in the SDSoC environment, you typically have to wrap the functions to insulate the SDSoC system compilers from non-portable data types or unsupported language constructs.

The Synthesizeable FIR Filter example demonstrates a standard idiom to use such a library function that in this case, computes a finite-impulse response digital filter. This example uses a filter class constructor and operator to create and perform sample-based filtering. To use this class within the SDSoC environment, the example wraps within a function wrapper as follows.

```
void cpp_FIR(data_t x, data_t *ret)
{
    static CF<coef_t, data_t, acc_t> fir1;
    *ret = fir1(x);
}
```

This wrapper function becomes the top-level hardware function that can be invoked from application code.

Matrix Multiplication

Matrix multiplication is a common compute-intensive operation for many application domains. The SDSoC IDE provides template examples for all base platforms, and the code for these provide instructive use of SDSoC environment system optimizations for memory allocation and memory access described in [Improving System Performance](#) in *SDSoC Environment Profiling and Optimization Guide*, and Vivado HLS optimizations like function inlining, loop unrolling and pipelining, and array partitioning, described in [Optimization Guidelines](#) in *SDSoC Environment Profiling and Optimization Guide*.

Using a C-Callable RTL Library

The SDSoC system compilers can incorporate libraries with hardware functions that are implemented using IP blocks written in register transfer level (RTL) in a hardware description language (HDL) like VHDL or Verilog. The process of creating such a library is described in [Using C-Callable IP Libraries](#). This example demonstrates how to incorporate the library in an SDSoC project.

To build this example in the SDSoC IDE, create a new SDSoC project and select the C-callable RTL Library template. As described in `src/SDSoC_project_readme.txt`, you must first build the library from an SDSoC terminal window at the command line.

To use the library and build the application, you must add the `-l` and `-L` linker options as described in [Using C-Callable IP Libraries](#). Right-click on the project in the **Project Explorer** and select **C/C++ Build Settings** → **SDS++ Linker** → **Libraries**, to add the `-lrtl_arraycopy` and `-L<path to project>` options.

C++ Design Libraries

A number of design libraries are provided with the SDSoC™ Environment installation. The C libraries allow common hardware design constructs and functions to be modeled in C and synthesized to RTL. The following C libraries are provided:

- reVISION and Machine Learning libraries
- Arbitrary Precision Data Types Library
- HLS Stream Library
- HLS Math Library
- HLS Video Library

- HLS IP Library
- HLS Linear Algebra Library
- HLS DSP Library

You can use each of the C libraries in your design by including the library header file. These header files are located in the `include` directory in the SDSoC Environment installation area (`$HOME_SDSOC/Vivado_HLS/include`).



IMPORTANT!: *The header files for the Vivado® HLS C libraries do not have to be in the include path if the C++ code is used in the SDSoC Environment.*

Managing Platforms and Repositories

When you are creating a project, you can manage the platforms that are available for use in SDx application projects, from the **Platform Selection** page of the SDx New Project wizard. This lets you add a new platform for a project as it is being created.


You can also manage the platforms and repositories from an opened project by clicking the Browse button () next to the **Platform** link in the **General** panel of the **Project Settings** window.

Figure 27: **SDAccel Platform Browse**

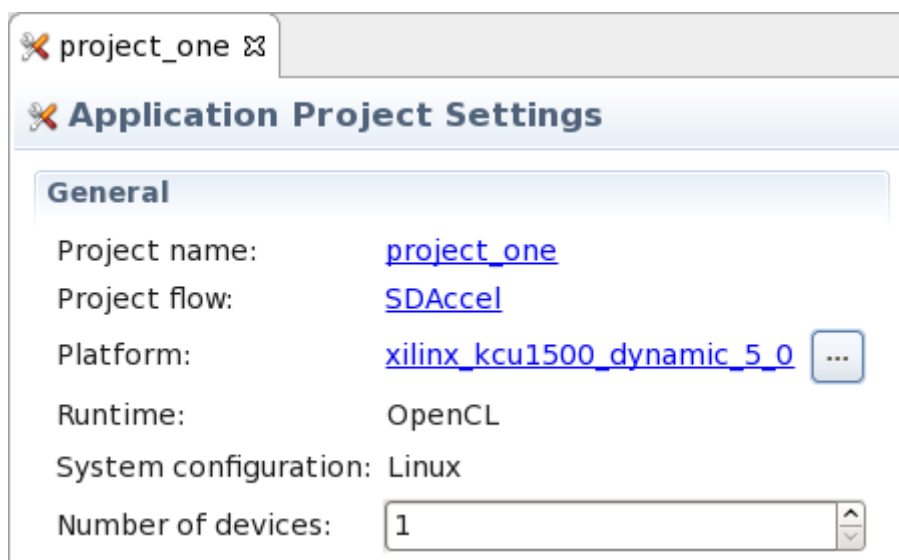
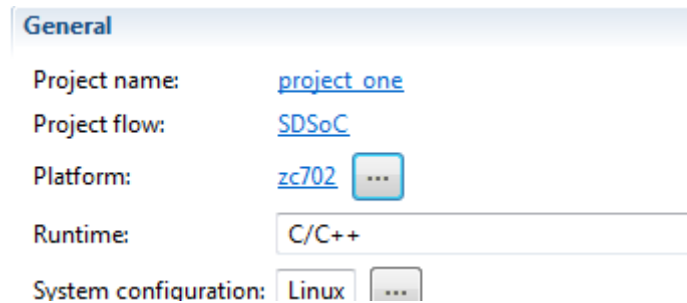


Figure 28: SDSoC Platform Browse



General

Project name: [project one](#)

Project flow: [SDSoC](#)

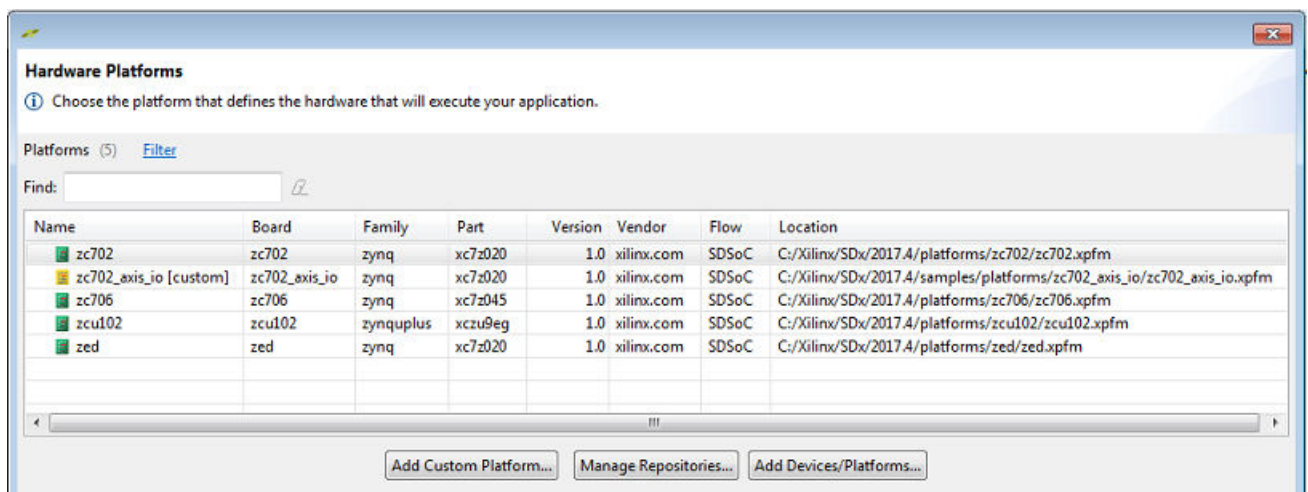
Platform: [zc702](#) ...

Runtime:

System configuration: [Linux](#) ...

This opens the Hardware Platforms dialog box, which lets you manage the available platforms and platform repositories.

Figure 29: Platform Selection



- Add Custom Platform:** Lets you add your own platform to the list of available platforms. Navigate to the top-level directory of the custom platform, select it, and click **OK** to add the new platform. The custom platform is immediately available for selection from the list of available platforms. See *SDSoC Environment Platform Development Guide (UG1146)* for information on creating custom platforms. The **Xilinx → Add Custom Platform** command can also be used to directly add custom platforms to the tool.
- Manage Repositories:** Lets you add or remove standard and custom platforms. If a custom platform is added, the path to the new platform is automatically added to the repositories. Removing any platform from the list of repositories removes the platform from the list of available platforms.
- Add Devices/Platforms:** Lets you manage which Xilinx® devices and platforms are installed. If a device or platform was not selected during the installation process, you can add it at a later time using this command. This command launches the SDx Installer to let you select extra content to install. The **Help → Add Devices/Platforms** command can also be used to directly add custom platforms to the tool.

SDSCC/SDS++ Compiler Commands and Options

This section describes the SDSoC `sdscc/sds++` compiler commands and options.

Compiler Commands

```
sdscc - SDSoC C compiler  
sds++ - SDSoC C++ compiler
```

Command Synopsis

```
sdscc | sds++ [hardware_function_options] [system_options]  
              [performance_estimation_options]  
              [options_passed_through_to_cross_compiler]  
              [-sds-pf platform_name] [-sds-pf-info platform_name]  
              [-sds-pf-list] [-sds-sys-config configuration_name]  
              [-sds-proc processor_name] [-target-os os_name]  
              [-sds-pf-path path] [-sds-image image_name]  
              [-verbose] [-version] [--help] [files]
```

Hardware Function Options

```
[-sds-hw function_name source_file [-clkid clock_id_number]  
[-files hls_file_list] [-hls-tcl hls_tcl_directives_file]  
[-shared-aximm] -sds-end]*
```

Performance Estimation Options

```
[[[-perf-funcs function_name_list -perf-root function_name] |  
[-perf-est data_file] [-perf-est-hw-only]]
```

Note: The `sdscc/sds++` performance estimation options are discussed in *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#)).

System Options

```
[[-ac function_name:clock_id_number]*[-apm] [-bsp-config-file mss_file]
[-bsp-config-merge-file mss_file][--disable-ip-cache] [--dm-sharing <0-3>] [-
dmclkid clock_id_number]
[-emulation mode] [-impl-strategy <strategy>]
[-instrument-stub] [-maxthreads number] [-mno-bitstream][--mno-boot-files]
[-rebuild-hardware]
[-remote-ip-cache cache_directory]
[-synth-strategy <strategy>] [--trace] [--trace-buffer depth] [--trace-no-sw]
[-maxjobs <number>]
[-sdcard <data_directory>][--vpl-ini ini_file] [--xp parameter_value]]
```

The `sdscc/sds++` compilers compile and link C/C++ source files into an application-specific hardware/software system on chip implemented on a Zynq®-7000 All Programmable SoC or Zynq UltraScale+™ MPSoC.

The command usage and options are identical for `sdscc` and `sds++`.

