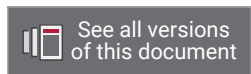


SDSoC Environment User Guide

UG1027 (v2019.1) May 22, 2019



Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
Software Acceleration with SDSoC	Updated description.
Getting Started	Updated description.
Chapter 3: Creating an SDSoC Application	Added description.
Importing C/C++ Sources	Added tip note and File and Folder figure.
Importing C-Callable IP Libraries	Updated numbered list.
Building an SDSoC Library	Added a cross-reference.
Chapter 4: C-Callable IP Libraries	Added sentence to end of first paragraph.
Chapter 5: Debugging Techniques	Updated menu cascade.
Targeting System Emulation	Minor update to description.
Building an SDSoC Library	Added new section.
Shared Library	Removed figures.
Installing Examples	Updated figures and link to Local Copies.
C++ Design Libraries	Updated links.
Appendix B: Managing Platforms and Repositories	Added developer sentence.
01/24/2019 Version 2018.3	
Execution Model of an SDSoC Application	Updated data mover line and async important note.
Chapter 2: SDSoC Environment	Removed table note.
Importing a Project	Added new section.
Selecting Clock Frequencies	Added ZCU102 platform code example in Command Line Options.
C++ Design Libraries	Updated to <Vivado_Install_Dir> filepath.
Guidelines for Invoking SDSOCC/SDS++	Removed section.
12/05/2018 Version 2018.3	
Chapter 2: SDSoC Environment	Updated description and figure.
Software Acceleration with SDSoC	Updated description.
Elements of SDSoC	Updated platform description.
Using an SDx Workspace	Updated launch description and figure.
Creating an Application Project	Updated whole section.
Importing Sources	Updated Imported File figure and C-Callable IP Libraries section.
Selecting Functions for Hardware Acceleration	Added Add Hardware Functions and Hardware Function Panel figures.
Selecting Clock Frequencies	Updated figures and code examples.
Targeting Hardware	Updated figures and code examples.
Targeting System Emulation	Updated figures.

Section	Revision Summary
Creating C-Callable IP Libraries	Updated description.
Using C-Callable IP Libraries	Updated description.
Appendix C: Configuring SDSoC Settings through the GUI	Updated appendix.
07/02/2018 Version 2018.2	
Design Flow Overview	Updated Design Flow Overview diagram.
Throughout the document:	Minor text updates. Updated Figures.
Chapter 1: SDSoC Introduction and Overview	Updated content throughout chapter.
Chapter 2: The SDSoC Environment	Updated content throughout chapter. Updated the PS/PL Block Diagram. Updated Flow Diagram.
Chapter 3: The SDSoC Environment	Updates in all sections. Design Flow Overview diagram was updated to include Emulation.
Chapter 4: C-Callable IP Libraries	Updates throughout chapter.
06/06/2018 Version 2018.2	
Throughout the document:	Updated Figures.
SDSoC Introduction and Overview	Added Chapter.
The SDSoC Environment	Updated Chapter. <ul style="list-style-type: none"> Added introductory information. Modified the PS/PL C-Callable block diagram. Added Pipelined Data Transfer and Compute figure. Added information and links to topics in the Chapter.
	Design Flow Overview
	Understanding the SDx GUI
	Using an SDx Workspace
Creating an SDSoC Application	Changed Chapter title. Made extensive changes to all sections.
	Using a Workspace : Consolidated information about environment variables and command shells.
	Creating an Application Project
	Working with Code : Updated all topics in this section.
Building the SDSoC Project	<ul style="list-style-type: none"> Creating a Hardware Project: Separated topic. Running Emulation: Separated figures for QEMU interface and Waveform window. Guidelines for Invoking SDSCC/SDS++: Moved topic from Command Line Options Chapter; that chapter was deleted.
Targeting System Emulation	Updated topic.
Chapter 4: C-Callable IP Libraries	Updated C-Callable Libraries Chapter.
Debugging Techniques	Added Chapter.

Section	Revision Summary
Chapter 6: Profiling and Optimization	Updated Chapter.
Appendix A: Getting Started with Examples	Modified examples.
Managing Platforms and Repositories	Updated Appendix.
Compiling and Running Applications	Added Appendix.
04/04/2018 Version 2018.1	
The SDSoC Environment	Updated information to more correctly reflect the SDSoC Environment.
Hardware/Software System Runtime Environment	Updated code examples.
Design Flow Overview	Updated content to more correctly reflect the behavior of SDSoC.
Creating and Using C-Callable IP Libraries.	Minor edits to C-Callable IP contents.

Table of Contents

Revision History	2
Chapter 1: SDSoC Introduction and Overview	7
Software Acceleration with SDSoC.....	8
Execution Model of an SDSoC Application.....	9
SDSoC Build Process.....	11
SDSoC Development Methodologies.....	13
Best Practices for Acceleration with SDSoC.....	15
Chapter 2: SDSoC Environment	17
Getting Started.....	20
Elements of SDSoC.....	20
Design Flow Overview.....	22
Understanding the SDx GUI.....	24
Chapter 3: Creating an SDSoC Application	27
Using an SDx Workspace.....	27
Creating an Application Project.....	29
Working with Code.....	34
Building the SDSoC Project.....	43
Building an SDSoC Library.....	52
Chapter 4: C-Callable IP Libraries	54
Creating C-Callable IP Libraries.....	55
Using C-Callable IP Libraries.....	65
Chapter 5: Debugging Techniques	68
Chapter 6: Profiling and Optimization	69
Appendix A: Getting Started with Examples	74
Installing Examples.....	74
C++ Design Libraries.....	76

Appendix B: Managing Platforms and Repositories.....	78
Appendix C: Configuring SDSoC Settings through the GUI.....	80
SDSoC Project Settings.....	80
SDSoC Build Configuration Settings.....	81
SDS++/SDSCC Compiler Options.....	83
SDS++ Linker Settings.....	86
Appendix D: Compiling and Running Applications.....	89
Compiling and Running Applications on a MicroBlaze Processor.....	89
Compiling and Running Applications on an Arm Processor.....	90
Appendix E: Additional Resources and Legal Notices.....	92
Xilinx Resources.....	92
Documentation Navigator and Design Hubs.....	92
References.....	92
Training Resources.....	93
Please Read: Important Legal Notices.....	94

SDSoC Introduction and Overview

The SDSoC™ environment provides a framework for developing and delivering hardware accelerated embedded processor applications using standard programming languages. It includes a familiar embedded processor development flow with an Eclipse-based integrated development environment (IDE), compilers for the embedded processor application and for hardware functions implemented on the programmable logic resources of the Xilinx® device. The `sdscc/sds++` (referred to as `sds++`) system compiler analyzes a program to determine the dataflow between software and hardware functions, generating an application-specific SoC supporting bare metal, Linux, and FreeRTOS as the target operating system. The `sds++` system compiler generates hardware IP and software control code that automatically implements data transfers and synchronizes hardware accelerators and application software, therefore pipelining communication and computation.

Using SoC devices from Xilinx, such as the Zynq®-7000 SoC and the Zynq UltraScale+™ MPSoC, you can implement elements of your application into hardware accelerators, running many times faster than optimized code running on a processor. Xilinx FPGAs and SoC devices offer many advantages over traditional CPU/GPU acceleration, including a custom architecture capable of implementing any function that can run on a processor, resulting in better performance at lower power dissipation. To realize the advantages of software acceleration on a Xilinx device, you should look to accelerate large compute intensive portions of your application in hardware. Implementing these functions in custom hardware allows you to achieve an ideal balance between performance and power. The SDSoC environment provides tools and reports to profile the performance of your embedded processor application and determines where the opportunities for acceleration are. The tools also provide automated runtime instrumentation of cache, memory, and bus utilization to track real-time performance on the hardware.

Developers of hardware accelerated applications can make use of a familiar software-centric programming workflow to take advantage of FPGA acceleration with little or no prior FPGA or hardware design experience. As a software programmer, calling a hardware function is the same as calling a software function, letting the compiler implement the hardware/software partitioning. However, developers can also create predefined hardware accelerators for use in an embedded processor application, using a hardware-centric approach working through the Vivado® HLS compiler, or creating and packaging optimized RTL accelerators for distribution as a library of C-Callable IP.

The SDSoC environment provides predefined platforms for standard ZCU102, ZCU104, ZCU106, ZC702, and ZC706, which are Zynq-based development boards. Third-party platforms are also available including: the Zedboard, Microzed, Zybo, Avnet Embedded Vision Kit, Video and Imaging Kit, SDR kit, and more. You can also create a custom platform to meet your specific market requirements. An SDSoC platform consists of a hardware portion defining the embedded processor, the hardware function, and any peripherals supported by the platform; and a software portion defining the operating system boot images, drivers, and the application code. You can start your project using one of the standard SDSoC platforms to evaluate a design concept, to be later implemented on a custom platform for production.

Software Acceleration with SDSoC

When compared with processor architectures, the structures that comprise the programmable logic (PL) in a Xilinx device enable a high degree of parallelism in application execution. The custom processing architecture generated by the `sds++/sdsc` (referred to as `sds++`) system compiler for a hardware function in an accelerator presents a different execution paradigm from CPU execution, and provides an opportunity for significant performance gains. While you can re-target an existing embedded processor application for acceleration in PL, writing your application to use the source code libraries of existing hardware functions, such as the [Xilinx xfOpenCV library](#), or modifying your code to better use the PL device architecture, yields significant performance gains and power reduction.

CPUs have fixed resources and offer limited opportunities for parallelization of tasks or operations. A processor, regardless of its type, executes a program as a sequence of instructions generated by processor compiler tools, which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the target processor. Even a simple operation, such as the multiplication of two values, results in multiple assembly instructions that must be executed across multiple clock cycles.

An FPGA is an inherently parallel processing device capable of implementing any function that can run on a processor. Xilinx devices have an abundance of resources that can be programmed and configured to implement any custom architecture and achieve virtually any level of parallelism. Unlike a processor, where all computations share the same ALU, the FPGA programming logic acts as a blank canvas to define and implement your acceleration functions. The FPGA compiler creates a unique circuit optimized for each application or algorithm; for example, only implementing multiply and accumulate hardware for a neural net—not a whole ALU.

The `sds++` system compiler invoked with the `-c` option compiles a file into a hardware IP by invoking the Vivado High-Level Synthesis (HLS) tool on the desired function definition. Before calling the HLS tool, the `sds++` compiler translates `#pragma SDS` into pragmas understood by the HLS tool. The HLS tool performs hardware-oriented transformations and optimizations, including scheduling, pipelining, and dataflow operations to increase concurrency.

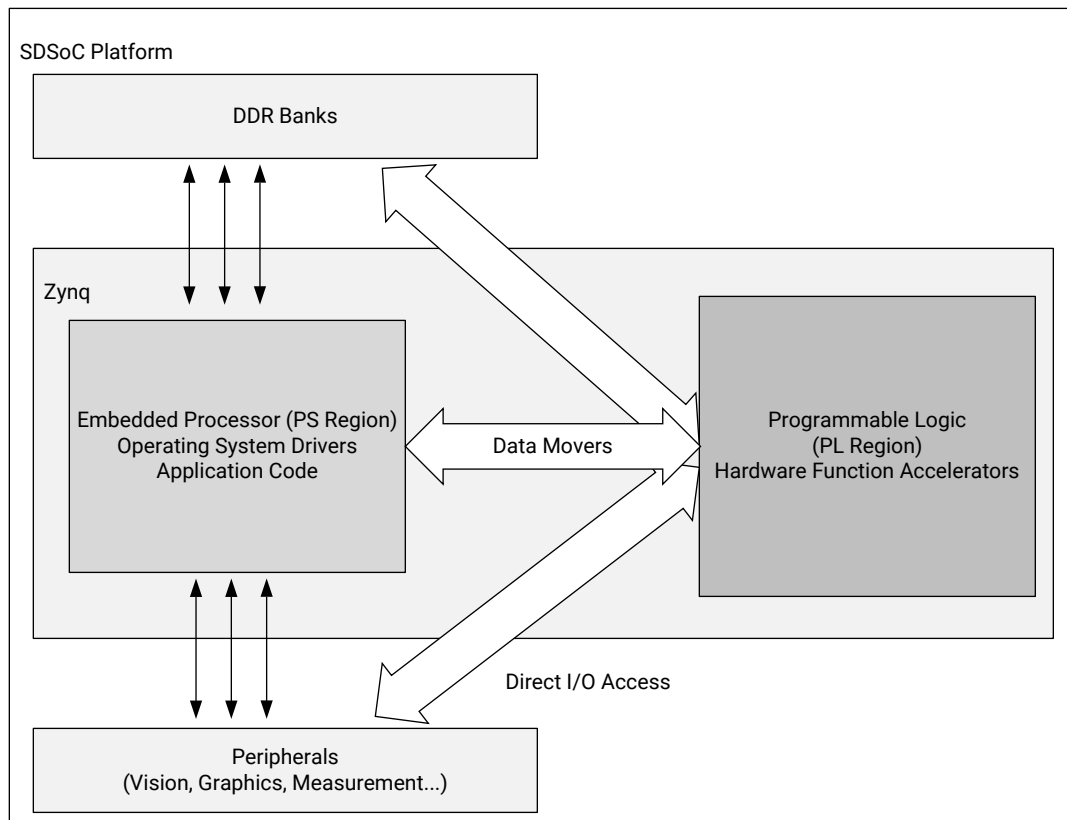
The `sds++` linker analyzes program dataflow involving calls into and between hardware functions, mapping into a system hardware data motion network, and software control code (called stubs) to orchestrate accelerators and data transfers through data movers. As described in the following section, the `sds++` linker performs data transfer scheduling to identify operations that can be shared, and to insert wait barrier API calls into stubs to ensure program semantics are preserved.

Execution Model of an SDSoC Application

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

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

Figure 1: Architecture of an SDSoC System



X21358-082418

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

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

At a high-level, the SDSoC program execution model includes the following steps:

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

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



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

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



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

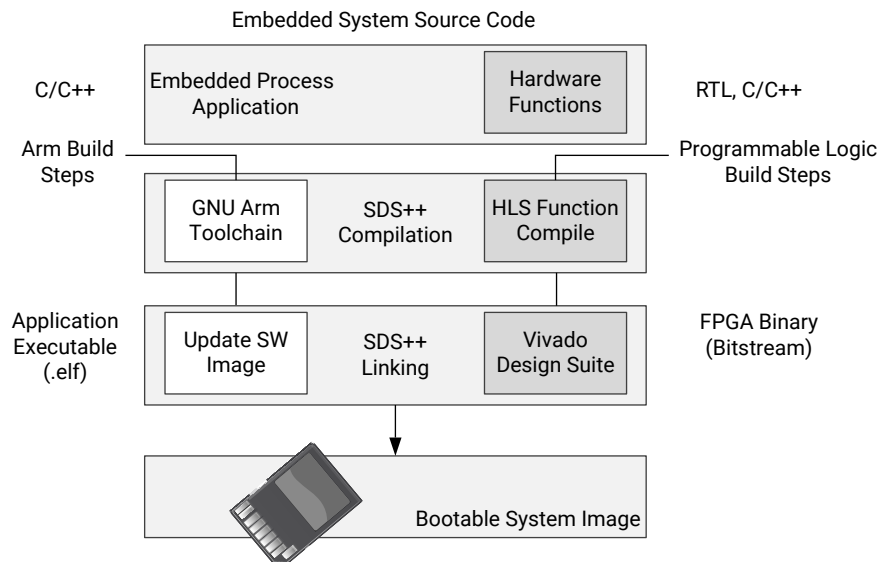
SDSoC Build Process

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

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

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

Figure 2: SDSoC Build Process



X21126-041119

The compilation process includes the following tasks:

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

After compilation, the linking process includes the following tasks:

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

SDSoC Development Methodologies

The SDSoC environment supports two primary use cases:

- **Software-centric design:** The development of an accelerated application written by software programmers using standard programming languages, accelerating compute intensive functions into programmable logic, or identifying application bottlenecks for acceleration by profiling the application.
- **Hardware-centric design:** The development of predefined accelerated functions for use in embedded processor applications like a library of intrinsic functions. This design methodology can be driven from a top-down approach of writing the hardware function in a standard programming language like C or C++, and then synthesized into RTL for implementation into programmable logic; or by using standard RTL design techniques to create and optimize the accelerated function.

The two use-cases are often combined, letting software and hardware developer teams define hardware accelerators and developing embedded processor applications to use them. This combined methodology involves different components of the application, developed by different people, and potentially from different companies. You can use predefined hardware functions from libraries available for use in your accelerated application, such as the [Xilinx xfOpenCV library](#), or develop all the accelerators within your own team.

Software-Centric Design

The software-centric approach to accelerated application development, or accelerator development, begins with the use of the C or C++ programming language. The code is written as a standard software program, with some attention to the specific architecture of the code. The software-centric development flow typically uses the following steps:

Table 1: Software-Centric Design Flow

Task	Steps
Profile the embedded processor application.	<ul style="list-style-type: none"> • Baseline the performance, identify bottlenecks, and functions to accelerate. • Assess acceleration potential, plan budgets, and requirements.
Code the desired accelerators.	<ul style="list-style-type: none"> • Convert the desired functions to define the hardware function code without optimization.
Verify functionality, iterate as needed.	<ul style="list-style-type: none"> • Run system emulation to generate application and accelerator profiling data including: <ul style="list-style-type: none"> ◦ Estimated FPGA resource usage. ◦ Overall application performance. ◦ Visual timeline showing application calls and accelerator start/stop times. • Address design recommendations provided by tool guidance.

Table 1: **Software-Centric Design Flow** (cont'd)

Task	Steps
Optimize for performance, iterate as needed.	<ul style="list-style-type: none"> • Analyze the profile summary and application timeline. • Optimize data movement throughout system: <ul style="list-style-type: none"> ◦ Application to DDR, DDR to accelerator, and hardware function interface to local buffers (bursting) ◦ Maximize DDR bandwidth usage with efficient transfer sizes ◦ Overlapping of transfers ◦ Prefetching • Optimize the accelerator code for performance: <ul style="list-style-type: none"> ◦ Task-level parallelism (dataflow) ◦ Instruction-level parallelism (pipelining and loop unrolling) ◦ Match datapath size to interface bandwidth (arbitrary bit-width)

Hardware-Centric Design

A hardware-centric flow first focuses on developing and optimizing the accelerators and typically leverages advanced FPGA design techniques to create a library of C-Callable IP. This begins with the definition of the hardware function in C or C++ for use in Vivado HLS, or the use of an RTL language, or an existing IP design or block design in the Vivado Design Suite. The hardware function is defined in RTL code, synthesized, and implemented into the programmable logic of the target device. A software function signature is needed to use the C-Callable IP in the accelerator application, or a compiled library of functions is created for use across multiple applications. The hardware-centric development flow typically uses the following steps:

 Table 2: **Hardware-Centric Design Methodology**

Task	Steps
Study the SDSoC platform specification, and the Zynq-7000 SoC device specification and programming model.	<ul style="list-style-type: none"> • Hardware platform, software platform, data movers, AXI interface, DDR.
Identify cycle budgets and performance requirements.	
Define the accelerator architecture and interfaces.	
Develop the accelerator.	<ul style="list-style-type: none"> • Use Vivado HLS for C or C++ hardware functions. • Use traditional RTL design techniques in the Vivado Design Suite.
Verify functionality and performance, iterate as needed.	<ul style="list-style-type: none"> • Run hardware/software co-simulation in Vivado HLS. • Run logic simulation in the Vivado simulator.
Optimize the quality of results to reduce resource utilization and increase frequency, iterate as needed.	<ul style="list-style-type: none"> • For HLS, ensure the design rules check (DRC) is clean. • Run the Vivado implementation flow, using the techniques specified in the <i>UltraFast Design Methodology Guide for the Vivado Design Suite</i> (UG949). • Use best practices for out-of-context synthesis and estimation.

Table 2: **Hardware-Centric Design Methodology** (cont'd)

Task	Steps
Import the C-Callable IP into the SDSoC environment.	<ul style="list-style-type: none"> For the HLS flow, import the C or C++ code into your SDSoC project. For RTL flow, use the C-Callable IP wizard. See C-Callable Libraries for more information.
Develop sample application code to test the hardware function.	<ul style="list-style-type: none"> Test sample applications with a dummy function having the same interfaces as the C-Callable IP. See C-Callable Libraries for more information.
Verify the hardware function works properly with application, iterate as needed.	<ul style="list-style-type: none"> Use system emulation for debug. Use the Hardware debug methodology for complex internal debug problems.
Optimize host code for performance, iterate as needed:	<ul style="list-style-type: none"> Use the Profile Summary report, the Activity Timeline, and event timers in the host application to measure performance. Ensure the DRC is clean. Work to achieve an Activity Timeline that matches the desired performance. Techniques: Overlapping transactions, out-of-order (OOO) synthesis queues, and sub-devices.
Finalize the Software Acceleration Layer deliverable (API, share lib, plug-in...).	

Best Practices for Acceleration with SDSoC

The following shows best practices when developing your application code and hardware function in the SDSoC environment:

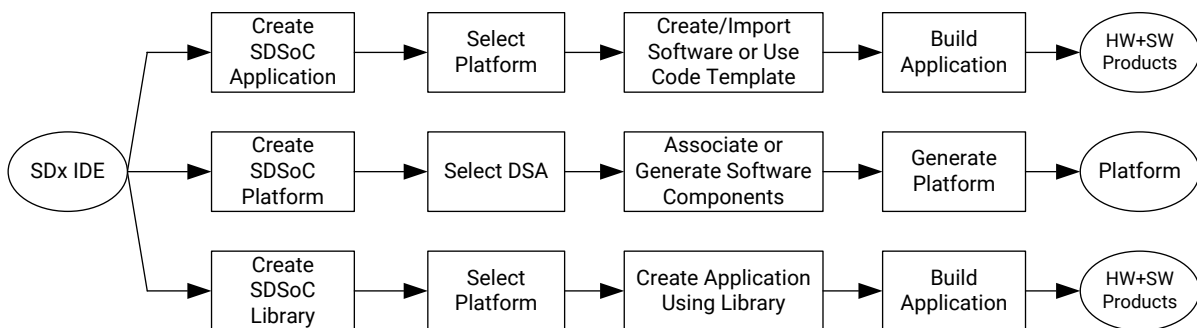
- General guidelines:
 - Reduce resource utilization and improve parallelism by streaming data instead of copying data into the PL region. For example, in an image processing application, stream rows of pixels that make up a frame instead of copying the image frame in one long data transfer.
 - Reuse the data local to the PL region rather than transferring it back and forth to limit DMA.
 - Look to accelerate functions that have:
 - A high compute time to data transfer time ratio.
 - Predictable communication streams.
 - Self-contained control structure not needing control logic outside the accelerator.
 - Look for opportunities to increase task-level parallelization by launching multiple accelerators concurrently, or multiple instances of an accelerator.

