

# SDSoC Environment

## *User Guide*

UG1027 (v2015.2) July 20, 2015

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/20/2015	2015.2	First version of the document.

# Table of Contents

Revision History .....	2
Table of Contents .....	3
<b>Chapter 1: The SDSoC Environment .....</b>	<b>5</b>
Getting Started .....	5
Feature Overview .....	6
<b>Chapter 2: User Design Flows .....</b>	<b>7</b>
Creating a Project for a Target Platform .....	8
Compiling and Running Applications on an ARM Processor .....	10
Profiling and Instrumenting Code to Measure Performance.....	11
Moving Functions into Programmable Logic.....	12
SDSCC/SDS++ Performance Estimation Flow Options .....	14
<b>Chapter 3: SDSoC Environment Troubleshooting.....</b>	<b>15</b>
Troubleshooting Compile and Link Time Errors .....	15
Troubleshooting Runtime Errors .....	16
Troubleshooting Performance Issues .....	17
Debugging an Application .....	18
<b>Chapter 4: Improving System Performance .....</b>	<b>19</b>
Memory Allocation .....	20
Copy and Shared Memory Semantics.....	21
Data Cache Coherency .....	22
Increasing System Parallelism and Concurrency .....	22
<b>Chapter 5: Coding Guidelines.....</b>	<b>26</b>
Guidelines for Invoking SDSCC/SDS++ .....	26
Makefile Guidelines .....	27
General C/C++ Guidelines.....	27
Hardware Function Argument Types.....	28
Hardware Function Call Guidelines .....	29
<b>Chapter 6: A Programmer's Guide to Vivado High-Level Synthesis .....</b>	<b>30</b>
Top-Level Hardware Function Guidelines .....	30
Optimization Guidelines.....	31
Hardware Function Interface Details.....	40
<b>Chapter 7: Using C-Callable IP Libraries.....</b>	<b>45</b>
<b>Chapter 8: Using Vivado Design Suite HLS Libraries.....</b>	<b>46</b>
<b>Chapter 9: Exporting an Application as a Library .....</b>	<b>47</b>
Linking to an Application Library .....	49

<b>Chapter 10: Debugging an Application .....</b>	<b>51</b>
Debugging Linux Applications in the SDSoC IDE.....	51
Debugging Standalone Applications in the SDSoC IDE .....	51
Debugging FreeRTOS Applications .....	52
Peeking and Poking IP Registers.....	52
Debugging Performance Tips .....	52
<b>Chapter 11: Target Operating System Support .....</b>	<b>54</b>
Linux Applications.....	54
Standalone Target Applications .....	55
FreeRTOS Target Applications .....	56
<b>Chapter 12: Representative Example Designs .....</b>	<b>59</b>
File I/O Video Example .....	59
Synthesizeable FIR Filter.....	60
Matrix Multiplication .....	60
Using a C-callable RTL Library .....	60
<b>Chapter 13: SDSoC Pragma Specification.....</b>	<b>61</b>
Data Transfer Size .....	61
Memory Attributes .....	62
Data Mover Type .....	63
SDSoC Platform Interfaces to External Memory .....	65
Hardware Buffer Depth .....	65
Asynchronous Function Execution .....	66
Partition Specification .....	66
<b>Chapter 14: SDSoC Environment API.....</b>	<b>68</b>
<b>Chapter 15: SDSoC/SDS++ Compiler Commands and Options .....</b>	<b>70</b>
Name .....	70
Command Synopsis.....	70
General Options.....	71
Hardware Function Options.....	73
Compiler Macros.....	75
System Options.....	77
<b>Appendix A: Additional Resources and Legal Notices .....</b>	<b>80</b>
Xilinx Resources .....	80
Solution Centers .....	80
References.....	80
Please Read: Important Legal Notices.....	81

## The SDSoC Environment

The SDSoC™ (Software-Defined System On Chip) environment is an Eclipse-based Integrated Development Environment (IDE) for implementing heterogeneous embedded systems using the Zynq®-7000 All Programmable SoC platform. The SDSoC system compilers (`sdscc/sds++`) transform C/C++ programs into complete hardware/software systems based on command line options that specify target platform, and functions within the program to compile into programmable hardware.

To achieve high performance, each hardware function runs as an independent thread. The SDSoC system compilers generate hardware and software components that preserve program semantics and ensure synchronization between hardware and software threads, while enabling pipelined computation and communication. Application code can involve many hardware functions, multiple instances of a specific hardware function, and calls to a hardware function from different parts of the program. The system compiler analyzes a program to determine the data flow between software and hardware functions, and generates an application-specific system on chip to realize the program.

The SDSoC IDE supports standard software development workflows, including profiling, compilation, linking, and debugging, as well as a fast performance estimation. In addition, the SDSoC environment provides a fast performance estimation capability to enable "what if" exploration of the hardware/software interface before committing to a full hardware compile.

The SDSoC system compilers compile hardware functions with the Vivado® High-Level Synthesis (HLS) tool to programmable logic, and then generate a complete hardware system based on the selected platform, including DMAs, interconnects, hardware buffers, and other IPs. It then generates an FPGA bitstream by invoking the Vivado Design Suite tools. The SDSoC system compilers also generate system-specific software stubs and configuration data, which they compile and link with the application code using a standard GNU toolchain into an application binary.

To orchestrate data transfers and control the hardware accelerators, the system compilers automatically generate hardware-specific software configuration code, integrating into the program any associated drivers for generated IP blocks. By generating complete applications from "single source", the system compilers allow you to iterate over design and architecture changes by refactoring at the program level, dramatically reducing the time to working program running on the target.

---

## Getting Started

Download and install the SDSoC™ environment according to the directions provided in [SDSoC Environment User Guide: Getting Started \(UG1028\)](#). The Getting Started guide provides detailed instructions and hands-on tutorials to introduce the primary work flows for project creation, specifying functions to run in programmable logic, system compilation, debugging, and performance estimation. Working through these tutorials is the best way to get an overview of the SDSoC environment, and should be considered prerequisite to application development.

Note the following:

- When running the SDSoC system compilers from the command-line or through makefile flows, you must set the shell environment as described in [SDSoC Environment User Guide: Getting Started \(UG1028\)](#) or the tools will not function properly.
- The SDSoC environment includes the entire tools stack to create a bitstream, object code, and executables. If you have installed the Xilinx® Vivado® Design Suite and Software Development Kit tools independently, you should not attempt to combine these installations with the SDSoC environment.

---

## Feature Overview

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

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

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

The principle of design reuse underlies workflows you employ with the SDSoC environment, using well established platform-based design methodologies. The SDSoC system compiler generates an application-specific system on chip by extending a target platform. The SDSoC environment includes a number of platforms for application development and others are provided by Xilinx partners. [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#) describes how to capture platform metadata so that a pre-existing design built using the Vivado Design Suite, and corresponding software run-time environment can be used to build an SDSoC platform and used in the SDSoC environment.

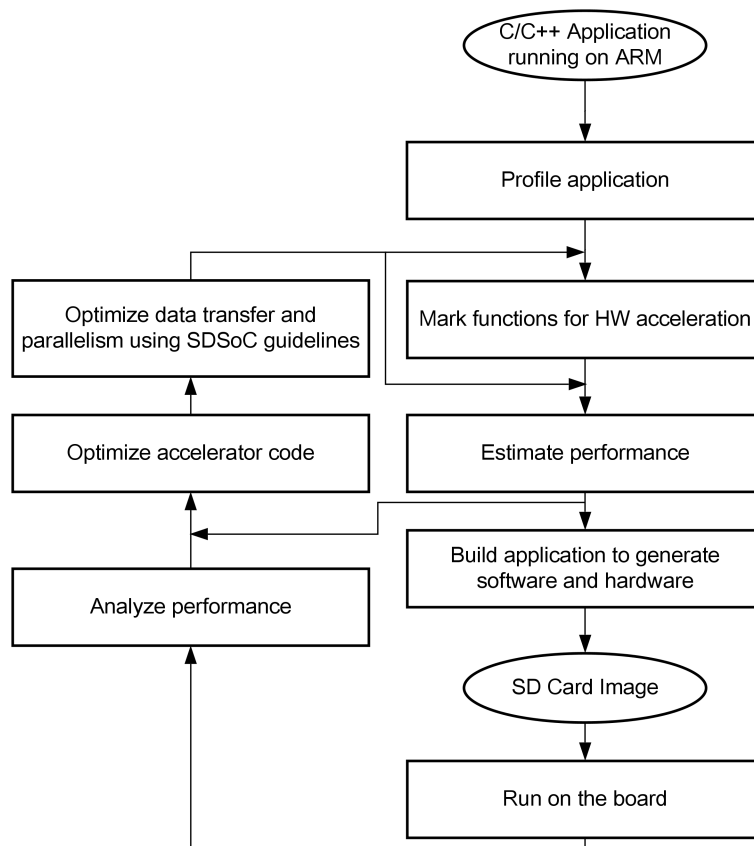
An SDSoC platform defines a base hardware and software architecture and application context, including processing system, external memory interfaces, custom input/output, and software run time including operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root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

## User Design Flows

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

The figure below shows a representative top-level user visible design flow that involves key components of the tool suite. For the purposes of exposition, the design flow proceeds linearly from one step to the next, but in practice you are free to choose other work flows with different entry and exit points. Starting with a software-only version of the application that has been cross-compiled for ARM CPUs, the primary goal is to identify portions of the program to move into programmable logic and to implement the application in hardware and software built upon a base platform.

**Figure 2–1: User Design Flow**



X14740-070215

The first step is to select a development platform, cross-compile the application, and ensure it runs properly on the platform. You then identify compute-intensive hot spots to migrate into programmable logic to improve system performance, and to isolate them into functions that can be compiled into hardware. You then invoke the SDSoC system compiler to generate a complete system-on-chip and SD card boot image for your application. You can instrument your code to analyze performance, and if necessary, optimize your system and hardware functions using a set of directives and tools within the SDSoC environment.

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

Because a complete system compile can be time-consuming compared with an "object code" compile for a CPU, the SDSoC environment provides a faster performance estimation capability that allows you to approximate the expected speed up over a software-only implementation for a given choice of hardware functions. This estimate is based on properties of the generated system and estimates for the hardware functions provided by the IPs when available.

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

It is assumed that you have already worked through the introductory tutorials (see [SDSoC Environment User Guide: Getting Started \(UG1028\)](#) ) and are familiar with project creation, hardware function selection, compilation, and running a generated application on the target platform. If you have not done so, it is recommended you do so before continuing.

---

## Creating a Project for a Target Platform

In the SDSoC IDE, click on **File > New > SDSoC Project** to create a new project and open up the **New Project** wizard. After entering the project name, the first step is to select a platform target for development from the **Platform** pull-down menu. The platform includes a base hardware system, software runtime (including operating system), boot loaders, and root file system. For an SDSoC environment project, the platform is fixed and the command line options are automatically inserted into every makefile. To retarget a project to a new platform, you must create a new project with the new platform and copy the source files from your current project into the new project.



If you are writing makefiles outside of the SDSoC IDE, you must include the `-sds-pf` command line option on every call to `sdscc`.

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

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

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

In addition to the available base platforms, you can find additional sample platforms in the `<sdsoc_root>/samples/platforms` directory. To create a new project for one of these platforms within the SDSoC IDE, create a new project, select **Other** for the platform and navigate to the desired sample platform.

## Data Motion Network Clock

Every platform supports one or more clock sources, one of which is selected by default if you do not make an explicit choice. This default clock is defined by the platform provider, and is used for the data motion network generated by `sdscc` during system generation. You can view the platform clocks by selecting the **Platform** tab at the bottom of the **SDSoC Project Overview** window. You can select a different platform clock frequency with the **Data Motion Clock Frequency** pull-down menu in the **SDSoC Project Overview Options** window, or on the command line with the `-dmclockid` option.

```
sdscc -sds-pf zc702 -dmclockid 1
```

To see the available clocks for a platform from the command line, execute the following:

```
$ sdscc -sds-pf-info zc702
Platform Description
=====
Basic platform targeting the ZC702 board, which includes 1GB of DDR3, 16MB Quad-
SPI Flash and an SDIO card interface. More information at http://www.xilinx.com/
products/boards-and-kits/EK-Z7-ZC702-G.htm
Platform Information
=====
Name: zc702
Device
-----
Architecture: zynq
Device: xc7z020
Package: clg484
Speed grade: -1
System Clocks
-----
Clock ID Frequency
-----|-----
666.666687
0 166.666672
1 142.857132
2 100.000000
3 200.000000
```

## Compiling and Running Applications on an ARM Processor

A first step in application development is to cross-compile your application code to run on the target platform. Every platform included in the SDSoC environment includes a pre-built SD card image from which you can boot and run cross-compiled application code. When you do not select any functions for hardware in your project, this pre-built image is used.

When you make code changes, including changes to hardware functions, it is valuable to rerun a software-only compile to verify 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/software debugging.

Like the Xilinx SDK upon which it is built, the SDSoC environment includes two distinct toolchains for the ARM CPUs within Zynq® architecture devices.

1. `arm-xilinx-linux-gnueabi` - for developing Linux applications
2. `arm-xilinx-gnueabi` - for developing standalone ("bare-metal") and FreeRTOS applications

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

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

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

- `README.TXT` - contains brief instructions on how to run the application
- `BOOT.BIN` - the boot image contains first stage boot loader (FSBL), boot program (u-boot), and the FPGA bitstream
- `uImage`, `devicetree.dtb`, `uramdisk.image.gz` - Linux boot image
- `<app>.elf` - the application binary executable

To run the application, copy the contents of `sd_card` directory onto an SD card and insert into the target board. Open a serial terminal connection to the target and power up the board (for more information see [SDSoC Environment User Guide: Getting Started \(UG1028\)](#) for more information). 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`.

For standalone applications, the ELF, bitstream, and board support package (BSP) are contained within `BOOT.BIN`, which automatically runs the application after the system boots.