Options not recognized by `sdscc` are passed to the ARM® cross-compiler. Compiler options within an `-sds-hw ... -sds-end` clause are ignored for the `-c foo.c` option when `foo.c` is not the file containing the specified hardware function.

When linking the application ELF, `sdscc` creates and implements the hardware system, and generates an SD card image containing the ELF and boot files required to initialize the hardware system, configure the programmable logic and run the target operating system.

When linking application ELF files for non-Linux targets, for example Standalone or FreeRTOS, default linker scripts found in the folder `<install_path>/platforms/<platform_name>` are used. If a user-defined linker script is required, it can be specified using the `-Wl, -T -Wl,<path_to_linker_script>` linker option.

When building a system containing no functions marked for hardware implementation, `sdscc` uses pre-built hardware when available for the target platform. To force bitstream generation, use the `-rebuild-hardware` option.

Report files are found in the folder `_sds/reports`.

When running Linux applications that use shared libraries, the libraries must be contained in the root file system or SD card, and the path to the libraries added to the `LD_LIBRARY_PATH` environment variable.

Optional PL Configuration After Linux Boot

When `sdscc/sds++` creates a bitstream `.bin` file in the `sd_card` folder, it can be used to configure the PL after booting Linux and before running the application ELF. The embedded Linux command used is `cat bin_file > /dev/xdevcfg`.

General Options

The following command line options are applicable to any `sdscc` invocation or display information for the user.

-sds-pf platform_name

Specify the target platform that defines the base system hardware and software, including operation system and boot files. The `platform_name` can be the name of a platform in the SDSoC™ environment installation, or a file path to a folder containing platform files, with the last component of the path matching the platform name. The platform defines the base hardware and software, including operation system and boot files. Use this option when compiling accelerator source files and when linking the ELF file. Use the `-sds-pf-list` option to list available platforms.

-sds-pf-info platform_name

Display general information about a platform. Use the `-sds-pf-list` option to list available platforms. The information displayed includes available system configurations that can be specified with the `-sds-sys-config system_configuration` option.

-sds-pf-list

Display a list of available platforms and exit (if no other options are specified). The information displayed includes available system configurations that can be specified with the `-sds-sys-config system_configuration` option.

-sds-sys-config configuration_name

Specify the system configuration that defines the software platform used, which includes the target operating system and other settings. The `-sds-pf-list` and `-sds-pf-info` options can be used to list the available system configurations for a platform. When the `-sds-sys-config` option is used, do not specify the `-target-os` option. If the `-sds-sys-config` option is not specified, the default system configuration is used.

-sds-proc processor_name

Specify the processor name to use with the system configuration defined by the `-sds-sys-config` option. A system configuration normally specifies a target CPU, and this option is not required.

-target-os os_name

Specify the target operating system. The selected OS determines the compiler toolchain used, and include file and library paths added by `sdscc.os_name` can be one of the following:

- `linux` : for the Linux OS. This is the default if the command line contains no `-target-os` option
- `standalone` : for standalone or bare-metal applications
- `freertos` : for FreeRTOS. The FreeRTOS Xilinx® port is the same implementation included with the Xilinx Software Development Kit (SDK).

If the `-sds-sys-config system_configuration` option is specified, do not specify the `-target-os` option, because a system configuration itself defines a target operating system. If you do not specify the `-sds-sys-config` but do specify the `-target-os` option, SDSoC searches for a system configuration with an OS that matches the one specified by `-target-os`.

-sds-pf-path path

Specify a search path for platforms. The specified path can contain one or more sub-folders, each of which is a platform folder.

-sds-image image_name

Used with the `-sds-sys-config` option, this specifies the SD card image to use. If this option is not specified, the default image will be used.

-verbose

Print verbose output to STDOUT.

-version

Print the `sdscc` version information to STDOUT.

--help

Print command line help information. Note that two consecutive hyphen or dash characters `--` are used.

The following command line options are applicable only to `sdscc` invocations used to compile a source file.

Hardware Function Options

Hardware function options provide a means to consolidate `sdscc/sds++` options within a `Makefile` to simplify command line calls and make minimal modifications to a pre-existing `Makefile`.

The `-sds-hw` and `-sds-end` options are used in pairs:

- The `-sds-hw` option begins the description of a single function being moved into hardware.
- `-sds-end` option terminates the list of configuration details for that function.

For the next function moved into hardware, there is another pair, with `-sds-hw` as the start of the configuration, and `-sds-end` as the terminator.

The `Makefile` fragment below illustrates the use of `-sds-hw` blocks to collect all options in the `SDSFLAGS` `Makefile` variable and to replace an original definition of `CC` with `sdscc` `{SDSFLAGS}` or `sds++` `{SDSFLAGS}`. Thus the original `Makefile` for an application can be converted to an `sdscc/sds++` compiler `Makefile` with minimal changes.

```
APPSOURCES = add.cpp main.cpp
EXECUTABLE = add.elf

CROSS_COMPILE = arm-xilinx-linux-gnueabi-
AR = ${CROSS_COMPILE}ar
LD = ${CROSS_COMPILE}ld
#CC = ${CROSS_COMPILE}g++
PLATFORM = zc702
SDSFLAGS = -sds-pf ${PLATFORM} \
           -sds-hw add add.cpp -clkid 1 -sds-end \
           -dmclkid 2
CC = sds++ ${SDSFLAGS}

INCDIRS = -I..
LDDIRS =
LDLIBS =
CFLAGS = -Wall -g -c ${INCDIRS}
LDFLAGS = -g ${LDDIRS} ${LDLIBS}

SOURCES := $(patsubst %,../%,${APPSOURCES})
OBJECTS := ${APPSOURCES:.cpp=.o}

.PHONY: all

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
    ${CC} ${OBJECTS} -o $@ ${LDFLAGS}

%.o: ../%.cpp
    ${CC} ${CFLAGS} $<
```

**-sds-hw function_name file [[-clkid <n>] [-files file_list] [-hls-tcl hls_tcl_directives_file]]
-sds-end**

An `sdscc` command line can include zero or more `-sds-hw` blocks. Each block is associated with a top-level hardware function specified as the first argument and its containing source file specified as the second argument. If the file name associated with an `-sds-hw` block matches the source file to be compiled, the options are applied. Options outside of `-sds-hw` blocks are applied where applicable.

When using the AuvizCV library, the `function_name` is the template function instantiation enclosed in double quotes, for example `"auCanny<1080,1920,0,0,3,2,1,1,1>"`, and the file is the source file containing the template function instantiation, for example `au_canny_tb.cpp`.

-clkid <n>

Set the accelerator clock ID to `<n>`, where `<n>` has one of the values listed in the table below. (You can use the command `sdscc -sds-pf-info platform_name` to display the information about a platform.) If the `clkid` option is not specified, the default value for the platform is used. Use the command `sdscc -sds-pf-list` to list available platforms and settings.

Clock ID Values by Platform

Platform	Value of <n>
zc702	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zc706	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zed	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zcu102	0 – 75 MHz
	1 – 100 MHz
	2 – 150MHz
	3 – 200 MHz
	4 – 300 MHz
	5 – 400 MHz
	6 – 600 MHz

-files file_list

Specify a comma-separated list (without white space) of one or more files required to compile the current top-level function into hardware using Vivado® HLS. If any of these files contain source code that is not used by HLS but is required to produce the application executable, they must be compiled separately to create object files (.o), and linked with other object files during the link phase.

When using the AuvizCV library, the `-files` option specifies the path to the source file containing the function template definition, for example `au_canny.hpp`.

-hls-tcl hls_tcl_directives_file

When using the Vivado® HLS tool to synthesize the hardware accelerator, source the specified Tcl file containing HLS directives. During HLS synthesis, `sdsc` creates a `run.tcl` file used to drive the Vivado HLS tool and in this Tcl file, the following commands are inserted:

```
# synthesis directives
create_clock -period <clock_period>
set_clock_uncertainty 27.0%
config_rtl -reset_level low
source <sdsoc_generated_tcl_directives_file>
# end synthesis directives
```

If the `-hls-tcl` option is used, the user-defined Tcl file is sourced after the synthesis directives generated by the SDSoc environment.

-shared-aximm

Share AXIMM ports instead of enabling multiple ports.

-sds-end

Specifies the end of the `-sds-hw` options for the specified `function_name`.

Compiler Macros

Predefined macros allow you to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters `'_'`. The `__SDSCC__` macro is defined whenever `sdsc` or `sds++` is used to compile source files; it can be used to guard code depending on whether it is compiled by `sdsc/sds++` or another compiler, for example GCC.

When `sdsc` or `sds++` compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is defined to be used to guard code depending on whether high-level synthesis is run or not.

The code fragment below illustrates the use of the `__SDSCC__` macro to use the `sds_alloc()` and `sds_free()` functions when compiling source code with `sdscc/sds++`, and `malloc()` and `free()` when using other compilers.

```
#ifdef __SDSCC__
#include <stdlib.h>
#include "sds_lib.h"
#define malloc(x) (sds_alloc(x))
#define free(x) (sds_free(x))
#endif
```

In the example below, the `__SDSVHLS__` macro is used to guard code in a function definition that differs depending on whether it is used by Vivado HLS to generate hardware or used in a software implementation.

```
#ifdef __SDSVHLS__
void mmult(ap_axiu<32,1,1,1> A[A_NROWS*A_NCOLS],
          ap_axiu<32,1,1,1> B[A_NCOLS*B_NCOLS],
          ap_axiu<32,1,1,1> C[A_NROWS*B_NCOLS])
#else
void mmult(float A[A_NROWS*A_NCOLS],
          float B[A_NCOLS*B_NCOLS],
          float C[A_NROWS*B_NCOLS])
#endif
```

In addition, the macro, `HLS_NO_XIL_FPO_LIB`, is defined prior to the include option for Vivado® HLS headers and is visible to Vivado HLS, SDSoc™ analysis tools, and target cross-compilers. This macro disables the use of bit-accurate, floating-point simulation models, instead using the faster (although not bit-accurate) implementation from your local system. Bit-accurate simulation models are not provided for Zynq® and Zynq UltraScale+™ ARM® targets.

System Options

-ac <function_name>:<clock_id_number>

Use the specified clock ID number for an RTL accelerator function instead of the default clock ID.

-apm

Insert an AXI Performance Monitor (APM) IP block to monitor all generated hardware/software interfaces. Within the SDSoc™ IDE, in the Debug Perspective, you can activate the APM prior to running your application by clicking the **Start** button within the Performance Counters View. See the *SDSoC Environment Tutorial* ([UG1028](#)) for more information.

-bsp-config-file <mss_file>

Specify the path to a board support package (BSP) configuration file (.mss) to use instead of an automatically-generated file for a bare-metal based target OS, for example Standalone or FreeRTOS. When using this option, also add an include option specifying the path to your BSP header files: `-I</path/to/includes>`

-bsp-config-merge-file <mss_file>

Specify the path to a board support package (BSP) configuration file (.mss) to use for the base platform and merge using hardware information from the final design to create a BSP configuration file contain user settings for the base platform plus settings for hardware added to the base platform; for example, DMA drivers. This merged BSP configuration file is used instead of an automatically generated file for a bare-metal based target OS; for example, Standalone or FreeRTOS. When using this option, also add an include option specifying the path to your BSP header files: `-I</path/to/includes>`.