- For a software-centric approach:
 - Use good memory management techniques, such as having known array sizes, and using `sds_alloc()/sds_free()` to allocate/de-allocate physically contiguous memory, thereby reducing the device footprint and increasing baseline performance.
 - Use system emulation to validate your code frequently to ensure it is functionally correct.
 - Write/migrate hardware functions to separate C/C++ files as to not re-compile the entire design for incremental changes.
- For a hardware-centric approach using C-Callable IP:
 - Keep track of the AXI4 Interface offsets for an IP, or accelerator, and what function definition parameters require what data type. The interfaces need to be byte aligned.
 - Maintain the original Vivado IP project so that modifications to it can be quickly implemented.
 - Keep the static library (.a) file and corresponding header file together.

SDSoC Environment

The software-defined system-on-chip (SDSoC™) environment provides the tools necessary to implement heterogeneous embedded systems for Zynq® UltraScale+™ MPSoC or Zynq-7000 devices. The design tasks and exploration of hardware/software partitioning is accomplished by working in an Eclipse-based integrated development environment referred to as the SDx™ IDE. The SDx IDE is designed to be familiar to users of software development IDEs. Actions carried out with the SDx IDE include creating an Application project, creating a Platform project, and creating a Library project. The figure below shows an overview of these design flows. Application and Library projects are discussed in this user guide, whereas Platform projects are covered in more detail in the *SDSoC Environment Platform Development Guide* ([UG1146](#)).

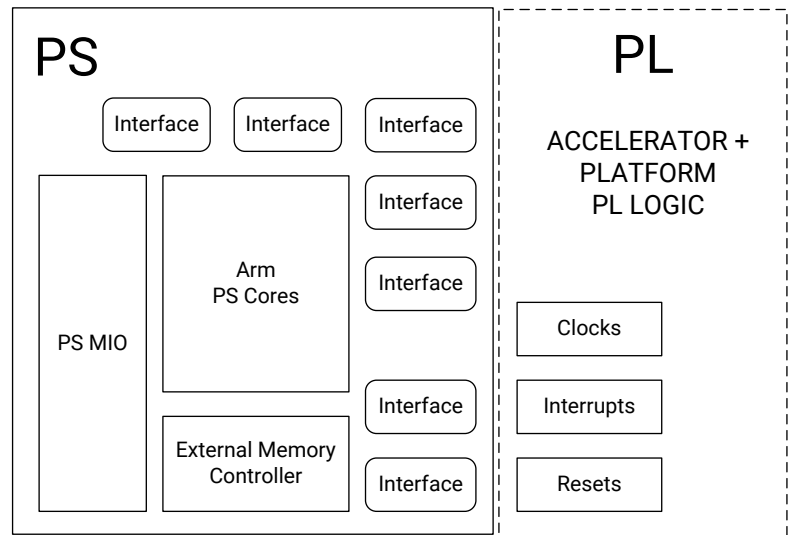
Figure 3: SDx Design Flows



X21844-110618

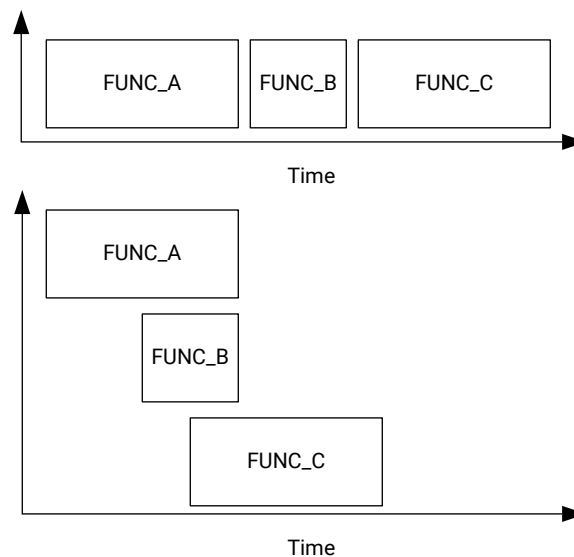
The concept of a platform is integral to the SDSoC environment as it defines the hardware, software, and meta-data components on which SDSoC applications are built. Multiple base platforms are available within the SDx IDE and can be used to create SDSoC applications without first having to create a custom platform. The SDx IDE utilizes the `sds++` system compiler to convert C/C++ code into high-performance hardware accelerators that attach to platform interfaces as determined by the platform designer and by application code pragmas. Declarations within the platform meta-data identify interface ports, clocks, interrupts, and reset blocks for use by the system compiler when it attaches hardware accelerators to the base platform.

Figure 4: Platform Block Diagram



The system compiler analyzes a program to determine the dataflow between software and hardware functions and generates an application-specific system-on-chip. The `sds++` system compiler generates hardware IP and software control code that implements data transfers and synchronizes the hardware accelerators with application software. Performance is achieved by pipelining communication and computation, thereby producing hardware functions that can run with maximum parallelism as illustrated in the following figure.

Figure 5: Pipelined Data Transfer and Compute



X21846-110618

The `sds++` system compiler invokes the Vivado® High-Level Synthesis (HLS) tool to transform software functions into a bitstream that defines and configures the programmable logic (PL) portion of the SoC. In addition, stub functions are generated so application software compiled and linked using the standard GNU toolchain transparently uses the implemented hardware functions. All necessary drivers and libraries are automatically included in this system compilation process.

The final output of system compilation is the generated `sd_card` directory, which at minimum is populated with a Zynq bootable `BOOT.BIN` file, the executable and linkable format (ELF) file application code, and a `README.txt` boot instructions file. The `BOOT.BIN` file contains any necessary bootloaders, bitstreams, and application code to boot the generated system on a target board. For systems that run Linux on the target board, the `sd_card` directory also contains the Linux image file used during the boot process.

The SDSoC system compilers generate complete applications and let users iterate over design and architectural features by re-factoring at the program level, reducing the time necessary to achieve working applications on target platforms. To achieve high-performance, each hardware function runs independently; the system compilers generate hardware and software components that ensure synchronization between the hardware functions and the application software 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.

For the SDSoC environment, this reflects the resources and performance available within the Zynq-7000 SoC or the Zynq UltraScale+ MPSoC device family. When creating applications that require specific real-time behavior, it is important to be aware of the execution environment.

The Zynq-7000 family includes a processor system (PS) with dedicated Arm® processing cores, on-chip memories, embedded peripherals, interconnect blocks, a DDR memory controller, and PL fabric used by the SDSoC-generated accelerators.

Ideal processor, memory, and AXI interface performance are shown in the following table using switching characteristics from the Zynq-7000 SoC and Zynq UltraScale+ MPSoC data sheets.

Table 3: Processor, Memory, and AXI Interface Performance

Clock or Interface	Zynq UltraScale+ MPSoC	Zynq-7000 SoC
Max APU clock frequency	Arm Cortex™-A53 64-bit Quad-Core: 1500 MHz	Arm Cortex-A9 32-bit Dual-Core: 1000 MHz
Max RPU clock frequency	ArmCortex-R5 32-bit Dual-Core: 600 MHz	N/A
DDR type and bit width	DDR4: x32, x64	DDR3: x16, x32
DDR Max performance	2400 Mb/s	1333 Mb/s
DDR Max Ideal Throughput	153.6 Gb/s	42.6 Gb/s
AXI Interface width	128-bit, 64-bit, 32-bit	64-bit, 32-bit
AXI Interface Max Frequency	333 MHz	250 MHz

Table 3: Processor, Memory, and AXI Interface Performance (cont'd)

Clock or Interface	Zynq UltraScale+ MPSoC	Zynq-7000 SoC
AXI Interface Max Ideal Throughput	42.6 Gb/s	16 Gb/s
Number of AXI Interface Ports	12	6
Total AXI Throughput	511.2 Gb/s	96.0 Gb/s

Getting Started

Download and install the SDSoC tool suite according to the directions provided in the *SDSoC Environments Release Notes, Installation, and Licensing Guide* ([UG1294](#)).

After installing the SDSoC tools, you can find 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 in the *SDSoC Environment Getting Started Tutorial* ([UG1028](#)). Working through the tutorial and its labs is the best way to get an overview of the SDSoC environment, and should be considered a prerequisite to application development.

Note: The SDSoC tool suite includes the entire tool 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 tools. Ensure that your tools are derived from an SDSoC installation (which includes the Vivado Design Suite and SDK tools).



RECOMMENDED: 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.

Elements of SDSoC

The SDSoC environment includes the `sds++` system compiler to generate 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. Elements of the SDx tools include:

- Eclipse-based IDE
- The `sds++` system compiler
- High-Level Synthesis (HLS)
- Vivado Design Suite

- IP integrator and IP libraries
- Vivado-generated SDx Platforms
- SDx-generated hardware accelerators and associated control software
- SDx-generated data movers and associated control software
- The Target Communication Framework (TCF)
- GNU software development tools

The SDSoC environment includes the GNU toolchains and standard libraries (for example, `glibc`), a performance analysis perspective within the Eclipse C/C++ Development Tooling (CDT)-based GUI, and command-line tools.

The SDSoC system compiler employs underlying tools from the Vivado Design Suite HLS Editions including Vivado HLS, IP integrator, and IP libraries for data movement and interconnect, and the RTL synthesis, implementation, and bitstream generation tools.

The principle of design reuse underlies workflows you employ with the SDSoC environment, using 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 built-in 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 runtime environment to build a platform for use in the SDSoC environment.

An SDSoC platform defines a base hardware and software architecture and application context, which includes the following:

- Processing system
- External memory interfaces
- Custom input/output
- Software runtime including: Operating system (for example, Linux, FreeRTOS, or Standalone), boot loaders, drivers for platform peripherals, and the root file system

Every project you create within the SDSoC environment targets a specific platform. Using the SDx IDE to build on the base platform foundation with application-specific hardware accelerators and data motion networks and connecting accelerators to the platform, you can create customized, application-specific SoC designs for different base platforms, and use base platforms for many different applications.

You are provided the option to use either a predefined platform from the SDx installation or a custom platform. Custom platforms are generated from a device support archive (DSA) hardware specification exported from the Vivado tools, or derived from a predefined platform.

See the *SDSoC Environments Release Notes, Installation, and Licensing Guide* ([UG1294](#)) for the most up-to-date list of supported devices and required software.

Design Flow Overview

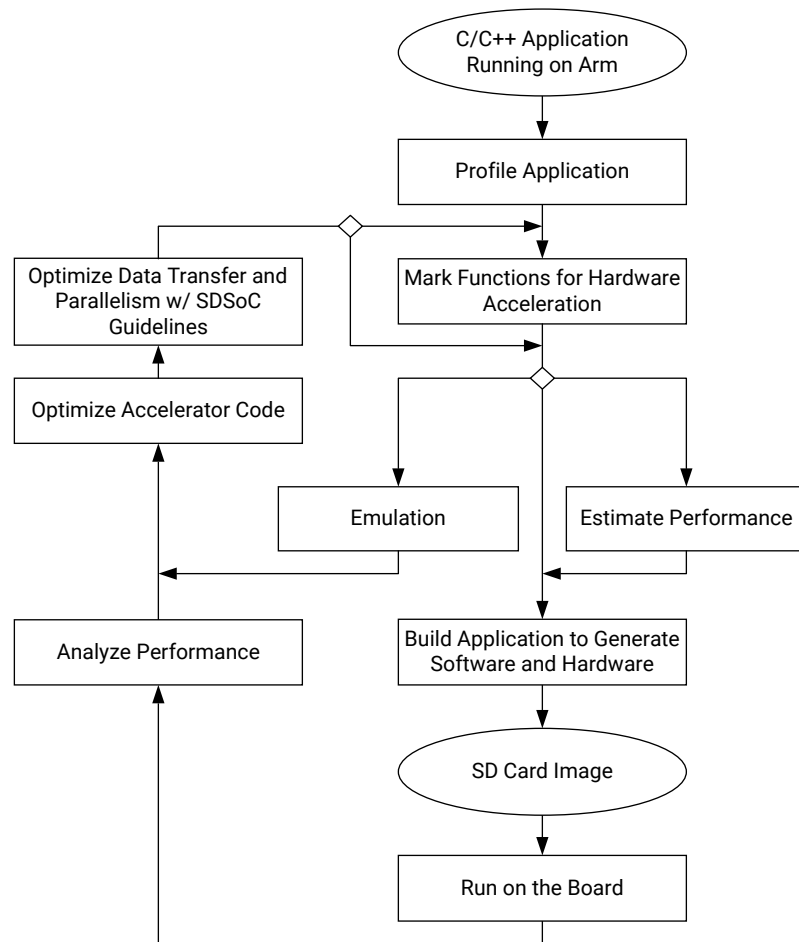
The SDSoC environment is a tool suite for building efficient SoC applications, starting from a platform that provides the base hardware and target software architecture. A boot image and the executable application code are generated by the SDSoC tools.

The following figure shows a representative top-level design flow that shows 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 compiled for 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.

Note: Emulation only works on the base platforms. For more on debug or emulation, see *SDSoC Environment Debugging Guide* ([UG1282](#)).

Figure 6: User Design Flow



X14740-041119

The steps are:

1. Select a development platform, compile the application, and ensure it runs properly on the platform.
2. 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. See [Selecting Functions for Hardware Acceleration](#).
3. Invoke the SDSoC system compiler to generate a complete SoC and SD card image for your application. See [Working with Code](#).

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 environment. *SDSoC Environment Profiling and Optimization Guide (UG1235)* for profiling and optimization best practices.

The `sds++` system compilers orchestrate the system generation process either through the IDE or in the terminal shell using command lines and makefiles. You select functions to run in hardware, specify accelerator and system clocks, and set properties on data 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 a conventional 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. Also, this can be functionally verified and analyzed through system emulation. The system emulation feature uses a quick emulation (QEMU) model executing the software and RTL model of the hardware functions to enable fast and accurate analysis of the system.

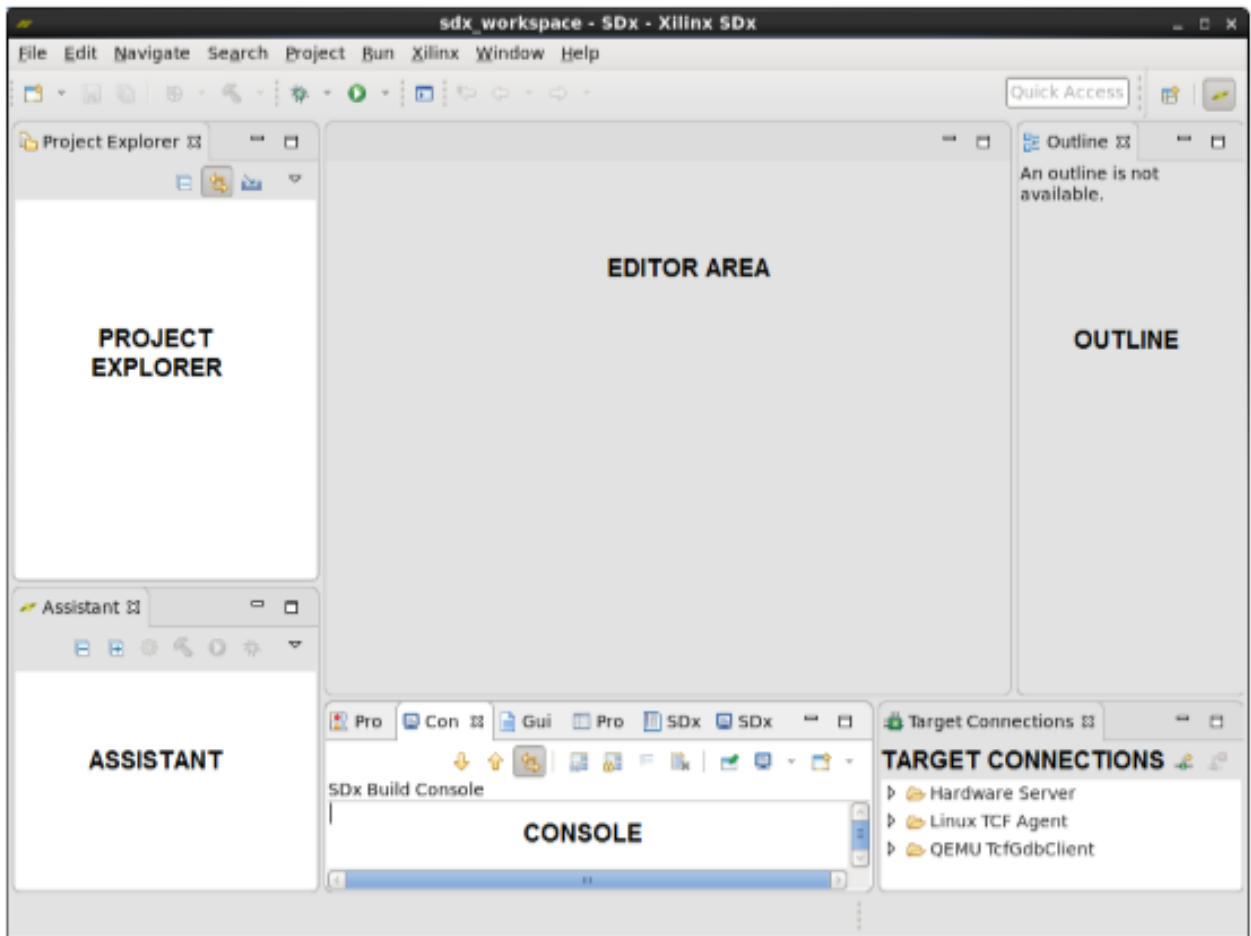
The overall design process involves iterating the steps until the generated system achieves your performance and cost objectives.

To run through the introductory tutorial and become familiar with creating a project, selecting hardware functions, and compiling and running a generated application on the target platform, see *SDSoC Environment Getting Started Tutorial* ([UG1028](#)).

Understanding the SDx GUI

When you open a project in the SDx IDE, the workspace is arranged in a series of different views and editors, also known as a *perspective* in the IDE. The tool opens with the SDx (default) perspective shown in the following figure.

Figure 7: SDx – Default Perspective




Some key views/editors in the default perspective are:

- **Project Explorer:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool.
- **Assistant:** Provides a central location to view/edit settings, build and run your SDSoC application, launch profiling and debug sessions, and open reports.
- **Editor Area:** Displays project settings, build configurations, and provides access to many commands for working with the project.
- **Console Area:** Presents multiple views including the command console, design guidance, project properties, logs and terminal views.
- **Outline:** Displays an outline of the current source file opened in the Editor Area.
- **Target Connections:** Provides status for different targets connected to the SDx tool, such as the Vivado hardware server, Target Communication Framework (TCF), and quick emulator (QEMU) networking.

To close a view, click the **Close** button (x) on the tab of the view. To open a view, select **Window** → **Show View** and select a view. You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE.

To save the arrangement of views as a perspective, select **Window** → **Perspective** → **Save Perspective As**. This defines different perspectives for initial project editing, report analysis, and debug for example. Any changes made without saving as a perspective are stored with the workspace. To restore the default arrangement of views, select **Window** → **Perspective** → **Reset Perspective**.

To open different perspectives, select **Window** → **Perspective** → **Open Perspective**.

To restore the SDx (default) perspective, click the SDx button  on the right side of the main toolbar.

Creating an SDSoC Application

An SDSoC™ application can be created through the SDx™ IDE or by using a command line interface. In this chapter, the SDx IDE shows how to create a workspace, generate an application project for a selected platform, and use the available application templates. Working with code, adjusting hardware accelerator settings and targeting either emulation or a board is also covered. Excerpts of how to perform the IDE operations using command line equivalents is interspersed with the GUI illustrations.

Using an SDx Workspace



IMPORTANT! *Linux host is strongly recommended for SDSoC™ platform development, and required for creating a platform supporting a target Linux OS.*

1. Launch the SDx™ IDE directly from the desktop icon or from the command line by one of the following methods:
 - Using either of the following commands from the command prompt:

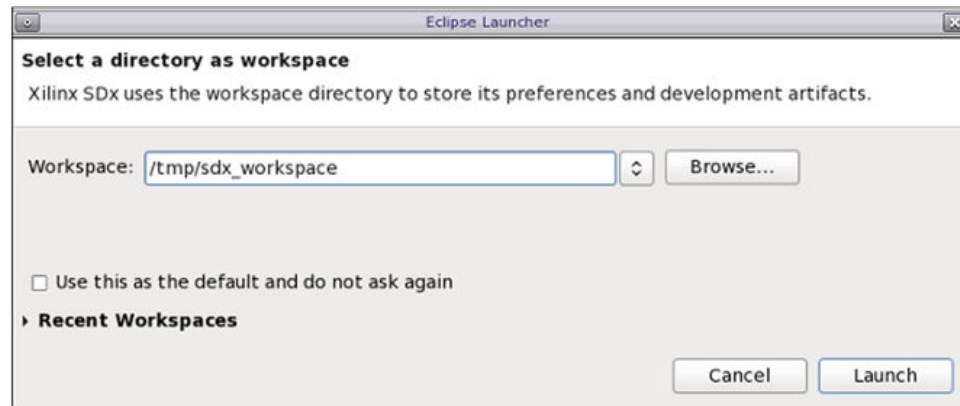
```
sdx
```



```
or
```

```
sdx -workspace <workspace_name>
```
 - Double-clicking the **SDx** icon to start the program.
 - Launching from the **Start** menu in the Windows operating system.
2. The SDx IDE opens and prompts you to select a workspace, as shown in the following figure.

Figure 8: Specify the SDx Workspace



IMPORTANT! When opening a new shell to enter an SDx command, ensure that you first source the `settings64` and `setup` scripts to set up the tool environment. On Windows, run the `settings64.bat` file from the command shell. See the *SDSoC Environments Release Notes, Installation, and Licensing Guide (UG1294)* for more information.

The SDx workspace is the 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. The following instructions show you how to define a workspace for an SDSoC project.

1. Click 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** check box to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.
3. Click **Launch**.



TIP: You can change the current workspace from within the SDx IDE by selecting **File → Switch Workspace**.

You have now created an SDx workspace and can populate the workspace with projects. Platform and application projects are created to describe the SDx tool flow for creating an SDSoC platform.

The SDx IDE can populate the workspace with three types of user selected project types:

- Application Project
- Platform Project
- Library Project

The following sections describe how to use the Platform and Application project types while constructing the example SDSoC platforms.

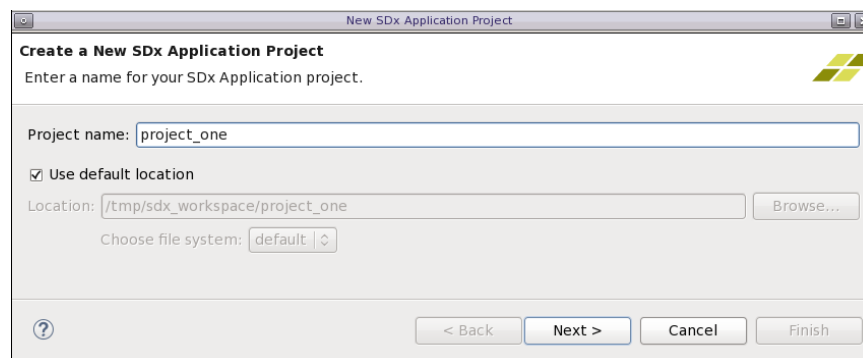
Creating an Application Project



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.