## Profiling and Instrumenting Code to Measure Performance

The first major task in creating a software-defined SoC is to identify portions of application code that are suitable for implementation in hardware, and that significantly improve overall performance when run in hardware. Program hot-spots that are compute-intensive are good candidates for hardware acceleration, especially when it is possible to stream data between hardware and the CPU and memory to overlap the computation with the communication. Software profiling is a standard way to identify the most CPU-intensive portions of your program.

The SDSoC environment includes all performance and profiling capabilities that are included in the Xilinx SDK, including `gprof`, the non-intrusive Target Communication Framework (TCF) Profiler, and the Performance Analysis perspective within Eclipse.

To run the TCF Profiler for a standalone application, run the following steps:

1. Set the **Active Build Configuration** to **SDDebug**.
2. In the **SDSoC Project Overview** window, click on **Debug application**. Note: the board must be connected to your computer and powered on. The application automatically breaks at the entry to `main()`.
3. Launch the TCF Profiler by selecting **Window > Show View > Other > Debug > TCF Profiler**.
4. Start the TCF Profiler by clicking on the green **Start** button at the top of the **TCF Profiler** tab. Enable **Aggregate per function** in the **Profiler Configuration** dialog box.
5. Start the profiling by clicking on the **Resume** button. The program runs to completion and breaks at the `exit()` function.
6. View the results in the **TCF Profiler** tab.

Profiling provides a statistical method for finding hot spots based on sampling the CPU program counter and correlating to the program in execution. Another way to measure program performance is to instrument the application to determine the actual duration between different parts of a program in execution.

The `sds_lib` library included in the SDSoC environment provides a simple, source code annotation based time-stamping API that can be used to measure application performance.

```
/*
 * @return value of free-running 64-bit Zynq(TM) global counter
 */
unsigned long long sds_clock_counter(void);
```

By using this API to collect timestamps and differences between them, you can determine duration of key parts of your program. For example, you can measure data transfer or overall round trip execution time for hardware functions as shown in the following code snippet:

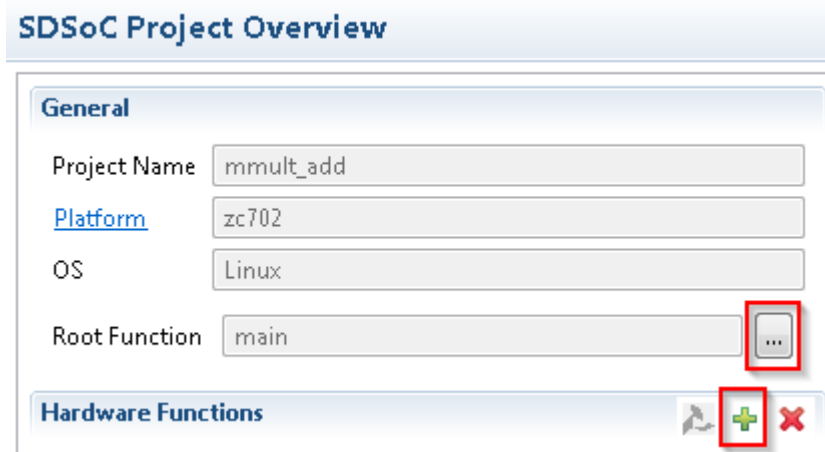
```
#include "sds_lib.h"
unsigned long long total_run_time = 0;
unsigned int num_calls = 0;
unsigned long long count_val = 0;
#define sds_clk_start() { \
    count_val = sds_clock_counter(); \
    num_calls++; \
}
#define sds_clk_stop() { \
    long long tmp = sds_clock_counter(); \
    total_run_time += (tmp - count_val); \
}
#define avg_cpu_cycles() (total_run_time / num_calls)
#define NUM_TESTS 1024
extern void f();
void measure_f_runtime()
{
    for (int i = 0; i < NUM_TESTS; i++) {
        sds_clock_start();
        f();
        sds_clock_stop();
    }
    printf("Average cpu cycles f(): %ld\n", avg_cpu_cycles());
}
```


The performance estimation feature within the SDSoC environment employs this API by automatically instrumenting functions selected for hardware implementation, measuring actual run-times by running the application on the target, and then comparing actual times with estimated times for the hardware functions.

**NOTE:** While off-loading CPU-intensive functions is probably the most reliable heuristic to partition your application, it is not guaranteed to improve system performance without algorithmic modification to optimize memory accesses. A CPU almost always has much faster random access to external memory than you can achieve from programmable logic, due to multi-level caching and a faster clock speed (typically 2x to 8x faster than programmable logic). Extensive manipulation of pointer variables over a large address range, for example, a sort routine that sorts indices over a large index set, while very well-suited for a CPU, may become a liability when moving a function into programmable logic. This does not mean that such compute functions are not good candidates for hardware, only that code or algorithm restructuring may be required. This issue is also well-known for DSP and GPU coprocessors.

## Moving Functions into Programmable Logic

When you have created a new project, you can open up the **SDSoC Project Overview** by double-clicking on the `project.sdsoc` file in the **Project Explorer**.



Click on the  symbol in the **Hardware Functions** panel to display the list of candidate functions within your program. This list consists of functions in the call graph rooted at the Root Function listed in the **General** panel, by default `main`, but changeable by clicking on the `...` button and selecting an alternative function root.

From within the popup window, you can select one or more functions for hardware acceleration and click **OK**. The selected functions appear in the list box. Note that the Eclipse CDT indexing mechanism is not foolproof, and you might need to close and reopen the selection popup to view available functions. If a function does not appear in the list, you can navigate to its containing file in the **Project Explorer**, expand the contents, right-click on the function prototype, and select **Toggle HW/SW**.

From the command line, select a function `foo` in the file `foo_src.c` for hardware with the following `sdscc` command line option.

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

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

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

Although the data motion network runs off of a single clock, it is possible to run hardware functions at different clock rates to achieve higher performance. In the **Hardware Functions** panel, select functions from the list and use the **Clock Frequency** pull-down menu to choose their clocks. Be aware that it might not be possible to implement the hardware system with some clock selections.

To set a clock on the command-line, determine the corresponding clock id using `sdscc -sds-pf-info <platform>` and use the `-clockid` option.

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

When moving a function optimized for CPU execution into programmable logic, you usually need to revise the code to achieve best performance. See [A Programmer's Guide to Vivado HLS](#) and [Coding Guidelines](#) for programming guidelines.

## SDSCC/SDS++ Performance Estimation Flow Options

A full bitstream compile can take much more time than a software compile, so `sdscc` provides performance estimation options to compute the estimated run-time improvement for a set of hardware function calls. In the SDSoC IDE Project Overview window, invoke the estimator by clicking on **Estimate speedup**, which switches the project to the SDEstimate build configuration.

Estimating the speed-up is a two phase process. First, the SDSoC IDE compiles the hardware functions and generates the system. Instead of synthesizing the system to bitstream, `sdscc` computes an estimate of the performance based on estimated latencies for the hardware functions and data transfer time estimates for the callers of hardware functions. In the generated Performance report, select **Click Here** to run an instrumented version of the software on the target to determine a performance baseline and the performance estimate (see [SDSoC Environment User Guide: Getting Started \(UG1028\)](#) for more information).

You can also generate a performance estimate from the command line. As a first pass to gather data about software runtime, you use the `-perf-funcs` option to specify functions to profile and `-perf-root` to specify the root function encompassing calls to the profiled functions. The `sdscc` compiler then automatically instruments these functions to collect run-time data when the application is run on a board. When you run an "instrumented" application on the target, the program creates a file on the SD card called `sw_perf_data.xml`, which contains the run-time performance data for the run.

Copy `sw_perf_data.xml` to the host and run a build that estimates the performance gain on a per hardware function caller basis and for the top-level function specified by the `-perf-root` function in the first pass run. Use the `-perf-est` option to specify `sw_perf_data.xml` as input data for this build.

The following table specifies the `sdscc` options normally used to build an application.

Option	Description
<code>-perf-funcs function_name_list</code>	Specify a comma separated list of all functions to be profiled in the instrumented software application.
<code>-perf-root function_name</code>	Specify the root function encompassing all calls to the profiled functions. The default is the function <code>main</code> .
<code>-perf-est data_file</code>	Specify the file contain runtime data generated by the instrumented software application when run on the target. Estimate performance gains for hardware accelerated functions. The default name for this file is <code>sw_perf_data.xml</code> .
<code>-perf-est-hw-only</code>	Run the estimation flow without running the first pass to collect software run data. Using this option provides hardware latency and resource estimates without providing a comparison against baseline.



**CAUTION!** After running the `sd_card` image on the board for collecting profile data, run `cd /; sync; umount /mnt;`. This ensures that the `sw_perf_data.xml` file is written out to the SD card.

A complete example of the makefile-based flow for performance estimation can be found in `<sdsoc_root>/samples/mmult_performance_estimation`.

# SDSoC Environment Troubleshooting

There are three common types of issues you might encounter using the SDSoC™ environment flow.

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

---

## Troubleshooting Compile and Link Time Errors

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

Some tips for dealing with SDSoC environment specific errors follow.

- Tool errors reported by tools in the SDSoC environment chain.
  - Check whether the corresponding code adheres to [Coding Guidelines](#).
  - Check the syntax of pragmas.
  - Check for typos in pragmas that might prevent them from being applied to the correct function.
- Vivado Design Suite High-Level Synthesis (HLS) cannot meet timing requirement.
  - Select a slower clock frequency for the accelerator in the SDSoC IDE (or with the `sdsccl/sds++` command line parameter).
  - Modify the code structure to allow HLS to generate a faster implementation. See [A Programmer's Guide to High-Level Synthesis](#) for more information on how to do this.
- Vivado tools cannot meet timing.
  - In the SDSoC IDE, select a slower clock frequency for the data motion network or accelerator, or both (from the command line, use `sdsccl/sds++` command line parameters).
  - Synthesize the HLS block to a higher clock frequency so that the synthesis/implementation tools have a bigger margin.
  - Modify the C/C++ code passed to HLS, or add more HLS directives to make the HLS block go faster.
  - Reduce the size of the design in case the resource usage (see the Vivado tools report in `_sds/ipi/*.log` and other log files in the subdirectories there) exceeds 80% or so. See the next item for ways to reduce the design size.
- Design too large to fit.
  - Reduce the number of accelerated functions.
  - Change the coding style for an accelerator function to produce a more compact accelerator. You can reduce the amount of parallelism using the mechanisms described in [A Programmer's Guide to High-Level Synthesis](#).
  - Modify pragmas and coding styles (pipelining) that cause multiple instances of accelerators to be created.
  - Use pragmas to select smaller data movers such as AXIFIFO instead of AXIDMA\_SG.
  - Rewrite hardware functions to have fewer input and output parameters/arguments, especially in cases where the inputs/outputs are continuous stream (sequential access array argument) types that prevent sharing of data mover hardware.

---

## Troubleshooting Runtime Errors

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



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

Consider the following code fragment that results in streaming input/output from a hardware function.

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

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

Notice that the loop reads the `in_a` stream 19 times but the size of `in_a[]` is 20, so the caller of `f1` would wait forever (or hang) if it waited for `f1` to consume all the data that was streamed to it. Similarly, the caller would wait forever if it waited for `f1` to send 20 int values because `f1` sends only 19. Program errors that lead to such “hangs” can be detected by instrumenting the code to flag streaming access errors such as non-sequential access or incorrect access counts within a function and running in software. Streaming access issues are typically flagged as `improper streaming access` warnings in the log file, and it is left to the user to determine if these are actual errors.

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

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

## Troubleshooting Performance Issues

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

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

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

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

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

---

## Debugging an Application

The SDSoC™ environment allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE (user-defined makefiles) and debugged either on the command line or using the SDSoC IDE.

See [SDSoC Environment User Guide: Getting Started \(UG1028\)](#), [Tutorial: Debugging Your System](#) for information on using the interactive debuggers in the SDSoC IDE.

# Improving System Performance

There are many factors that affect overall system performance. A well-designed system generally balances computation and communication so that all hardware components remain occupied doing meaningful work. Some applications will be compute-bound, and for these, you should concentrate on maximizing throughput and minimizing latency in hardware accelerators. Others may be memory-bound, in which case you might need to restructure algorithms to increase temporal and spatial locality in the hardware, for example, by adding copy-loops or memcpy to pull blocks of data into hardware rather than making random array accesses to external memory.

This section describes underlying principles and inference rules within the SDSoC system compiler to assist the programmer in controlling the compiler to improve overall system performance through

- Improved access to external memory from programmable logic
- Increased concurrency and parallelism in programmable logic

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

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

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

Every transfer between the software program and a hardware function requires a data mover, which consists of a hardware component that moves the data, and an operating system-specific library function. The following table lists supported data movers and various properties for each.

**Figure 4–1: SDSoC Data Movers Table**

SDSoC Data Mover	Vivado IP Data Mover	Accelerator IP Port Types	Transfer Size	Contiguous Memory Only
axi_lite	processing_system7	register, axilite		
axi_dma_simple	axi_dma	bram, ap_fifo, axis	< 8 MB	✓
axi_dma_sg	axi_dma	bram, ap_fifo, axis		
axi_dma_2d	axi_dma	bram		✓
axi_fifo	axi_fifo_mm_s	bram, ap_fifo, axis	(≤ 300 B)	
zero_copy	accelerator IP	aximm master		✓

X14762-070315

Scalar variables are always transferred over an AXI4-Lite bus interface with the `axi_lite` data mover. For array arguments, the data mover inference is based on transfer size, hardware function port mapping, and function call site information. The `axi_dma_simple` data mover is the most efficient bulk transfer engine, but only supports up to 8MB transfers, so for larger transfers, the `axi_dma_sg` (scatter-gather DMA) data mover is required. The `axi_fifo` data mover does not require as many hardware resources as the DMA, but due to its slower transfer rates, is preferred only for payloads of up to 300 bytes.

You can override the data mover selection by inserting a pragma into program source immediately before the function declaration, for example,

```
#pragma SDS data data_mover(A:AXIDMA_SIMPLE)
```

Note that `#pragma SDS` is always treated as a rule, not a hint, so you must ensure that their use conforms with the data mover requirements in [SDSoC Data Movers Table](#).