-disable-ip-cache

Do not use a cache of pre-synthesized IP cores. The use of IP caching for synthesis reduces the overall build time by eliminating the synthesis step for static IP cores. If the resources required to implement the hardware system exceeds available resources by a small amount, the `-disable-ip-cache` option forces SDSoc to synthesize all IP cores in the context of the design and may reduce resource usage enough to enable implementation.

-dmclkid <n>

Set the data motion network clock ID to <n>, where <n> has one of the values listed in the table below. (You can use the command `sdscc -sds-pf-info platform_name` to display the information about the platform.) If the `dmclkid` option is not specified, the default value for the platform is used. Use the command `sdscc -sds-pf-list` to list available platforms and settings.

Platform	Value of <n>
zc702	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zc706	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz

Platform	Value of <n>
zed	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zcu102	0 – 75 MHz
	1 – 100 MHz
	2 – 150MHz
	3 – 200 MHz
	4 – 300 MHz
	5 – 400 MHz
	6 – 600 MHz

-dm-sharing <n>

The `-dm-sharing <n>` option enables exploration of data mover sharing capabilities if the initial schedule can be relaxed. The level of sharing defaults to 0 (low) if not specified. Other values are 1 (medium), 2 (high) and 3 (maximum – schedule can be relaxed infinitely). For example, to enable maximum data mover sharing, add the `sdscc -dm-sharing 3` option.

-emulation <mode>

Generate files required to run emulation of the system using QEMU for the processing subsystem and the Vivado Logic Simulator for the programmable logic. The `<mode>` specifies the type of simulation models created for the PL, `debug` or `optimized`. In the same directory that you ran `sds++`, type the command `sdsoc_emulator` to run the emulation in the current shell.

-impl-strategy <strategy_name>

Specify the Vivado implementation strategy name to use instead of the default strategy, for example **Performance_Explore**. The strategy name can be found in the **Vivado Implementation Settings** dialog box in the **Strategy** menu, and the strategies are described in *Vivado Design Suite User Guide: Implementation* (UG904). When creating the Tcl file for synthesis and implementation, this command is added: `set_property strategy <strategy_name> [get_runs impl_1]`.

-instrument-stub

The `-instrument-stub` option instruments the generated hardware function stubs with calls to the counter function `sds_clock_counter()`. When a hardware function stub is instrumented, the time required to call send and receive functions, as well as the time spent for waits, is displayed for each call to the function.

-maxjobs <n>

The `-maxjobs <n>` option specifies the maximum number of jobs used for Vivado synthesis. The default is the number of cores divided by 2.

-maxthreads <n>

The `-maxthreads <n>` option specifies the number of threads used in multithreading to speed up certain tasks, including Vivado® placement and routing. The number of threads can be an integer from 1 to 8. The default value is 4, but the tools will not use more threads than the number of cores on the machine. Also, a general limit based on the OS applies to all tasks.

-mno-bitstream

Do not generate the bitstream for the design used to configure the programmable logic (PL). Normally a bitstream is generated by running the Vivado implementation feature, which can be time-consuming with runtimes ranging from minutes to hours depending on the size and complexity of the design. This option can be used to disable this step when iterating over flows that do not impact the hardware generation. The application ELF is compiled before bitstream generation.

-mno-boot-files

Do not generate the SD card image in the folder `sd_card`. This folder includes your application ELF and files required to boot the device and bring up the specified OS. This option disables the creation of the `sd_card` folder in case you would like to preserve an earlier version of this folder.

-rebuild-hardware

When building a software-only design with no functions mapped to hardware, `sdsc` uses a pre-built bitstream if available within the platform, but use this option to force a full system build.

-remote-ip-cache <cache_directory>

Specify the path to a directory used for IP caching for Vivado synthesis. The use of an IP cache can reduce the amount of time required for logic synthesis for subsequent runs. The option `--remote_ip_cache` is also accepted.

-sdcard <data_directory>

Specify an optional directory containing additional files to include in the SD card image.

-synth-strategy <strategy_name>

Specify the Vivado synthesis strategy name to use instead of the default strategy, for example **Flow_RuntimeOptimized**. The strategy name can be found in the **Vivado Synthesis Settings** dialog box in the **Strategy** menu, and the strategies are described in *Vivado Design Suite User Guide: Synthesis* ([UG901](#)). When creating the Tcl file for synthesis and implementation, this command is added: `set_property strategy <strategy_name> [get_runs synth_1]`.

-trace

The `-trace` option inserts hardware and software infrastructure into the design to enable tracing functionality.

-trace-buffer depth

The `-trace-buffer` option specifies the trace buffer depth, which must be at least 16 and a power of 2. If this option is not specified, the default value of 1024 is used.

-trace-no-sw

The `-trace-no-sw` option inserts hardware trace monitors into the design without instrumenting the software when enabling tracing functionality.

-vpl-ini <ini_file>

Specify an initialization file containing one `-xp <parameter_value>` per line, but do not include the `-xp` option itself. This is equivalent to specify multiple `-xp` options on the command line. Advanced users can use this option to customize the Vivado synthesis and implementation flows.

-xp <parameter_value>

Specify a Vivado synthesis or implementation property or parameter, optionally enclosed in double quotes. The `<parameter_value>` uses one of the following forms to set a Vivado property or parameter, respectively.

```
"vivado_prop:run.run_name.<prop_name>=<value>"
"vivado_param:<param_name>=<value>"
```

The following are examples.

The first two examples set a Vivado property to specify a post-synthesis and post-optimization Tcl script, respectively:

```
vivado_prop:run.synth_1.STEPS.SYNTH_DESIGN.TCL.POST=/path/to/postsynth.tcl "
"vivado_prop:run.impl_1.STEPS.OPT_DESIGN.TCL.POST=/path/to/postopt.tcl "
```

The following example sets the implementation strategy and is equivalent to using the `sdscc/sds++ -impl-strategy` option. It illustrates a method for setting a Vivado property for the implementation run named `impl_1`.

```
"vivado_prop:run.impl_1.strategy=Performance_Explore"
```

The following example below sets the maximum number of threads used by Vivado and is equivalent to using the `sdscc/sds++ -maxthreads` option. It illustrates a method for setting a Vivado parameter:

```
"vivado_param:general.maxThreads=1"
```

Advanced users can use the `-xp` option to customize the Vivado synthesis and implementation flows. The `--xp` option is also accepted.

Normally, Vivado implementation does not produce a bitstream if there are timing violations. To force `sds++` to skip the timing violation check and continue, allowing you to proceed and correct timing issues later, you can use the parameter below:

```
param:compiler.skipTimingCheckAndFrequencyScaling=1
```

Compiler Toolchain Support

The SDSoC™ Environment uses the same GNU ARM® cross-compiler toolchains included with the Xilinx® Software Development Kit (SDK).

The Linaro-based GCC compiler toolchains support the Zynq®-7000 and Zynq UltraScale+™ family devices, and this section includes additional information on toolchain usage that might be useful.

When compiling and linking applications, use only object files and libraries built using the same compiler toolchain and options as those used by the SDSoC Environment. All SDSoC provided software libraries and software components (Linux kernel, root filesystem, BSP libraries, and other pre-built libraries) are built with the included toolchains. If you use `sdscc` or `sds++` to compile object files, the tools automatically insert a small number of options, and if you invoke the underlying toolchains, you must use the same options.

For example, if you use a different Zynq-7000 floating-point application binary interface (ABI), your binary objects are incompatible and cannot be linked with SDSoC Zynq-7000 binary objects and libraries.

The following table summarizes the `sdscc` and `sds++` usage of Zynq-7000 toolchains and options. Where options are listed, you need to specify them only if you use the toolchain `gcc` and `g++` commands directly instead of invoking `sdscc` and `sds++`.

Usage	Description
Zynq-7000 ARM® bare-metal compiler and linker options	<code>-mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=hard</code>
Zynq-7000 ARM bare-metal linker options	<code>-Wl,--build-id=none -specs=<specfile></code> where the <code><specfile></code> contains *startfile: <code>crti%O%s crtbegin%O%s</code>
Zynq-7000 ARM bare-metal compiler	<code>\${SDSOC_install}/SDK/gnu/aarch32/<host>/gcc-arm-none-eabi/bin</code> Toolchain prefix: <code>arm-none-eabi</code> gcc executable: <code>arm-none-eabi-gcc</code> g++ executable: <code>arm-none-eabi-g++</code>
Zynq-7000 SDSoc bare-metal software (lib, include)	<code>\${SDSOC_install}/aarch32-none</code>
Zynq-7000 ARM Linux compiler	<code>\${SDSOC_install}/SDK/gnu/aarch32/<host>/gcc-arm-linux-gnueabi/bin</code> Toolchain prefix: <code>arm-linux-gnueabi</code> gcc executable: <code>arm-linux-gnueabi-gcc</code> g++ executable: <code>arm-linux-gnueabi-g++</code>
Zynq-7000 SDSoc Linux software (lib, include)	<code>\${SDSOC_install}/aarch32-linux</code>

The following table summarizes `sdscc` and `sds++` usage of Zynq UltraScale+ Cortex-A53 toolchains and options. Where options are listed, you only need to specify them if you use the toolchain `gcc` and `g++` commands directly instead of invoking `sdscc` and `sds++`.

Usage	Description
Zynq UltraScale+ ARM bare-metal compiler and linker options	<code>-mcpu=cortex-r5 -DARMR5 -mfloat-abi=hard -mfpv=vfpv3-d16</code>
Zynq UltraScale+ ARM bare-metal linker options	<code>-Wl,--build-id=none</code>
Zynq UltraScale+ ARM bare-metal compiler	<code>\${SDSOC_install}/SDK/gnu/aarch64/<host>/aarch64-none/bin</code> Toolchain prefix: <code>aarch64-none-elf</code> gcc executable: <code>aarch64-none-elf-gcc</code> g++ executable: <code>aarch64-none-elf-g++</code>
Zynq UltraScale+ SDSoc bare-metal software (lib, include)	<code>\${SDSOC_install}/aarch64-none</code>
Zynq UltraScale+ ARM Linux compiler	<code>\${SDSOC_install}/SDK/gnu/aarch64/<host>/aarch64-linux/bin</code> Toolchain prefix: <code>aarch64-linux-gnu</code> gcc executable: <code>aarch64-linux-gnu-gcc</code> g++ executable: <code>aarch64-linux-gnu-g++</code>
Zynq UltraScale+ SDSoc Linux software (lib, include)	<code>\${SDSOC_install}/aarch64-linux</code>

The following table summarizes the `sdscc` and `sds++` usage of Zynq® UltraScale+™ Cortex™-R5 toolchains and options. Where options are listed, you need to specify them only if you use the toolchain `gcc` and `g++` commands directly instead of invoking `sdscc` and `sds++`.