1. After launching the SDx IDE you can create a new Project. Select **File** → **New** → **SDx Application Project**, or if this is the first time the SDx IDE has been launched, you can select **Create Application Project** on the Welcome screen.
2. The Create a New SDx Application Project wizard opens.
3. In the Create a New SDx Application Project page, you can specify the project name as shown. Specify the name of the project in the **Project name** field.

Figure 9: Create a New SDx Application Project



4. The **Use default location** is selected by default to locate your project in a folder in the SDx workspace. You can uncheck this check box to specify that the project is created in a **Location** of your choice.
5. If you specify the location, you can use **Choose file system** to select the **default** file system, JSch, or enable the Eclipse Remote File System Explorer (**RSE**).



IMPORTANT! The project location cannot be a parent folder of an SDx workspace.

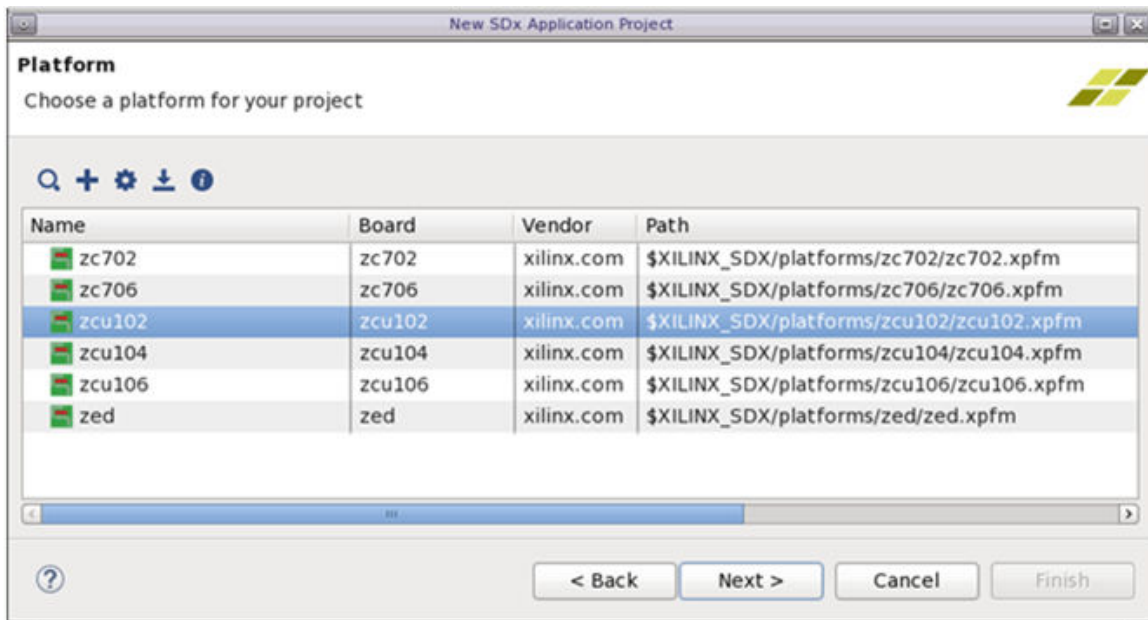
6. Click **Next**.

The Platform dialog box, similar to the one shown in the following figure, displays the available installed platforms. For installing additional platforms, see the "Installing Platform-Specific Packages" section in *SDAccel Environment Release Notes, Installation, and Licensing Guide* ([UG1238](#)).



IMPORTANT! Be sure to select the right platform for your project, as subsequent processes are driven by this choice.

Figure 10: Specify SDSoC Platform



A platform is composed of a shell, which describes the base hardware design, the meta-data used in attaching accelerators to declared interfaces, and the software environment, which can include operating system images (for example, Linux), as well as boot-up and runtime files.

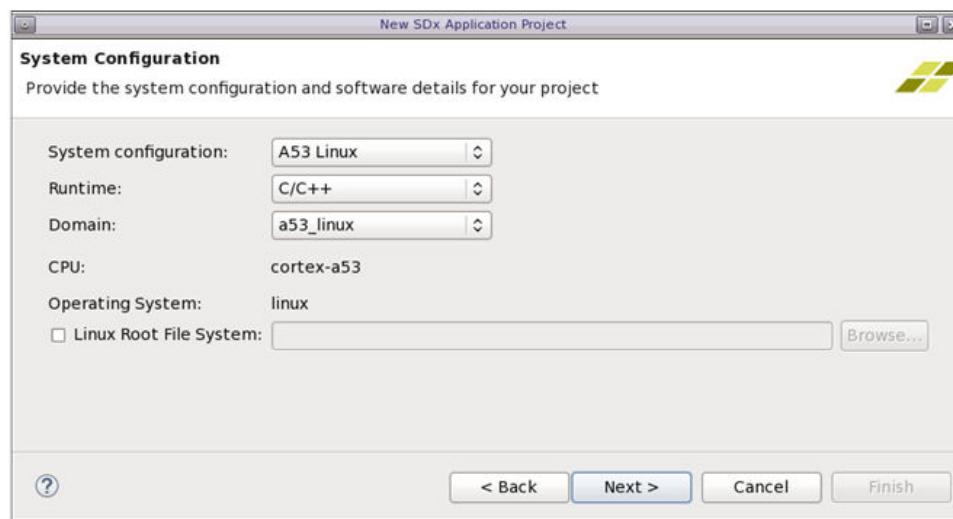
The SDSoC development environment offers base-platforms for specific boards based on the following:

- Zynq®-7000
 - zc702
 - zc706
- Zynq UltraScale+™ MPSoC
 - zcu102
 - zcu104
 - zcu106
 - zed

You can add custom defined or third-party platforms into a repository. See [Appendix B: Managing Platforms and Repositories](#) for more information.

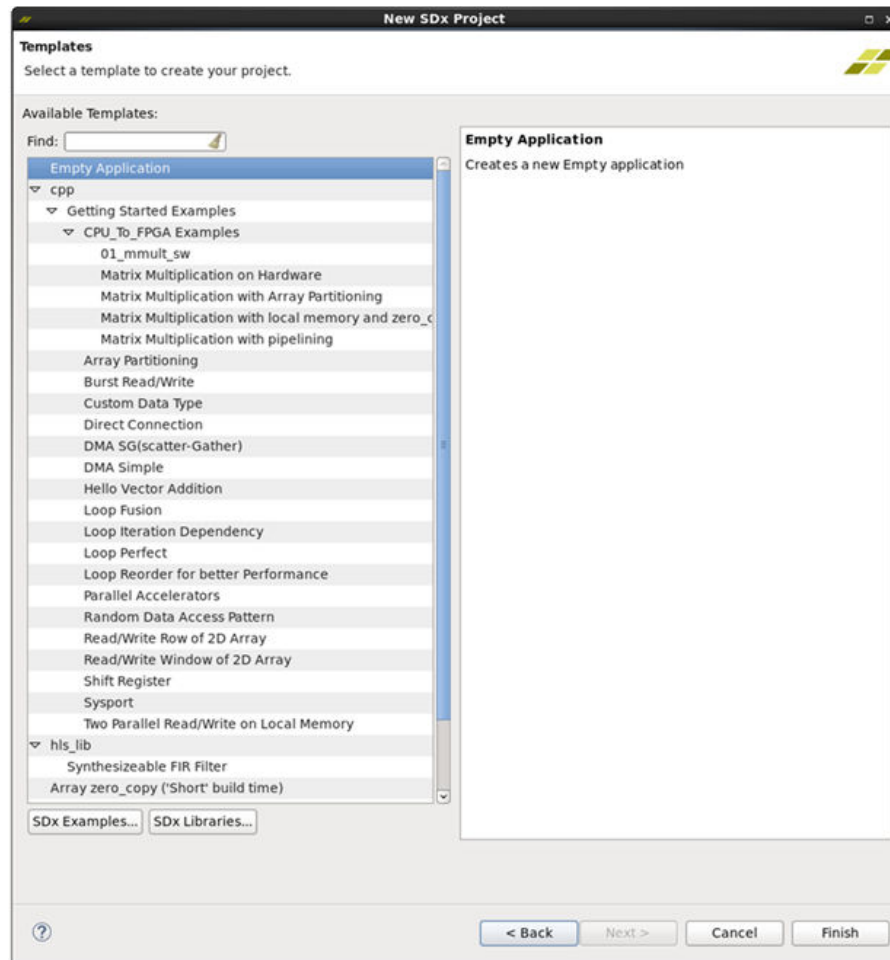
1. Select the target platform for your project from the listed platforms. A series of actions are also available in this Platform dialog through the five icons immediately above the platform list. From left-to-right, clicking an icon invokes a platform text string search, adding a custom platform, managing a platform repository, adding new devices or platforms, and listing additional information about a selected platform.
2. Select one of the predefined, installed platforms and click **Next**.
3. The System configuration page opens, as shown in the following figure. Select **System configuration** and **Runtime** from a list of those defined for the selected platform. The System Configuration defines the software environment that runs on the hardware platform. It specifies the operating system and the available runtime settings for the processors in the hardware platform.

Figure 11: Specify System Configuration



4. When setting the system configuration, you can also check the **Domain** and specify a **Linux Root File System** if a Linux-based configuration is selected. The **Linux Root File System** is a sysroot directory structure that provides the necessities for a system to run and locate headers.
5. After selecting the **System Configuration** and clicking **Next**, the Templates page displays, as shown in the following figure. Specify an application template for your new project. The `samples` directory within the SDx tools installation contains multiple source code example templates.
6. Initially, you have the option in the Template dialog box of an **Empty Application** or one of the provided application templates. You can download additional SDx examples or update the installed examples by clicking the **SDx Examples** button, which retrieves content from the Xilinx [GitHub](#) as discussed in [Appendix A: Getting Started with Examples](#).

Figure 12: Application Templates



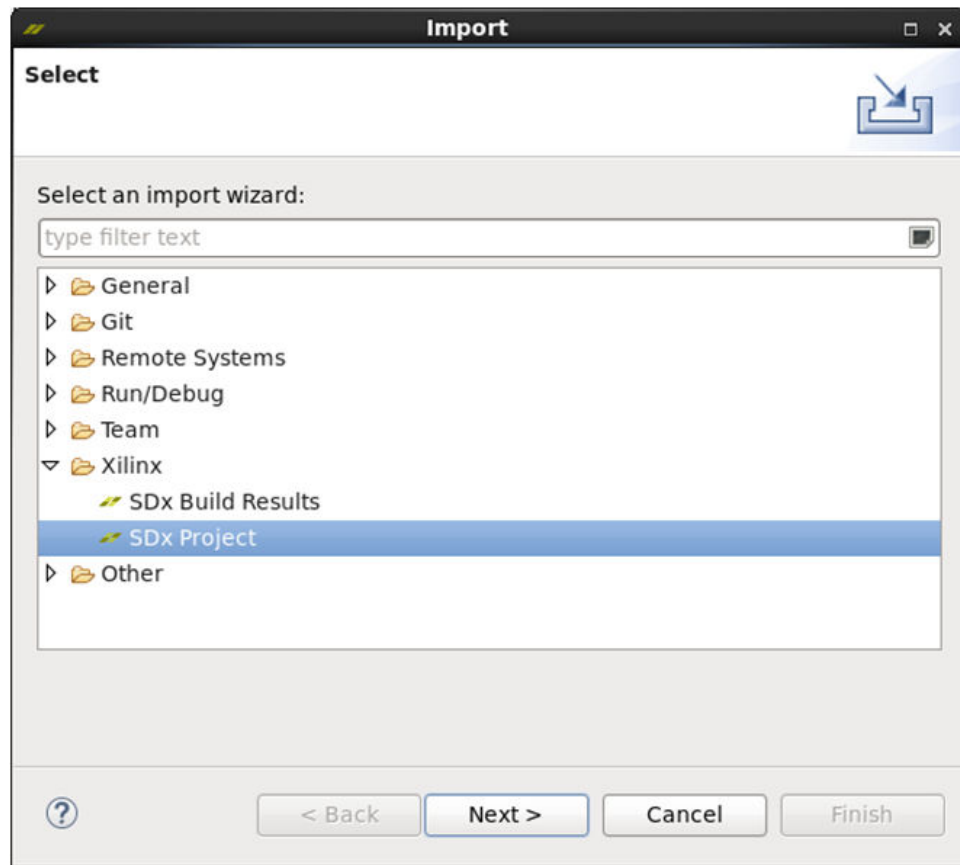
7. You can use the template projects as examples to learn about the SDx tool and acceleration kernels or as a foundation for your new project. Note that you must select a template. You can select **Empty Application** to create a blank project into which you can import files and build your project from scratch.
8. Click **Finish** to close the New SDx Project wizard and open the project.

Importing a Project

A previously exported project can be used as the source for a new project.

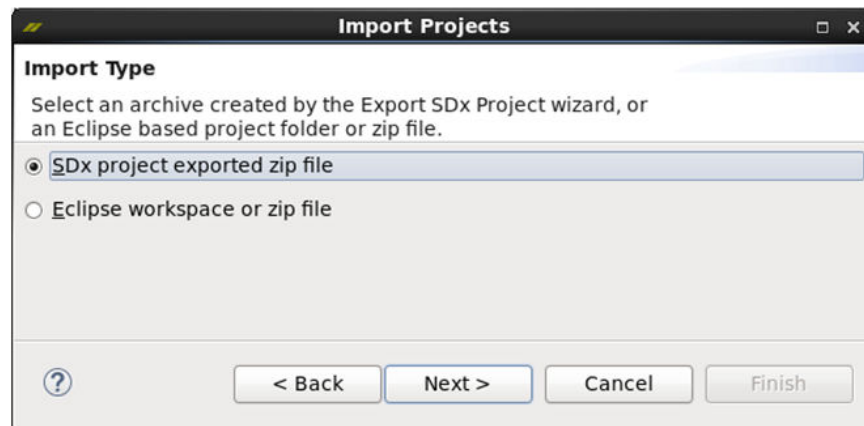
1. Select **File** → **Import** menu and click **Xilinx** → **SDx Project**, or go directly through the Import Project selection on the Welcome screen.

Figure 13: Select an SDx Project



2. Chose the import type as an **SDx project exported zip file** which opens up a dialog for browsing to the exported zip file. Click **Next**.

Figure 14: Import Type



3. Select and click **Open** provides a list of Application projects that can be brought into the workspace.



RECOMMENDED: Xilinx recommends using SDx exported zip files to move projects between workspaces. Also, this is the last SDx release, so use the import sources as opposed to the remote sources.

Working with Code

The SDx environment provides a GUI-based IDE as well as command line control to invoke the `sds++` system compiler with user-specified command options from a shell prompt.

Application code generally consists of C/C++ source files, C/C++ header files, and libraries created for shared or static use. The SDx tools help you identify and convert C/C++ source code functions into hardware accelerators. By analyzing function arguments, argument types, and any applied directives or pragmas, the SDx tools generate data movers and pipelined dataflows to feed data into and out of an accelerator. Typical data sources and data sinks are memories and I/O streams.

The SDSoC-generated accelerators reside in the PL and need to fit in the PL resources. See [Execution Model of an SDSoC Application](#). The interfaces between the PS and PL are user-configurable. For instance, SDx tools automatically generate data movers for crossing the PS-PL boundary with interface direction as set by your C/C++ source code.

Source files to create an SDx application with example SDx pragmas are provided within the IDE by selecting **Xilinx → SDx Examples**. Additional examples are available for download from the [SDSoC Examples GitHub](#).

Importing Sources

Importing C/C++ Sources

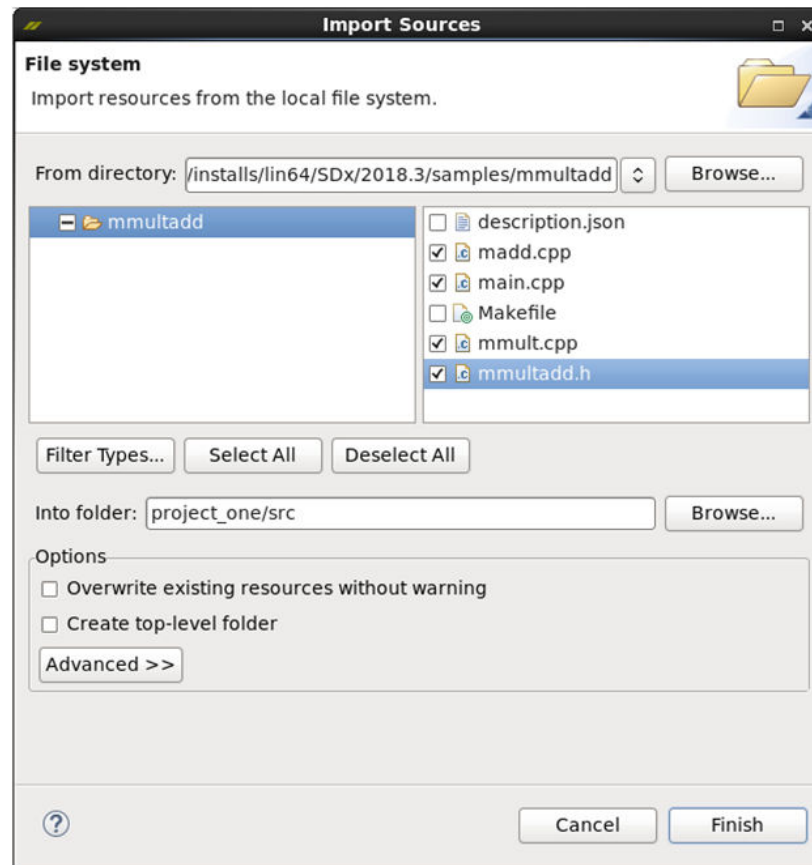
1. To add files in the Project Explorer, right-click the `src` folder and select the **Import Sources** command.



IMPORTANT! When you import source files into a workspace, it copies the files into the workspace (by default, you can deselect it). Any changes to the files are lost if you delete the workspace.

2. This displays the Import dialog box that enables you to specify the source of the files 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 to import and click **Next**.
3. The displayed dialog box depends on the source of files you selected in the prior step. In the following figure, the File system dialog box shows the result of choosing to import sources from the file system.

Figure 15: Import File System Sources



4. The File system dialog box allows you to 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.



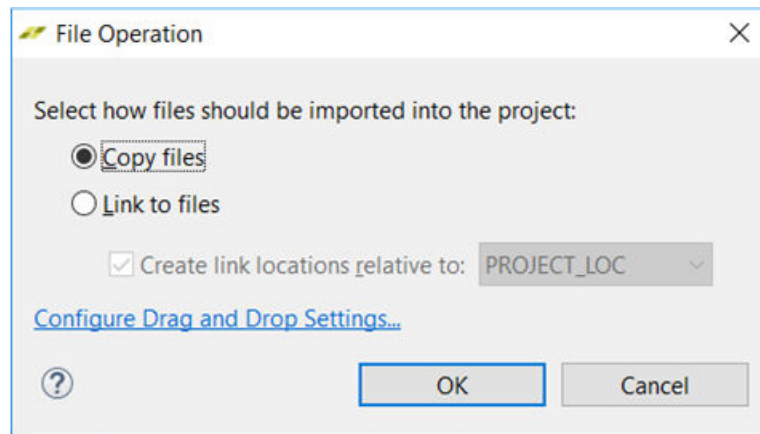
TIP: Having each hardware accelerator function exist in a separate file enables parallel compilation of the accelerator, thereby speeding up the compilation time in SDx.

5. In the Options, select the **Overwrite existing resource without warning** check box to overwrite any existing files, and select the **Create top-level folder** check box to have the files imported into a directory structure that matches the source file structure.

Note: If this check box is not enabled, which is the default, then the files are imported into the folder listed in the **Into folder** option.

6. 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 SDx IDE Project Explorer. When you perform this, the tool prompts you to specify how to add the files to your project, as shown in the following figure.

Figure 16: File and Folder Operation



7. 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 the **Configure Drag and Drop Settings** option, which allows you to specify how the tool should handle these types of drag and drop operations by default. You can also access these settings through the **Windows → Preferences** menu command.
8. After adding source files to your project, you are ready to begin configuring, compiling, and running the application.



RECOMMENDED: 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.

Importing C-Callable IP Libraries

In addition to C/C++ source files, you can incorporate pre-existing hardware functions in your design with use of a C-Callable IP library, that is published as an Arm® .a static library file. Code examples that use C-Callable IP are available in the SDSoC installation tree under the `samples/rtl` directory. You create and add your own C-Callable IP libraries through the SDx IDE.

Note: The static library (.a files) might need to be rebuilt for the appropriate Arm processor (Cortex™-A9 or Cortex-A53) before adding it to the project. Add the associated header file (.h) for the C-Callable IP function to the project, as follows:

1. In the Project Explorer, right-click the application name or its `project.sdx` file and select **Properties**.
2. Select **C/C++ Build → Settings → SDS++ Linker → Libraries**.
3. Add `<libname_without_leading_lib_text_characters>` to the `-l` libraries list.
4. Add the directory that contains the `<libname.a>` object file to the `-L` Library search path.
5. Click **Apply**.

6. Click **OK**.
7. Ensure that a C-Callable IP header (`.h/ .hpp`) file is present in project source tree.
8. Build the project.

See [Chapter 4: C-Callable IP Libraries](#) for more information on creating and using C-Callable IP libraries.

Selecting Functions for Hardware Acceleration

The first task 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.

Before marking any functions for acceleration you should profile the software. Self-contained compute intensive functions with limited external control logic are good starting points, especially when its possible to stream data between hardware, the CPU, and memory to overlap the computation with the communication. You can build and run your application on one of the platforms provided in the SDSoC environment install to identify compute intensive functions on the Arm processor.

Every base 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 do not have any functions selected for acceleration on the hardware platform. Running the application this way enables you to profile the original application code to identify candidates for acceleration.

See [Chapter 6: Profiling and Optimization](#) for more information on profiling your application. Also, the *SDSoC Environment Profiling and Optimization Guide (UG1235)* provides more extensive detail.

After determining the function or functions to move into hardware, with a project, you can select the function from the Add Hardware Functions portion of the window.



TIP: If the Editor Area window is not open, you can double-click the `<project>.sdx` file in the **Project Explorer** to open it, or you can select **Add Hardware Function** from the context menu that is available when you right-click the project.