## Memory Allocation

The `sdscc/sds++` compilers analyze your program and selects data movers to match the requirements for each hardware function call between software and hardware, based on payload size, hardware interface on the accelerator, and properties of the function arguments. When the compiler can guarantee an array argument is located in physically contiguous memory, it can use the most efficient data movers. Allocating or memory-mapping arrays with the following `sds_lib` library functions can inform the compiler that memory is physically contiguous.

```
sds_alloc(size_t size); // guarantees physically contiguous memory
sds_mmap(void *paddr, size_t size, void *vaddr); // paddr must point to contiguous memory
sds_register_dmabuf(void *vaddr, int fd); // assumes physically contiguous memory
```

It is possible that due to the program structure, the `sdscc` compiler cannot definitively deduce the memory contiguity, and when this occurs, it issues a warning message, as shown:

```
WARNING: [SDSoC 0-0] Unable to determine the memory attributes passed to foo_arg_A of function
foo at foo.cpp:102
```

You can inform the compiler that the data is allocated in a physically contiguous memory by inserting the following pragma immediately before the function declaration (note: the pragma does not guarantee physically contiguous allocation of memory; your code must use `sds_alloc` to allocate such memory).

```
#pragma SDS data mem_attribute (A:PHYSICAL_CONTIGUOUS) // default is NON_PHYSICAL_CONTIGUOUS
```

## Copy and Shared Memory Semantics

By default, hardware function calls involve copy-in, copy-out semantics for function arguments. It is possible to impose a shared memory model for hardware function arguments, but you must keep in mind that while throughput on burst transfers is quite good, the latency to external DDR from the programmable logic is significantly higher than it is for the CPU. The following pragma, inserted immediately preceding the function declaration, is used to declare that a variable transfer employs shared memory semantics.

```
#pragma SDS data zero_copy(A[0:<array_size>]) // array_size = number of elements
```

Within a synthesizable hardware function, it is usually inefficient to read/write single words from the shared memory (specified using the zero-copy pragma). A more efficient approach is to employ `memcpy` to read/write data from memory in bursts and store it in a local memory.

For copy and zero copy memory semantics, another efficient alternative is to stream data between programmable logic and external DDR to maximize memory efficiency, storing data in local memory within a hardware function whenever you need to make non-sequential and repeated accesses to variables. For example, video applications typically have data coming in as pixel streams and implement line buffers in FPGA memory to support multiple accesses to the pixel stream data.

To declare to `sdscc` that a hardware function can admit streaming access for an array data transfer (that is, each element is accessed precisely once in index order), you can insert the following pragma immediately preceding the function prototype.

```
#pragma SDS data access_pattern(A:SEQUENTIAL) // access pattern = SEQUENTIAL | RANDOM
```

For arrays passed as pointer typed arguments to hardware functions, sometimes the compilers can infer transfer size, but if they cannot, they issue `ERROR: [SDSoC 0:0] The bound callers of accelerator foo have different/indeterminate data size for port p` and you must use the following to specify the size of the data to be transferred:

```
#pragma SDS data copy(p[0:<array_size>]) // for example, int *p.
```

You can vary the data transfer size on a per function call basis to avoid transferring data that is not required by a hardware function by setting `<array_size>` in the pragma definition to be an expression defined in the scope of the function call (that is, all variables in the size expression must be scalar arguments to the function), for example:

```
#pragma SDS data copy(A[0:L+2*T/3]) // scalar arguments L, T to same function
```

---

## Data Cache Coherency

The `sdscc/sds++` compilers automatically generate software configuration code for each data mover required by the system, including interfacing to underlying device drivers as needed. The default assumption is that the system compiler maintains cache coherency for the memory allocated to arrays passed between the CPU and hardware functions. Consequently, the compiler might generate code to perform a cache flush before transferring data to a hardware function and to perform a cache-invalidate before transferring data from a hardware function to the memory. Both actions are necessary for correctness, but have performance implications. When using Zynq® device HP ports, for example, you can override the default when you know that the CPU will not access the memory indicating that the correctness of the application does not depend on cache coherency. To avoid the overhead of unnecessary cache flushes use the following pragma inserted immediately before the function declaration.

```
#pragma SDS data mem_attribute(A:NON_CACHEABLE) // default is CACHEABLE
```

Declaring an array as non-cacheable means the compiler does not need to ensure the cache coherency when accessing the specified array in the memory, but it is your responsibility to do so when necessary. A typical use case is a video application where some frame buffers are accessed by programmable logic but not the CPU.

---

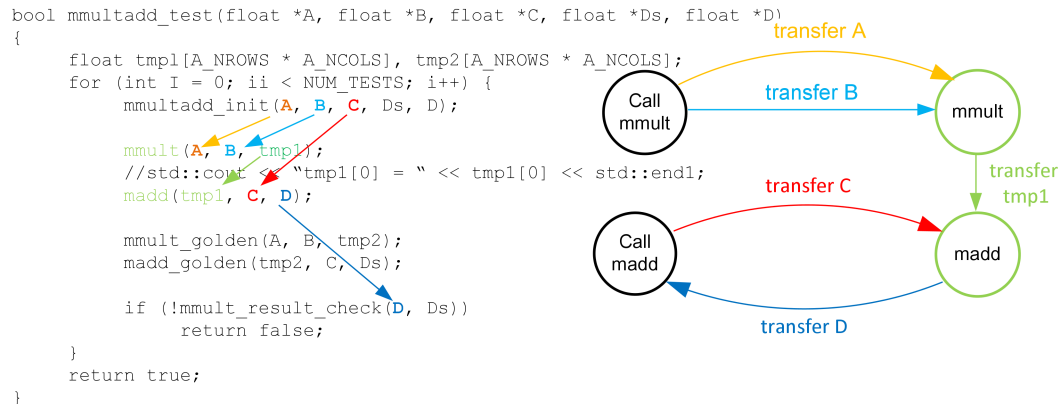
## Increasing System Parallelism and Concurrency

Increasing the level of concurrent execution is a standard way to increase overall system performance, and increasing the level of parallel execution is a standard way to increase concurrency. Programmable logic is well-suited to implement architectures with application-specific accelerators that run concurrently, especially communicating through flow-controlled streams that synchronize between data producers and consumers.

In the SDSoC environment, you influence the macro-architecture parallelism at the function and data mover level, and the micro-architecture parallelism within hardware accelerators. By understanding how the `sdscc` system compiler infers system connectivity and data movers, you can structure application code and apply pragmas as needed to control hardware connectivity between accelerators and software, data mover selection, number of accelerator instances for a given hardware function, and task level software control. You can control the micro-architecture parallelism, concurrency, and throughput for hardware functions within Vivado HLS or within the IPs you incorporate as C-callable/linkable libraries. [A Programmer's Guide to Vivado High-Level Synthesis](#) provides to guidelines and methodologies for creating efficient hardware function micro-architectures that can be employed within the SDSoC environment.

At the system level, the `sdscc` compiler chains together hardware functions when the data flow between them does not require transferring arguments out of programmable logic and back to system memory. For example, consider the code in the following figure, where `mmult` and `madd` functions have been selected for hardware.

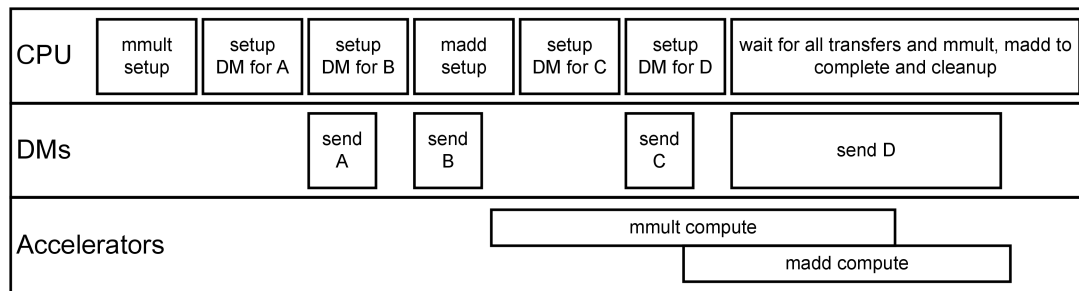
**Figure 4–2: Hardware /Software Connectivity with Direct Connection**



Because the intermediate array variable `tmp1` is used only to pass data between the two hardware functions, the `sdscc` system compiler chains the two functions together in hardware with a direct connection between them.

It is instructive to consider a time line for the calls to hardware as shown in the following figure.

**Figure 4–3: Timeline for mmult/madd Function Calls**

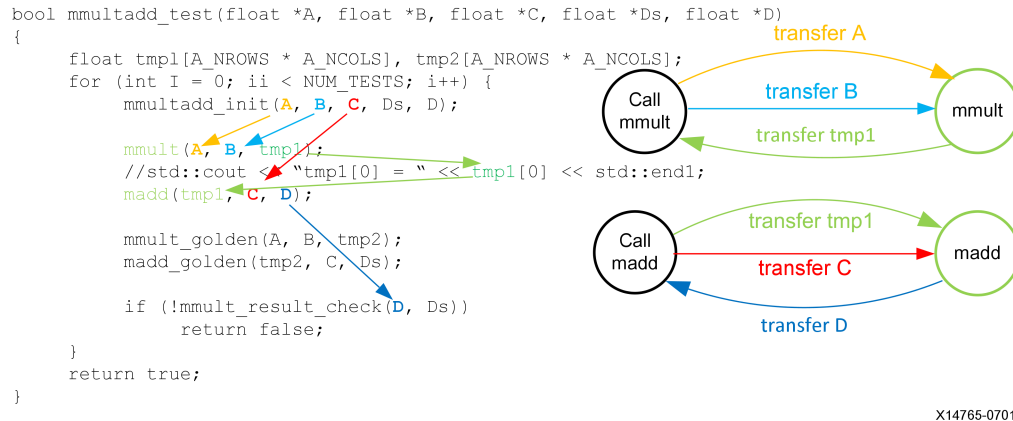


The program preserves the original program semantics, but instead of the standard ARM procedure calling sequence, each hardware function call is broken into multiple phases involving setup, execution, and cleanup, both for the data movers (DM) and the accelerators. The CPU in turn sets up each hardware function (that is, the underlying IP control interface) and the data transfers for the function call with non-blocking APIs, and then waits for all calls and transfers to complete. In the example shown in the diagram, the `mmult` and `madd` functions run concurrently whenever their inputs become available. The ensemble of function calls is orchestrated in the compiled program by control code automatically generated by `sdscc` according to the program, data mover, and accelerator structure.



In general, it is impossible for the `sdscc` compiler to determine side-effects of function calls in your application code (for example, `sdscc` may have no access to source code for functions within linked libraries), so any intermediate access of a variable occurring lexically between hardware function calls requires the compiler to transfer data back to memory. So for example, an injudicious simple change to uncomment the debug print statement (in the "wrong place") as shown in the figure below, can result in a significantly different data transfer graph and consequently, an entirely different generated system and application performance.

**Figure 4–4: Hardware/Software Connectivity with Broken Direct Connection**



A program can invoke a single hardware function from multiple call sites. In this case, the `sdscc` compiler behaves as follows. If any of the function calls results in "direct connection" data flow, then `sdscc` creates an instance of the hardware function that services every similar direct connection, and an instance of the hardware function that services the remaining calls between memory ("software") and programmable logic.

Structuring your application code with "direct connection" data flow between hardware functions is one of the best ways to achieve high performance in programmable logic. You can create deep pipelines of accelerators connected with data streams, increasing the opportunity for concurrent execution.

There is another way in which you can increase parallelism and concurrency using the `sdscc` compiler. You can direct the compiler to create multiple instances of a hardware function by inserting the following pragma immediately preceding a call to the function.

```
#pragma SDS async(<id>) // <id> a non-negative integer
```

This pragma creates a hardware instance that is referenced by `<id>`. The generated control code for the hardware function call returns to the caller as soon as all of the setup has completed without waiting for the function execution to complete. The program must correctly synchronize with the function call by inserting a matching `wait` pragma for the same `<id>` at an appropriate point in the program.

```
#pragma SDS wait(<id>) // <id> synchronizes to hardware function with <id>
```

A simple code snippet that creates two instances of a hardware function `mmult` is as follows.

```
{
    #pragma SDS async(1)
    mmult(A, B, C); // instance 1
    #pragma SDS async(2)
    mmult(D, E, F); // instance 2
    #pragma SDS wait(1)
    #pragma SDS wait(2)
}
```



The `async` mechanism gives the programmer ability to handle the "hardware threads" explicitly to achieve very high levels of parallelism and concurrency, but like any explicit multi-threaded programming model, requires careful attention to synchronization details to avoid non-deterministic behavior or deadlocks.

# Coding Guidelines

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

---

## Guidelines for Invoking SDSCC/SDS++

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

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

All other source files can safely be compiled with the ARM GNU toolchain.

A large software project may include many files and libraries that are unrelated to the hardware accelerator and data motion networks generated by `sdsc`. If the `sdsc` compiler issues errors on source files unrelated to the generated hardware system (for example, from an OpenCV library), you can compile these files through `GCC` instead of `sdsc` by right-clicking on the file (or folder) **Properties** > **C/C++ Build** > **Settings** and setting the **Command** to `GCC`.

## Makefile Guidelines

The makefiles provided with the designs in `<sdsoc_root>/samples` consolidate all `sdscc` hardware function options into a single command line. This is not required, but has the benefit of preserving the overall control structure and dependencies within a makefile without requiring change to the makefile actions for files containing a hardware function.