Usage	Description
Zynq UltraScale+ ARM bare-metal compiler and linker options	<code>-mcpu=cortex-r5 -DARMR5 -mfloat-abi=hard -mfpu=vfpv3-d16</code>
Zynq UltraScale+ ARM bare-metal linker options	<code>-Wl,--build-id=none</code>
Zynq UltraScale+ ARM bare-metal compiler	<code>\${SDSOC_install}/SDK/gnu/armr5/<host>/gcc-arm-none-eabi/bin</code> Toolchain prefix: <code>armr5-none-eabi</code> gcc executable: <code>armr5-none-eabi-gcc</code> g++ executable: <code>armr5-none-eabi-g++</code>
Zynq UltraScale+ SDSoC bare-metal software (lib, include)	<code>\${SDSOC_install}/armr5-none</code>



IMPORTANT!: When using `sdscc` and `sds++` to compile Zynq-7000 source files, be aware that SDSoC tools that are processing and analyzing source files issue errors if they contain NEON intrinsics. If hardware accelerator (or caller) source files contain NEON intrinsics, guard them using the `--SDSCC--` and `--SDSVHLS--` macros.

For source files that do not contain hardware accelerators or callers but do use NEON intrinsics, you can either compile them directly using the GNU toolchain and link the objects with `sds++`, or you can add the `sdscc/sds++` command line option `-mno-ir` for these source files. The option prevents clang-based tools from being invoked to create an intermediate representation (IR) used in analysis, you are programmatically aware that they are not required (such as no accelerators or callers). For the latter solution, if you are using the SDSoC Environment, you can apply the option on a per-file basis by right-clicking the source file, select **Properties** and go to the **Settings** dialog box under **C/C++ Build Settings** → **Settings**.

Coding Guidelines

This section contains general coding guidelines for application programming using the SDSoC system compilers, with the assumption of starting from application code that has already been cross-compiled for the ARM CPU within the Zynq® device, using the GNU toolchain included as part of the SDSoC environment.

General C/C++ Guidelines

- Hardware functions can execute concurrently under the control of a master thread. Multiple master threads are supported.
- A top-level hardware function must be a global function, not a class method, and it cannot be an overloaded function.
- There is no support for exception handling in hardware functions.
- It is an error to refer to a global variable within a hardware function or any of its sub-functions when this global variable is also referenced by other functions running in software.
- Hardware functions support scalar types up to 1024 bits, including double, long long, packed structs, etc.
- A hardware function must have at least one argument.
- An output or inout scalar argument to a hardware function can be assigned multiple times, but only the last written value will be read upon function exit.
- Use predefined macros to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters `'_'`. For examples, see [SDSCC/SDS++ Compiler Commands and Options](#).
 - The `__SDSCC__` macro is defined and passed as a `-D` option to sub-tools whenever `sdsc` or `sds++` is used to compile source files, and can be used to guard code dependent on whether it is compiled by `sdsc/sds++` or by another compiler, for example a GNU host compiler.
 - When `sdsc` or `sds++` compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is defined and passed as a `-D` option, and can be used to guard code dependent on whether high-level synthesis is run or not.

- In 2017.4 running on the Windows operating system, you will typically have to use these macros to guard code that will be synthesized within SDx with type long long (which should be 64 bits).
- Vivado HLS employs some 32-bit libraries irrespective of the host machine. Furthermore, the tool does not provide a true cross-compilation.

All object code for the ARM CPUs is generated with the GNU toolchains, but the `sdscc` (and `sds++`) compiler, built upon Clang/LLVM frameworks, is generally less forgiving of C/C++ language violations than the GNU compilers. As a result, you might find that some libraries needed for your application cause front-end compiler errors when using `sdscc`. In such cases, compile the source files directly through the GNU toolchain rather than through `sdscc`, either in your makefiles or by setting the compiler to `arm-linux-gnueabi-g++` by right-clicking on the file (or folder) in the **Project Explorer** and selecting **C/C++ Build** → **Settings** → **SDSCC/SDS++ Compiler**.

Hardware Function Argument Types

The SDSoC™ environment `sdscc/sds++` system compilers support hardware function arguments with types that resolve to a single or array of C99 basic arithmetic type (scalar), a `struct` or `class` whose members flatten to a single or array of C99 basic arithmetic type (hierarchical structs are supported), an array of `struct` whose members flatten to a single C99 basic arithmetic type. Scalar arguments must fit in a 1024-bit container. The SDSoC™ environment automatically infers hardware interface types for each hardware function argument based on the argument type and the following pragmas:

```
#pragma SDS data copy|zero_copy
#pragma SDS data access_pattern
```

To avoid interface incompatibilities, you should only incorporate Vivado® HLS interface type directives and pragmas in your source code when `sdscc/sds++` fails to generate a suitable hardware interface directive.

- Vivado® HLS provides arbitrary precision types `ap_fixed<int>`, `ap_int<int>`, and an `hls::stream` class. In the SDSoC environment, `ap_fixed<int>` types must be specified as having widths greater than 7 but less than 1025 ($7 < \text{width} < 1025$). The `hls::stream` data type is not supported as the function argument to any hardware function.
- By default, an array argument to a hardware function is transferred by copying the data, that is, it is equivalent to using `#pragma SDS data copy`. As a consequence, an array argument must be either used as an input or produced as an output, but not both. For an array that is both read and written by the hardware function, you must use `#pragma SDS data zero_copy` to tell the compiler that the array should be kept in the shared memory and not copied.

- To ensure alignment across the hardware/software interface, do not use hardware function arguments that are an array of `bool`.



IMPORTANT!: *Pointer arguments for a hardware function require special consideration. Hardware functions operate on physical addresses, which typically are not available to userspace programs, so pointers cannot be embedded in data structures passed to hardware functions.*



IMPORTANT!:

By default, in the absence of any pragmas, a pointer argument is taken to be a scalar parameter, even though in C/C++ it might denote a one-dimensional array type. The following are the permitted pragmas.

- This pragma provides pointer semantics using shared memory.

```
#pragma SDS data zero_copy
```

- This pragma maps the argument onto a stream, and requires that array elements are accessed in index order. The data copy pragma is only required when the `sdscc` system compiler is unable to determine the data transfer size and issues an error.

```
#pragma SDS data copy(p[0:<p_size>])
#pragma SDS data access_pattern(p:SEQUENTIAL)
```



IMPORTANT!: *When you require non-sequential access to the array in the hardware function, you should change the pointer argument to an array with an explicit declaration of its dimensions, for example, `A[1024]`.*

Hardware Function Call Guidelines

- Stub functions generated in the SDSoC™ environment transfer the exact number of bytes according the compile-time determinable array bound of the corresponding argument in the hardware function declaration. If a hardware function admits a variable data size, you can use the following pragma to direct the SDSoC environment to generate code to transfer data whose size is defined by an arithmetic expression:

```
#pragma SDS data copy|zero_copy(arg[0:<C_size_expr>])
#pragma SDS data zero_copy(arg[0:<C_size_expr>])
```

where the `<C_size_expr>` must compile in the scope of the function declaration.

The `zero_copy` pragma directs the SDSoC environment to map the argument into shared memory.



IMPORTANT!: *Be aware that mismatches between intended and actual data transfer sizes can cause the system to hang at runtime, requiring laborious hardware debugging.*

- Align arrays transferred by DMAs on cache-line boundaries (for L1 and L2 caches). Use the `sds_alloc()` API provided with the SDSoC environment instead of `malloc()` to allocate these arrays.
- Align arrays to page boundaries to minimize the number of pages transferred with the scatter-gather DMA, for example, for arrays allocated with `malloc`.
- You must use `sds_alloc` to allocate an array for the following two cases:
 1. You are using zero-copy pragma for the array.
 2. You are using pragmas to explicitly direct the system compiler to use Simple-DMA.

Note that in order to use `sds_alloc()` from `sds_lib.h`, it is necessary to include `stdlib.h` before including `sds_lib.h`. `stdlib.h` is included to provide the `size_t` type.

Using C-Callable IP Libraries

Using a C-callable IP library is similar to using any software library. You `#include` header files for the library in appropriate source files and use the `sdscc -I<path>` option to compile your source. For example:

```
sdscc -c -I<path to header> -o main.o main.c
```

When you are using the SDSoC IDE, you add these `sdscc` options by right-clicking on your project, selecting **C/C++ Build Settings** → **SDSCC Compiler** → **Directories** (or **SDS++ Compiler** → **Directories** for C++ compilation).

To link the library into your application, you use the `-L<path>` and `-l<lib>` options.

```
sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -l<library_name> -o myApp.elf
```



TIP: As with the standard GNU linkers, for a library called `libMyLib.a`, you use `-lMyLib`.

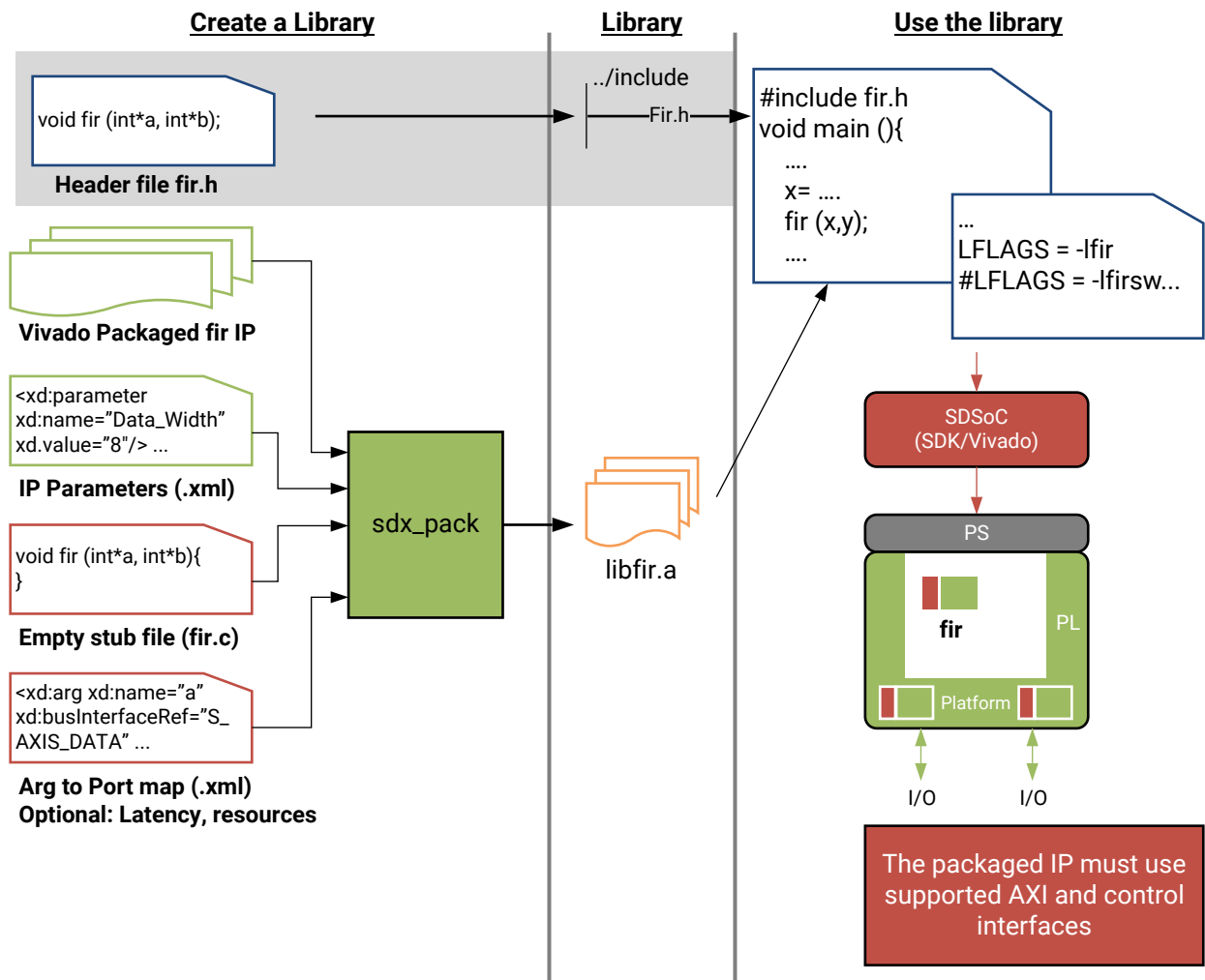
When you are using the SDSoC IDE, you add these `sdscc` options by right-clicking on your project, selecting **C/C++ Build Settings** → **SDS++ Linker** → **Libraries**.