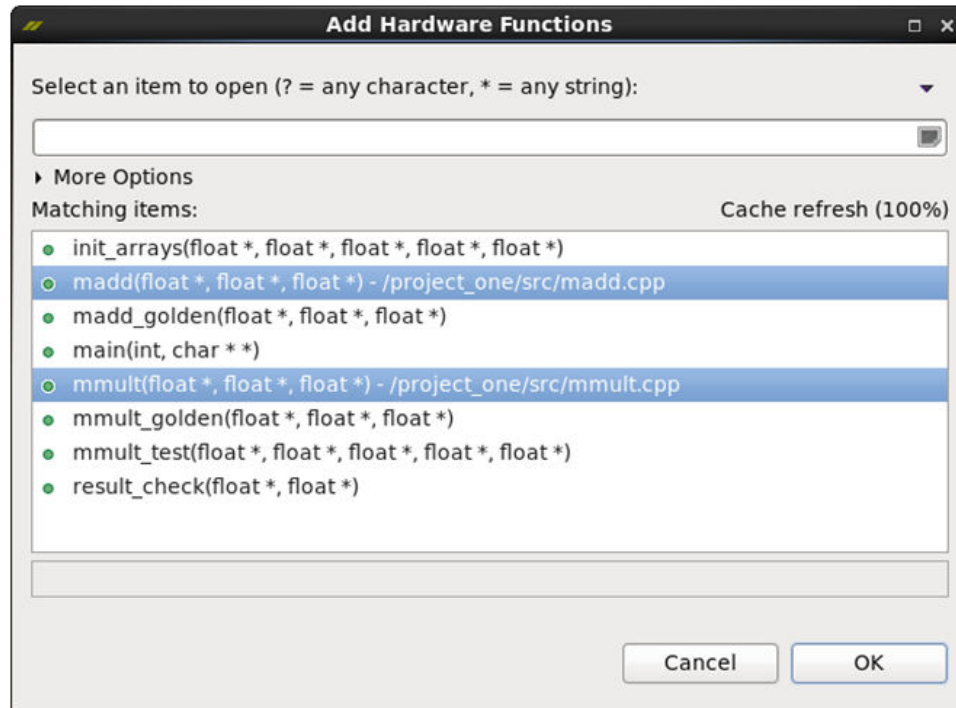
Click the **Hardware Functions** button  of the Project Editor window to display the list of candidate functions within your program. This displays the Add Hardware Functions dialog box, that lets you select one or more functions from a list of functions in the call graph of the `main` function by default.

Figure 17: Add Hardware Functions Dialog Box



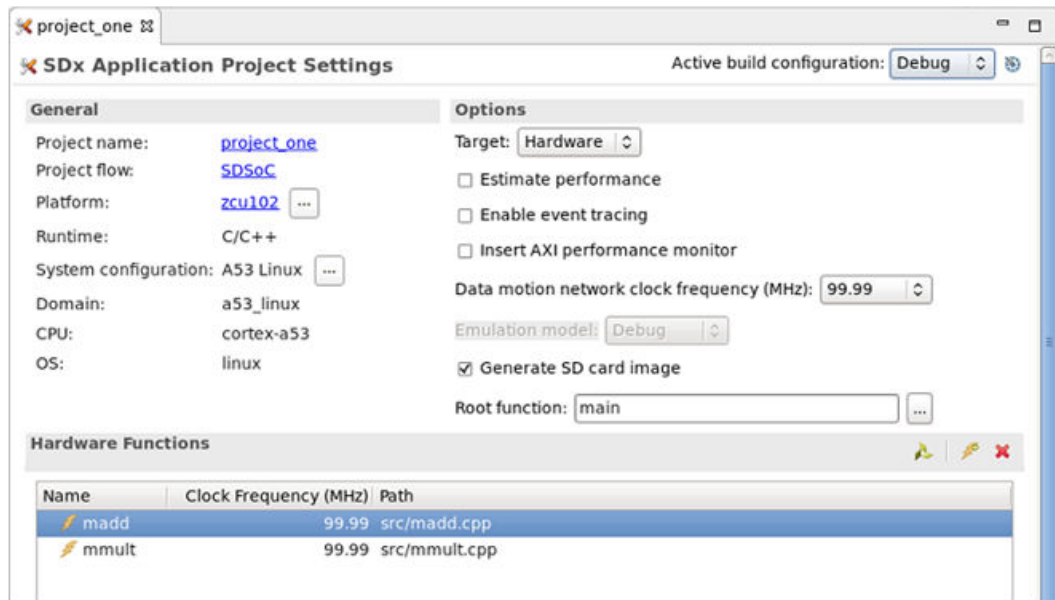
From within the Add Hardware Functions dialog box, select one or more functions for hardware acceleration and click **OK**.

The list of functions starts at the **Root function** as defined in the **Options** panel of the Project window and is set to `main` by default. You can change the **Root function** by clicking the **Browse**

() command and selecting an alternate root.

The functions display in the Hardware Functions panel of the SDx Application Project Settings window as shown in the following figure.

Figure 18: Hardware Function Panel



TIP: If you do not see a function that you expect in the Add Hardware Function dialog box, 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 the "Programming Hardware Functions" section in the *SDSoC Environment Programmers Guide* ([UG1278](#)).

For accelerators using the xOpenCV library, right-click and select **Toggle Hardware** from the associated header files in the project included in Project Explorer. See the *Xilinx OpenCV User Guide* ([UG1233](#)) for more information.

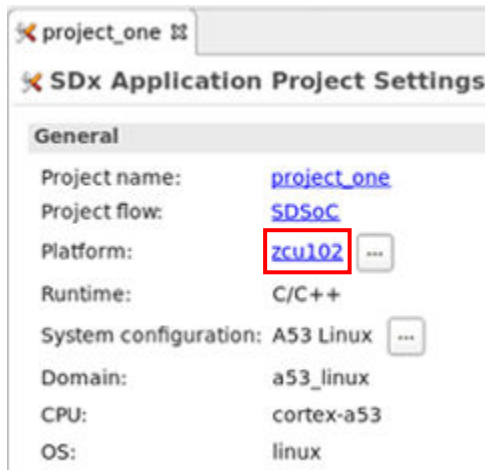
Selecting Clock Frequencies

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](#)). Setting the clock frequency of the hardware function determines the clock setting for the hardware function's logic as well as its associated data mover. Setting the clock frequency of the Data Motion network determines the clock used for the `axi_lite` control buses. By default, the Data Motion network shares its clock with the hardware accelerator generated during system generation. You can select the Data Motion network clock and the hardware function clock from the SDx IDE or the command line.

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

Figure 19: SDx IDE - General



IMPORTANT! Be aware that it might not be possible to implement the hardware system with some faster clock selections. If this occurs, reduce the clock frequency.

The function clock displays in the **SDx Application Project Settings** window, in the **Hardware Functions** panel.

Select a function from the list, like `madd` 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.

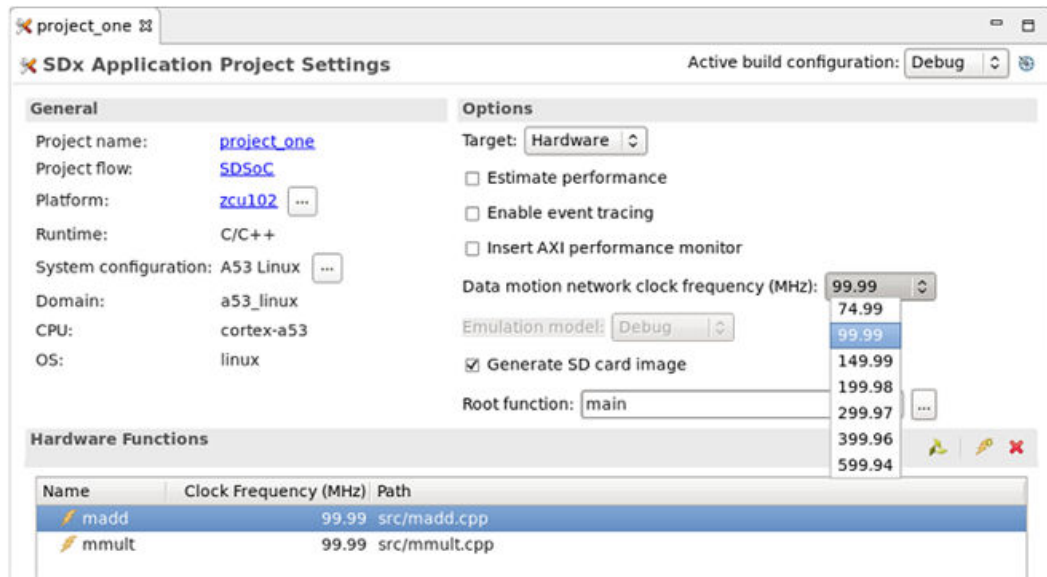
Figure 20: Select Function Clock Frequency

Hardware Functions		
Name	Clock Frequency (MHz)	Path
madd	99.99	src/madd.cpp
mmult	74.99	src/mmult.cpp

The 'Clock Frequency (MHz)' column for the 'madd' function is expanded to show a list of available frequencies: 99.99, 149.99, 199.98, 299.97, 399.96, and 599.94.

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

Figure 21: Data Motion Network Clock Frequency



Command Line Options

You can set clock frequencies for either the Hardware Accelerator or the Data Motion network at the command line, as shown in the following examples.

Set the clock frequency for a function from the command line, by specifying the Clock ID (clkid).

```
$ sds++ -c mmult.cpp -o mmult.o -sds-pf zcu102 -sds-hw mmult mmult.cpp -
clkid 1 -sds-end
```

To set the clock frequency for a hardware function in a C-callable IP library, use the `-ac` option.

```
$ sds++ -lmyIpLib -ac myAc:3 -o main.elf main.o
```

Select a Data Motion clock frequency from the command line with the `-dmclkid` option. For example:

```
$ sds++ -sds-pf zcu102 -dmclkid 1
```

You can use the following command to see the available clocks for a platform, and determine the clock ID:

```
$ sds++ -sds-pf-info <platform_name>
```

As an example for the ZCU102 platform, running the `sds++ -sds-pf-info zcu102` command displays the following output. The clock ID and its frequency is shown for each declared system clock. A list of available system configurations and system port interfaces is also provided.

```
Platform Information
=====
Name: zcu102

Device
-----
Architecture: zynqplus
Device: xczu9eg
Package: ffvb1156
Speed grade: -2

System Clocks
-----
Clock ID      Frequency
-----|-----
          1199.880127
0          74.992500
1          99.990000
2         149.985000
3         199.980000
4         299.970000
5         399.960000
6         599.940000

Platform:      zcu102 (<SDx_Install_Dir>/platforms/zcu102)

Description:
A basic platform targeting the ZCU102 evaluation board, which includes
4GB of DDR4 for the Processing System, 512MB of DDR4 for the Programmable
Logic, 2x64MB Quad-SPI Flash and an SDIO card interface. More information
at https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

Available system configurations:
a53_linux (a53_linux)
a53_standalone (a53_standalone)
r5_standalone (r5_standalone)

System Ports

Use the system port name in a sysport pragma, for example
#pragma SDS data sysport(parameter_name:system_port_name)

System Port Name (Vivado BD instance name, Vivado BD port name)
ps_e_S_AXI_HPC0_FPD (ps_e, S_AXI_HPC0_FPD)
ps_e_S_AXI_HPC1_FPD (ps_e, S_AXI_HPC1_FPD)
ps_e_S_AXI_HP0_FPD (ps_e, S_AXI_HP0_FPD)
ps_e_S_AXI_HP1_FPD (ps_e, S_AXI_HP1_FPD)
ps_e_S_AXI_HP2_FPD (ps_e, S_AXI_HP2_FPD)
ps_e_S_AXI_HP3_FPD (ps_e, S_AXI_HP3_FPD)
```

Building the SDSoC Project

After you have added source code, identified the functions to accelerate, and selected the accelerator and data mover clock frequencies, you can build for a Hardware target or an Emulation target.

For Hardware targets, see [Targeting Hardware](#).

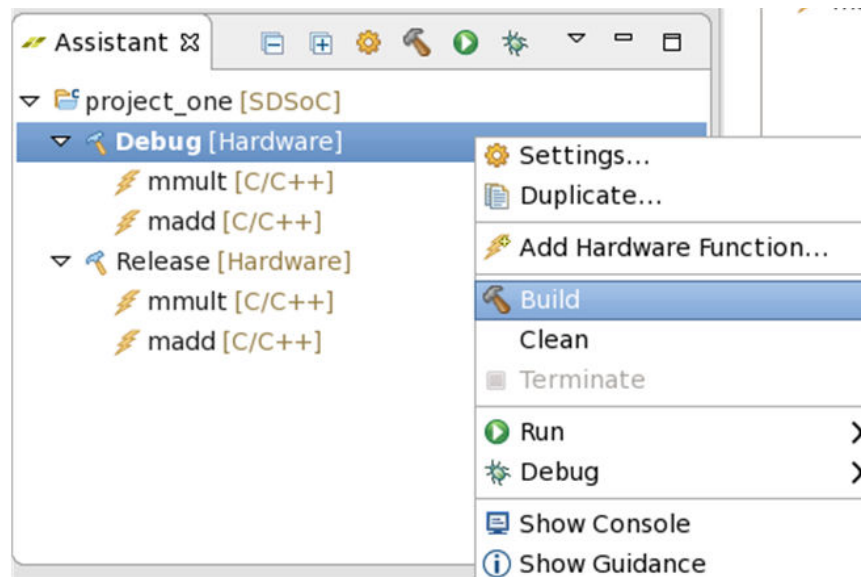
For Emulation targets, see [Targeting System Emulation](#).

The following section describes the SDx project to target a Hardware Project build.

Targeting Hardware

1. Select the **Build** button (🔧) or right-click the project name and select **Build Project** in the Project Explorer window.
2. Initiate a build through the Assistant window by right-clicking either the **Debug [Hardware]** or **Release [Hardware]** selections.
3. Right-click **Debug [Hardware]** and, from the context menu, select **Build**.

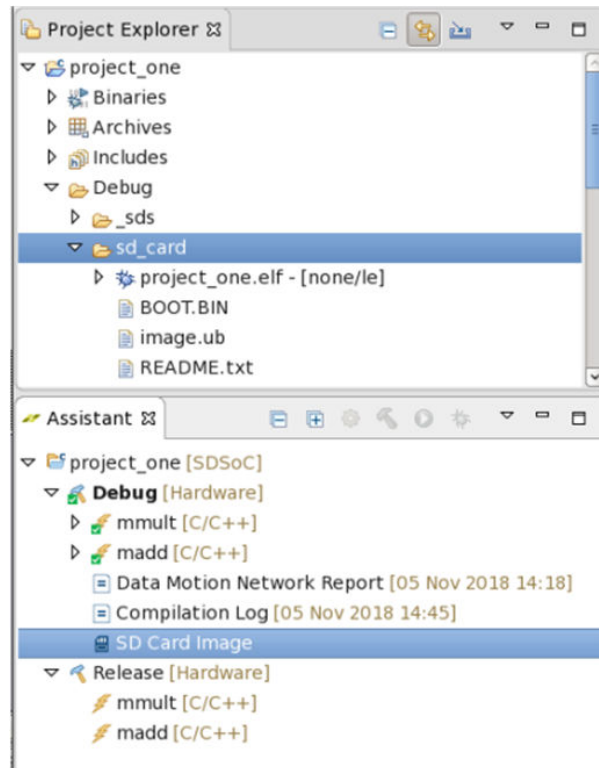
Figure 22: Assistant Window



This generates a bitstream and bootable SD card image, based on the check-marked project options. The build process also produces a **Compilation Log** file and a **Data Motion Network Report** that you can access through the Assistant window.

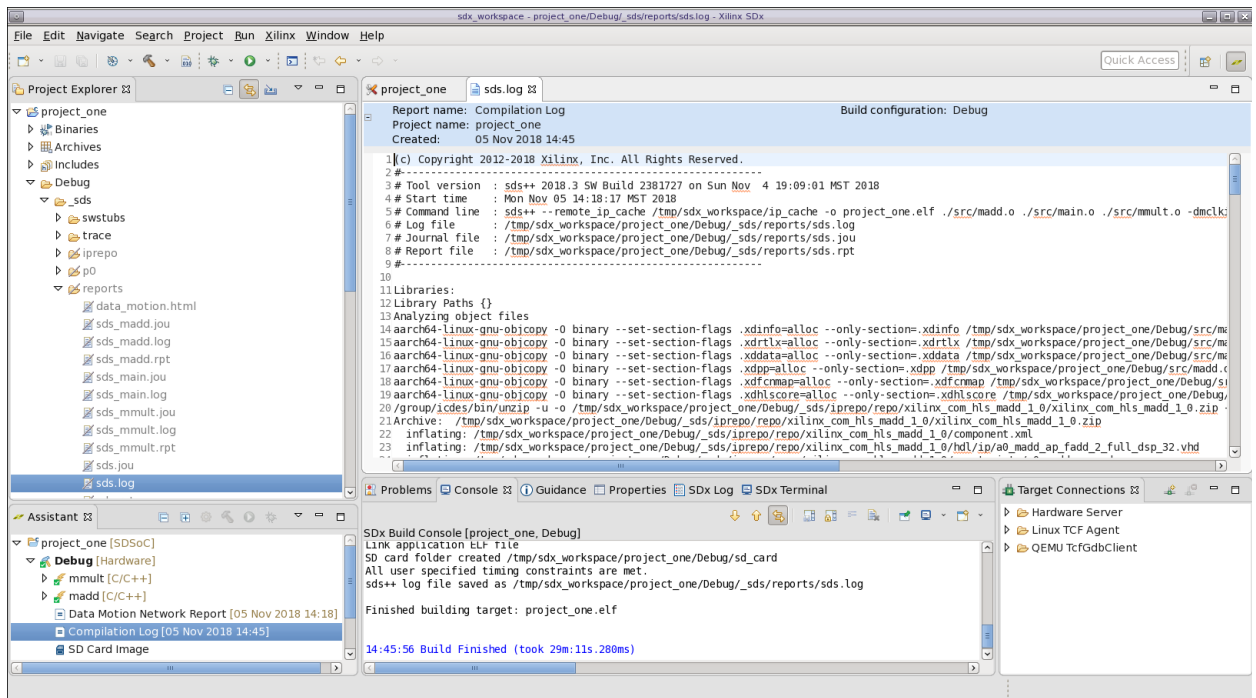
- The results of the build process produce files to populate an SD card for booting and running on a target board. Using the Assistant window, the set of generated files can be viewed by right-clicking and selecting **SD Card Image** → **Open** → **Open in File Browser**.

Figure 23: Build Results



- SDx creates detailed reports of the compilation process and saves those files in the `Debug/_sds/reports` directory. Access these files through the Project Explorer or the Assistant window. The following figure shows a Project Explorer expansion of the `reports` directory and a view into the `sds.log` compilation log file.

Figure 24: Compilation Log



An excerpt of a log file showing the equivalent system compiler commands run by the IDE for creating the accelerators added as part of the matrix multiply and matrix add example, shown in the following code snippet. The `-sds-pf` option identifies the platform, and each accelerated function is identified by placing the function name and its defining source file between their own set of `-sds-hw` and `-sds-end` hardware function options. The accelerator clock ID is also chosen within the hardware function options by setting the `-clkid` option.

Compilation of function `madd()`:

```
sds++ -Wall -O0 -g -I../src -c -fmessage-length=0 -MTsrc/madd.o\
-MMD -MP -MFsrc/madd.d -MTsrc/madd.o -o src/madd.o ../src/madd.cpp\
-sds-hw mmult mmult.cpp -clkid 1 -sds-end\
-sds-hw madd madd.cpp -clkid 1 -sds-end\
-sds-sys-config a53_linux -sds-proc a53_linux -sds-pf zcu102
```

Compilation of function `main()`:

```
sds++ -Wall -O0 -g -I ../src -c -fmessage-length=0 -MTsrc/main.o\
-MMD -MP -MFsrc/main.d -MTsrc/main.o -o src/main.o ../src/main.cpp\
-sds-hw mmult mmult.cpp -clkid 1 -sds-end\
-sds-hw madd madd.cpp -clkid 1 -sds-end\
-sds-sys-config a53_linux -sds-proc a53_linux -sds-pf zcu102
```

Compilation of function `mmult()`:

```
sds++ -Wall -O0 -g -I../src -c -fmessage-length=0 -MTsrc/mmult.o\
-MMD -MP -MFsrc/mmult.d -MTsrc/mmult.o -o src/mmult.o ../src/mmult.cpp\
-sds-hw mmult mmult.cpp -clkid 1 -sds-end\
-sds-hw madd madd.cpp -clkid 1 -sds-end\
-sds-sys-config a53_linux -sds-proc a53_linux -sds-pf zcu102
```

The compiled object files are then linked by the SDx tools to produce the single executable file for the matrix multiply and matrix add example containing the application code as well as the code to invoke the accelerated functions. This executable is targeted for Linux as specified with the `-sds-sys-config` and `-sds-proc` options.

Linking object files to produce an executable file (ELF) and boot files for SD card:

```
sds++ --remote_ip_cache /tmp/sdx_workspace/ip_cache -o project_one.elf\
./src/madd.o ./src/main.o ./src/mmult.o\
-dmclkid 1 -sds-sys-config a53_linux\
-sds-proc a53_linux -sds-pf zcu102
```

XP Option: Advanced Feature for Controlling a Vivado Build

While the `sds++` system compiler automatically invokes the Vivado® design tools to implement the hardware system, users who are familiar with Vivado tool options have the ability to further customize the flow by passing arguments on the command line. The `-xp` option of the `sds++` system compiler can be used to pass a `parameter-value` or `property-value` pair into the Vivado tools for guiding accelerator implementation.

The following is an example of specifying a Vivado synthesis option:

```
sds++ <command_options> -xp
"vivado_prop:run.synth_1.STEPS.SYNTH_DESIGN.TCL.POST=<full path to
postsynth.tcl>"
```

This example `-xp` option specifies a post-synthesis execution of a Vivado tool Tcl file. Multiple `-xp` options can be specified on a single system compiler invocation.

However, for synthesis and implementation strategies, the `sds++` command directly takes the `-impl-strategy` and `-synth-strategy` options, and does not use the `-xp` option.

For example:

```
sds++ <command_options> -synth-strategy <strategy_name>
```

```
sds++ <command_options> -impl-strategy <strategy_name>
```

```
sds++ <command_options> -synth-strategy <strategy_name> -impl-strategy
<strategy_name>
```

When using the IDE, the `-xp` options can be added to a build configuration as described in [SDS+ + Linker Settings](#), under the Miscellaneous options.