- You can define make variables to capture the entire SDSoC environment command line, for example: `CC = sds++ ${SDSFLAGS}` for C++ files, invoking `sdscc` for C files. In this way, all SDSoC environment options are consolidated in the `CC` variable. Define the platform and target OS once in this variable.
- There must be a separate `-sds-hw/-sds-end` clause in the command line for each file that contains a hardware function. For example:

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

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

## General C/C++ Guidelines

- Hardware functions can execute concurrently under the control of a master thread. A program can have multiple threads and processes, but must have only a single master thread that controls hardware functions.
- A top-level hardware function must be a global function, not a class method, and it cannot be an overloaded function.
- There is no support for exception handling in hardware functions.
- It is an error to refer to a global variable within a hardware function or any of its sub-functions when this global variable is also referenced by other functions running in software.
- If a hardware function returns a value, then the return type must be a scalar type that fits in a 32-bit container.
- A hardware function must have at least one argument.
- An output or inout scalar argument to a hardware function should be assigned once. Create a local variable when multiple assignments to an output or inout scalar are required within a hardware function.
- Use predefined macros to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters `'_'`. For examples, see [SDSSC/SDS++ Compiler and Linker Options](#).
  - The `__SDSCC__` macro is defined and passed as a `-D` option to sub-tools whenever `sdscc` or `sds++` is used to compile source files, and can be used to guard code dependent on whether it is compiled by `sdscc/sds++` or by another compiler, for example a GNU host compiler.
  - When `sdscc` or `sds++` compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is defined and passed as a `-D` option, and can be used to guard code dependent on whether high-level synthesis is run or not.

## Hardware Function Argument Types

The SDSoC™ environment `sdscc/sds++` system compilers support hardware function arguments with types that resolve to a C99 basic arithmetic type, a pointer to, array of, or a `struct` whose members flatten to a C99 basic arithmetic type (hierarchical structs are supported). Scalar arguments must fit in a 32-bit container. The SDSoC™ environment automatically infers hardware interface types for each hardware function argument based on the argument type and the following pragmas:

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

To avoid interface incompatibilities, you should only incorporate Vivado® HLS interface type directives and pragmas in your source code as described in [Vivado HLS Function Argument Types](#) when `sdscc` fails to generate a suitable hardware interface directive.

- Vivado® HLS provides arbitrary precision types `ap_fixed<int>`, `ap_int<int>`, and an `hls::stream` class. In the SDSoC environment, arguments to top-level hardware functions must have width of 8, 16, 32, or 64 bits, and you must guard such declarations with `#ifndef __SDS_VHLS__` to coerce to a like-sized C99 type such as `char`, `short`, `int`, or `long long`. Vivado HLS `hls::stream` arguments must be presented to `sdscc/sds++` as arrays. The example `<sdsoc_install_dir>/samples/hls_if/hls_stream` demonstrates how to use HLS `hls::stream` typed arguments in the SDSoC environment.
- By default, an array argument to a hardware function is transferred by copying the data, that is, it is equivalent to using `#pragma SDS data copy`. As a consequence, an array argument must be either used as an input or produced as an output, but not both. For an array that is both read and written by the hardware function, you must use `#pragma SDS data zero_copy` to tell the compiler that the array should be kept in the shared memory and not copied.
- To ensure alignment across the hardware/software interface, do not use hardware function arguments that have type `long`, or an array of `bool` or `struct`.



**IMPORTANT:** *Pointer arguments for a hardware function require special consideration. Although pointers are common and powerful abstractions, they can lead to challenges for the `sdscc` and `vivado_hls` tools, due to the way they are synthesized by the latter.*



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

- `#pragma SDS data zero_copy` which provides pointer semantics using shared memory
- 

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

*which maps the argument onto a stream, and requires that array elements are accessed in index order. The data copy pragma is only required when the `sdscc` system compiler is unable to determine the data transfer size and issues an error. When you require non-sequential access to the array in the hardware function, you should change the pointer argument to an array with an explicit declaration of its dimensions, for example, `A[1024]`.*

## Hardware Function Call Guidelines

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

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

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

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

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

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

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

# A Programmer's Guide to Vivado High-Level Synthesis

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

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

---

## Top-Level Hardware Function Guidelines

This section describes coding guidelines to ensure that a Vivado HLS hardware function has a consistent interface with object code generated by the ARM GNU toolchain.

### Use Standard C99 Data Types for Top-Level Hardware Function Arguments

1. Avoid using the `long` data type. The `long` data type is not portable between 64-bit architecture (such as x64) or 32-bit architecture (such as the ARM A9 in Zynq® devices).
2. Avoid using arrays of `bool`. An array of `bool` has different memory layout between ARM GCC and Vivado® HLS.
3. Avoid using arrays of `struct`. An array of `struct` has different memory layout between ARM GCC and Vivado HLS. In the future, the SDSoC environment will support array of `struct` with a compatible memory layout between ARM GCC and Vivado HLS.
4. Avoid using `ap_int<>`, `ap_fixed<>`, `hls::stream`, except with data width of 8, 16, 32 or 64 bits. Navigate to <SDSoC Installation Path>/samples/hls\_if/hls\_stream for a sample design on how to use `hls::stream` in the SDSoC environment.

## Omit HLS Interface Directives for Top-Level Hardware Function Arguments

A top-level hardware function should not contain any `HLS interface` pragmas. In this case, the SDSoC environment generates appropriate HLS interface directives. There are two SDSoC environment pragmas you can specify for a top-level hardware function to guide the SDSoC environment to generate the desired HLS interface directives.

`#pragma SDS data zero_copy()` can be used to generate a shared memory interface implemented as an AXI master interface in hardware.

`#pragma SDS data access_pattern(argument:SEQUENTIAL)` can be used to generate a streaming interface implemented as a FIFO interface in hardware.




---

**IMPORTANT:** *If you specify the interface using `#pragma HLS interface` for a top-level function argument, the SDSoC environment does not generate a HLS interface directive for that argument, and it is your responsibility to ensure that the generated hardware interface is consistent with all other function argument hardware interfaces. Because a function with incompatible HLS interface types can result in cryptic `sdscc` error messages, it is strongly recommended (though not absolutely mandatory) that you omit `HLS interface` pragmas.*

---

## Optimization Guidelines

This section documents several fundamental HLS optimization techniques to enhance hardware function performance. These techniques are: function inlining, loop and function pipelining, loop unrolling, increasing local memory bandwidth and streaming data flow between loops and functions.

### Function Inlining

Similar to function inlining of software functions, it can be beneficial to inline hardware functions.

Function inlining replaces a function call by substituting a copy of the function body after resolving the actual and formal arguments. After that, the inlined function is dissolved and no longer appears as a separate level of hierarchy. Function inlining allows operations within the inlined function be optimized more effectively with surrounding operations, thus improves the overall latency or the initiation interval for a loop.

To inline a function, put `#pragma HLS inline` at the beginning of the body of the desired function. The following code snippet directs Vivado HLS to inline the `mmult_kernel` function:

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                  float in_B[A_NCOLS][B_NCOLS],
                  float out_C[A_NROWS][B_NCOLS])
{
    #pragma HLS INLINE
    int index_a, index_b, index_d;
    // rest of code body omitted
}
```

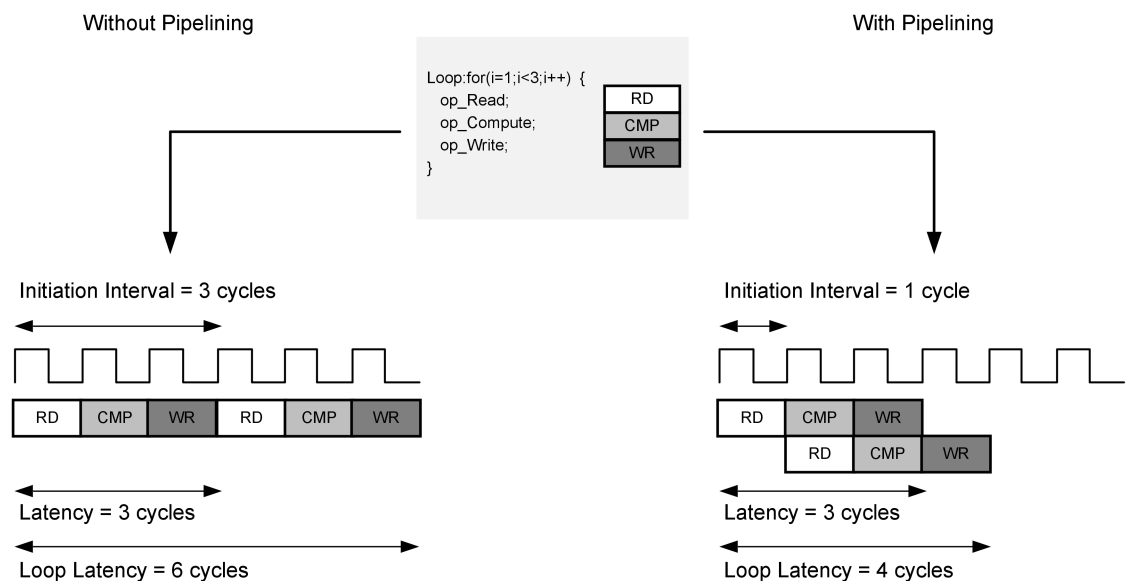
## Loop Pipelining and Loop Unrolling

Both loop pipelining and loop unrolling improve the hardware function's performance by exploiting the parallelism between loop iterations. The basic concepts of loop pipelining and loop unrolling and example codes to apply these techniques are shown and the limiting factors to achieve optimal performance using these techniques are discussed.

### Loop Pipelining

In sequential languages such as C/C++, the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in the following figure.

**Figure 6–1: Loop Pipelining**



X14770-070115

As shown in the above figure, without pipelining, there are three clock cycles between the two RD operations and it requires six clock cycles for the entire loop to finish. However, with pipelining, there is only one clock cycle between the two RD operations and it requires four clock cycles for the entire loop to finish, that is, the next iteration of the loop can start before the current iteration is finished.

An important term for loop pipelining is called **Initiation Interval (II)**, which is the number of clock cycles between the start times of consecutive loop iterations. In **Loop Pipelining** the Initiation Interval (II) is one, because there is only one clock cycle between the start times of consecutive loop iterations.



To pipeline a loop, put `#pragma HLS pipeline` at the beginning of the loop body, as illustrated in the following code snippet. Vivado HLS tries to pipeline the loop with minimum **Initiation Interval**.

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
    for (index_b = 0; index_b < B_NCOLS; index_b++) {
        #pragma HLS PIPELINE II=1
        float result = 0;
        for (index_d = 0; index_d < A_NCOLS; index_d++) {
            float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
            result += product_term;
        }
        out_C[index_a * B_NCOLS + index_b] = result;
    }
}
```

## Loop Unrolling

Loop unrolling is another technique to exploit parallelism between loop iterations. It creates multiple copies of the loop body and adjust the loop iteration counter accordingly. The following code snippet shows a normal rolled loop:

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

After the loop is unrolled by a factor of 2, the loop becomes:

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

So unrolling a loop by a factor of N basically creates N copies of the loop body, the loop variable referenced by each copy is updated accordingly ( such as the `a[i+1]` in the above code snippet ), and the loop iteration counter is also updated accordingly ( such as the `i+=2` in the above code snippet ).

Loop unrolling creates more operations in each loop iteration, so that Vivado HLS can exploit more parallelism among these operations. More parallelism means more throughput and higher system performance. If the factor N is less than the total number of loop iterations (10 in the example above), it is called a "partial unroll". If the factor N is the same as the number of loop iterations, it is called a "full unroll". Obviously, "full unroll" requires the loop bounds be known at compile time but exposes the most parallelism.

To unroll a loop, simply put `#pragma HLS unroll [factor=N]` at the beginning of the desired loop. Without the optional `factor=N`, the loop will be fully unrolled.

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    #pragma HLS unroll factor=2
    sum += a[i];
}
```

## Factors Limiting the Parallelism Achieved by Loop Pipelining and Loop Unrolling

Both loop pipelining and loop unrolling exploit the parallelism between loop iterations. However, parallelism between loop iterations is limited by two main factors: one is the data dependencies between loop iterations, the other is the number of available hardware resources.

A data dependence from an operation in one iteration to another operation in a subsequent iteration is called a loop-carried dependence. It implies that the operation in the subsequent iteration cannot start until the operation in the current iteration has finished computing the data input for the operation in subsequent iteration. Loop-carried dependencies fundamentally limit the initiation interval that can be achieved using loop pipelining and the parallelism that can be exploited using loop unrolling.

The following example demonstrates loop-carried dependencies among operations producing and consuming variables a and b.

```
while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

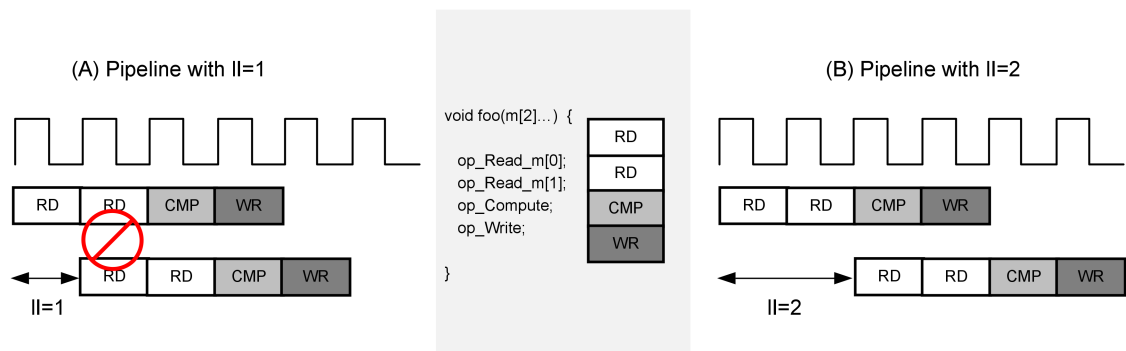
Obviously, operations in the next iteration of this loop can not start until the current iteration has calculated and updated the values of a and b. Array accesses are a common source of loop-carried dependencies, as shown in the following example:

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

In this case, the next iteration of the loop must wait until the current iteration updates the content of the array. In case of loop pipelining, the minimum Initiation Interval is the total number of clock cycles required for the memory read, the add operation, and the memory write.

Another performance limiting factor for loop pipelining and loop unrolling is the number of available hardware resources. The following figure shows an example the issues created by resource limitations, which in this case prevents the loop to be pipelined with an initiation interval of 1.