You can find code examples that employ C-callable libraries in the SDSoC™ environment installation under the `samples/fir_lib/use` and `samples/rtl/arraycopy` directories.

C-Callable Libraries

This section describes how to create a C-callable library for IP blocks written in a hardware description language like VHDL or Verilog. User applications can statically link with such libraries using the SDSoC system compilers, and the IP blocks will be instantiated into the generated hardware system. A C-callable library can also provide `sdscc`-compiled applications access to IP blocks within a platform (see [Creating a Library](#)).

Figure 30: Create and Use a C-Callable Library



X14779-011918

The following is the list of elements that are part of an SDSoC platform software callable library:

- [Header File](#)
 - Function prototype
- [Static Library](#)
 - Function definition
 - IP core
 - IP configuration parameters
 - Function argument mapping

Header File

A library must declare function prototypes that map onto the IP block in a header file that can be #included in user application source files. These functions define the function call interface for accessing the IP through software application code.

For example:

```
// FILE: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

Static Library

An SDSoC environment static library contains several elements that allow a software function to be executed on programmable resources.

Function Definition

The function interface defines the entry points into the library, as a function or set of functions that can be called in user code to target the IP. The function definitions can contain empty function bodies since the SDSoC compilers will replace them with API calls to execute data transfers to/from the IP block. The implementation of these calls depend upon the data motion network created by the SDSoC system compilers. The function definition must #include `stdlib.h` and `stdio.h`, which are used when the function body is replaced and compiled.

For example:

```
// FILE: fir.c
#include "fir.h"
#include <stdlib.h>
#include <stdio.h>
void fir(signed char X[N], short Y[N])
{
    // SDSoC replaces function body with API calls for data transfer
}
```



IMPORTANT!: Application code that links to the library must also #include `stdlib.h` and `stdio.h`, which are required by the API calls in the stubs generated by the SDSoC system compilers.

IP Core

An HDL IP core for a C-callable library must be packaged using the Vivado® tools. This IP core can be located in the Vivado tools IP repository or in any other location. When the library is used, the corresponding IP core is instantiated in the hardware system.

You must package the IP for the Vivado Design Suite as described in the *Vivado Design Suite User Guide: Designing with IP* (UG896) and *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118). The Vivado IP Packager tool creates a directory structure for the HDL and other source files, and an IP Definition file (`component.xml`) that conforms to the IEEE-1685 IP-XACT standard. In addition, the packager creates an archive zip file that contains the directory and its contents required by the Vivado Design Suite.

The IP can export AXI4, AXI4-Lite, and AXI4 Stream interfaces. The IP control register must exist at address offset `0x0`, and can support two different task protocols:

1. 'none' - in this mode, the control register must be tied to a constant value `0x6`. The core then is assumed to run continuously upon power up, with all data synchronized through AXI4 stream interfaces or through asynchronous read or writes to memory-mapped registers via an axilite bus.
2. 'axilite' - in this mode, the control register must conform to the following specification, which coincides with the `axilite` control interface for an IP generated by Vivado HLS.

The control signals are generally self-explanatory. The `ap_start` signal initiates the IP execution, `ap_done` indicates IP task completion, and `ap_ready` indicates that the IP is can be started. For more details, see the Vivado HLS documentation for the `ap_ctrl_hs` bus definition.

```
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// (COR = Clear on Read, COH = Clear on Handshake)
```



IMPORTANT!: For details on how to integrate HDL IP into the Vivado Design Suite, see *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

IP Configuration Parameters

Most HDL IP cores are customizable at synthesis time. This customization is done through IP parameters that define the IP core's behavior. The SDSoc environment uses this information at the time the core is instantiated in a generated system. This information is captured in an XML file.

The `xd:component` name is the same as the `spirit:component` name, and each `xd:parameter` name must be a parameter name for the IP. To view the parameter names in IP integrator, right-click on the block and select **Edit IP Meta Data** to access the IP Customization Parameters.

For example:

```
<!-- FILE: fir.params.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="fir_compiler">
  <xd:parameter xd:name="DATA_Has_TLAST" xd:value="Packet_Framing"/>
  <xd:parameter xd:name="M_DATA_Has_TREADY" xd:value="true"/>
  <xd:parameter xd:name="Coefficient_Width" xd:value="8"/>
  <xd:parameter xd:name="Data_Width" xd:value="8"/>
  <xd:parameter xd:name="Quantization" xd:value="Integer_Coefficients"/>
  <xd:parameter xd:name="Output_Rounding_Mode" xd:value="Full_Precision"/>
  <xd:parameter xd:name="CoefficientVector"
    xd:value="6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6"/> </
xd:component>
```

Function Argument Map

The SDSoC system compiler requires a mapping from any function prototypes in the library onto the hardware interface defined by the IP block that implements the function. This information is captured in a "function map" XML file. XML attribute literals, for example array sizes, must be constants and not macros (the SDSoC environment does not use macros in header files to resolve literals in the XML file).

The information includes the following.

- XML namespace - the namespace must be defined as `xmlns:xd="https://www.xilinx.com/xd"`
- Function name - the name of the function mapped onto a component
- Component reference - the IP type name from the IP-XACT Vendor-Name-Library-Version identifier.
 - If the function is associated with a platform, then the component reference is the platform name. For example, see *SDSoC Environment Platform Development Guide* ([UG1146](#)).
- C argument name - an address expression for a function argument, for example `x` (pass scalar by value) or `*p` (pass by pointer).

Note: argument names in the function map must be identical to the argument in the function definition, and they must occur in precisely the same order.

- Function argument direction - either `in` (an input argument to the function) or `out` (an output argument to the function). Currently the SDSoC environment does not support `inout` function arguments.
- Bus interface - the name of the IP port corresponding to a function argument. For a platform component, this name is the platform interface `xd:name`, not the actual port name on the corresponding platform IP.
- Port interface type - the corresponding IP port interface type, which currently must be either `aximm` (slave only), `axis`.
- Address offset - hex address, for example, `0x40`, required for arguments mapping onto `aximm` slave ports (this must be a constant).

- Data width – number of bits per datum (this must be a constant).
- Array size – number of elements in an array argument (this must be a constant).

The function mapping for a configuration of the Vivado FIR Filter Compiler IP from `samples/fir_lib/build` is shown below.

```
<!-- FILE: fir.fcnmap.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="https://www.xilinx.com/xd">
  <xd:fcnMap xd:fcnName="fir" xd:componentRef="fir_compiler">
    <xd:arg xd:name="X"
      xd:direction="in"
      xd:portInterfaceType="axis"
      xd:dataWidth="8"
      xd:busInterfaceRef="S_AXIS_DATA"
      xd:arraySize="32"/>
    <xd:arg xd:name="Y"
      xd:direction="out"
      xd:portInterfaceType="axis"
      xd:dataWidth="16"
      xd:busInterfaceRef="M_AXIS_DATA"
      xd:arraySize="32"/>
    <xd:latencyEstimates xd:worst-case="20" xd:average-case="20"
      xd:best-case="20"/>
    <xd:resourceEstimates xd:BRAM="0" xd:DSP="1" xd:FF="200"
      xd:LUT="200"/>
  </xd:fcnMap>
</xd:repository>
```

Packaging RTL IP for SDx

The SDSoC tools include a command line utility, `sdx_pack`, that encapsulates internal data files required to incorporate RTL IP into SDSoC application projects.

Usage

```
sdx_pack -header<header.h/pp>-ip <component.xml> [-param <name>="value"]
[configuration options]
```

Configuration Options

Description	Option
-header <header.h/pp>	(required) Header file with function prototype
-ip<component.xml>	(required) IP packed by the Vivado® IP integrator
-param<name>="value"	IP parameter values
-func <function_name>	(required) Function name

Description	Option
<code>-control<protocol> [=<port> [:offset]]</code>	(required) IP control protocol options: <ul style="list-style-type: none"> • <code>AP_CTRL</code> • <code>AXI</code> • <code>none</code>
<code>-map<sw_name>=<hw_name>:direction[:offset[aximm_name]:direction]]</code>	(required) SW function argument to IP port mapping: <code><swName>=<hwName>:direction[:offset[<aximm_name>:direction]]</code>
<code>-target-cpu<cpu_type></code>	Specify target CPU: <ul style="list-style-type: none"> • <code>cortex-a9</code> • <code>cortex-a53</code> • <code>cortex-r5</code> • <code>microblaze</code> • <code>x86</code>
<code>-target-os<name></code>	Specify target Operating System <ul style="list-style-type: none"> • <code>linux</code> (default) • <code>standalone(bare-metal)</code>
<code>-stub<file.c/pp></code>	Stub file with entry points; can be multiple functions mapping to a single IP.
<code>-verbose</code>	Print verbose output to STDOUT.
<code>-version</code>	Print the <code>sdx_pack</code> version information to STDOUT.
<code>-help</code>	Display information about this command.

Example:

```
sdx_pack -header count.hpp -ip ../ip/component.xml -func count \
-control AXI=S_AXI:0 -map start_value=S_AXI:in:0x8 -map return=S_AXI:out:4
-target-os standalone
```

Where:

- The `count.hpp` specifies the header file defining the function prototype for the `count` function.
- The `component.xml` of the IP generates the packaged Vivado IP for SDx.
- The `-control` specifies the IP control protocol.
- The `-map` specifies the mapping of an argument from the software function to a port on the Vivado IP. Notice the option is used twice in the example above to map `start_value` and `return` arguments to IP ports.

- The `-target-os` option specifies the target operating system.

The `sdx_pack` utility automatically generates:

- `<function_name>.o`: Compiled object code for the specified function.
- `<function_name>.fcnmap.xml`: mapping IP ports to function arguments
- `<function_name>.params.xml`: IP parameters
- `stub.cpp`: C++ file with entry points; can include multiple functions mapping to a single IP

Note: You will need to create an archive file (`.a`) of the `sdx_pack` compiled object code (`.o`) using an appropriate GNU tool chain utility, like `arm-none-eabi-ar`, so that the function can be linked to the appropriate IP during runtime.

Testing a Library

To test a C-callable library, create a program that uses the library, and `#include` the appropriate header file in your source code.

When compiling the code that calls a library function, provide the path to the header file using the `-I` switch:

```
sdscc -c -I<path to header> -o main.o main.c
```

To link against a library, use the `-L` and `-l` switches:

```
sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -lcount -o count.elf
```

In the example above, the compiler uses the library `libcount.a` located at `<path to library>`.

C-Callable Library Examples

You can find examples of Vivado tools-packaged RTL IP that can be compiled as C-callable libraries in the `samples/rtl` directory of the SDx software installation. These example include `aximm_arraycopy`, `axis_arraycopy`, and `count`.

In each of these examples there are three folders:

- `app`: Contains the top-level application example that uses the function from the C-callable library.
- `ip`: Contains the RTL IP for packaging by the Vivado Design Suite, that defines the hardware function.
- `src`: Contains the header definition (`.h/.hpp`) file for the software function.

Each of these folders contain a Makefile that can be run in order, as well as a top-level Makefile that can be used to run all three of the individual Makefiles. The order of operations to define the C-callable library is:

1. Create the packaged RTL IP using the Vivado Design Suite by running the `../ip/Makefile`. This makefile creates the `component.xml` for the packaged IP that is required for the `sdx_pack` utility.
2. Create the C-callable library object and archive by running the `../src/Makefile`. This makefile uses `sdx_pack` to create the C-Callable library archive, and copies the needed files into the `../lib` and `../inc` folders.
3. Compile the top-level application by running the `../app/Makefile`. This makefile uses the `sdscc/sds++` compiler to compile the main application, linking to the C-callable library.

To build the sample applications open an SDx Terminal window, change directory to the specific example, `../samples/rtl/aximm_arraycopy` folder for example, and run the top-level make file, or run each separate make file in the sequence described above.



TIP: The makefiles run on Linux as written. To run on the Windows Operating System you must remove the comment lines beginning with `#`, and edit the `mkdir` command to remove the `-p` option and use the `'\'` character. An edited Makefile for the `aximm_arraycopy/src` folder is shown below:

```
DEP_FILES = arraycopy.hpp
OBJ_FILES = arraycopy.o

all: $(DEP_FILES) $(OBJ_FILES)
    arm-none-eabi-ar crs libarraycopy.a arraycopy.o

    rm -rf ../lib
    mkdir ../lib
    cp libarraycopy.a ../lib

    rm -rf ../inc
    mkdir ../inc
    cp arraycopy.hpp ../inc

$(OBJ_FILES): $(DEP_FILES)
    sdx_pack -func arraycopy -header arraycopy.hpp -ip ../ip/component.xml \
        -control AXI=s_axi_axilite:0 -map \
        input=s_axi_axilite:in:0x10,m_axi_copy_in:in \
        -map output=s_axi_axilite:in:0x18,m_axi_copy_out:out -map \
        qty=s_axi_axilite:in:0x20 \
        -target-os standalone -verbose

clean:
    rm -rf _sds .Xil
    rm -rf libarraycopy.a
    rm -rf sdx_pack.*

ultraclean: clean
    rm -rf $(OBJ_FILES)
```

The `arraycopy` examples contain two IP cores, each of which copies `M` elements of an array from its input to its output, where `M` is a scalar parameter that can vary with each function call.

- `arraycopy_aximm` - array transfers using an AXI master interface in the IP.
- `arraycopy_axis` - array transfers using AXI4-Stream interfaces.

The register mappings for the IPs are as follows.

```
// arraycopy_aximm
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of a
// bit 31~0 - a[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of b
// bit 31~0 - b[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x2c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on Handshake)

// arraycopy_axis
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x1c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on Handshake)
```

Exporting a Library for GCC

This chapter demonstrates how to use the `sdscc/sds++` compiler to build a library with entry points into hardware functions implemented in programmable logic. This library can later be linked into applications using the standard GCC linker for Zynq®-7000 All Programmable SoCs. In addition to the library, `sdscc` generates a complete boot image that includes an FPGA bitstream containing the hardware functions and data motion network. You can then develop software applications that call into the hardware functions (and fixed hardware) using the standard GCC toolchains. Such code will compile quickly and will not change the hardware. You are still targeting the same hardware system and using the `sdscc`-generated boot environment, but you are then free to develop your software using the GNU toolchain in the software development environment of your choice.

Note: In the current SDSoC release, libraries are not thread-safe, so they must be called into from a single thread within an application, which could consist of many threads and processes.

Note: In the current SDSoC release, shared libraries can be created only for Linux target applications.

Building a Shared Library

To build a shared library, `sdscc` requires at least one accelerator. This example provides three entry points into two hardware accelerators: a matrix multiplier and a matrix adder. You can find these files in the `samples/libmatrix/build` directory.

- `mmult_accel.cpp` – Accelerator code for the matrix multiplier
- `mmult_accel.h` – Header file for the matrix multiplier
- `madd_accel.cpp` – Accelerator code for the matrix adder
- `madd_accel.h` – Header file for the matrix adder
- `matrix.cpp` – Code that calls the accelerators and determines the data motion network
- `matrix.h` – Header file for the library

The `matrix.cpp` file contains functions that define the accelerator interfaces as well as how the hardware functions communicate with the platform (i.e., the data motion networks between platform and accelerators). The function `madd` calls a single matrix adder accelerator, and the function `mmult` calls a single matrix multiplier accelerator. Another function `mmultadd` is implemented using two hardware functions, with the output of the matrix multiplier connected directly to the input of the matrix adder.

```
/* matrix.cpp */
#include "madd_accel.h"
#include "mmult_accel.h"

void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
{
    madd_accel(in_A, in_B, out_C);
}

void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
{
    mmult_accel(in_A, in_B, out_C);
}

void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
in_C[MSIZE*MSIZE],
float out_D[MSIZE*MSIZE])
{
    float tmp[MSIZE * MSIZE];

    mmult_accel(in_A, in_B, tmp);
    madd_accel(tmp, in_C, out_D);
}
```

The `matrix.h` file defines the function interfaces to the shared library, and will be included in the application source code.

```
/* matrix.h */
#ifndef MATRIX_H_
#define MATRIX_H_

#define MSIZE 16

void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE]);

void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE]);

void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
in_C[MSIZE*MSIZE],
float out_D[MSIZE*MSIZE]);

#endif /* MATRIX_H_ */
```

The Makefile shows how the project is built by specifying that the functions `mmult_accel`, `madd`, and `mmult_add` must be implemented in programmable logic.

```
SDSFLAGS = \
    -sds-pf ${PLATFORM} \
    -sds-hw mmult_accel mmult_accel.cpp -sds-end \
    -sds-hw madd_accel madd_accel.cpp -sds-end
```

As is the case for normal shared libraries, object files are generated with position independent code (`-fpic` option).

```
sds++ ${SDSFLAGS} -c -fpic -o mmult_accel.o mmult_accel.cpp
sds++ ${SDSFLAGS} -c -fpic -o madd_accel.o madd_accel.cpp
sds++ ${SDSFLAGS} -c -fpic -o matrix.o matrix.cpp
```

To link the objects files we also follow the standard method and use the `-shared` switch.

```
sds++ ${SDSFLAGS} -shared -o libmatrix.so mmult_accel.o madd_accel.o
matrix.o
```

After building the project, these files will be generated

- `libmatrix.so` – Shared library suitable for linking using GCC and for runtime use
- `sd_card` – Directory containing an SD card image for booting the board

Delivering a Library

The following structure allows compiling and linking into applications using GCC in standard ways.

```
<path_to_library>/include/matrix.h
<path_to_library>/lib/libmatrix.so
<path_to_library>/sd_card
```

Note: The `sd_card` folder is to be copied into an SD card and used to boot the board. This image includes a copy of the `libmatrix.so` file that is used at runtime.

Compiling and Linking Against a Library

The following is an example of using the library with a GCC compiler. The library is used by including the header file `matrix.h` and then calling the necessary library functions.

```
/* main.cpp (pseudocode) */
#include "matrix.h"

int main(int argc, char* argv[])
{
    float *A, *B, *C, *D;
    float *J, *K, *L;
    float *X, *Y, *Z;
    ...
    mmultadd(A, B, C, D);
    ...
    mmult(J, K, L);
    ...
    madd(X, Y, Z);
    ...
}
```

To compile against a library, the compiler needs the header file. The path to the header file is specified using the `-I` switch. You can find example files in the `samples/libmatrix/use` directory.

Note: For explanation purposes, the code above is only pseudocode and not the same as the `main.cpp` file in the directory. The file has more code that allows full compilation and execution.

```
gcc -I <path_to_library>/include -o main.o main.c
```

To link against the library, the linker needs the library. The path to the library is specified using the `-L` switch. Also, ask the linker to link against the library using the `-l` switch.