See the following resources for more information:

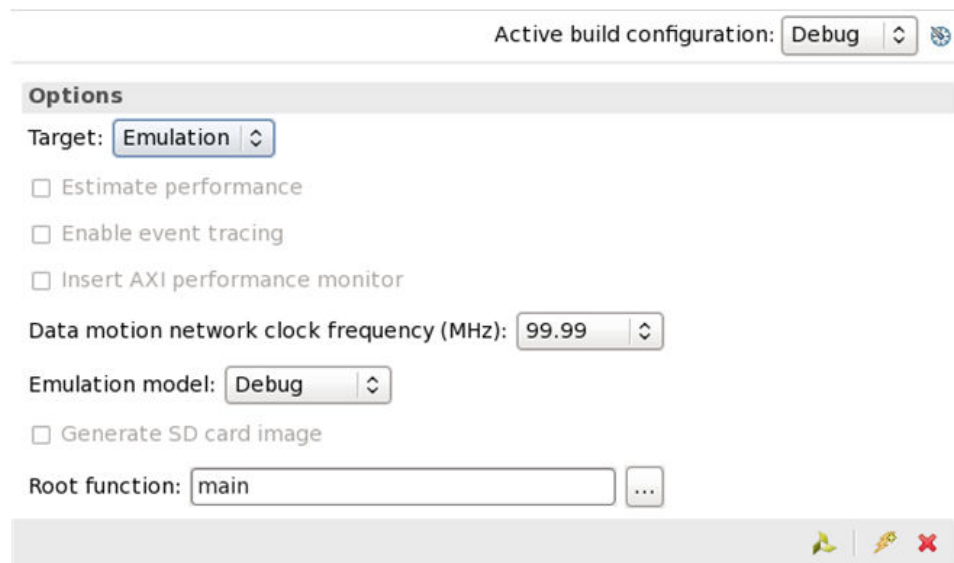
- *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#))
- *Vivado Design Suite User Guide: Synthesis* ([UG901](#))
- *Vivado Design Suite User Guide: Implementation* ([UG904](#))
- *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#))
- *SDx Command and Utility Reference Guide* ([UG1279](#))

Targeting System Emulation

After the hardware functions are identified, the logic can be compiled, and the entire system (PS and PL) verified using emulation on Xilinx base platforms (such as zc702, zc706, zcu102, zcu104, and zcu106). System emulation allows you to verify and debug the system with the same level of accuracy as a full bitstream compilation, without requiring a bitstream. This can significantly reduce design iteration time, and allow faster iteration through debug cycles more.

To enable system emulation, from the Editor Area, click the **Target** field, and select **Emulation**.

Figure 25: Emulation Target




System emulation offers **Debug** and **Optimized** modes, which can be specified by clicking in the **Emulation Model** field.

- **Debug:** Builds the system through RTL generation, and the Vivado 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. If there is a functional issue with the code, use this option.

- **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 emulation options, click the **Build** () command to compile the active build configuration. The Build command invokes the system compilers to build your application project. There are two build configurations available:

- **Debug:** This compiles the project use in software debug. The compiler produces extra information for use by a debugger to facilitate debug and allows you to 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 debug with this version.

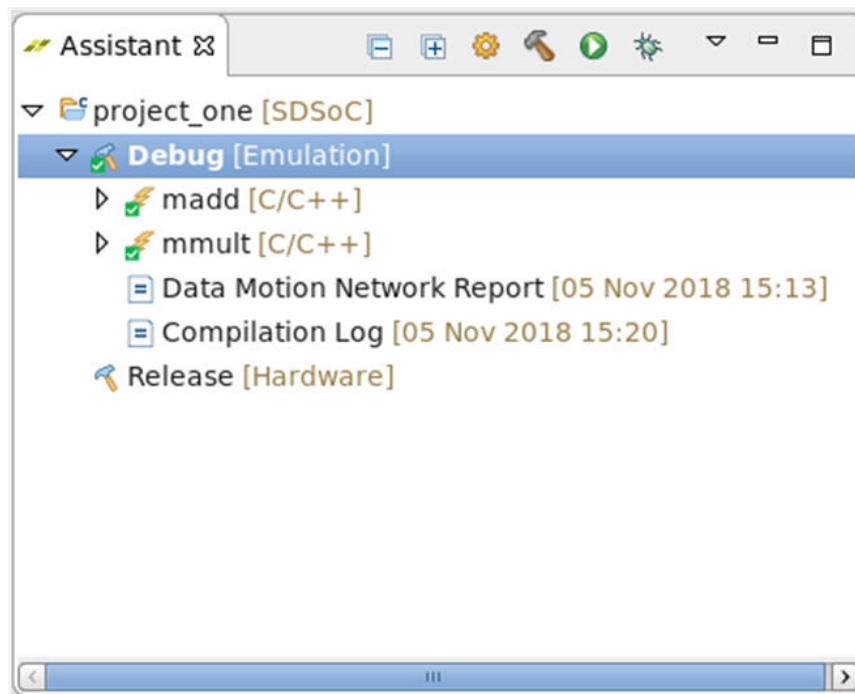


TIP: *Debug and Release modes describe how the software code is compiled; it does not affect the compilation and implementation of the hardware functions.*

You can launch the build from the Assistant window also, by selecting **Debug[Emulation] → Build**.

After system emulation has completed, the Assistant lists the **Data Motion Network Report** and the **Compilation Log** as shown in the following figure:

Figure 26: Assistant Window Display

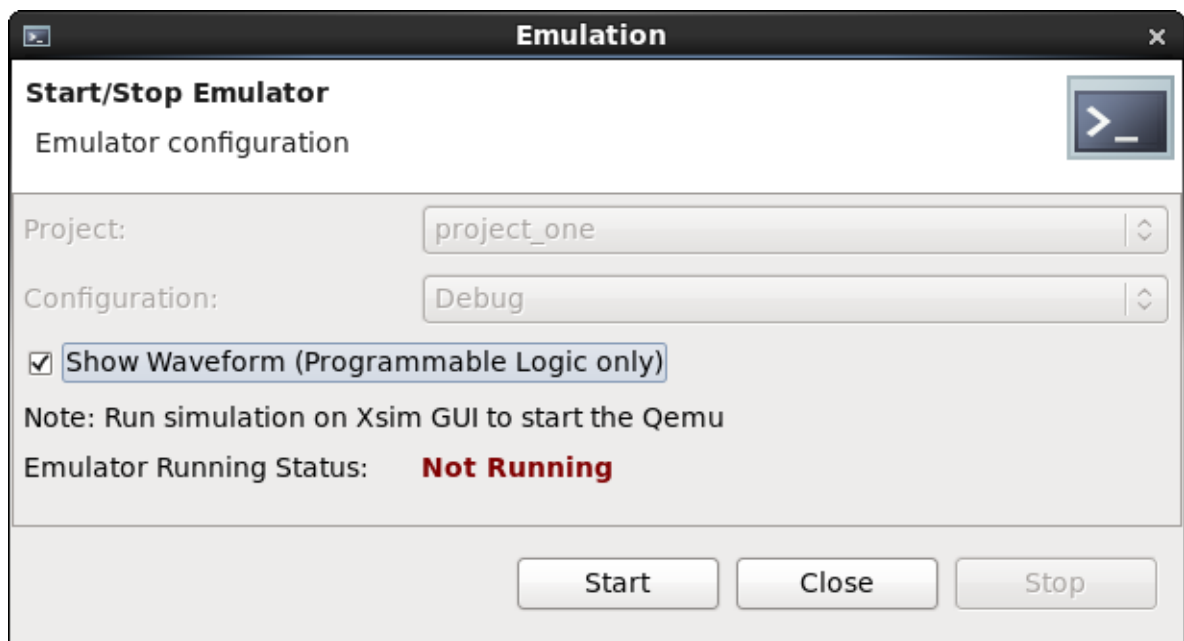


The build process for hardware emulation can take some time, depending on your application code, the size of your hardware functions, and the various options you have selected. To compile the hardware functions, the tool stack includes SDx, Vivado HLS, and the Vivado Simulator.

After the system is compiled for emulation, you can invoke the system emulator using the **Xilinx** → **Start/Stop Emulator** menu command or using `sdsoc_emulator` from the command line.

When the Start/Stop Emulator dialog box opens, if the specified emulation mode is **Debug**, you can choose to run emulation with or without waveforms. If the emulation mode is **Optimized**, the **Show waveforms** check-box is disabled and cannot be changed.

Figure 27: **Start/Stop Emulator**



Disabling the **Show Waveform** option allows you to 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 PL-resident IP with a waveform window in the Xsim tool. The 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. Select the **Run → Run All** option to start updates to the waveform window. For more information on working with the Vivado simulator waveform window, see the *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

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

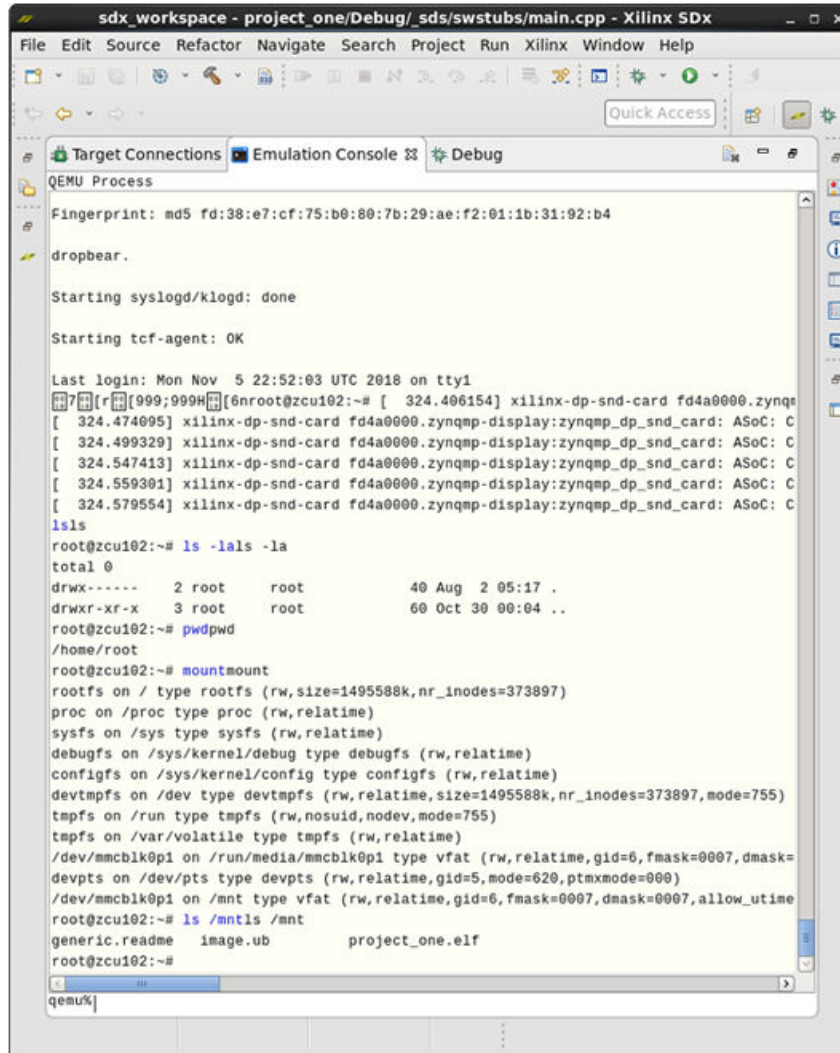
The system emulation can also be started by selecting the active project in the Project Explorer view and right-clicking to select **Run As → Launch on Emulator** menu command, or the **Debug As → Launch on Emulator** menu command. Launching the emulator from the **Debug As** menu prompts you to change to the debug perspective to arrange the windows and views to facilitate debugging the project. See [Understanding the SDx GUI](#) for more information on changing perspectives.

The program output is displayed in the console tab and if the **Show Waveform** option is selected, you also see any appropriate response in the hardware functions in the RTL-PL waveform. During any pause in the execution of the code, the RTL-PL 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**.

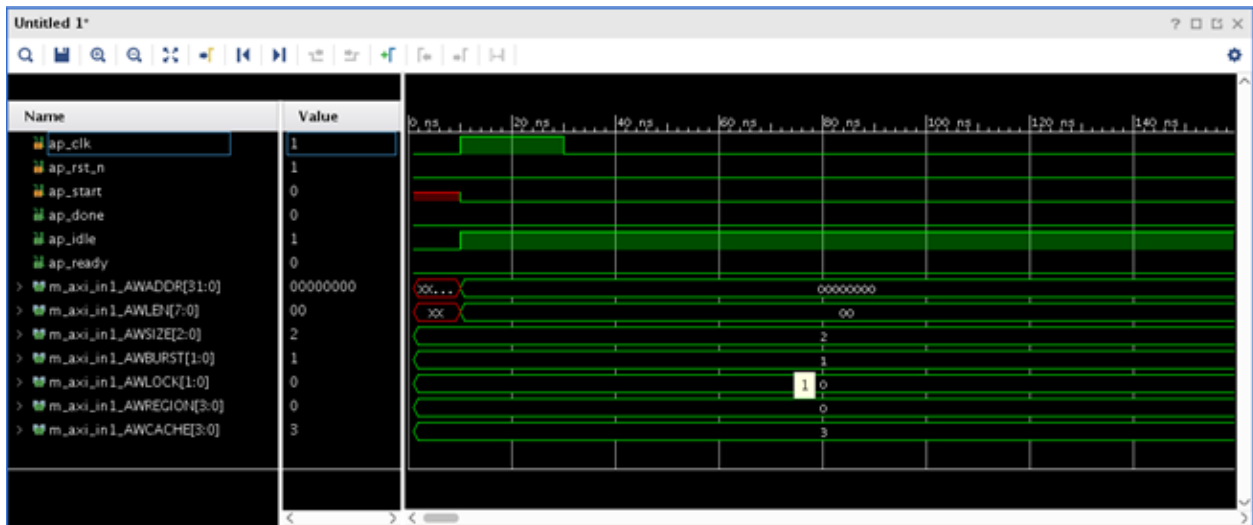
A system emulation session with the quick emulation (QEMU) console is shown in the following figure.

Figure 28: QEMU Console



As shown in the following figure, the PL waveform displays if you selected the waveform option and the **Run All** option:

Figure 29: Waveform Window



The Assistant window enables you to right-click **Debug** → **Run** → **Launch on Emulator (SDx Application Debugger)**.

You can find more information about emulation in the *SDSoC Environment Debugging Guide* ([UG1282](#)).

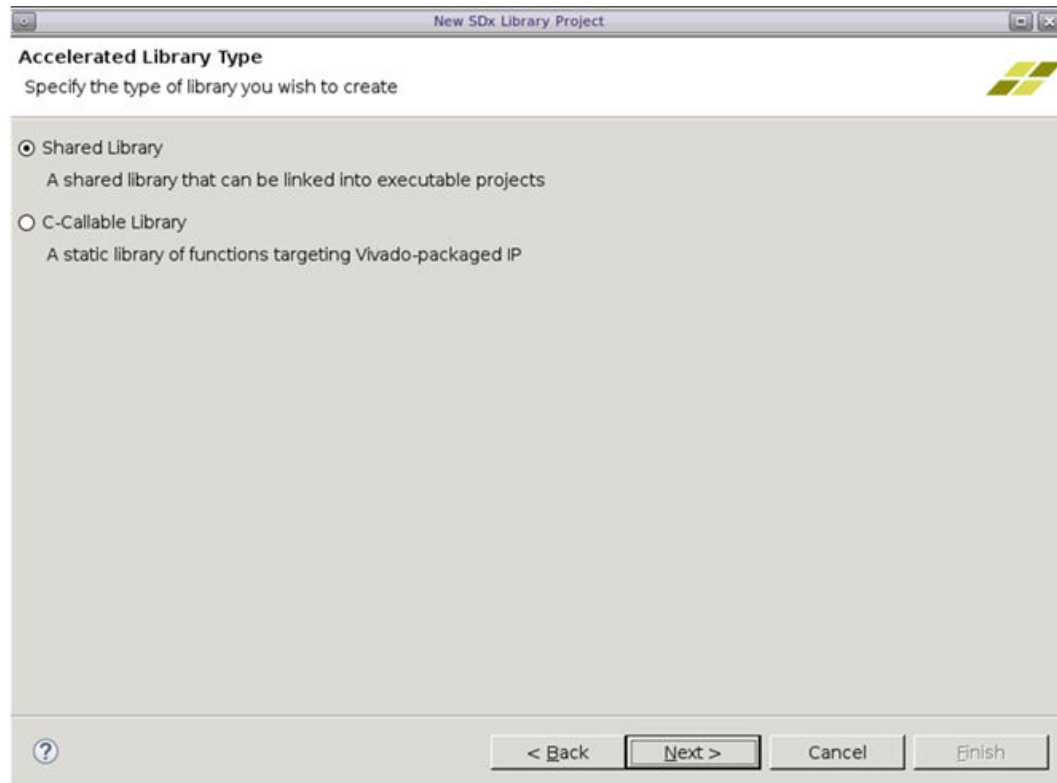


TIP: To generate an example project demonstrating emulation, create a new SDx project using the **Emulation Example** template. The `README.txt` file in the project has a step-by-step guide for performing emulation on both the SDx GUI and the command line.

Building an SDSoC Library

Libraries can be used to encapsulate one or more accelerated functions into a shared or statically linked file. The SDx IDE menu selection **File** → **New** → **SDx Library Project** provides a means to create a library project. The shared library flow produces a library file which can be linked to an SDSoC application through the SDx IDE or the command line. The SDSoC library flow is intended for exporting a library for use with `gcc/g++`. Using this library, you can call the functions which are accelerated in hardware. The C-Callable IP library flow is used for linking a Vivado packaged IP block to an SDSoC application and is also described in [Chapter 4: C-Callable IP Libraries](#).

Figure 30: Specify Library Type



Shared Library

The shared library includes a Matrix Shared Library template with a multiply and add example provided with the SDx IDE. This library demonstrates how three different functions with unique entry points in the shared library can be called from a software application.

When running the results of an application build invoked through the SDx IDE, the `BOOT.BIN` file in the `sd_card` directory of the library project should be used to boot the board as it contains the accelerated hardware functions.

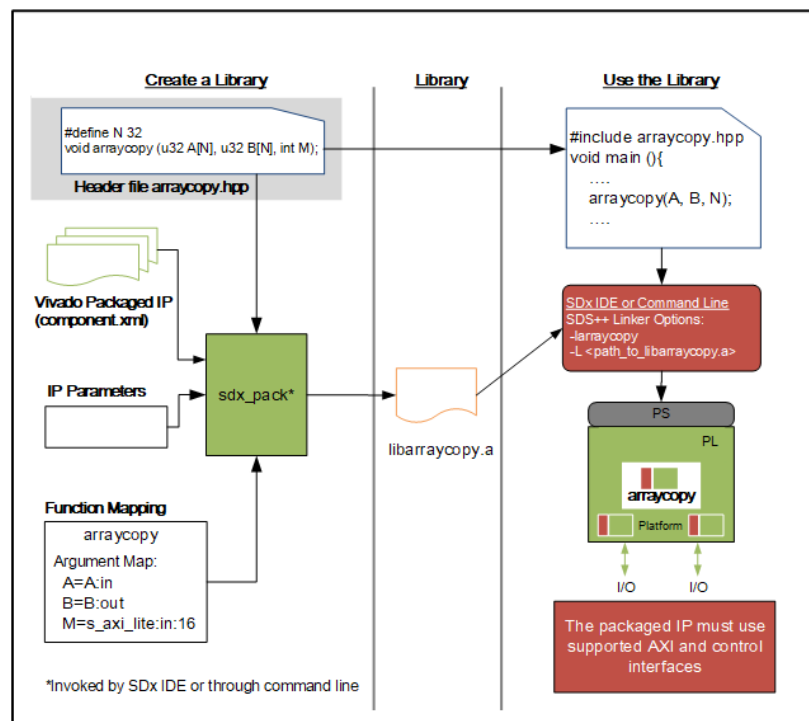
For command line use, an example is available in the `<SDx_Install_Dir>/samples/libmatrix` directory. The `libmatrix` example contains makefiles that demonstrate how to build static and shared libraries as well as how to use the libraries with an application.

For more information on building a shared library, see the "Building a Shared Library" section in *SDSoC Environment Programmers Guide* ([UG1278](#)).

C-Callable IP Libraries

This section describes how to create a C-Callable IP 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 compiler, and the IP blocks are instantiated into the generated hardware system. Using the SDx™ IDE and its command line equivalents are shown.

Figure 31: Create and Use a C-Callable Library



SDSoC applications can use C-Callable IP libraries to access packaged IP blocks written in a hardware description language (HDL) such as VHDL or Verilog or with a high-level synthesis (HLS) tool like Vivado® HLS. At times, hardware specific optimizations or micro-architectures are easier to implement through an HDL and can be delivered encapsulated within a C-Callable library.

Using a C-Callable IP library provides both the design reuse flexibility of a software library and the performance gain of optimized hardware IP blocks. With a bottom-up approach individual IP blocks can be designed, implemented, and tested prior to being placed into a C-callable library for broader use. The library of hardware-accelerated functions allows a means to insulate hardware and software development teams from low-level implementation details while still ensuring that both teams are cognizant of the functional interfaces.

Creating C-Callable IP Libraries

The SDx™ installation contains examples of C-Callable IP in the `<SDx_Install_Dir>/samples/rtl` directory. These examples show single-function as well as multi-function accelerator libraries where function arguments are passed as registers, memory references, or AXI streams, which are highlighted (`axilite_arraycopy`, `aximm_arraycopy`, `axis_arraycopy`). The multi-function examples (`mfa_fir`, `mfa_scalar128_none`, `mfa_scalar_axi`) highlight single accelerators with multiple software entry points.

The SDx IDE's Library Project flow is used to create a C-Callable IP library with foundational support from the `sdx_pack` utility. A description of the `sdx_pack` command line arguments can be found in the *SDx Command and Utility Reference Guide (UG1279)*. For inspecting interfaces within Vivado-packaged IP, `sdx_pack` provides the capability to query IP settings including their hardware interfaces. The SDx IDE uses this feature to populate menu selections with interfaces and pull-down choices relevant to the IP being transformed into a C-Callable IP library.

A procedure for creating a C-Callable IP library using the `axis_arraycopy` example for a zcu102 platform is described below. The `axis_arraycopy` example contains the Vivado-packaged IP and the files used to create the `libarraycopy.a` library. The `arraycopy.hpp` header file contains the software function declarations associated with the hardware functionality provided in the packaged IP. The `component.xml` contains the meta-data used by the SDx IDE and the underlying `sdx_pack` tool to build the library.

The following steps are necessary to create the C-Callable IP library with the SDx IDE:

1. Create a C-Callable IP Library project.
 - Each library is created for a specific `device_family`, `cpu_type`, and `OS_type` tuple.
2. Import source files from the `<SDx_Install_Dir>/samples/rtl` directory.
 - Import both the header file and the packaged IP.
3. Identify the header file (`.hpp`) and the IP meta-data file (`component.xml`) to use as inputs to build the C-Callable IP library.
4. Show how to customize the IP.
5. Indicate how the arguments of each function in the C-Callable IP library maps to the hardware IP.

The following tables guide you on how to complete the SDx dialogs.

- The top row of each table contains the SDx IDE menu selection to begin each task.
- The **Dialog** column lists the names of the subsequent dialog boxes that open.
- **Selection** and **Action** columns indicate how to fill out the dialog boxes to complete the task.