**Figure 6–2: Resource Contention**



In this example, if the loop is pipelined with an initiation interval of one, there are two read operations. If the memory has only a single port, then the two read operations cannot be executed simultaneously and must be executed in two cycles. So the minimal initiation interval can only be two, as shown in part (B) of the figure. The same can happen with other hardware resources. For example, if the `op_compute` is implemented with a DSP core which cannot accept new inputs every cycle, and there is only one such DSP core. Then `op_compute` cannot be issued to the DSP core each cycle, and an initiation interval of one is not possible.

## Increasing Local Memory Bandwidth

This section shows several ways provided by Vivado HLS to increase local memory bandwidth, which can be used together with loop pipelining and loop unrolling to improve system performance.

Arrays are intuitive and useful constructs in C/C++ programs. They allow the algorithm to be easily captured and understood. In Vivado HLS, each array is by default implemented with a single port memory resource. However, such memory implementation may not be the most ideal memory architecture for performance oriented programs. At the end of previous section, an example of resource contention caused by limited memory ports is shown.

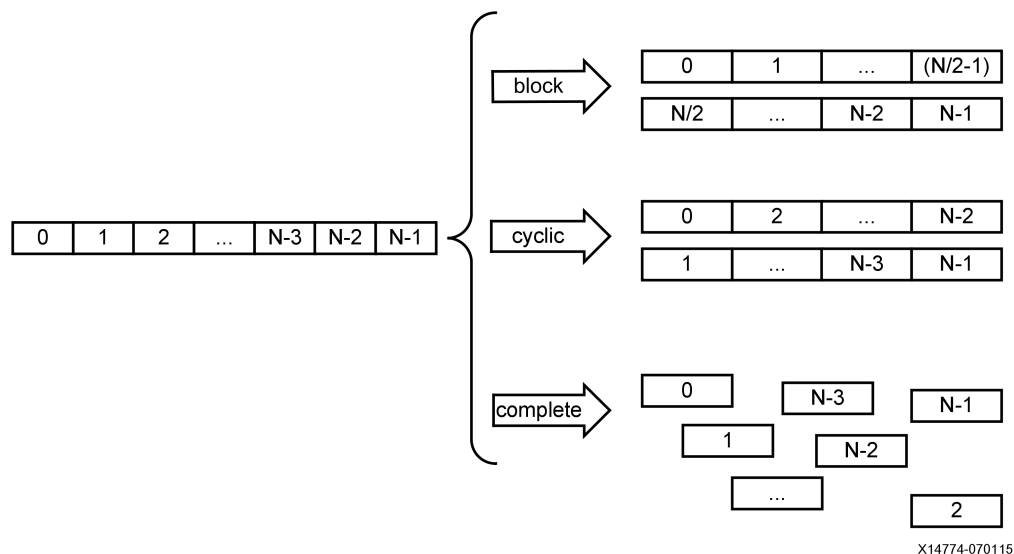
### Array Partitioning

Arrays can be partitioned into smaller arrays. Physical implementation of memories have only a limited number of read ports and write ports, which can limit the throughput of a load/store intensive algorithm. The memory bandwidth can sometimes be improved by splitting up the original array (implemented as a single memory resource) into multiple smaller arrays (implemented as multiple memories), effectively increasing the number of load/store ports.

Vivado HLS provides three types of array partitioning, as shown in [Array Partitioning](#).

1. **block**: The original array is split into equally sized blocks of consecutive elements of the original array.
2. **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
3. **complete**: The default operation is to split the array into its individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

**Figure 6–3: Array Partitioning**



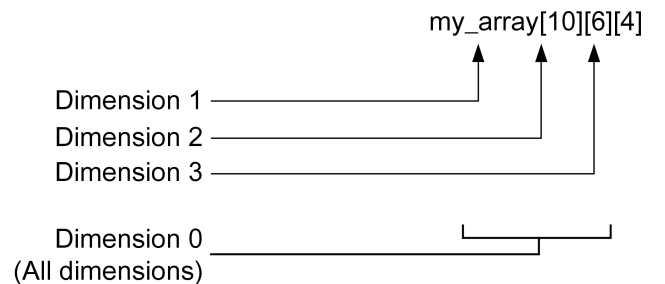
To partition an array in Vivado HLS, insert

```
#pragma HLS array_partition variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>
```

in the hardware function source code. For **block** and **cyclic** partitioning, the `factor` option can be used to specify the number of arrays which are created. In the figure, [Array Partitioning](#), a factor of two is used, dividing the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the last array will have fewer than average elements.

When partitioning multi-dimensional arrays, the `dim` option can be used to specify which dimension is partitioned. The following figure shows an example of partitioning different dimensions of a multi-dimensional array.

**Figure 6–4: Multi-dimension Array Partitioning**

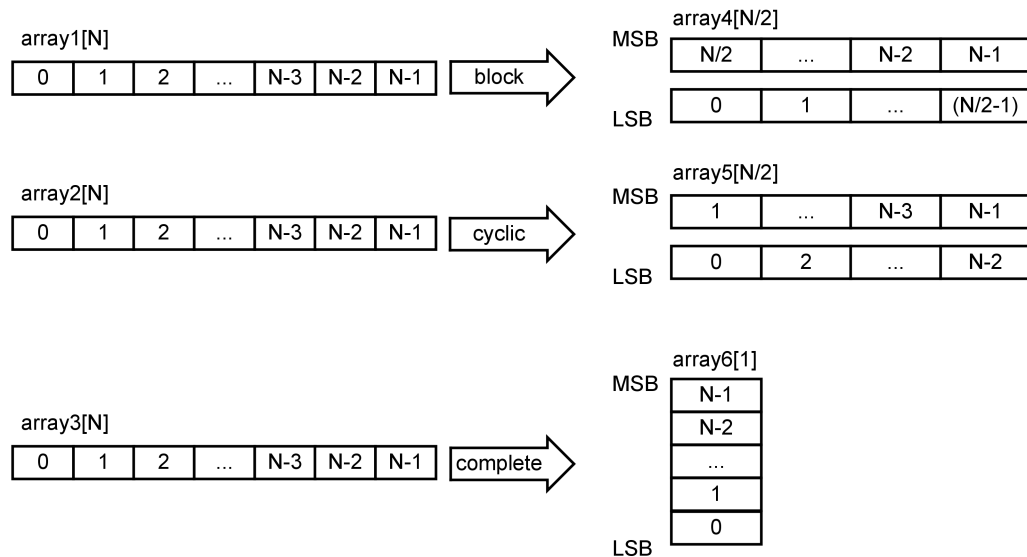


X14769-070115

## Array Reshaping

Arrays can also be reshaped to increase the memory bandwidth. Reshaping takes different elements from a dimension in the original array, and combines them into a single wider element. Array reshaping is similar to array partitioning, but instead of partitioning into multiple arrays, it widens array elements. The following figure illustrates the concept of array reshaping.

**Figure 6–5: Array Reshaping**



X14773-070115

To use array reshaping in Vivado HLS, insert

```
#pragma HLS array_reshape variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>
```

in the hardware function source code. The options have the same meaning as the array partition pragma.

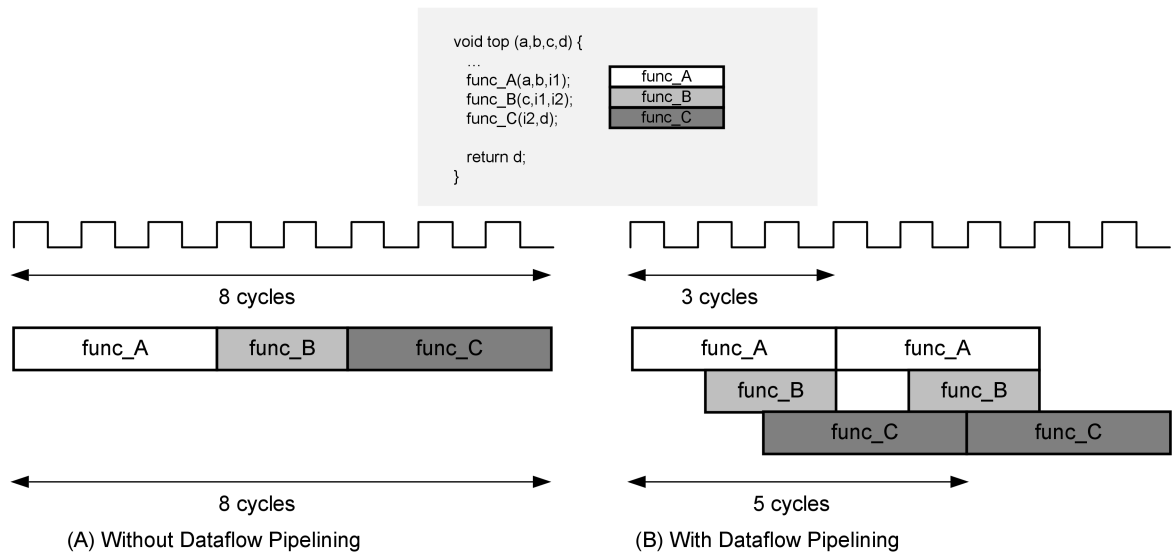
## Data Flow Pipelining

The previously discussed optimization techniques are all "fine grain" parallelizing optimizations at the level of operators, such as multiplier, adder, and memory load/store operations. These techniques optimize the parallelism between these operators. Data flow pipelining on the other hand, exploits the "coarse grain" parallelism at the level of functions and loops. Data flow pipelining can increase the concurrency between functions and loops.

## Function Data Flow Pipelining

The default behavior for a series of function calls in Vivado HLS is to complete a function before starting the next function. Part (A) in [Function Data Flow Pipelining](#), shows the latency without function data flow pipelining. Assuming, it takes 8 cycles for the three functions to complete, the code requires eight cycles before a new input can be processed by "func\_A" and also eight cycles before an output is written by "func\_C" (assume the output is written at the end of "func\_C").

**Figure 6–6: Function Data Flow Pipelining**



X14772-070115

An example execution with data flow pipelining is shown in the part (B) of the figure above. Assuming the execution of `func_A` takes 3 cycles, `func_A` can begin processing a new input every three clock cycles rather than waiting for all the three functions to complete, resulting in increased throughput. The complete execution to produce an output then requires only five clock cycles, resulting in shorter overall latency.

Vivado HLS implements function data flow pipelining by inserting "channels" between the functions. These channels are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data.

- If a function parameter (producer or consumer) is an array, the corresponding channel is implemented as a multi-buffer using standard memory accesses (with associated address and control signals).
- For scalar, pointer and reference parameters as well as the function return, the channel is implemented as a FIFO, which uses less hardware resources (no address generation) but requires that the data is accessed sequentially.

To use function data flow pipelining, put `#pragma HLS dataflow` where the data flow optimization is desired. The following code snippet shows an example:

```

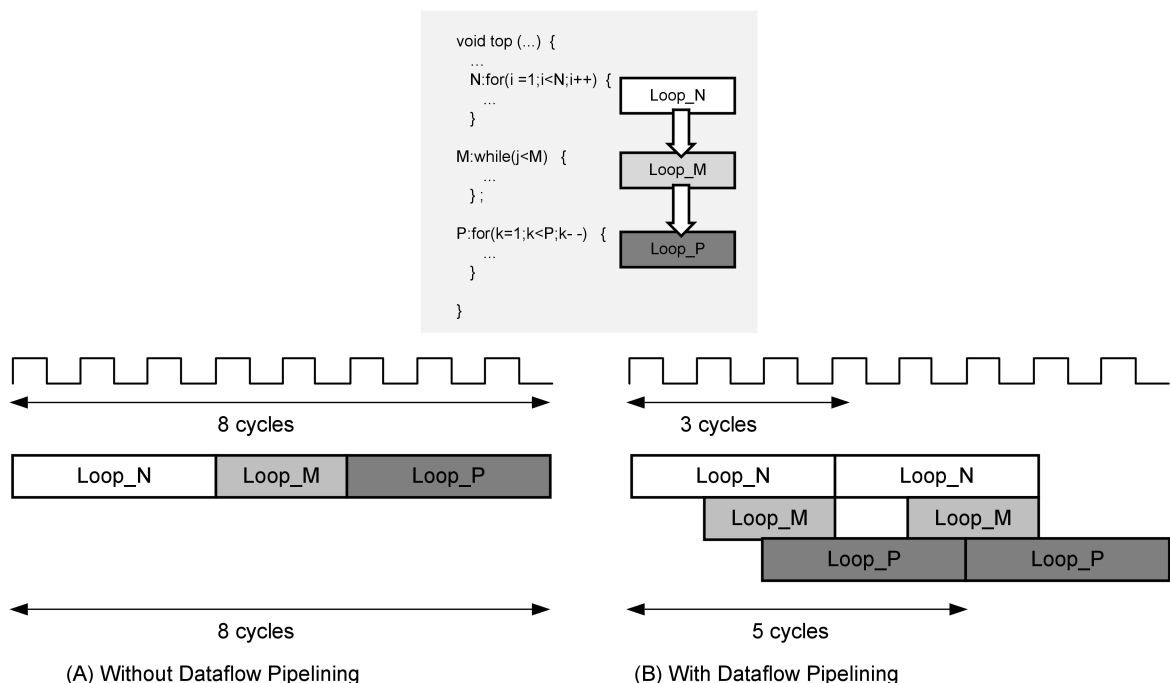
void top(a, b, c, d) {
  #pragma HLS dataflow
  func_A(a, b, i1);
  func_B(c, i1, i2);
  func_C(i2, d);
}
  
```

## Loop Data Flow Pipelining

Data flow pipelining can also be applied to loops in similar manner as it can be applied to functions. It enables a sequence of loops, normally executed sequentially, to execute concurrently. Data flow pipelining should be applied to a function, loop or region which contains either all function or all loops: do not apply on a scope which contains a mixture of loops and functions.