```
gcc -I <path_to_library>/lib -o main.elf main.o -lmatrix
```

For detailed information on using the GCC compiler and linker switches refer to the GCC documentation.

Use a library at runtime

At runtime, the loader will look for the shared library when loading the executable. After booting the board into a Linux prompt and before executing the ELF file, add the path to the library to the `LD_LIBRARY_PATH` environment variable. The `sd_card` created when building the library already has the library, so the path to the mount point for the `sd_card` must be specified.

For example, if the `sd_card` is mounted at `/mnt`, use this command:

```
export LD_LIBRARY_PATH=/mnt
```

Exporting a Shared Library

The following steps demonstrate how to export an SDSoc environment shared library with the corresponding SD card boot image using the SDSoc environment GUI.

1. Select **File** → **New** → **SDSoC Project** to bring up the **New Project** dialog box.
2. Create a new SDSoc project.
 - a. Type `libmatrix` in the **Project name** field.
 - b. Select **Platform** to be `zc702`.
 - c. Put a checkmark on the **Shared Library** checkbox.
 - d. Click **Next**.
3. Choose the application template.
 - a. Select `Matrix Shared Library` from the **Available Templates**.
 - b. Click **Finish**.

A new SDSoc shared library application project called `libmatrix` is created in the **Project Explorer** view. The project includes two hardware functions `mmult_accel` and `madd_accel` that are visible in the **SDSoC Project Overview**.

4. Build the library.
 - a. In the **Project Explorer** view, select the `libmatrix` project.
 - b. Select **Project** → **Build Project**.

After the build completes, there will be a boot SD card image under the Debug (or current configuration) folder.

Compiling and Running Applications on an ARM Processor

When you make code changes, including changes to hardware functions, it is valuable to rerun a software-only compile to verify that your changes did not adversely change your program. A software-only compile is much faster than a full-system compile, and software-only debugging is a much quicker way to detect logical program errors than hardware and software debugging.

The SDSoC™ Environment includes two distinct toolchains for the ARM®Cortex™-A9 CPU within Zynq®-7000 SoCs.

1. `arm-linux-gnueabihf` - for developing Linux applications
2. `arm-none-eabi` - for developing standalone ("bare-metal") and FreeRTOS applications

For ARM® Cortex™-A53 CPUs within the Zynq® UltraScale+™ MPSoCs, the SDSoC Environment includes two toolchains:

- `aarch64-linux-gnu` - for developing Linux applications
- `aarch64-none-elf` - for developing standalone ("bare-metal") applications

For the ARM Cortex-R5 CPU provided in the Zynq UltraScale+ MPSoCs, the following toolchain is include in the SDSoC environment:

- `armr5-none-eabi` - for developing standalone ("bare-metal") applications

The underlying GNU toolchain is defined when you select the operating system during project creation. The SDSoC system compilers (`sdscc/sds++`) automatically invoke the corresponding toolchain when compiling code for the CPUs, including all source files not involved with hardware functions.

The SDSoC system compilers generate an SD card image by default in a project subdirectory named `sd_card`. For Linux applications, this directory includes the following files:

- `README.TXT` - contains brief instructions on how to run the application
- `BOOT.BIN` - the boot image contains first stage boot loader (FSBL), boot program (U-Boot), and the FPGA bitstream
- `image.ub` - contains the Linux boot image (platforms can be created that include `uImage`, `devicetree.dtb`, and `uramdisk.image.gz` files)

- `<app>.elf`- the application binary executable

To run the application:

1. Copy the contents of `sd_card` directory onto an SD card and insert into the target board.
2. Open a serial terminal connection to the target and power up the board.

Linux boots, automatically logs you in as `root`, and enters a bash shell. The SD card is mounted at `/mnt`, and from that directory you can run `<app>.elf`.

Compiling and Running Applications on a MicroBlaze Processor

A MicroBlaze™ platform in the SDSoC™ Environment is a standard MicroBlaze processor system built using the Vivado® tools and SDK that must be a self-contained system with a local memory bus (LMB) memory, MicroBlaze Debug Module (MDM), UART, and AXI timer.

The SDSoC Environment includes the standard SDK toolchain for MicroBlaze processors, including `microblaze-xilinx-elf` for developing standalone ("bare-metal") and FreeRTOS applications.

By default, the SDSoC system compilers do not generate an SD card image for projects targeting a MicroBlaze platform. You can package the bitstream and corresponding ELF executable as needed for your application.

To run an application, the bitstream must be programmed onto the device before the ELF can be downloaded to the MicroBlaze core. The SDSoC Environment includes Vivado tools and SDK facilities to create MCS files, insert an ELF file into the bitstream, and boot the system from an SD card.

Using External I/O

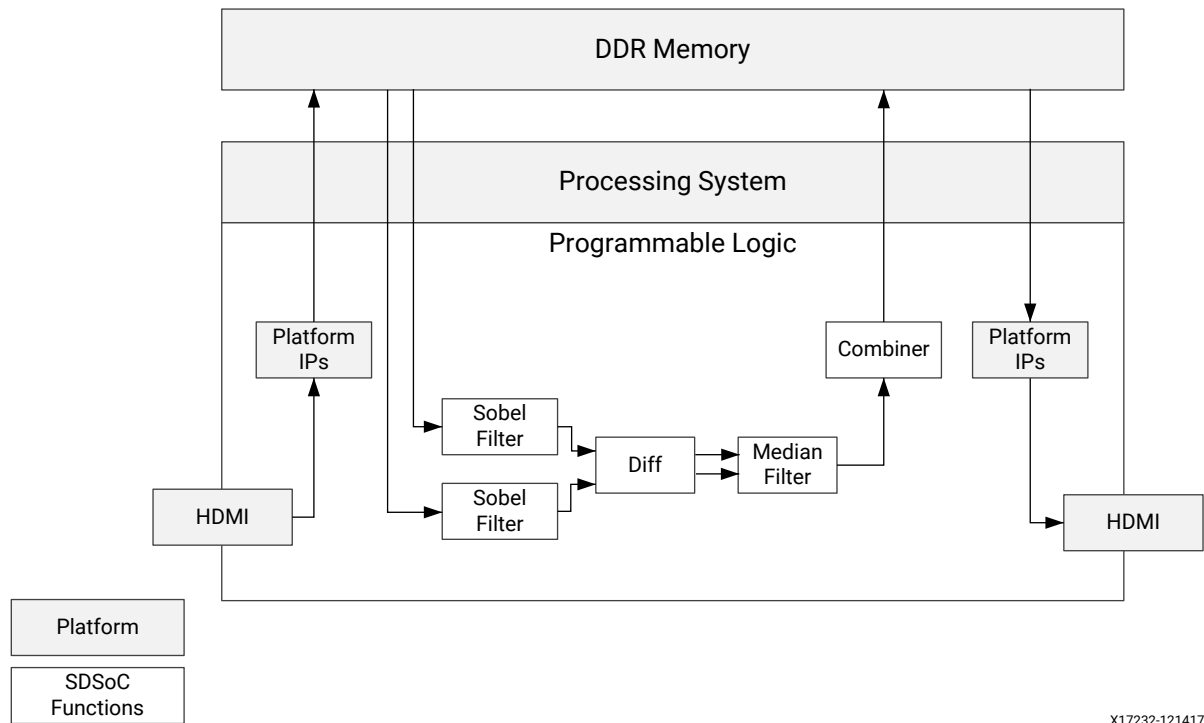
Hardware accelerators generated in the SDSoC™ environment can communicate with system inputs and outputs either directly through hardware connections, or through memory buffers (e.g., a frame buffer). Examples of system I/O include analog-to-digital and digital-to-analog converters, image, radar, LiDAR, and ultrasonic sensors, and HDMI™ multimedia streams. A platform exports stream connections in hardware that are accessed in software by calling platform library functions as described in the following sections. Direct hardware connections are implemented over AXI4-Stream channels, and connections to memory buffers are realized through function calls implemented by the standard data movers supported in the SDSoC Environment.

For information and examples that show how to create SDSoC platforms, refer to the *SDSoC Environment Platform Development Guide* ([UG1146](#))

Accessing External I/O using Memory Buffers

This section uses the motion-detect ZC702 + HDMI IO FMC or ZC706 + HDMI IO FMC platform found on the [SDSoC Downloads Page](#). The following figure shows an example of how a design is configured. The preconfigured SDSoC™ platform is responsible for the HDMI data transfer to external memory. The application must call the platform interfaces to process the data from the frame buffer in DDR memory.

Figure 31: Motion Detect Design Configuration



X17232-121417

The SDSoC Environment accesses the external frame buffer through an accelerator interface to the platform. The `zc702_hdmi` platform provides a software interface to access the video frame buffer through the `Video4Linux2 (V4L2)` API. The V4L2 framework provides an API accessing a collection of device drivers supporting real-time video capture in Linux. This API is the application development platform I/O entry point. In the `motion_demo_processing` example, the following code snippet from `m2m_sw_pipeline.c` demonstrates the function call interface.

```
extern void motion_demo_processing(unsigned short int *prev_buffer,
    unsigned short int *in_buffer,
    unsigned short int *out_buffer,
    int fps_enable,
    int height, int width, int stride);

.
.
.
unsigned short *out_ptr = v_pipe->drm.d_buff[buf_next->index].drm_buff;
unsigned short *in_ptr1 = buf_prev->v4l2_buff;
unsigned short *in_ptr2 = buf_next->v4l2_buff;
v_pipe->events[PROCESS_IN].counter_val++;

motion_demo_processing(in_ptr1, in_ptr2, out_ptr,
    v_pipe->fps_enable,
    (int)m2m_sw_stream_handle.video_in.format.height,
    (int)m2m_sw_stream_handle.video_in.format.width,
    (int)m2m_sw_stream_handle.video_in.format.bytesperline/2);
```

The application accesses this API in `motion_detect.c`, where `motion_demo_processing` is defined and called by the `img_process` function.

```
void motion_demo_processing(unsigned short int *prev_buffer,
                           unsigned short int *in_buffer,
                           unsigned short int *out_buffer,
                           int fps_enable,
                           int height, int width, int stride)
{
    int param0=0, param1=1, param2=2;