SDx Library Project

Begin by launching the SDx IDE and specifying a workspace (for example, `sdx_workspace`) to create a Library project (**File** → **New** → **SDx Library Project**).

The project name becomes the name of the library created by concatenating the prefix `lib` and the `.a` suffix (`lib<project_name>.a`).

Table 4: SDx Library Project

Dialog	Selection or Field Name	Action
Project Type	Application	Click Next
Create a New SDx Project	Project name:	<code>arraycopy</code>
	Use default location	Check-mark
		Click Next
Accelerated Library Type	Type:	C-Callable Library
Platform	Name	zcu102
		Click Next
System Configuration	System configuration:	A53_Linux
	Runtime:	C/C++
	Domain:	a53_linux
(pre-set)	CPU:	cortex-a53
(pre-set)	OS:	linux
	Linux Root File System:	Leave unchecked
		Click Next
Templates	Empty Application	Click Finish

Import Sources Dialog Options

The following table shows the menu selection for importing sources.

Table 5: Import Sources Dialog Options

Dialog	Selection or Field Name	Action
File system	From directory:	Browse to <code>axis_arraycopy</code> directory in <code>samples/rtl</code>
		Click OK
	Files: <code>src/arraycopy.hpp</code>	Check-marked
	Directory: <code>ip</code>	Check-marked
	Into folder:	<code>arraycopy/src</code>
		Click Finish

Add IP Customizations

In the IP Customizations window, click **Add IP Customizations** (file icon with stylized "h"). The following figure and table show how to add IP customization.



Table 6: Add IP Customizations

Dialog	Selection or Field Name	Action
Add IP Customizations	Header File:	Click Select
	Files:	Select <code>arraycopy.hpp</code>
	Qualifier:	Select <code>src/src/arraycopy.hpp</code>
		Click OK
	IP Path:	Click Select
	Files:	Select <code>component.xml</code>
	Qualifier:	Select <code>src/ip/component.xml</code>
		Click OK
	Accelerator control:	Protocol: ap_ctrl_hs
		Port: s_axi_lite
		Offset: 0
	Primary Clock: <code>ap_clk</code>	10.0
	Derived Clock:	(no change)
IP Parameters	(no change)	
	Click OK	

Add Function Mapping

This `axis_arraycopy` example uses the contents of the provided `samples/rtl/axis_arraycopy/src/Makefile` to complete the dialog box option.

The `component.xml` and, if provided, the `register_map.txt` files associated with the IP block can also be queried for information on the how the function arguments map to the hardware.

In the IP Customizations window, click **Add Function Mapping** ("+" icon) shown in the figure and table.

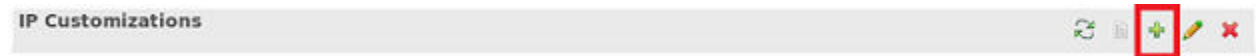


Table 7: Add Function Mapping

Dialog	Selection or Field Name	Action
Add Function Mapping	Function name:	Click "+" icon on the right-side
		Select <code>arraycopy</code>
		Click OK
	Arguments and Function Return mapped to AXILite Interface	Click Add Function Argument Map ("+" icon) above the table for this interface type
	Argument (Click within field to expose pull-down menu)	M
	AXILite Interface (Click within field to expose pull-down menu)	s_axi_lite
	Direction (Click within field to expose pull-down menu)	IN
	Register Info (Click within field to expose pull-down menu)	M , at offset 16
	Array Arguments mapped to AXIS Interface	Click Add Function Argument Map ("+" icon) above the table for this interface type
	Argument (Click within field to expose pull-down menu)	A
	AXIS Interface (Click within field to expose pull-down menu)	A
	Direction (Click within field to expose pull-down menu)	IN
	Array Arguments mapped to AXISinterface	Click Add Function Argument Map ("+" icon) above the table for this interface type
	Argument (Click within field to expose pull-down menu)	B
	AXIS Interface (Click within field to expose pull-down menu)	B
	Direction (Click within field to expose pull-down menu)	OUT
	Complete the Add Function Mapping dialog box	Click OK

Building the C-Callable IP Project

To build the project, the C-Callable IP library is generated and placed in the build output directory of the application (for example, the `Release` directory) with the name `lib<application>.a` (`libarraycopy.a` for this example).

In addition to the SDx IDE method of creating a C-Callable IP library, a command line method that directly invokes the `sdx_pack` tool is available. The equivalent `sdx_pack` command to match the actions taken with the SDx IDE for the `axis_arraycopy` example is:

```
sdx_pack -header arraycopy.hpp -lib libarraycopy.a \
-func arraycopy -map A=A:in -map B=B:out -map M=s_axi_lite:in:16 -func-end \
-ip ../ip/component.xml -control ap_ctrl_hs=s_axi_lite:0 \
-primary-clk ap_clk=10.0 -target-family zynqplus \
-target-cpu cortex-a53 -target-os linux \
-verbose
```

Note: The C-Callable function and its argument map is listed between `-func` and `-func-end` options of the `sdx_pack` call.

Multi-Function Accelerator Libraries

The `axis_arraycopy` example is a library with a single accelerator function. Other examples, in particular the ones that begin with the `mfa_` prefix are multi-function accelerator (MFA) libraries where more than one function is mapped onto one IP block. Below is the `sdx_pack` command for the `mfa_scalar_128_none` example that generates the `libmfa.a`.

The C-Callable IP library contains eight functions and is shown in the following code example. This accelerator library uses control protocol `none`, indicating that the user explicitly controls the IP. This MFA example also demonstrates 128-bit scalar function arguments that map to AXI4-Lite interfaces as well as 128-bit array arguments that map to master AXI4-Stream interfaces.

```
sdx_pack -header mfa.hpp -I inc -lib libmfa.a \
-func mfa_reset -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_init -map inA=inA:in -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_copy -map outB=outB:out -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_sum -map result=axi_AXILiteS:out:0x2c -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_status -map return=axi_AXILiteS:out:0x10 -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_status2 -map status=s_axi_AXILiteS:out:0x10 -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_result -map result=s_axi_AXILiteS:out:0x2c -func-end \
-func mfa_stop -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-ip ../ip/component.xml -control none \
-add-ip-repo ../dummy_ip \
-add-ip-repo ../dummy_ip_repo \
-primary-clk ap_clk=10.0 \
-target-family zynqplus -target-cpu cortex-a53 -target-os linux -verbose
```

Another example of an MFA type of C-Callable library is the `mfa_scalar_axi` accelerator. This accelerator library uses an `ap_ctrl_hs` control protocol and shows the use of scalar function arguments that map to AXI4-Lite interfaces as well as array arguments that map to master AXI4 interfaces.

```

sdx_pack -header mfa.hpp -I inc -lib libmfa.a \
    -func mfa_reset -map status=s_axi_AXILiteS:out:0x20 -map
inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_init -map inA=s_axi_AXILiteS:in:0x10,m_axi_inA:in -map
status=s_axi_AXILiteS:out:0x20 \
    -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_copy -map outB=s_axi_AXILiteS:in:0x18,m_axi_outB:out -map
status=s_axi_AXILiteS:out:0x20 \
    -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_sum -map result=s_axi_AXILiteS:out:0x28 -map
status=s_axi_AXILiteS:out:0x20 \
    -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_status -map status=s_axi_AXILiteS:out:0x20 -func-end \
    -func mfa_stop -map status=s_axi_AXILiteS:out:0x20 -map
inst=s_axi_AXILiteS:in:0x34 -func-end \
    -ip ../ip/component.xml -control AXI=s_axi_AXILiteS:0x0 \
    -primary-clk ap_clk=10.0 \
    -target-family zynqplus -target-cpu cortex-a53 -target-os linux -
verbose
    
```

A register map showing the bit-level definition of the AXI4-Lite control protocol signals used in the `mfa_scalar_axi` example is provided in the `mfa_scalar_axi/ip/register_map.txt` file and excerpted below. In general, IP register mapping information is provided by the IP developer.

```

// =====
// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and
// SystemC
// Version: 2018.3
// Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
//
// =====
// AXILiteS
// 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
// 0x04 : Global Interrupt Enable Register
//      bit 0 - Global Interrupt Enable (Read/Write)
//      others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//      bit 0 - Channel 0 (ap_done)
//      bit 1 - Channel 1 (ap_ready)
//      others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//      bit 0 - Channel 0 (ap_done)
//      bit 1 - Channel 1 (ap_ready)
//      others - reserved
// 0x10 : Data signal of inA_offset
//      bit 31~0 - inA_offset[31:0] (Read/Write)
    
```

```
// 0x14 : reserved
// 0x18 : Data signal of outB_offset
//         bit 31~0 - outB_offset[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of status
//         bit 31~0 - status[31:0] (Read)
// 0x24 : Control signal of status
//         bit 0 - status_ap_vld (Read/COR)
//         others - reserved
// 0x28 : Data signal of result
//         bit 31~0 - result[31:0] (Read)
// 0x2c : Data signal of result
//         bit 31~0 - result[63:32] (Read)
// 0x30 : Control signal of result
//         bit 0 - result_ap_vld (Read/COR)
//         others - reserved
// 0x34 : Data signal of inst
//         bit 31~0 - inst[31:0] (Read/Write)
// 0x38 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
// Clear on Handshake)
```

The `mfa_fir` C-Callable IP library example highlights instantiating IP parameters, using AXI4-Stream interfaces, a 24-bit data type, and a control protocol selection of **none**.

```
ibfir.a: fir.hpp
sdx_pack -header fir.hpp -lib libfir.a \
-func fir -map X=S_AXIS_DATA:in -map Y=M_AXIS_DATA:out -func-end \
-func fir_reload -map H=S_AXIS_RELOAD:in -func-end \
-func fir_config -map H=S_AXIS_CONFIG:in -func-end \
-ip ../ip/fir_compiler_v7_2/component.xml -control none \
-param DATA_Has_TLAST="Packet_Framing" \
-param M_DATA_Has_TREADY="true" \
-param Coefficient_Width="8" \
-param Data_Width="8" \
-param Quantization="Integer_Coefficients" \
-param Output_Rounding_Mode="Full_Precision" \
-param Coefficient_Reload="true" \
-param Coefficient_Structure=Non_Symmetric \
-primary-clk aclk_intf=10.0 \
-target-family zynqplus -target-cpu cortex-a53 -target-os linux -
verbose
```

Considerations for C-Callable IP Libraries

1. Function arguments of `TYPE *a` or `TYPE &a` are interpreted as an OUTPUT scalar.
2. Arrays must be declared as `TYPE a[N]` or `TYPE a[]`.
3. Function return type can only be a scalar in the format of `TYPE`: `TYPE*` or `TYPE&` are not allowed.
4. C-Callable IP library header files cannot have SDS pragmas that use MACRO as parameters: `#pragma sds data copy (A[0:SIZE])` is not allowed when `SIZE` is a macro (for example, `#define SIZE 16`).

5. Overlapped function calls of multi-function accelerators (MFAs) are not allowed, as there is only one IP instance in the hardware; therefore, `async` pragmas around MFA functions are very risky and not recommended unless there is no chance of overlapping during runtime.
6. Argument sizes must match between the software declared argument and the hardware port pair: `<project_name>/Release/reports/sdx_pack.html` report file can be used to double-check if the library implemented the expected argument sizes, offsets, and bus interfaces.
7. This supports up to one AXI4-Lite interface per top-level IP.
8. For C-Callable functions, all pragmas must be applied when building the library using `sdx_pack`. Any pragmas added into the header file after the library is already built are ignored by the tool.

sdx_pack Command

The following are examples of the `sdx_pack` command:

```
sdx_pack -header <header.h/pp>-ip <component.xml>
```

```
[-param <name>="value"] [configuration options]
```

The following table provides further details on the `sdx_pack` tool.

Table 8: sdx_pack Command Options

Option	Description
<code>-header header.h/.hpp</code>	(required) Header file (.h, .hpp) with function prototypes. <code>sdx_pack</code> generates C++ style library for a .hpp file and a C-style library for a .h file. Only one top-level header file is allowed per library. The top-level header can include other header files, using the <code>-I</code> option.
<code>-ip component.xml</code>	(required) Path to Vivado packaged IP. Only one top-level IP per library. The top-level IP can invoke other IP blocks, using the <code>-add-ip-repo</code> option.
<code>-control protocol [=port [:offset]]</code>	(required) IP control protocol options: <ul style="list-style-type: none"> • <code>ap_ctrl_hs</code> Automatic control protocol based on an AXI4-Lite control register, typically at offset 0x0 (for example: <code>-control ap_ctrl_hs=s_axi_AXILiteS:0</code>) • <code>none</code> User application explicitly controls the IP (for example <code>-control none</code>)

Table 8: `sdx_pack` Command Options (cont'd)

Option	Description
<pre>-func function_name -map swName=hwNAME:direction[:offset[aximm_name:direction]] -func-end</pre>	<p>(required)</p> <p>Specify a list of C-Callable IP functions. Each function is listed between a <code>-func</code> and <code>-func-end</code> option pair. Function arguments are mapped to each IP port with the <code>-map</code> option. The <code>-map</code> option is then used as follows:</p> <ul style="list-style-type: none"> • Scalars map onto an AXI4-Lite interface <ul style="list-style-type: none"> ◦ Map input scalar (for example, <code>int a</code>) with <code>-map a=s_axi_AXILiteS:in:offset</code> ◦ Map output scalar (for example: <code>int *a</code>, or <code>int &a</code>) with <code>-map a=s_axi_AXILiteS:out:offset</code> ◦ Map function return scalar (return type can only be a scalar) with <code>-map return=s_axi_AXILiteS:out:offset</code> • Arrays map onto AXI4, AXI4-Stream, or AXI4-Lite interfaces. The arrays must be one-dimensional (for example, <code>int a[N]</code>, or <code>int a[]</code>). <ul style="list-style-type: none"> ◦ Map to AXI4, with <code>-map a=s_axiAXILiteS:in:offset,a_hwName:direction</code>. Not allowed when <code>control=none</code>. The first part is mapping to the address and the second part is mapping to the data port. ◦ Map to AXI4-Stream with <code>-map a=a_hwName:direction</code>. ◦ Map to AXI4-Lite with <code>-map a=s_axi_AXILiteS:in:offset</code>. Array must be one-dimensional and of constant size.
<pre>-param name="value"</pre>	IP parameter name-value pairs to instantiate IP parameters. Use one <code>-param</code> option per pair.
<pre>-lib libname</pre>	Use specified <code>libname</code> for naming the generated library. By default <code>lib_header.a</code> is used.
<pre>-I path</pre>	If the file named with the <code>-header</code> option includes other files, this option specifies the path to the additionally included files. Multiple <code>-I</code> options can be used. For easier library distribution, place all include files into a single directory.
<pre>-add-ip-repo path</pre>	Add all IP found in the listed repository into the library. Although multiple <code>-add-ip-repo</code> options can be used to specify multiple paths. Xilinx recommends to place all required IP into a single directory and use a single <code>-add-ip-repo</code> option.
<pre>-primary-clk clk_interface=min_clk_period</pre>	Specify the primary clock interface and its minimum clock period in nanoseconds.
<pre>-derived-clk clk_interface=multiplier:divisor</pre>	Specify a phase-aligned derived clock interface and its multiplier and divisor in units of integers. Only two phase-aligned clocks are supported.
<pre>-target-family device_family</pre>	The target device family supported by the IP (for example, <code>zynq</code> (default), <code>zynqplus</code>).

Table 8: `sdx_pack` Command Options (cont'd)

Option	Description
<code>-target-cpu cpu_type</code>	Specify target CPU: <ul style="list-style-type: none"> <code>cortex-a9</code> (default) <code>cortex-a53</code> <code>cortex-r5</code> <code>microblaze</code>
<code>-target-os name</code>	Specify target OS: <ul style="list-style-type: none"> <code>linux</code> (default) <code>standalone</code> (bare-metal)
<code>-query-target type</code>	Query one of: supported device families, cpu types, or OS type for the IP [<code>family, cpu, os</code>]
<code>-query-interface type</code>	Query interfaces and parameters of the IP. Multiple query types supported [<code>all, aximm, axilite, axis, clock, control, param, misc</code>] Note: This requires that the IP has packaged all necessary information needed by the query.
<code>-o output.json</code>	User-specified JSON file to save query results.
<code>-verbose</code>	Print verbose output to <code>STDOUT</code> .
<code>-version</code>	Print the <code>sdx_pack</code> version information to <code>STDOUT</code> .
<code>-h, -help, --help</code>	Display <code>sdx_pack</code> option usage and descriptions.

Here is an example of the code:

```
sdx_pack -header arraycopy.hpp -lib libarraycopy.a \
-func arraycopy -map A=A:in -map B=B:out \
-map M=s_axi_lite:in:16 -func-end \
-ip ../ip/component.xml -control AXI=s_axi_lite:0 \
-target-family zynqplus -target-cpu cortex-a53 -target-os standalone \
-verbose
```

Where:

- `arraycopy.hpp` specifies the header file defining the function prototype for the `arraycopy` function.
- `component.xml` of the IP generates the packaged Vivado IP for SDx.
- `-control` specifies the IP control protocol.
- `-map` specifies the mapping of an argument from the software function to a port on the Vivado IP. Notice the option is used three times in the example above to map function arguments A, B, and M to IP ports.
- The `-target-os` option specifies the target operating system.

The `sdx_pack` utility generates a C-Callable IP library to match the name in the `-lib` option, `libarraycopy.a` in this case.

Using C-Callable IP Libraries

After generating the C-Callable library, create a new SDx Application project to use the library. Continuing with the example of the `axis_arraycopy` C-Callable IP library built in the previous section, use the generated library (`libarraycopy.a`) from the library build's Release directory. The result of building the Application project is an executable file (ELF) that is linked with the C-Callable IP Library.

1. Create an SDx application project to output an executable file. Click **File → New → SDx Application Project**.

Note: The tuple consisting of `device_family`, `cpu_type`, and `os_type` must match that of the C-Callable IP library.

Table 9: SDx Application Project

Dialog Box	Selection or Field Name	Action
Project Type	Application	Click Next
Create a New SDx Project	Project name:	<code>app_arraycopy</code>
	Use default location	Check-marked
		Click Next
Platform	Name	<code>zcu102</code>
		Click Next
System Configuration	System configuration:	A53_Linux
	Runtime:	C/C++
	Domain:	a53_linux
(pre-set)	CPU:	cortex-a53
(pre-set)	OS:	linux
	Linux Root File System:	Unchecked
		Click Next
Templates	Empty Application	Click Finish

2. Import the function declarations header file (`.hpp`) common to both the library and the application. In the Project Explorer window, right-click `app_arraycopy` and select **Import Sources**.

Table 10: Select Import Sources

Dialog	Selection or Field Name	Action
File system	From directory:	Browse to <code>axis_arraycopy/src</code> directory in <code><SDx_Install_Dir>/samples/rtl</code> .
		Click OK
	Files: arraycopy.hpp	Check-marked
	Into folder:	<code>app_arraycopy/src</code>

Table 10: Select Import Sources (cont'd)

Dialog	Selection or Field Name	Action
		Click Finish

- Open the Importing Sources dialog box again to get the example main application code from the `<SDx_Install_Dir>/samples/rtl` directory. In the Project Explorer window, right-click `app_arraycopy` and select **Import Sources**.

Table 11: Select Import Sources

Dialog	Selection or Field Name	Action
File system	From directory:	Browse to the <code>axis_arraycopy/app</code> directory in <code><SDx_Install_Dir>/samples/rtl</code>
		Click OK
	Files: main.cpp	Check-marked
	Into folder:	<code>app_arraycopy/src</code>
		Click Finish

Now that the source files for the `app_arraycopy` application have been imported, update the **C/C++ Build Settings** to have the `sds++` linker use the `arraycopy.a` C-Callable IP library when building the application.

- Update the **C/C++ Build Settings** in the Project Explorer window, right-click `app_arraycopy` and select **C/C++ Build Settings** as shown in the table.

Table 12: C/C++ Build Settings

Dialog	Selection or Field Name	Action
Settings → Tool Settings	SDS++ Linker	Select Libraries
		Click Add symbol (with "+" icon) in the Libraries (-1) window
	Libraries(-1)	<code>arraycopy</code>
		Click OK
		Click Add symbol (with "+" icon) in the Library search path (-L) window
		Click Workspace
	Folder:	Navigate to and select arraycopy/Release
		Click OK
(pre-set)	Directory:	<code>\${workspace_loc:/arraycopy/Release}</code>
		Click OK
		Click Apply and Close

To create the application, you can use the Assistant window to build the `app_arraycopy` application.

1. In the Assistant window under **app_arraycopy[SDSoC]**, right-click **Debug[Hardware]** and select **Build**. The Console window shows the build progression including the `sds++` system compiler invocation.
2. After the application successfully builds the target executable file (`app_arraycopy.elf`), the Assistant window populates with a **Data Motion Network Report**, the **Compilation Log**, and an **SD Card Image** menu. Through the **SD Card Image** menu, the contents of the generated (`sd_card`) files directory is available to view using the Project Explorer, a file browser, or a command shell window.
3. When the build completes, you can write the contents of the generated `sd_card` directory to the root of a FAT32-formatted SD card and boot and run the `app_arraycopy.elf` application on a ZCU102 board. The `sd_card` directory includes a `README.txt` for boot setup instructions, a bootable `BOOT.BIN` file, and the `image.ub` file used to boot Linux.

The SDx IDE builds the application with the `sds++` system compiler using the C-callable library and the application code. The main application is compiled to produce an object file and then it is linked with the C-callable library (`arraycopy`).

The following examples show the issued commands.

Compilation of `main.cpp`:

```
sds++ -Wall -O0 -g -I../src -c -fmessage-length=0 -MTsrc/main.o -MMD -MP -
MFsrc/main.d \
  -MTsrc/main.o -o src/main.o ../src/main.cpp \
  -sds-sys-config a53_linux -sds-proc a53_linux -sds-pf zcu102
```

Linking `main.o` with `arraycopy` library to produce executable application

`app_arraycopy.elf`:

```
sds++ -L<path_to_arraycopy/Release> --remote-ip-cache ../ip_cache \
  -o app_arraycopy.elf ../src/main.o -larraycopy -dmclkid 1 \
  -sds-sys-config a53_linux -sds-proc a53_linux -sds-pf zcu102
```

Debugging Techniques

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

When debugging an SDSoC application with a board attached to the debug Host machine, you can use the Assistant view and right-click the **Debug[Hardware] → Debug → Launch on Hardware** option to begin a debug session. You can also set the options through the Assistant by selecting **Debug[Hardware] → Debug → Debug Configurations**.

Note: As the debug environment is initialized, Xilinx recommends that users switch to the **Debug** perspective when prompted by the SDx™ IDE.

The debug perspective view provides the ability to debug the standard C/C++ portions of the application, by single-stepping code, setting and removing breakpoints, displaying variables, dumping registers, viewing memory, and controlling the code flow with *run until* and *jump to type* debugging directives. Inputs and outputs can be observed pre- and post- function call to determine correct behavior.

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

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

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

Profiling and Optimization

There are two distinct areas for you to consider 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.

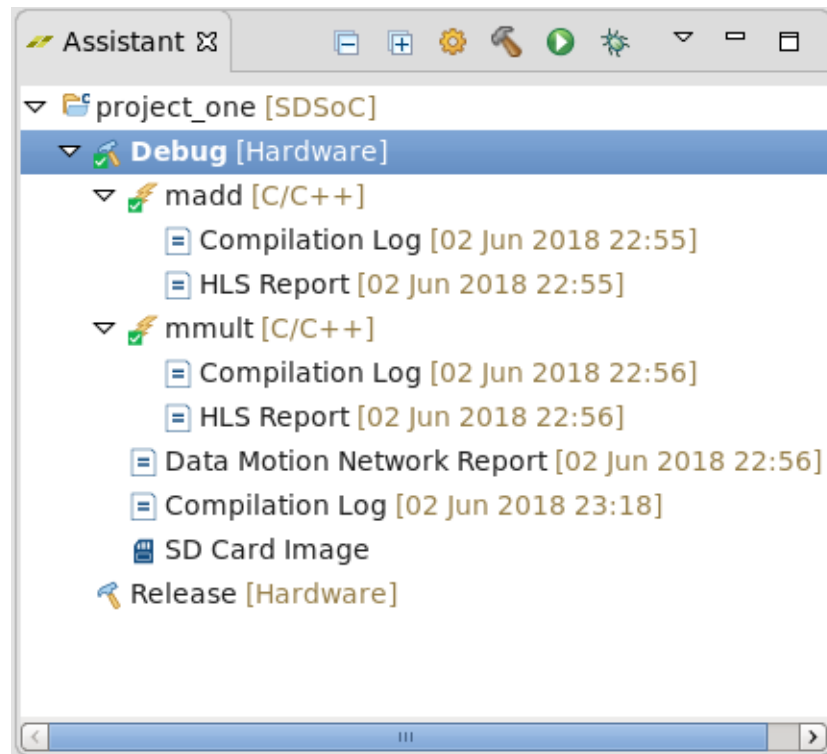
As a first step, programmers should optimize their overall program performance independently of the final target. The main difference between SDSoC and general purpose software is: in SDSoC projects, part of the core compute algorithms are pushed onto the FPGA. This implies that the developer must also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

Generally, you need to 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 generated automatically when you build the project, and listed in the Assistant view of the SDx™ IDE, as shown in the following figure. Double-click a listed report to open it.

Figure 32: Assistant View

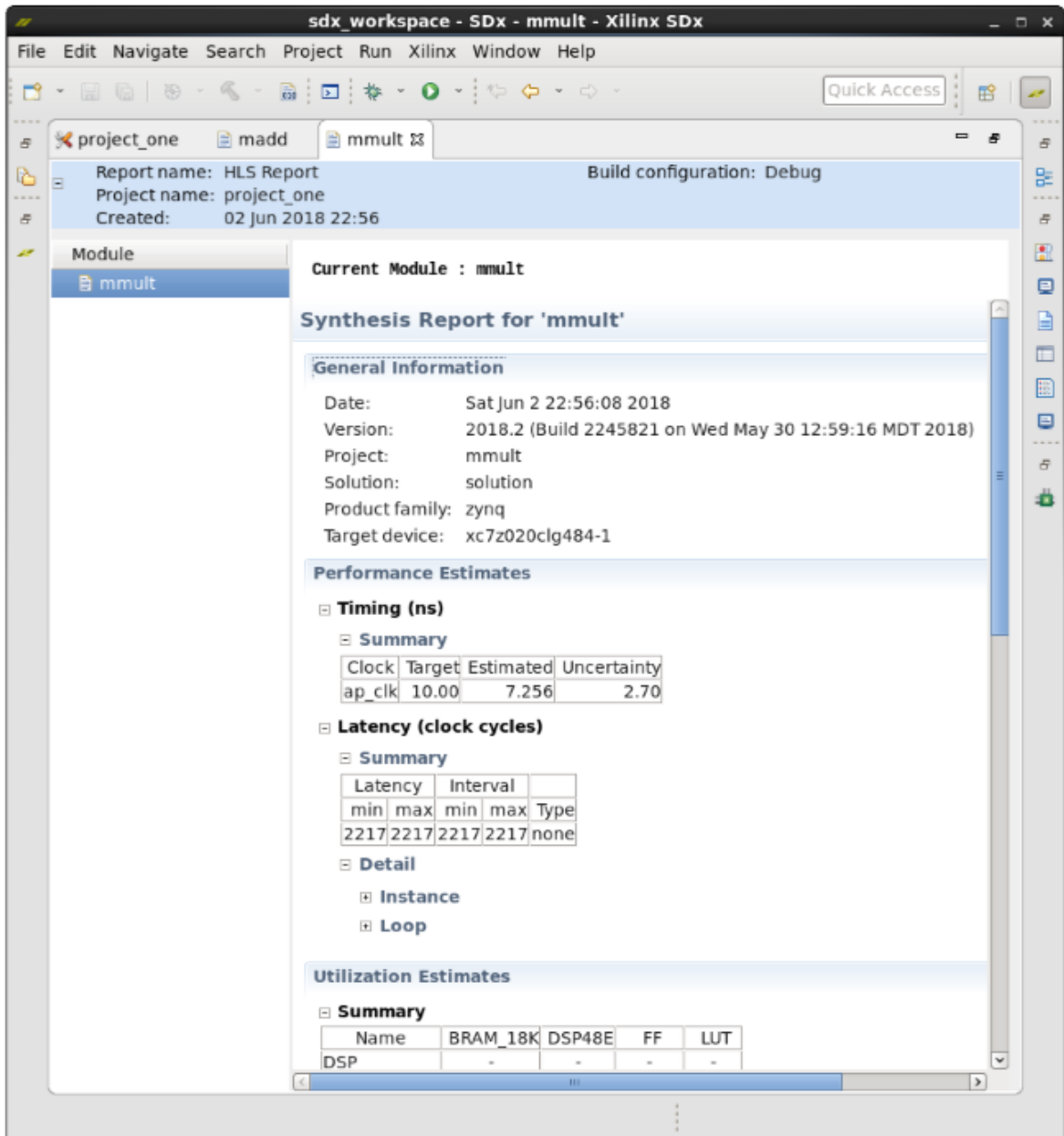


The following figures show the two main reports:

- High-Level Synthesis (HLS)
- Data Motion Network

To access these reports from the GUI, ensure the **Assistant** view is visible. This view is typically below the **Project Explorer** view. You can use the **Window → Show View → Assistant** menu command to display the Assistant view if it is not displayed.

Figure 33: HLS Report Window



The HLS Report provides details about the HLS process program that translates the C/C++ model into a hardware description language responsible for implementing the functionality on the FPGA. The details of this report enables you to see the impact of the design on the hardware implementation. You can then optimize the hardware function(s) based on the information.

Figure 34: Data Motion Network Report

Report name: Data Motion Network Report
 Project name: project_one
 Created: 02 Jun 2018 22:56
 Build configuration: Debug

Partition 0

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		mmult_1:C
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_1	A	A	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		madd_1:A

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time(CPU cycles)
madd_1	main.cpp:128:11	A	4096	paged		
		B	4096	contiguous	1112	7976
		C	4096	contiguous	1112	7976
mmult_1	main.cpp:127:11	A	4096	contiguous	1112	7976
		B	4096	contiguous	1112	7976
		C	4096	paged		

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 data mover is used for transport of each hardware function argument, and to which system port that data mover is attached. The **Pragmas** shows any SDS-based pragmas used for the hardware function.

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

- Accelerator instance name and Accelerator argument.
- Name of the port on the IP that pertains to the Accelerator argument (typically the same as the previous, except when bundling).
- Direction of the data motion transfer.
- Size, in bytes, of data to be transferred, to the degree in which the compiler can deduce that size. If the runtime determines the transfer size, this is zero.
- List of pragmas related to this argument.
- System Port and data mover `<system port>:<datamover>`, if applicable indicates which platform port and which data mover is used for transport of this argument.

- Accelerator(s) that 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 absence 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 Data Motion 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](#)).

Getting Started with Examples

All Xilinx[®] SDx[™] environments are provided with example 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.

Every platform provided within the SDx environment contains sample designs to get you started, and are accessible through the project creation flow as described in [Creating an Application Project](#). Furthermore, each of these designs, which are found in `<SDx_Install_Dir>/samples` provides a makefile so you can build, emulate, and run the code working entirely on the command line if you prefer.

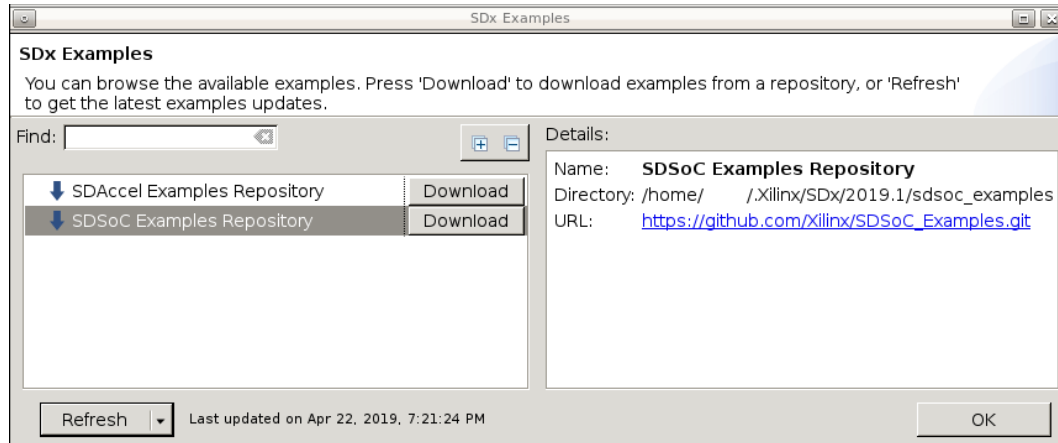
Many example designs and tutorials can be downloaded from the Xilinx [GitHub](#) repository. The [example design repository](#) contains the latest examples to get you started with application optimization targeting Xilinx PCIe[®] FPGA acceleration boards. All examples are ready to be compiled and executed on SDAccel[™] supported boards and accelerated cloud service partners.

In addition, the [tutorial repository](#) provides step-by-step instructions on a range of topics including building an application, emulation, along with advanced topics such as mixing C++ and RTL kernels, and optimizing host code.

Installing Examples

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, by selecting **Xilinx → SDx Examples**.

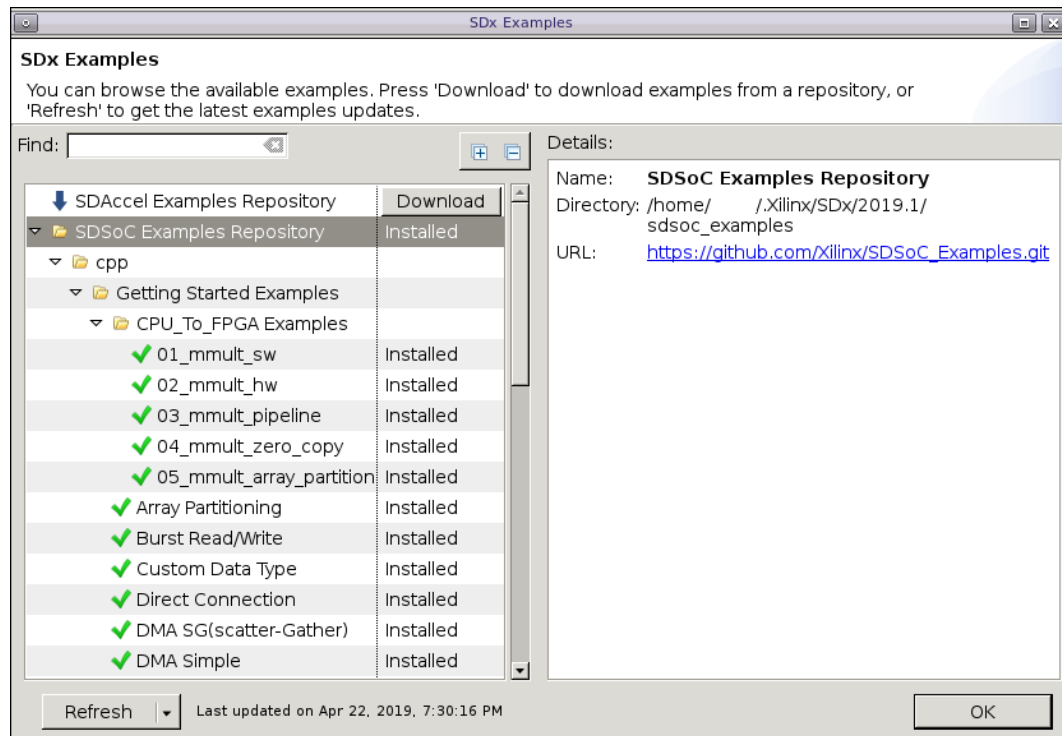
Figure 35: SDSoC Examples – Empty



The left side of the dialog box shows SDSoC™ Examples, and has a download command for each category. The right side of the dialog box shows the directory to where the examples downloaded and the URL from where the examples are downloaded. Customizing the location of the download directory is accomplished using the directions in the [Using Local Copies](#) section.

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

Figure 36: SDSoC 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.

Note: Corporate firewalls can restrict outbound connections. Specific proxy settings might be necessary.

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/SDx/<version>` folder:

- On Windows: `C:\Users\<<user_name>\.Xilinx\SDx\<<version>`
- On Linux: `~/.Xilinx/SDx/<version>`

The download directory cannot be changed from the SDx Examples dialog box. You might want to download the example files to a different location from the `.Xilinx` folder. To perform this, use the `git` command from a command shell to specify a new destination folder for the downloaded examples:

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

When you clone the examples using the `git` command as shown above, you can use the example files as a resource for 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. To make the files local, locate the files and manually make them local to your project.

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

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

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:

- [GitHub xfOpenCV](#)
- Arbitrary Precision Data Types
- HLS Stream
- HLS Math
- HLS Video
- HLS IP
- HLS Linear Algebra
- HLS DSP

You can use each of the C/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 (`<Vivado_Install_Dir>/include`).



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

Wrapping HLS Functions

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 system compilers from non-portable data types or unsupported language constructs.

The Synthesizable FIR Filter example demonstrates a standard idiom to use such a library function that 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.

See also: [Coding Guidelines](#) in the *SDSoC Environment Programmers Guide (UG1278)*.

Managing Platforms and Repositories

The SDx™ environment comes with built-in platforms. If you need to use a custom platform for your project, you must make that platform available for application implementation.


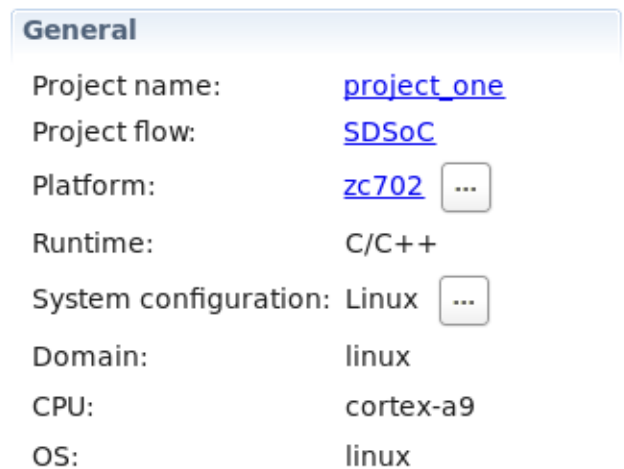
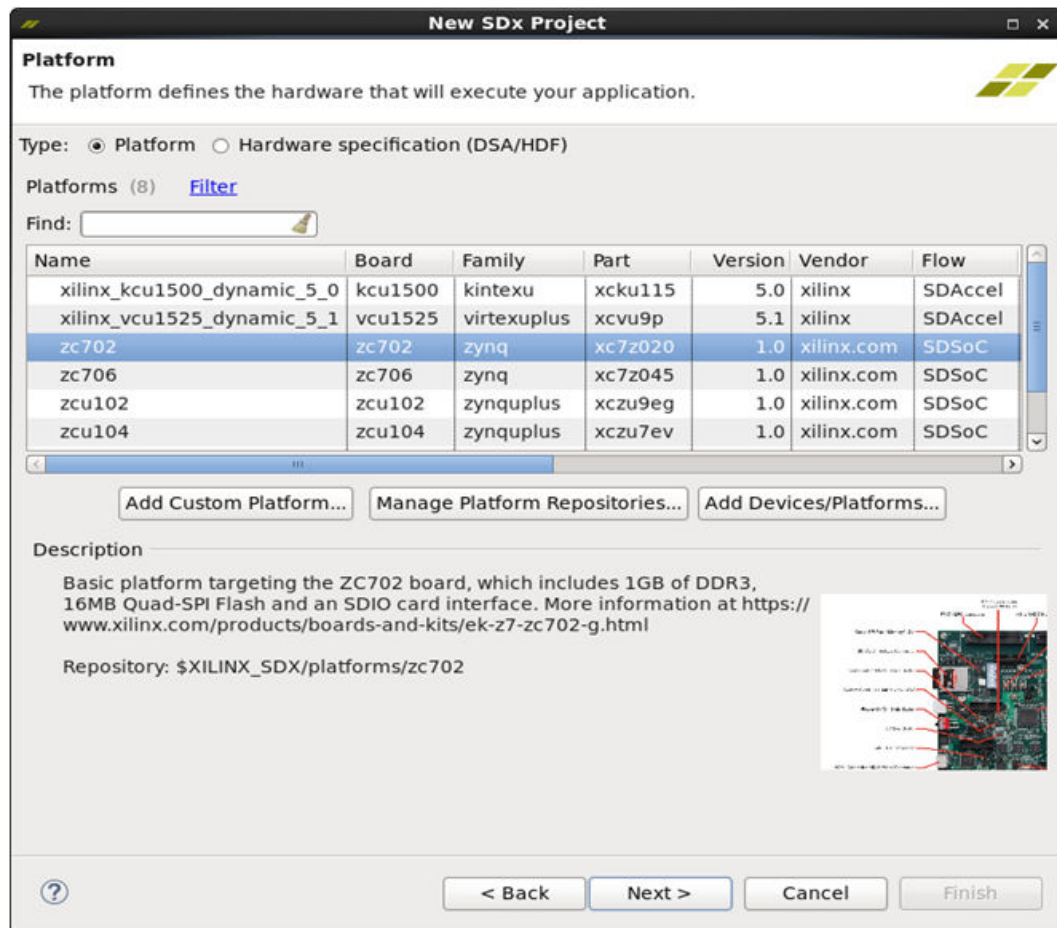
You can manage the platforms and repositories from an opened project by clicking the **Browse** () button next to the **Platform** link in the General panel of the Project Settings window. For developers using build scripts, the command line can be changed to reference the desired platform.

Figure 37: SDSoC Platform Browse



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

Figure 38: Platform Selection



- **Add Custom Platform:** 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. Select **Xilinx** → **Add Custom Platform** to directly add custom platforms to the tool.
- **Manage Repositories:** 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:** 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. Select **Help** → **Add Devices/Platforms** to directly add custom platforms to the tool.

Configuring SDSoC Settings through the GUI

The SDx™ GUI provides different views for you to manage SDSoC™ projects and builds, debug the design, view the design, and analyze the design.

The Assistant view in the SDx GUI displays the project and all of the build configurations that are part of the project. All of the settings for the project, and for building and debugging the project are accessible from the Assistant view, as described in the following sections.



TIP: Because the SDx IDE is based on Eclipse, many of the dialog boxes and settings are standard options available through the Eclipse environment. You can view the Eclipse help at: <http://help.eclipse.org>.

SDSoC Project Settings


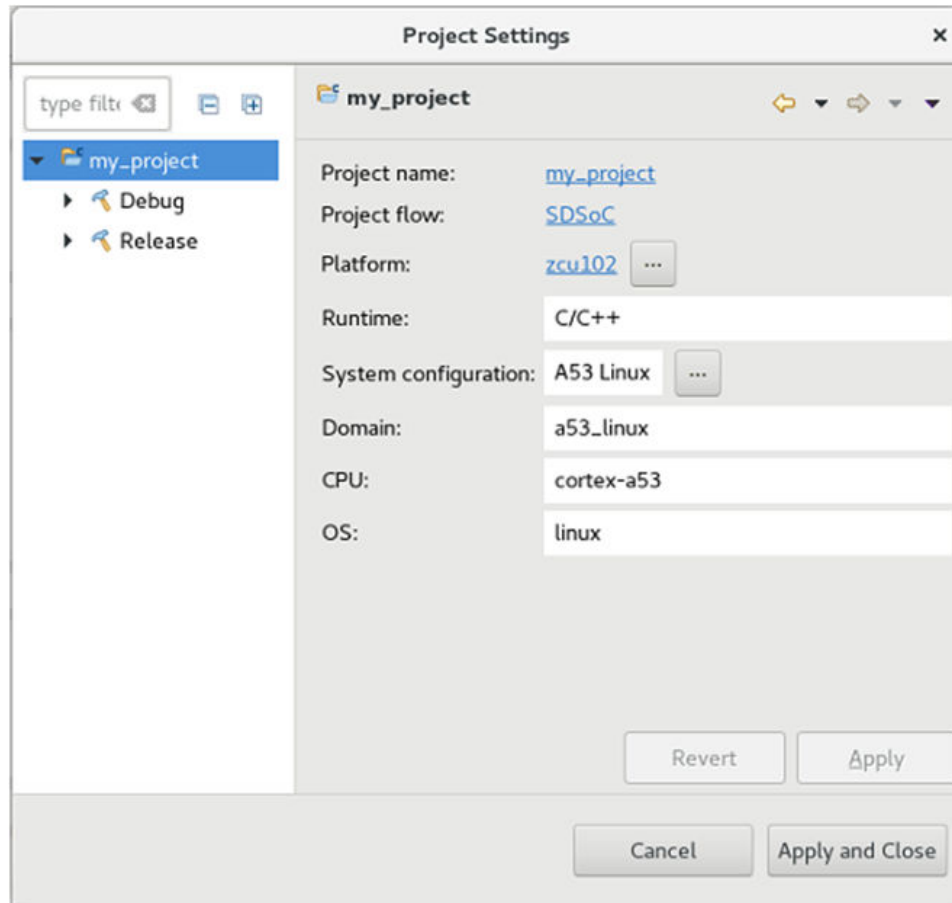
To edit the SDSoC project settings, select the project in the **Assistant** view and click the **Settings** icon () to bring up the Project Settings window.