    img_process(prev_buffer, in_buffer, out_buffer, height, width,
stride);
}
```

Finally, `img_process` calls the various filters and transforms to process the data.

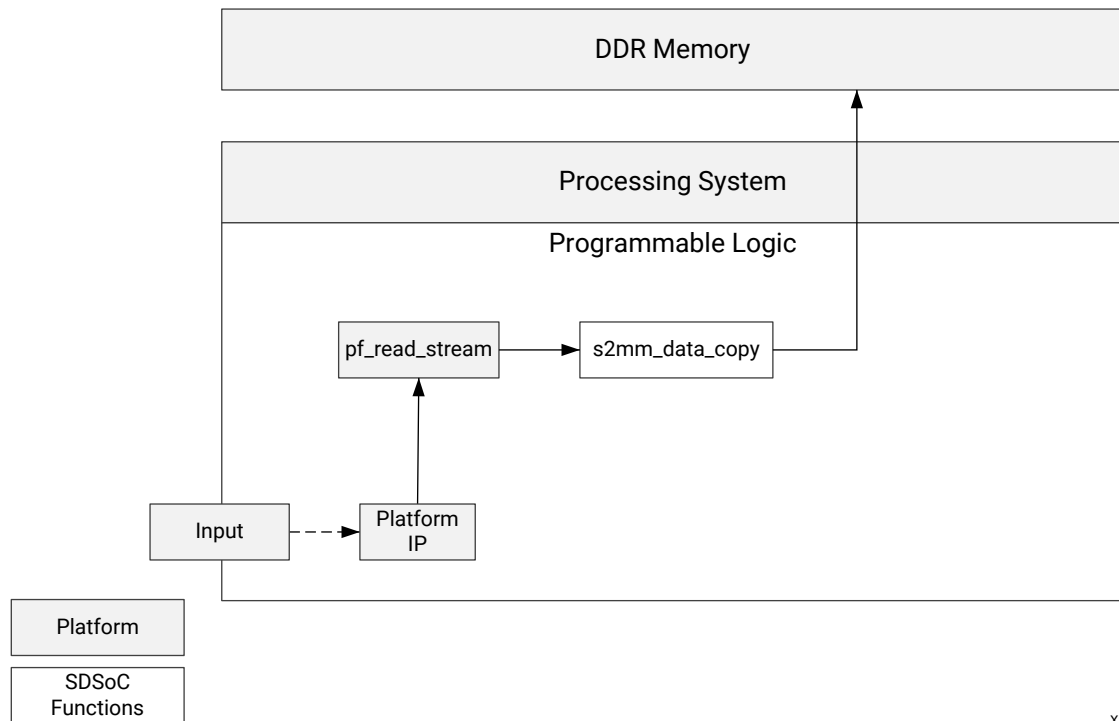
```
void img_process(unsigned short int *frame_prev,
                 unsigned short int *frame_curr,
                 unsigned short int *frame_out,
                 int param0, int param1, int param2)
{
    ...
}
```

By using a platform API to access the frame buffers, the application developer does not program at the driver level to process the video frames. You can find the platform used for the code snippets on the [SDSoC Downloads Page](#) with the name ZC702[ZC706] + HDMI IO FMC. To access the project in the SDSoC environment, create a new project, name the project, and select **Add Custom Platform**. From the **Target Platform** menu, select the downloaded platform named `zc702[zc706]_trd`, click **Next**, and use the template named `Motion Detect`.

Accessing External I/O using Direct Hardware Connections

Whereas the previous example demonstrated how applications can access system I/O through memory buffers, a platform can also provide direct hardware connectivity to hardware accelerators within an application generated in the SDSoC™ Environment. The following figure shows how a function `s2mm_data_copy` communicates to the platform using an AXI4-Stream channel, and writes to DDR memory using the `zero_copy` datamover (implemented as an AXI4 master interface). This design template is called `aximm` design example in the `samples/platforms/zc702_axis_io` platform.

Figure 32: AXI-MM DataMover Design



X17233-121417

In this example, the `zc702_axis_io` platform proxies actual I/O by providing a free-running binary counter (labeled Platform IP in the diagram) running at 50 MHz, connected to an AXI4-Stream Data FIFO IP block that exports an AXI4-Stream master interface to the platform clocked at the data motion clock (which might differ from the 50 MHz input clock).

The direct I/O interface is specified using the `sys_port` pragma for a stream port. In the code snippet below, the direct connection is specified for `stream_fifo_M_AXIS`.

```

#pragma SDS data sys_port (fifo:stream_fifo_M_AXIS)
#pragma SDS data zero_copy(buf)
int s2mm_data_copy(unsigned *fifo, unsigned buf[BUF_SIZE])
{
#pragma HLS interface axis port=fifo
    for(int i=0; i<BUF_SIZE; i++) {
#pragma HLS pipeline
        buf[i] = *fifo;
    }
    return 0;
}

```


Because the only use of the `rbuf0` output is the input to the `s2mm_data_copy` function, the linker creates a direct hardware connection over an AXI4-Stream channel. Because the `s2mm_data_copy` function transfers `buf` using the `zero_copy` data mover, the buffer must be allocated in physically contiguous memory using `sds_alloc`, and released using `sds_free`.

```
int main()
{
    unsigned *bufs[NUM_BUFFERS];
    bool error = false;
    unsigned* rbuf0;

    for(int i=0; i<NUM_BUFFERS; i++) {
        bufs[i] = (unsigned*) sds_alloc(BUF_SIZE * sizeof(unsigned));
    }

    // Flush the platform FIFO of start-up garbage
    s2mm_data_copy(rbuf0, bufs[0]);
    s2mm_data_copy(rbuf0, bufs[0]);
    s2mm_data_copy(rbuf0, bufs[0]);

    for(int i=0; i<NUM_BUFFERS; i++) {
        s2mm_data_copy(rbuf0, bufs[i]);
    }

    error = check(bufs);

    printf("TEST %s\n\r", (error ? "FAILED" : "PASSED"));

    for(int i=0; i<NUM_BUFFERS; i++) {
        sds_free(bufs[i]);
    }
    return 0;
}
```

Information on creating a platform using AXI4-Stream to write to memory directly can be found in the "SDSoC Platform Examples" appendix of *SDSoC Environment Platform Development Guide* ([UG1146](#)).

SDSoC Environment Troubleshooting

There are several common types of issues you might encounter using the SDSoC™ Environment flow.

- Compile/link time errors can 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 can be the result of general software issues such as null-pointer access, or SDSoCEnvironment-specific issues 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.

Troubleshooting Compile and Link Time Errors

Typical compile/link time errors are indicated by error messages issued when running `make`. To probe further, look at the log files and `rpt` files in the `_sds/reports` subdirectory 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.

Some tips for dealing with SDSoC Environment specific errors follow.

- Tool errors reported by tools in the SDSoC Environment chain.
 - Check whether the corresponding code adheres to [Coding Guidelines](#).
 - Check the syntax of pragmas.

- Check for typos in pragmas that might prevent them from being applied to the correct function.
- Vivado® Design Suite High-Level Synthesis (HLS) cannot meet timing requirement.
 - Select a slower clock frequency for the accelerator in the SDSoC IDE (or with the `sdscc/sds++` command line parameter).
 - Modify the code structure to allow HLS to generate a faster implementation. See [Improving Hardware Function Parallelism](#) for more information on how to do this.
- Vivado tools cannot meet timing.
 - In the SDSoC 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).
 - 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 case the resource usage (see the Vivado tools report in `_sds/p0/_vpl/ipi/*.log` and other log files in the subdirectories there) exceeds 80% or so. See the next item for ways to reduce the design size.
- Design too large to fit.
 - 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 [Improving Hardware Function Parallelism](#).
 - 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 sharing of data mover hardware.

Improving Hardware Function Parallelism

This section provides a concise introduction to writing efficient code that can be cross-compiled into programmable logic.

The SDSoC environment employs Vivado HLS as a programmable logic cross-compiler to transform C/C++ functions into hardware. By applying the principles described in this section, you can dramatically increase the performance of the synthesized functions, which can lead to significant increases in overall system performance for your application.

Troubleshooting System Hangs and Runtime Errors

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

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 Vivado HLS, 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];
    }
}
```

Notice that the loop reads the `in_a` stream 19 times but the size of `in_a[]` is 20, so the caller of `f1` would wait forever (or hang) if it waited for `f1` to consume all the data that was streamed to it. Similarly, the caller would wait forever if it waited for `f1` to send 20 int values because `f1` sends only 19. Program errors that lead to such “hangs” can be detected by using system emulation to review whether the data signals are static (review the associated protocol signals `TLAST`, `ap_ready`, `ap_done`, `TREADY`, etc.) or 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 it is left to the user to 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 will be able to 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.

The following list shows other sources of run-time errors:

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

Troubleshooting Performance Issues

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

Estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado HLS report files (`_sds/vhls/.../*.rpt`). In the SDSoC IDE Project Platform Details tab, you can determine the CPU clock frequency, and in the Project Overview you can determine the clock frequency for a hardware function. A latency of X accelerator clock cycles is equal to $X * (\text{processor_clock_freq} / \text{accelerator_clock_freq})$ processor clock cycles. Compare this with the time spent on the actual function call to determine the 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 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 `sdscc -sds-pf-info <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.

Debugging an Application

The SDSoC™ Environment lets you create and debug projects using the SDx IDE. Projects can also be created outside the SDx IDE, using makefiles for instance, and debugged either on the command line or using the SDx IDE.

See the *SDSoC Environment Tutorial* ([UG1028](#)) for information on using the interactive debuggers in the SDx IDE.

SDSoC Environment API

This section describes functions in `sds_lib` available for applications developed in the SDSoC Environment.

Note: To use the library, `#include "sds_lib.h"` in source files. You must include `stdlib.h` before including `sds_lib.h` to provide the `size_t` type declaration.

The SDSoC™ Environment API provides functions to map memory spaces, and to wait for asynchronous accelerator calls to complete.

- `void sds_wait(unsigned int id)`: Wait for the first accelerator in the queue identified by `id`, to complete. The recommended alternative is the use `#pragma SDS wait(id)`, as described in [pragma SDS async](#) in *SDx Pragma Reference Guide*.
- `void *sds_alloc(size_t size)`: Allocate a physically contiguous array of `size` bytes.
- `void *sds_alloc_non_cacheable(size_t size)`:

Allocate a physically contiguous array of `size` bytes that is marked as non-cacheable. Memory allocated by this function is not cached in the processing system. Pointers to this memory should be passed to a hardware function in conjunction with

```
#pragma SDS data mem_attribute (p:NON_CACHEABLE)
```

- `void sds_free(void *memptr)`: Free an array allocated through `sds_alloc()`
- `void *sds_mmap(void *physical_addr, size_t size, void *virtual_addr)`: Create a virtual address mapping to access a memory of `size` bytes located at physical address `physical_addr`.
 - `physical_addr`: physical address to be mapped.
 - `size`: size of physical address to be mapped.
 - `virtual_addr`:
 - If not null, it is considered to be the virtual-address already mapped to the `physical_addr`, and `sds_mmap` keeps track of the mapping.
 - If null, `sds_mmap` invokes `mmap()` to generate the virtual address, and `virtual_addr` is assigned this value.
- `void *sds_munmap(void *virtual_addr)`: Unmaps a virtual address associated with a physical address created using `sds_mmap()`.

- `unsigned long long sds_clock_counter(void)`: Returns the value associated with a free-running counter used for fine grain time interval measurements.
- `unsigned long long sds_clock_frequency(void)`: Returns the frequency (in ticks/second) associated with the free-running counter that is read by calls to `sds_clock_counter`. This is used to translate counter ticks to seconds.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

References

These documents provide supplemental material useful with this webhelp:

1. *SDx Environments Release Notes, Installation, and Licensing Guide* ([UG1238](#))
2. *SDSoC Environment User Guide* ([UG1027](#))
3. *SDSoC Environment Optimization Guide* ([UG1235](#))
4. *SDSoC Environment Tutorial: Introduction* ([UG1028](#))
5. *SDSoC Environment Platform Development Guide* ([UG1146](#))
6. [SDSoC Development Environment web page](#)
7. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
8. *Zynq-7000 All Programmable SoC Software Developers Guide* ([UG821](#))
9. *Zynq UltraScale+ MPSoC Software Developer Guide* ([UG1137](#))
10. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* ([UG850](#))
11. *ZCU102 Evaluation Board User Guide* ([UG1182](#))
12. *PetaLinux Tools Documentation: Workflow Tutorial* ([UG1156](#))
13. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))

14. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))
15. [Vivado® Design Suite Documentation](#)

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 2015-2018 Xilinx®, Inc. Xilinx®, the Xilinx® logo, Artix, ISE, Kintex, Spartan, 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. All other trademarks are the property of their respective owners.