Figure 39: SDSoC Project Settings Dialog Box



Project Settings provides quick access to the project settings through the **Project name:** link. The **Project flow:** link guides you to the www.xilinx.com web site for the **SDSoC** flow. You can also change the platform and the system configuration for the current project using the **browse** button.

SDSoC Build Configuration Settings

Build Configuration Settings


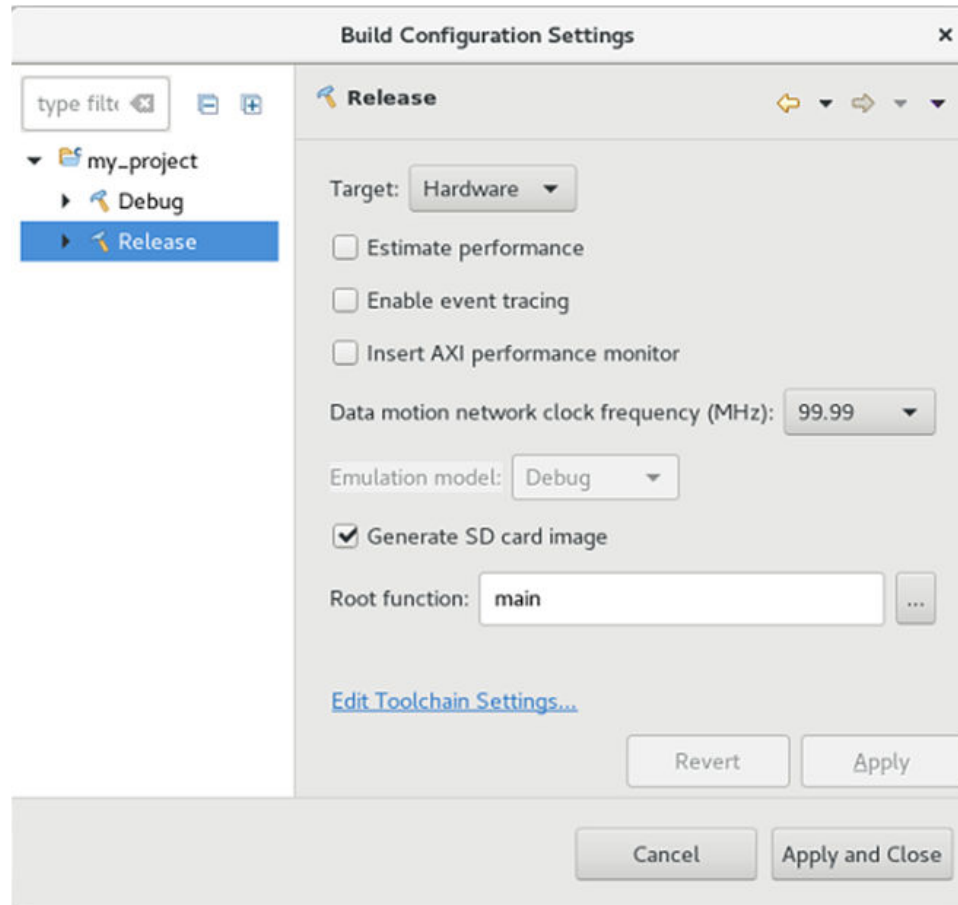
To edit the settings for any of the build configurations under the project, select a specific build configuration in the Assistant view and click the **Settings** icon () to bring up the window with the build configurations.

Figure 40: Build Configuration Settings



The Build Configuration Settings dialog provides a convenient way to make adjustments to your build configuration. You can change the build target as described in [Building the SDSoC Project](#).

You can enable analysis options like performance estimation, and event tracing as described in [Chapter 6: Profiling and Optimization](#), and specify the root function to exclude certain code from performance estimation. These options are only available when the build target is hardware. See [SDSoC Environment Getting Started Tutorial \(UG1028\)](#) for more information on using these options.



TIP: Hold the mouse over a setting to display an informative tooltip about what that setting does.

The **Data motion network clock frequency** drop-down shows available values for the clock frequency in between the platform and hardware accelerated functions. For more information, see [Selecting Clock Frequencies](#).

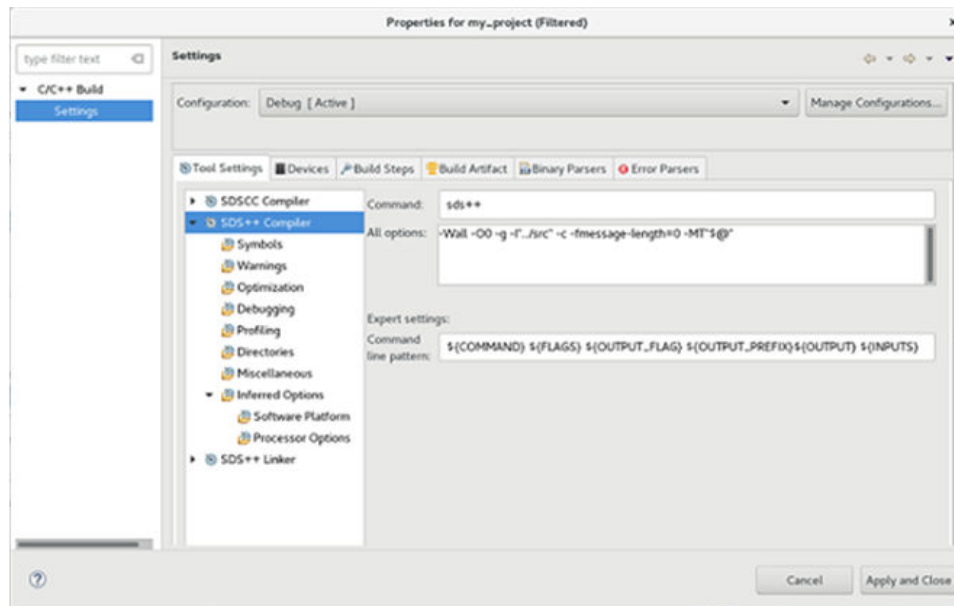
Use the **Generate SD card image** option to create the files required to allow booting your board from an SD Card. This option is only available when the build target is hardware.

Finally, the **Edit Toolchain Settings** link at the bottom of the dialog box opens the [SDS++/SDSCC Compiler Options](#) dialog box to let you set `sds++` system compiler and `sds++` linker options, including specifying directories, additional libraries, and command line options for the active configuration.

SDS++/SDSCC Compiler Options

The SDS++/SDSCC Compiler sections of the Tool Settings dialog box lets you set various options for the `sds++` and `sdscc` commands that are passed when the compiler is called. To access the Tool Settings dialog box, select the **Edit Toolchain Settings** link from the [Build Configurations](#) dialog box.

Figure 41: Compiler Command Options



The Settings dialog box lets you specify which build configuration you are specifying the settings for. Select the **Configuration** field at the top of the dialog box to specify any of the current build configurations, or select **All configurations** to change settings for all.



TIP: The SDS++ and SDSCC compilers are based on GCC, and therefore support many standard GCC options including some documented here. For more information refer to the [GCC Option Index](#).

Compiler Symbols Settings

Click **Symbols** under the SDS++ Compiler to define compile time macros that are passed with the `-D` option when calling the `sds++` command.


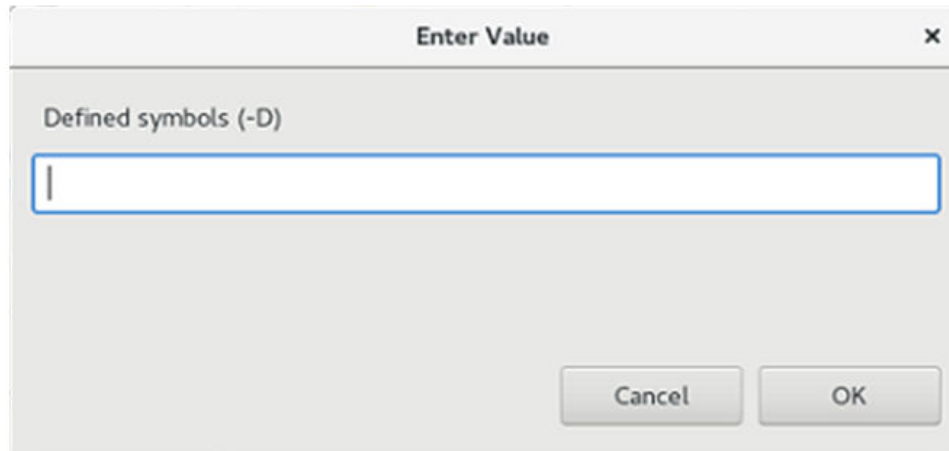
Under the Defined symbols (-D) field, you can add multiple symbols by clicking the add () icon to add each symbol.

Figure 42: Enter Value for Symbols

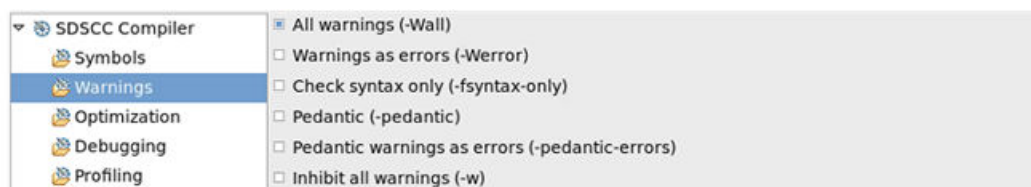


Under the Undefined symbols (-U) field, you can cancel any previous definition of a defined symbol.

Compiler Warnings Settings

Command options related to compiler warnings (-W) are provided through the Warnings section.

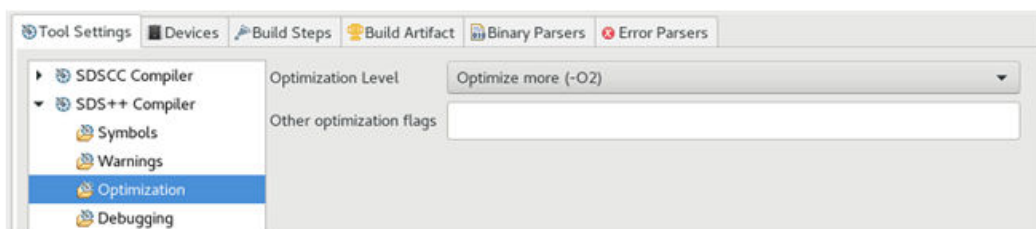
Figure 43: Compiler Warnings Settings



Compiler Optimization Settings

Compiler optimization flags (-O) and other optimization settings can be specified in this section.

Figure 44: Compiler Optimization Settings



Compiler Debugging Settings

Specify debug Level (`-g<level>`) and other debugging flags are specified through this section in the GUI.


Figure 45: Compiler Debugging Settings



Compiler Profiling Settings

Enable profiling (`-pg`) to generate extra code to write profile information for analysis of the application.

Compiler Directories Settings

Include Paths for the SDS++ Compiler are added under the Directories option. You can add directories to the Include Paths by clicking the add () icon to add each symbol.

Compiler Miscellaneous Settings

Any other flags that need to be passed to the SDS++ Compiler are added to the Miscellaneous section.

Note: These options can include any SDS++ Compiler options as described in *SDx Command and Utility Reference Guide (UG1279)*, as well as any GCC standard options not specifically addressed in other sections of this dialog box.

Figure 46: Compiler Miscellaneous Settings



Inferred Options

Software platform inferred flags and software platform include paths are added under the Inferred Options section.

Figure 47: Inferred Options

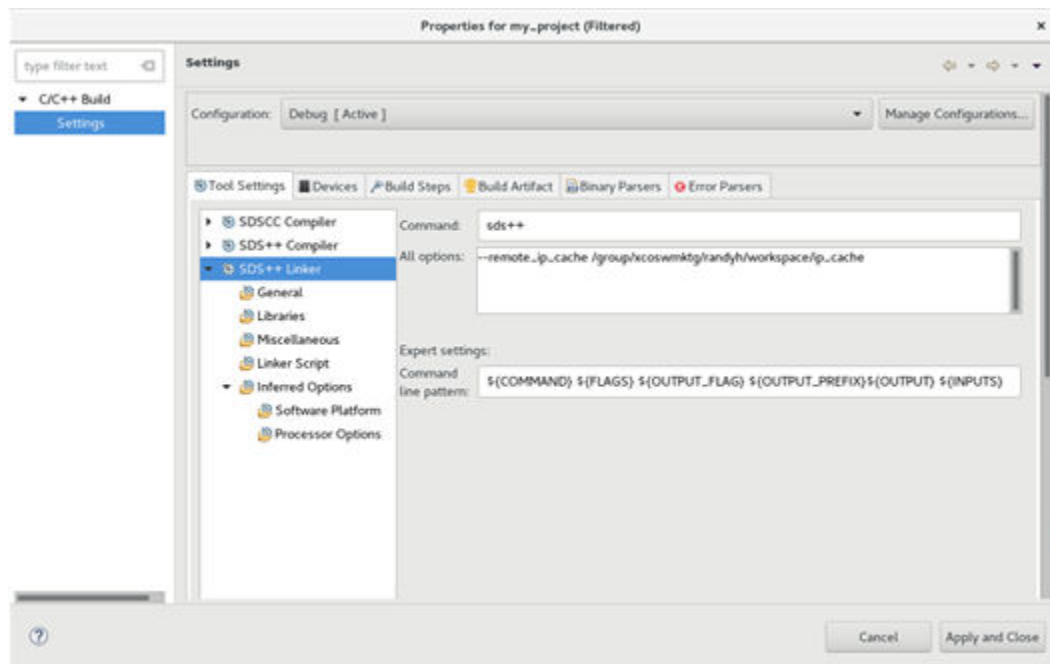


SDS++ Linker Settings

The SDS++ Linker sections of the Tool Settings dialog box lets you set various options for the `sds++` command that are passed when the linking stage of the build process is run. To access the Tool Settings dialog box, select the **Edit Toolchain Settings** link from the [Build Configurations](#) dialog box.

The SDS++ Linker page shows the `sds++` command and any additional options to be passed when calling `sds++` for the linking stage.

Figure 48: SDS++ Linker Options



The Settings dialog box lets you specify which build configuration you are specifying the settings for. Select the **Configuration** field at the top of the dialog box to specify any of the current build configurations, or select **All configurations** to change settings for all.

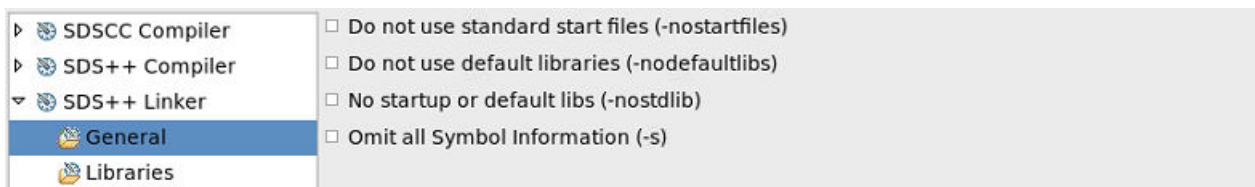


TIP: The SDS++ command is based on GCC, and therefore supports many standard GCC linking options including some documented here. For more information refer to the [GCC Option Index](#).

SDS++ Linker General Settings

Some general setting for the SDS++ linker are specified in this section.

Figure 49: SDS++ Linker General Settings



SDS++ Linker Libraries Settings


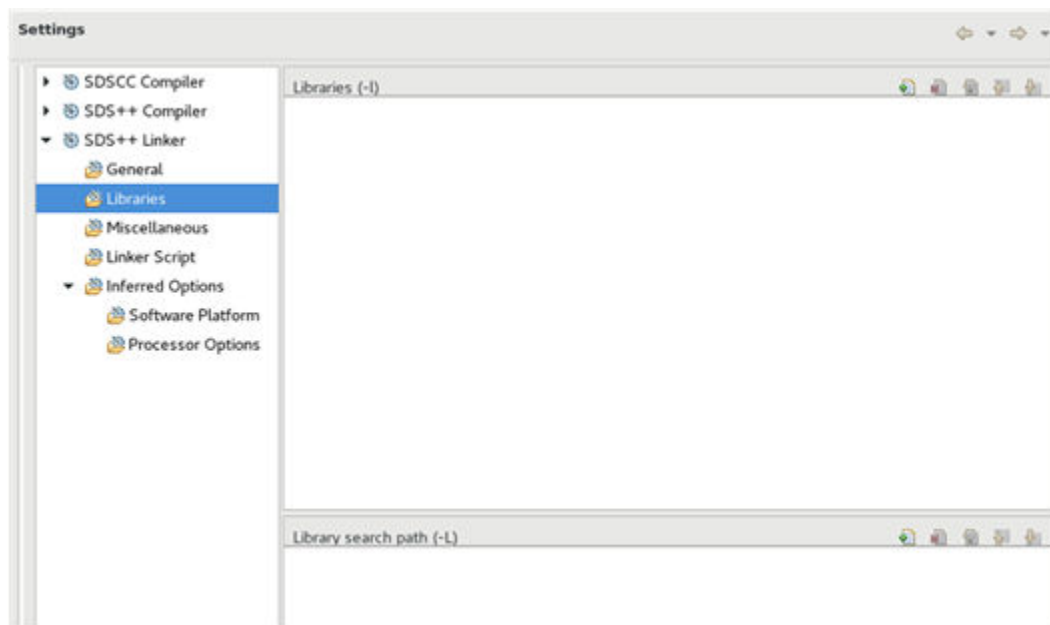
Specifies Libraries (-l) and the library search path (-L) for the SDS++ linker command. You can add libraries or library search paths by clicking the add () icon.

Figure 50: SDS++ Linker Libraries Settings



SDS++ Linker Miscellaneous Settings

Any other flags that needs to be passed to the SDS++ Linker can be provided through the Miscellaneous section.

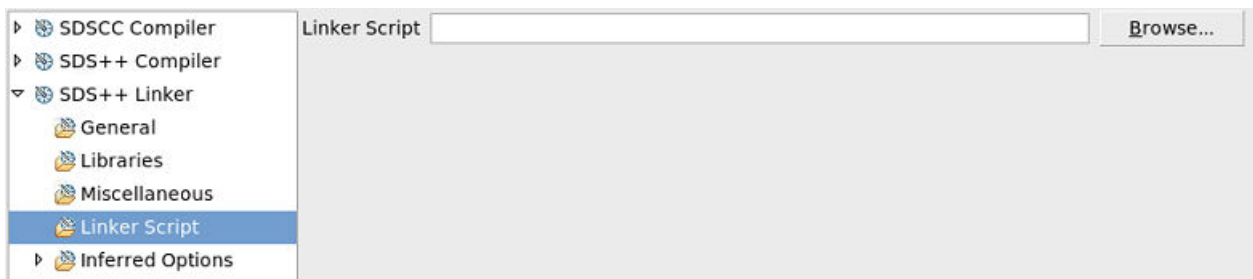
Figure 51: SDS++ Linker Miscellaneous Settings



SDS++ Linker Script Settings

The path and file name of the SDS++ Linker Script is provided in the Linker Script field.

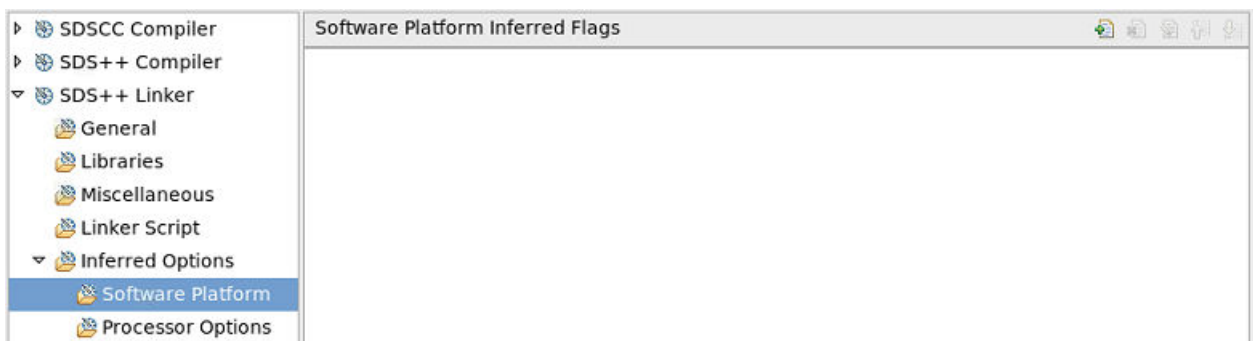
Figure 52: SDS++ Linker Script Settings



SDS++ Linker Inferred Options

Software platform inferred flags are added under the Inferred Options section.

Figure 53: SDS++ Linker Inferred Options



Compiling and Running Applications

This appendix contains the following topics:

- [Compiling and Running Applications on a MicroBlaze Processor](#)
- [Compiling and Running Applications on an Arm Processor](#)



RECOMMENDED: 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.

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.

Compiling and Running Applications on an Arm Processor



RECOMMENDED: 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 SoC.

- `arm-linux-gnueabi`: For developing Linux applications
- `arm-none-eabi`: For developing standalone (*bare-metal*) and FreeRTOS applications

For Arm Cortex-A53 CPUs within the Zynq devices, 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+™ MPSoC, the toolchain included in the SDSoC environment is the `armr5-none-eabi`. This develops 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++` referred to as `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 sub-directory 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`: 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 the following:
 - `uImage`
 - `devicetree.dtb`
 - `uramdisk.image.gz` files

- `<app>.elf`: 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`.

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoc™ and SDAccel™ development environments. To open it:

- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

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

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

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

References

These documents provide supplemental material useful with this guide:

1. *SDSoC Environments Release Notes, Installation, and Licensing Guide* ([UG1294](#))
2. *SDSoC Environment User Guide* ([UG1027](#))
3. *SDSoC Environment Getting Started Tutorial* ([UG1028](#))
4. *SDSoC Environment Tutorial: Platform Creation* ([UG1236](#))
5. *SDSoC Environment Platform Development Guide* ([UG1146](#))
6. *SDSoC Environment Profiling and Optimization Guide* ([UG1235](#))
7. *SDx Command and Utility Reference Guide* ([UG1279](#))
8. *SDSoC Environment Programmers Guide* ([UG1278](#))
9. *SDSoC Environment Debugging Guide* ([UG1282](#))
10. *SDx Pragma Reference Guide* ([UG1253](#))
11. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
12. *Zynq-7000 SoC Software Developers Guide* ([UG821](#))
13. *Zynq UltraScale+ MPSoC: Software Developers Guide* ([UG1137](#))
14. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide* ([UG850](#))
15. *ZCU102 Evaluation Board User Guide* ([UG1182](#))
16. *PetaLinux Tools Documentation: Reference Guide* ([UG1144](#))
17. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
18. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))
19. [SDSoC Development Environment web page](#)
20. [Vivado® Design Suite Documentation](#)

Training Resources

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

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright

© Copyright 2015-2019 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, ISE, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. HDMI, HDMI logo, and High-Definition Multimedia Interface are trademarks of HDMI Licensing LLC. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.