**Loop Data Flow Pipelining** shows the advantages data flow pipelining can produce when applied to loops. Without data flow pipelining, loop N must execute and complete all iterations before loop M can begin. The same applies to the relationship between loops M and P. In this example, it is eight cycles before loop N can start processing the next value and eight cycles before an output is written (assuming the output is written when loop P finishes).

**Figure 6–7: Loop Data Flow Pipelining**



X14771-070115

With data flow pipelining, these loops can operate concurrently. An example execution with data flow pipelining is shown in part (B) of the figure above. Assuming the loop M takes 3 cycles to execute, the code can accept new inputs every three cycles. Similarly, it can produce an output value every five cycles, using the same hardware resources. Vivado HLS automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next. As with data flow pipelining, the channels between the loops are implemented either as multi-buffers or FIFOs.

To use loop data flow pipelining, put `#pragma HLS dataflow` where the data flow optimization is desired.

---

## Hardware Function Interface Details

### Hardware Function Control Protocols

The SDSoC system compiler automatically determines the correct control protocol for a hardware function. This section includes reference material that is only needed when you are forced to write explicit Vivado® HLS pragmas in your source code, for example, because your function requires more than eight stream inputs or eight stream outputs, or because you are creating C-callable/C-linkable libraries for your IP and you want your RTL to mimic the hardware interfaces generated by Vivado® HLS.

The SDSoC™ environment supports the following hardware function control protocols, which are automatically inferred based on the hardware interface definition. The automatically generated software stub functions implement the control protocols, synchronizing data transfers with hardware function execution using `cf_send_i()`, `cf_receive_i()`, and `cf_wait()` APIs defined in `<sds_install_root>/arm-xilinx*-gnueabi/include/cf_lib.h`.



- **None** – no software control interface. The hardware function must self-synchronize entirely based on arguments mapped to AXI streams and cannot have any scalar arguments or arguments that are memory mapped. All AXI stream ports must include `TLAST` and `TKEEP` sideband signals.
- **axis\_acc\_adapter** – the default interface in the SDSoC environment for Vivado® Design Suite HLS hardware functions. The SDSoC environment automatically inserts an instance of the `axis_accelerator_adapter` IP to interface a Vivado HLS hardware function. This IP provides pipelined AXI4-Lite control and data interfaces for software pipelining, and clock domain crossing circuitry to run hardware functions at higher (or lower) clock rates than the data motion network to balance computation and communication. The adapter also provides optional multi-buffering for arguments that map to BRAM and FIFO interfaces, and automatically maps them into AXI4-Streams (see [Hardware Buffer Depth](#) for `buffer_depth` pragma). The hardware function interface cannot include any arguments with `#pragma HLS interface s_axilite`, but can contain any number of arguments that map onto a single AXI-MM master interface (with pragma attribute `offset=direct`) and onto AXI4-Stream interfaces that include `TLAST` and `TKEEP` sideband signals.

The `axis_accelerator_adapter` IP supports up to eight AXI4-Stream inputs and up to eight AXI4-Stream outputs each of which can map onto either a BRAM or FIFO interface. The IP also provides an AXI4-Lite register interface to support scalar arguments, with eight input registers, eight output registers, and eight input/output registers that can be used either for an input, output, or inout argument. Scalar arguments can be of type `bool`, `char`, `short`, `int`, or `float`. A function return value is mapped into an output scalar register. A hardware function that cannot adhere to these constraints must employ the `generic_axi_lite` control protocol.

- **generic\_axi\_lite** – the “native” Vivado HLS control interface when any of the arguments are mapped via `#pragma HLS interface s_axilite`. This interface is suitable for C-callable HDL IP, described in [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\), Creating a Library](#). The hardware control register must reside at offset 0x0 with the following bit encoding.

```
// 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
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

## Vivado HLS Function Argument Types

This section describes supported hardware interface types for hardware functions compiled by the SDSoC™ system compilers using Vivado® HLS. The compilers automatically determine hardware interface types based on the argument type, `#pragma SDS data copy|zero_copy` and `#pragma SDS data access_pattern`.

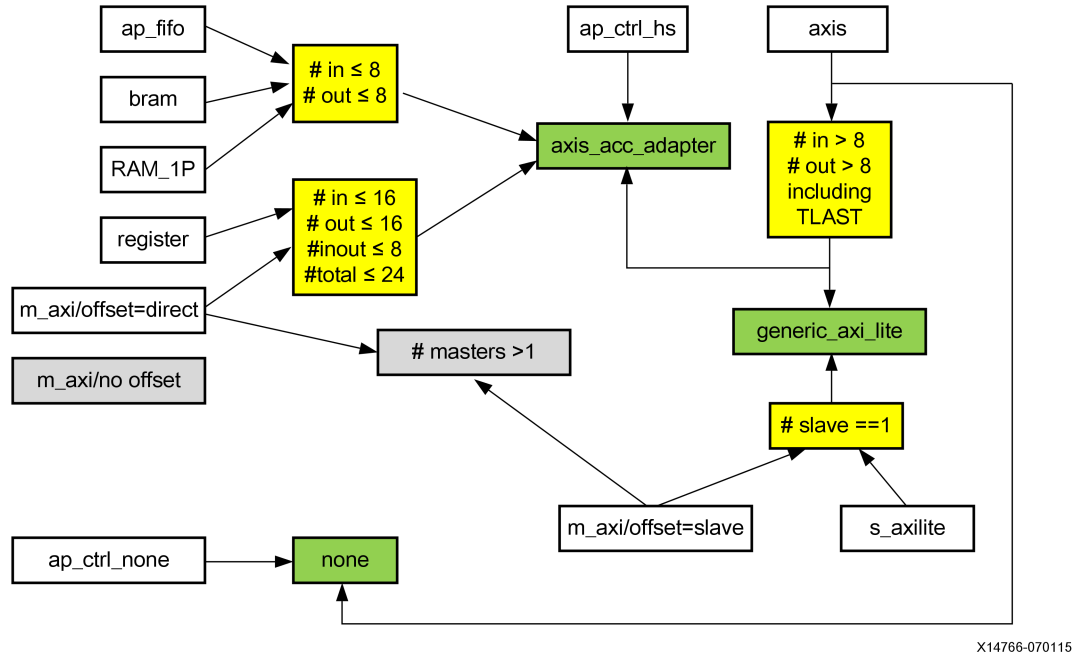


**IMPORTANT:** *To avoid interface incompatibilities, you should only incorporate Vivado® HLS interface type directives and pragmas in your source code when `sdscc` fails to generate a suitable hardware interface directive, and you should only use the HLS interface types described in this section.*

- Vivado® HLS provides arbitrary precision types `ap_fixed<int>`, `ap_int<int>`, and an `hls::stream` class. In the SDSoC environment, arguments to top-level hardware functions must have width of 8, 16, 32, or 64 bits, and you must guard such declarations with `#ifndef __SDS_VHLS__` to coerce to a like-sized C99 type such as `char`, `short`, `int`, or `long long`. Vivado HLS `hls::stream` arguments must be presented to `sdscc/sds++` as arrays. The example `<sdsoc_install_dir>/samples/hls_if/hls_stream` demonstrates how to use HLS `hls::stream` typed arguments in the SDSoC environment.
- By default, an array argument to a hardware function is transferred by copying the data, that is, it is equivalent to using `#pragma SDS data copy`. As a consequence, an array argument must be either used as an input or produced as an output, but not both. For an array that is both read and written by the hardware function, you must use `#pragma SDS data zero_copy` to tell the compiler that the array should be kept in the shared memory and not copied.

The `sdscc` compiler selects a hardware function control protocol based on the program structure, a hardware function prototype, and the types of its arguments. The remainder of this section describes the hardware interface types supported by the system compilers, but it should be emphasized that explicit use of Vivado HLS interface pragmas is discouraged to avoid inadvertent errors due to conflicts between tools defaults and requirements for the control protocols.

The following diagram describes supported hardware interface types (white boxes) and their relation to the supported function control protocols (green). Several mappings involve constraints (yellow). Unsupported HLS interface directives are in gray.

**Figure 6–8: Hardware Function Control Protocols and Supported Hardware Interfaces**


The SDSoC environment supports the following hardware interface types:

- RAM** – using `#pragma SDS data access_pattern(A:RANDOM)` immediately preceding the accelerator function declaration. The SDSoC environment automatically maps onto a packetized AXI4-Stream channel compatible with the DMA protocol, with optional multi-buffering at the accelerator. A hardware function can have no more than eight input bram or ap\_fifo arguments and no more than eight output bram or ap\_fifo arguments.

The example `<sdsoc_install_dir>/samples/hls_if/mmult_hls_bram` demonstrates how to use BRAM interfaces in the SDSoC environment.

- FIFO** – using `#pragma SDS data access_pattern(A:SEQUENTIAL)` immediately preceding the accelerator function declaration. The SDSoC environment automatically maps onto a packetized AXI4-Stream channel compatible with the DMA protocol. A hardware function can have no more than eight input bram or ap\_fifo arguments and no more than eight output bram or ap\_fifo arguments.

The example `<sdsoc_install_dir>/samples/hls_if/mmult_hls_ap_fifo` demonstrates how to use HLS ap\_fifo interfaces in the SDSoC environment.

- SCALAR** The SDSoC environment automatically maps arguments with basic arithmetic types (8, 16, or 32 bits) onto a register accessible over an AXI-Lite interface. The SDSoC environment treats registers as FIFOs to support task pipelining with multiple in-flight task calls. An HLS hardware function can have up to 8 inout scalar arguments, up to 16 input scalars or 16 output scalars, with no more than 24 scalar arguments total including a return value. If more scalar arguments are required, you must explicitly map all scalar arguments onto an HLS-generated AXI4-Lite interface using HLS pragmas.

A hardware function cannot contain both scalar register mapped and explicit axilite mapped arguments.

- AXI4-Lite** – using `#pragma HLS INTERFACE s_axilite port=arg` in the hardware function. Inclusion of this pragma also requires `#pragma HLS INTERFACE s_axilite port=return` to generate a memory mapped control interface in HLS. There are no FIFOs on the command interface or on scalar arguments. A hardware function can have

only one explicit AXI4-Lite interface; you must bundle all ports, including `ap_control`, into a single AXI4-Lite interface.

The example `<sdsoc_install_dir>/samples/hls_if/arraycopy_axilite` demonstrates how to use HLS AXI4-Lite interfaces in the SDSoc environment.

- **AXI-memory mapped (AXI-MM) master** – Using the VHLS pragma `#pragma HLS INTERFACE m_axi port=arg` to pass physical addresses over the AXI4-Lite interface. In this mode, the hardware function acts as its own data mover. When a hardware function maps an argument onto an AXI-MM master, it must also include an output scalar argument or a return value.

The example `<sdsoc_install_dir>/samples/hls_if/mmult_hls_aximm` demonstrates how to use HLS AXI-MM interfaces in the SDSoc environment.

- **AXI4-Stream** – using `#pragma HLS INTERFACE axis port=arg` in the hardware function. The SDSoc environment supports direct connections between hardware functions with commensurate AXI4-Stream interfaces. The example `<sdsoc_install_dir>/samples/hls_if/mmult_hls_axis` demonstrates how to use HLS AXI4-Stream interfaces in the SDSoc environment.




---

**IMPORTANT:** *It is recommended that you do not use this type unless absolutely required, for example, when a hardware function has more than eight input array arguments or eight array arguments that must be mapped onto AXI4-Stream transport channels. Otherwise, it is recommended you use the `#pragma SDS data mem_attribute(A:SEQUENTIAL)` attribute, which directs `sdsoc` to automatically map the array transfer onto an AXI4-Stream channel.*

---




---

**IMPORTANT:** *Data transport using a DMA data mover requires AXI4-Stream `TLAST`, `TKEEP` side band signals, which must be explicitly coded within HLS code.*

---

# Using C-Callable IP Libraries

The [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#), [Creating a Library](#) chapter describes the process for creating C-callable libraries that employ IP cores.

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

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

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

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

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

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

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

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

## Using Vivado Design Suite HLS Libraries

This section describes how to use Vivado HLS libraries with the SDSoC environment.

Vivado® Design Suite High-Level Synthesis (HLS) libraries are provided as source code with the Vivado HLS installation in the SDSoC environment. Consequently, you can use these libraries as you would any other source code that you plan to cross-compile for programmable logic using Vivado HLS. In particular, you must ensure that the source code conforms to the rules described in [Hardware Function Argument Types](#), which might require you to provide a C/C++ wrapper function to ensure the functions export a software interface to your application.

The synthesizable FIR example template for all basic platforms in the SDSoC IDE provides an example that uses an HLS library. You can find several additional code examples that employ HLS libraries in the `samples/hls_lib` directory. For example, `samples/hls_lib/hls_lib` contains an example to implement and use a square root function.

The file `my_sqrt.h` contains:

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_

#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

The file `my_sqrt.cpp` contains:

```
#include "my_sqrt.h"

void my_sqrt(float x, float *ret)
{
    *ret = sqrtf(x);
}
```

The makefile has the commands to compile these files:

```
sds++ -c -hw my_sqrt -sds-pf zc702 my_sqrt.cpp
sds++ -c my_sqrt_test.cpp
sds++ my_sqrt.o my_sqrt_test.o -o my_sqrt_test.elf
```

## Exporting an Application as a Library

When you create an application in the SDSoC environment, you select an SDSoC platform as a starting point and specify a set of functions to be implemented in hardware on top of it. The SDSoC system compiler creates a hardware design that includes the functions selected for hardware as well as the corresponding data movers, and also generates the required software to communicate with these accelerators and data movers. By default, the output of the system compilation is a complete boot image including bitstream, file system, operating system, and application executable.

You can change the SDSoC system compiler options to generate either a static or a shared library instead of an application binary, and you can then link to this library when developing the rest of your application using the standard GNU toolchain. You are still targeting the same hardware system and using the `sdscc`-generated boot environment, but you are then free to develop your software using the GNU toolchain in the software development environment of your choice.

One use case for a library flow is to partition the application into the hardware-specific portion and the rest of the application software that runs entirely on the CPU and does not need to be compiled through `sdscc`. After you have determined the hardware accelerators and built the application-specific hardware system, a library allows you to develop the rest of the software application using the ARM toolchain, with fast software compilation.

The entry points into the shared library are specific stub functions generated by `sdscc` during system compilation that you declare in the library header file.



**IMPORTANT:** *Because the same hardware is being used for the shared library, you must ensure that the entry points into your library enforce consistency with the generated system, including all assumptions on memory allocation, direct hardware connectivity between hardware functions, and data movers. When the overall application code is linked with the shared library, there are no additional consistency checks (for example, memory allocation for buffers passed to hardware functions) with the assumptions made by `sdscc` during system generation. It is strongly recommended that you wrap any "connected components" of multiple hardware functions that are directly connected in hardware, into a single function that controls access to all of the individual hardware functions.*

When `sdscc` generates a system, it automatically creates a static library with entry points for each of the hardware functions. For a project named `myApp` created in the SDSoC IDE, the library is `<build_configuration>/_sds/swstubs/libmyApp.a`. For `myApp.elf` built using the command line interface, the library is `_sds/swstubs/libmyApp.a`.

The stub function entry points to the library are not precisely the same as the set of hardware function prototypes. The `sdscc` compiler automatically renames hardware functions with "mangled" names to support multiple bitstreams (that is, partitions using `#pragma SDS partition`) and multiple hardware function instances. In general you do not need to be aware of this renaming, but when you export a design as a library, the library header file must declare the stub functions.

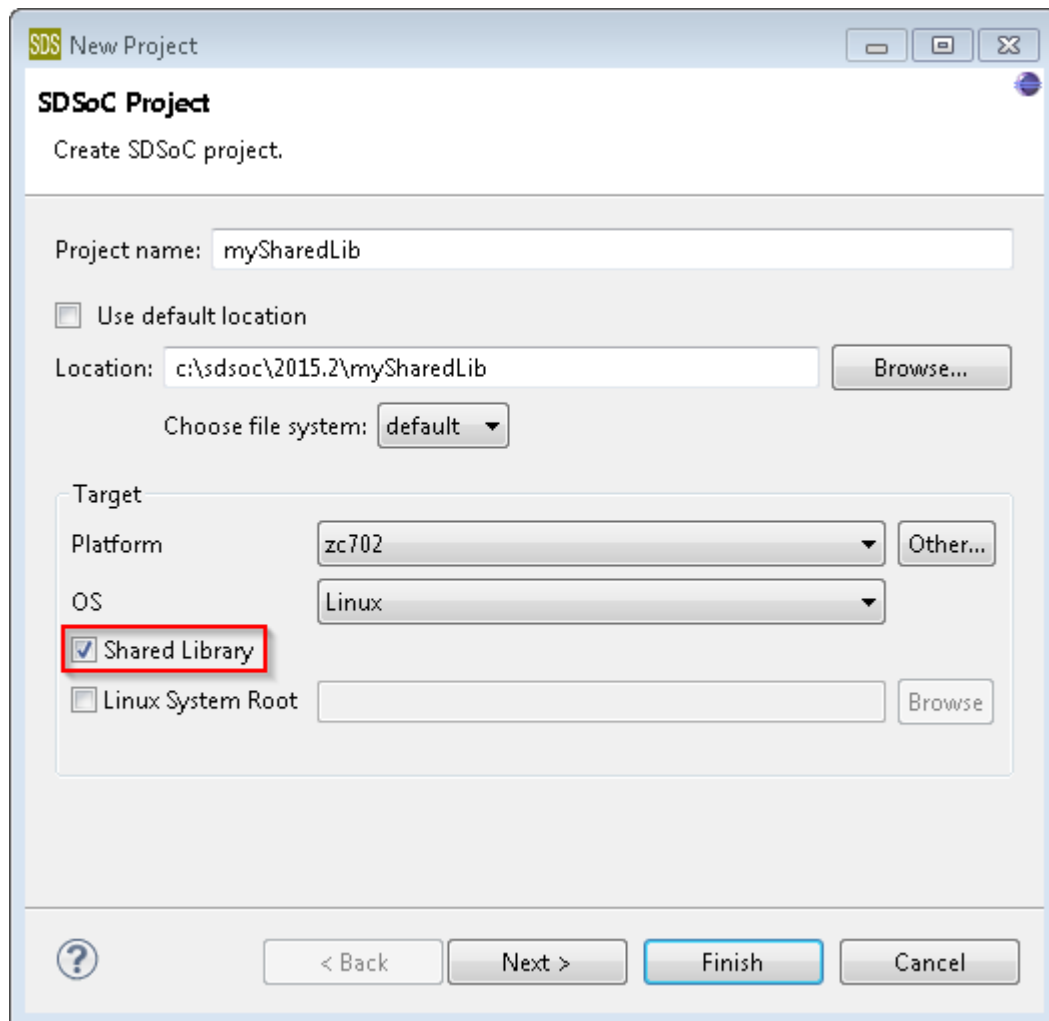
For example, a hardware function `mmult_accel` typically has the following declaration:

```
// mmult_accel.h
void _p0_mmult_accel_0(float[], float[], float[]); // hardware function mmult_accel
```

For any hardware function, the entry point should be straightforward to determine by expanding the library in the Project Explorer and inspecting functions within the library.

You can find a complete example in the `samples/mmult_static_lib/build` directory in the SDSoC environment install.

To create a shared library in the SDSoC IDE, you select the Shared Library check box when you create a new SDSoC environment project.



The shared library `libmySharedLib.so` is created along with the SD card boot image. You can export a design as a shared library from the command line by compiling source files containing the hardware functions and functions calling them with the `sdscc/sds++` position independent code flag (`-fPIC`) and linking using the `-shared` option.



The SDSoC IDE provides a Matrix Multiplication Shared Library example template when you select the Shared Library check box shown in the figure above. The connectivity of the hardware blocks is determined using a source file that includes a process function that defines how the user calls the library. The SDSoC system compiler then determines the system connectivity based on this function as usual.

```
File: mmult_call.c
#include "mmult_accel.h"

void mmult_call (float in_A[A_NROWS*A_NCOLS],
                 float in_B[A_NCOLS*B_NCOLS],
                 float out_C[A_NROWS*B_NCOLS])
{
    mmult_accel(in_A, in_B, out_C);
}
```

This example specifies that there is a single call to the function `mmult_accel` that is selected for hardware implementation, but you can specify multiple hardware functions for the library.

The hardware function is compiled using `sdscc` with an additional `-fPIC` flag to make the object code position independent.

```
sdscc -sds-pf zc702 -sds-hw mmult_accel mmult_accel.cpp -sds-end \
-c -fPIC mmult_accel.c -o mmult_accel.o
```

You must also compile the calling function code with the `-fPIC` flag.

```
sdscc -sds-pf zc702 -c -fPIC mmult_call.c -o mmult_call.o
```

Finally, link all object files and specify the shared library options.

```
sdscc -sds-pf zc702 -shared mmult_accel.o mmult_call.o -o libmmult_accel.so
```

This creates a `libmmult_accel.so` library that can be linked using the standard ARM GNU toolchain on the command line or in any software development environment.

The above command also creates an `sd_card` image that contains the boot files needed to execute the program that links against the library.

You can find a complete example in the `samples/mmult_shared_lib/build` directory in the SDSoC environment install.

## Linking to an Application Library

A library generated for an application in the SDSoC environment is linked like any other software library. You `#include` header files associated with the library into source files and compile them with the `GCC -I` option to specify the directory path to the header files. You link your application using the `GCC -L` option to specify the path to the library, and the `-l` option to declare the library name.

As an example, assume you created a file called `mmult.cpp` that contains the main function and calls an `mmult_accel` function in a shared library. Compile the file using:

```
arm-xilinx-linux-gnueabi-g++ -c -O3 mmult.cpp -o mmult.o
```

and link the application using:

```
arm-xilinx-linux-gnueabi-g++ -O3 mmult.o -L./lib -lmmult_accel -lpthread \  
-o mmult.elf
```

This creates an executable called `mmult.elf` that you copy into your SD card along with the boot files. The POSIX Threads (`pthread`) library is required for the software runtime code generated by `sdscc`.

To run the program, copy the `sd_card` directory created in the SDSoc environment into an SD card, boot the board and wait for the command prompt. Execute the following commands on the board:

```
sh-4.3# export LD_LIBRARY_PATH=/mnt  
sh-4.3# /mnt/mmult.elf
```

You can find complete examples in the `samples/mmult_shared_lib/use` and `samples/mmult_static_lib/use` directories in the SDSoc environment install.

# Debugging an Application

The SDSoC™ environment allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE (user-defined makefiles) and debugged either on the command line or using the SDSoC IDE.

See [SDSoC Environment User Guide: Getting Started \(UG1028\)](#), [Tutorial: Debugging Your System](#) for information on using the interactive debuggers in the SDSoC IDE.

---

## Debugging Linux Applications in the SDSoC IDE

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

1. Select the **SDDebug** as the active build configuration and build the project.
2. Copy the generated `SDDebug/sd_card` image to an SD card, and boot the board with it.
3. Make sure the board is connected to the network, and note its IP address, for example, by executing `ifconfig eth0` at the command prompt.
4. Select the **Debug As** option to create a new debug-configuration, and enter the IP address for the board
5. You now switch to the SDSoC environment debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

---

## Debugging Standalone Applications in the SDSoC IDE

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

1. Select **SDDebug** as the active build configuration and build the project.
2. Make sure the board is connected to your host computer using the JTAG Debug connector.

3. Select the **Debug As** option to create a new debug-configuration

You now switch to the SDSoC environment debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

In the SDSoC IDE Project Overview panel, click on the **Debug application** link, which provides a shortcut to the procedure described above.

---

## Debugging FreeRTOS Applications

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

---

## Peeking and Poking IP Registers

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

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

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




---

**CAUTION!** *Trying to access non-existent addresses can cause the system to hang.*

---



---

## Debugging Performance Tips

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

Estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado® Design Suite HLS report files (`_sds/vhls/.../*.rpt`). Latency of X accelerator clock cycles =  $X * (\text{processor\_clock\_freq} / \text{accelerator\_clock\_freq})$  processor clock cycles. Compare this with the time spent on the actual function call to determine the data transfer overhead.

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

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

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

# Target Operating System Support

The SDSoC environment currently supports Linux, standalone ("bare metal"), and FreeRTOS target operating systems. The SDSoC system compilers use target OS-specific characterization data in choosing data movers for each application, so the same application code running under different operating systems may result in different generated hardware systems.

---

## Linux Applications

The SDSoC™ environment supports Linux applications that run on Zynq® devices, which lets users compile their programs to run on the hardware with the Linux operating system. The SDSoC environment links in a library to communicate with the hardware using services provided by the operating system.

### Usage

In order to compile and link an SDSoC environment program for Linux, the makefile should include `-target-os linux` in CFLAGS, as well as LFLAGS. If the `-target-os linux` option is omitted, the SDSoC™ environment by default targets the Linux operating system.

The SD boot image consists of multiple files in the `sd_card` directory. `BOOT.BIN` contains the first stage boot loader (FSBL), which is invoked directly after powering on the board, and in turn invokes U-boot. Linux boot uses a device tree, Linux kernel and ramdisk image. Finally, SD boot image also includes the application ELF and hardware bitstream used to configure the programmable logic.

## Supported Platforms

Linux mode is supported for all SDSoC™ platforms.

## Limitations

The provided Linux operating system utilizes a pre-built kernel image (3.19, Xilinx branch `xilinx-v2015.2.01`) and a ramdisk containing BusyBox. To configure the Linux image or ramdisk image for your own platform or requirements, follow the instructions at [wiki.xilinx.com](http://wiki.xilinx.com) to download and build the Linux kernel. [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#), [Linux Boot Files](#) describes the Linux boot files and summarizes the process for creating them using Petalinux.

## Standalone Target Applications

In addition to Linux applications that run on Zynq® devices, the SDSoC™ environment supports standalone mode, which lets users compile their programs to run directly on the hardware without any operating system. The SDSoC environment links in a library that provides the services normally provided by the target operating system.

### Usage

In order to compile and link an SDSoC environment program for standalone mode, the makefile should include `-target-os standalone` in `CFLAGS`, as well as `LFLAGS`.

The SD boot image consists of a single file `BOOT.BIN` in the `sd_card` directory that contains the first stage boot loader (FSBL) as well as the user application, which is invoked directly after powering on the board.

### Supported Platforms

Standalone mode is supported for the following platforms:

- `zc702` (based on the Xilinx ZC702 evaluation board)
- `zc706` (based on the Xilinx ZC706 evaluation board)
- `zed` (based on the ZedBoard from [zedboard.org](http://zedboard.org))
- `microzed` (based on the MicroZed board from [zedboard.org](http://zedboard.org))
- `zybo` (based on the Zybo board from )

### Limitations

Standalone mode does not support multi-threading, virtual memory, or address protection as documented in [OS and Libraries Document Collection \(UG643\)](#). Access to the file system is not through the usual C API, but instead through a special API using `libxilffs`. The sample program `file_io_manr_sobel_standalone` shows an example of its use. This program can be compared with the Linux version `file_io_manr_sobel` to see what changes are necessary for accessing the file system. In general, the procedure to access the file system is to include a few extra files, use different types (for example, `FIL` instead of `FILE`), use a slightly different API for file system access (for example, `f_open` instead of `fopen`), and disable DCache before doing any file operations.



**IMPORTANT:** *On the ZedBoard, a serial connection to the board takes a couple of seconds. If your program runs for a time shorter than that, you will never see its output. When the ZedBoard is power cycled, the serial connection goes down and it is not possible to see the output in the subsequent run either. The ZC702 and ZC706 boards keep the serial connection alive across power cycles and do not suffer from this limitation.*

## FreeRTOS Target Applications

In addition to Linux applications that run on Zynq®-7000 AP SoC devices, the SDSoC™ environment supports applications that use the FreeRTOS real time operating system from Real Time Engineers Ltd ([www.freertos.org](http://www.freertos.org)), which allows users to compile their programs with a real time kernel using APIs for scheduling, inter-task communication, timing and synchronization.

The SDSoC environment includes FreeRTOS v8.2.1 header files and a pre-configured library containing the real time kernel, API functions and Zynq device-specific platform code. It also builds the standalone library that provides drivers and functions required to support a C/C++ bare-metal application.

### Usage

In order to compile and link an SDSoC™ environment program for FreeRTOS, the makefile should include the `-target-os freertos` option in all compiler and linker calls in the makefile. This is typically specified in an SDSoC environment variable, which in turn is included in a compiler toolchain variable, as shown below:

```
SDSFLAGS = -sds-pf zc702 -target-os freertos \
-sds-hw mmult_accel mmult_accel.cpp -sds-end \
-poll-mode 1
CPP = sds++ ${SDSFLAGS}
CC = sds ${SDSFLAGS}
:
all: ${EXECUTABLE}
${EXECUTABLE}: ${OBJECTS}
${CPP} ${LFLAGS} ${OBJECTS} -o $@
%.o: %.cpp
${CPP} ${CFLAGS} $< -o $@
:
```

When the SDSoC environment links the application ELF file, it builds a standalone (bare-metal) library for you, provides a predefined linker script and uses a pre-configured FreeRTOS kernel using headers and a pre-built library, and includes their paths when it calls the ARM GNU toolchain (you do not need to specify the paths in your makefile):

```
<path_to_install>/SDSoC/2015.2/arm-xilinx-eabi/include/freertos
<path_to_install>/SDSoC/2015.2/arm-xilinx-eabi/lib/freertos
```

The SD boot image consists of a single file `BOOT.BIN` in the `sd_card` directory that contains the first stage boot loader (FSBL) as well as the user application, which is invoked directly after powering on the board.

SDSoC environment GUI flows for working with FreeRTOS applications are the same as those for standalone (bare-metal) applications, except the target OS is specified as FreeRTOS. The user application code needs to include the following:

- Hardware configuration function
- Task functions and task creation calls using the `xTaskCreate()` API function
- Scheduler start call using the `vTaskStartScheduler()` API function
- Callback functions such as `vApplicationMallocFailedHook()`, `vApplicationStackOverflowHook()`, `vApplicationIdleHook()`, `vAssertCalled()`, `vApplicationTickHook()`, and `vInitialiseTimerForRunTimeStats`



Simple SDSoC environment applications based on the Zynq®-7000 AP SoC series demo included in the FreeRTOS v8.2.1 software distribution are available in the SDSoC GUI application wizard and in the SDSoC environment installation:

```
<path_to_install>/SDSoC/2015.2/samples/mmult_datasize_freertos  
<path_to_install>/SDSoC/2015.2/samples/mmult_optimized_sds_freertos
```

User or sample applications that normally target the Standalone BSP can be built using the `-target-os freertos` option compile and link, but the FreeRTOS linker script is used and predefined callback functions found in the pre-built FreeRTOS library are used. Applications built this way do not explicitly call FreeRTOS API functions and run as standalone applications. While it is possible to begin FreeRTOS application development in this way, Xilinx recommends that FreeRTOS API functions and callbacks be incorporated as early as possible.

## Supported Platforms

FreeRTOS mode is supported for two Zynq®-7000 AP SoC platforms:

- ZC702
- ZC706

## Limitations and Implementation Notes

The SDSoC environment FreeRTOS support uses the standalone board support package (BSP) library and includes the same limitations as standalone mode.

In performance estimation flows, a FreeRTOS application instrumented to collect software runtime data needs to exit for the data to be collected and merged with hardware performance data to create a report.

The SDSoC environment uses a pre-configured FreeRTOS v8.2.1 library that has been pre-built for the user, and a dynamically built (at application link time) standalone library. Characteristics of the FreeRTOS library include:

- Uses the standard FreeRTOS v8.2.1 distribution for platform independent code; platform dependent code uses the default `FreeRTOSConfig.h` file included as part of FreeRTOS v8.2.1 (see the FreeRTOS reference <http://www.freertos.org/a00110.html>, with downloads available at <http://sourceforge.net/projects/freertos/files/FreeRTOS> )
- Uses `heap_3.c` for its memory allocation implementation (see the FreeRTOS reference <http://www.freertos.org/a00111.html> )
- Uses source from the following FreeRTOS v8.2.1 distribution folders:
  - `Demo/CORTEX_A9_Zynq_ZC702/RTOSDemo/src`
  - `Source`
  - `Source/include`
  - `Source/portable/GCC/ARM_CA9`
  - `Source/portable/MemMang`
- Uses a linker script found in `<path_to_install>/SDSoC/2015.2/platforms/<platform>/freertos/lscript.ld` (to temporarily use a modified version of this file instead, make a copy of the file and add the linker option `-Wl,-T<path_to_your_linker_script>` to the `sdscc/sds++` command line used to create the ELF file)
- Is based on the porting description for Zynq ZC702 found at <http://www.freertos.org/RTOS-Xilinx-Zynq.html>, including replacement functions for `memcpy()`, `memset()`, and `memcmp()` as part of the pre-built library rather than user application code; does not use a Xilinx® SDK-based BSP package
- Includes predefined callback functions to enable standalone applications to be linked with the `sdscc/sds++ -target-os freertos` option (Xilinx recommends that you define your own versions of these functions as part of the application)
  - `vApplicationMallocFailedHook`
  - `vApplicationStackOverflowHook`
  - `vApplicationIdleHook`
  - `vAssertCalled`
  - `vApplicationTickHook`
  - `vInitialiseTimerForRunTimeStats`

# Representative Example Designs

When you create a new SDSoC environment project for one of the base platforms within the SDSoC IDE, you can optionally choose one of several representative designs.

- [File I/O Video Example](#) - a simple file-base video-processing example
- [Synthesizable FIR Filter](#) - example using a Vivado HLS library
- [Matrix Multiplication](#) - a standard linear algebra hardware accelerator
- [Using a C-Callable RTL Library](#) - example using a packaged C-callable IP written in a hardware description language (HDL)

---

## File I/O Video Example

It is sometimes useful to read video data from a file and write back the processed data to a file, instead of reading and writing frame buffers. A simple example called `file_io_manr_sobel` illustrates the methodology. The example uses the base ZC702 platform. The overall structure of the `main()` function is:

```
int main()
{
    // code omitted
    read_frames(in_filename, frames, rows, cols, ...);
    process_frames(frames, ...);
    write_frames(out_filename, frames, rows, cols, ...);
    // code omitted
}
```

Since there is no need for synchronization of the input and output with the video hardware, the software loop in `process_frames()` is straightforward, creating a hardware function pipeline when `manr` and `sobel_filter` are selected for hardware implementation.

```
for (int loop_cnt = 0; loop_cnt < frames; loop_cnt++) {
    // set up manr_in_current and manr_in_prev frames
    manr( nr_strength, manr_in_current, manr_in_prev, yc_out_tmp);
    sobel_filter(yc_out_tmp, out_frames[frame]);
}
```

The input and output video files are in YUV422 format. The platform directory contains sources for converting these files to/from the frame arrays used in the accelerator code. The makefile in the top level directory compiles the application sources along with the platform sources to generate the application binary.

## Synthesizeable FIR Filter

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

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

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

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

## Matrix Multiplication

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

## Using a C-callable RTL Library

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

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

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

# SDSoC Pragma Specification

This section describes pragmas (directives) for the SDSoC system compilers `sdscc/sds++` to assist system optimization.

All pragmas specific to the SDSoC environment are prefixed with `#pragma SDS` and should be inserted into C/C++ source code, prior either to a function declaration or at a function call site.

There is no single dominant industry standard in wide use for compilers that target heterogeneous embedded systems that employ hardware accelerators, but the pragmas and pragma syntax has been defined to be consistent with standards like OpenACC. In a future release, the SDSoC environment might adopt an industry standard pragmas should a suitable standard become widely adopted.

---

## Data Transfer Size

The syntax for this pragma is:

```
#pragma SDS data copy|zero_copy(ArrayName[offset:length])
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- The `data_copy` implies that data is explicitly copied from the processor memory to the hardware function. A suitable data mover as described in [Improving System Performance](#) performs the data transfer. The `data_zero_copy` means that the hardware function accesses the data directly from shared memory. For the latter, the hardware function must access the array through an AXI4 bus interface.
- For a multi-dimensional array, each dimension should be specified. For example, for a 2-dimensional array, use `ArrayName[offset_dim1:length_dim1][offset_dim2:length_dim2]`
- Multiple arrays can be specified in the same pragma, separated by a comma(. For example: `copy(ArrayName1[offset1:length1], ArrayName2[offset2:length2])`
- The `[offset:length]` part is optional
- `ArrayName` must be one of the formal parameters of the function definition, that is, not from the prototype (where parameter names are optional) but from the function definition.
- `offset` is the number of elements from the first element in the corresponding dimension. It must be a compile-time constant. This is currently ignored.
- `length` is the number of elements transferred for that dimension. It can be an arbitrary expression as long as the expression can be resolved at runtime inside the function. For example:

```
#pragma SDS data copy(InData[0:num_rows+3*num_coeffs_active + L*(P+1)])
#pragma SDS data copy(OutData[0:2*(L-M-R+2)+4*num_coeffs_active*(1+num_rows)])
void evw_accelerator (uint8_t M,
                     uint8_t R,
                     uint8_t P,
                     uint16_t L,
                     uint8_t num_coeffs_active,
                     uint8_t num_rows,
                     uint32_t InData[InDataLength],
                     uint32_t OutData[OutDataLength]);
```

This pragma specifies the number of elements to be transferred for an array argument to a hardware function, and applies to all calls to the function. As shown in the example above, `length` need not be a constant; it can be a C arithmetic expression involving other scalar parameters to the same function.

If this pragma is not specified for an array argument, the SDSoc environment first checks the argument type. If the argument type has a compile-time array size, the compiler uses that as the data transfer size. Otherwise, the SDSoc environment analyzes the calling code to determine the transfer size based on the memory allocation APIs for the array (for example, `malloc` or `sds_alloc`). If the analysis fails or there is inconsistency between callers about the transfer size, the compiler generates an error message so that the user can modify the source code.

## Memory Attributes

For an operating system like Linux that supports virtual memory, user-space allocated memory is paged, which can affect system performance. To improve system performance, the pragmas in this section can be used to declare arguments that have been allocated in physically contiguous memory, or to tell the compiler that it need not enforce cache coherency.

## Physically Contiguous Memory and Data Caching



**IMPORTANT:** *The syntax and implementation of this pragma might be revised in a future release.*

The syntax for this pragma is:

```
#pragma SDS data mem_attribute(ArrayName:cache|contiguity)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- `ArrayName` must be one of the formal arguments of the function.
- `cache` must be either `CACHEABLE` or `NON_CACHEABLE`. The default value is set to be `CACHEABLE`. `CACHEABLE` means that the compiler must maintain cache coherency between the CPU and accelerator for the memory allocated to the array. To maintain the cache coherency, it may be necessary (for example, when using HP ports) to perform a cache flush before transferring the data to an accelerator and to perform a cache-invalidate when transferring the data from the accelerator to the memory.  
  
`NON-CACHEABLE` means that the compiler does not need to ensure the cache coherency of the specified memory. It is then the user's responsibility to do so when necessary. It gives compiler more freedom in allocating memory ports. A typical use case is in video applications where:
  - Cache flushing/invalidating for a large chunk of video data can significantly decrease the system performance
  - Software code does not read or write the video data so the cache coherency between processor and accelerator is not required.
- `Contiguity` must be either `PHYSICAL_CONTIGUOUS` or `NON_PHYSICAL_CONTIGUOUS`. The default value is set to be `NON_PHYSICAL_CONTIGUOUS`. `PHYSICAL_CONTIGUOUS` means that all memory corresponding to the associated `ArrayName` is allocated using `sds_alloc`, while `NON_PHYSICAL_CONTIGUOUS` means that all memory corresponding to the associated `ArrayName` is allocated using `malloc`. This helps the SDSoC compiler select the optimal data mover.
- Multiple arrays can be specified in one pragma, separated by commas.

## Data Mover Type



**IMPORTANT:** *This pragma is not recommended for normal use. Only use this pragma if the compiler-generated data mover type does not meet the design requirement.*

The syntax for this pragma is:

```
#pragma SDS data data_mover(ArrayName:DataMover)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- Multiple arrays can be specified in one pragma, separated by a comma (,). For example:

```
#pragma SDS data_mover(ArrayName:DataMover, ArrayName:DataMover)
```

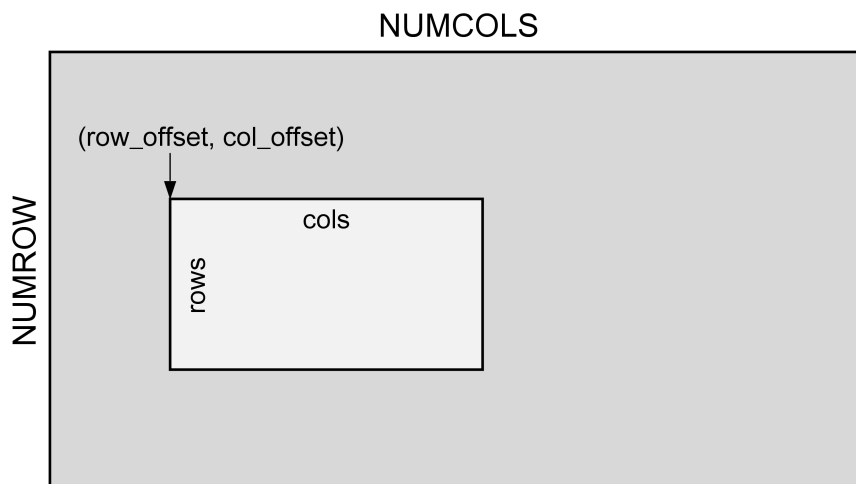
- ArrayName must be one of the formal parameters of the function.
- DataMover must be either AXIFIFO or AXIDMA\_SG or AXIDMA\_SIMPLE or AXIDMA\_2D.

This pragma specifies the data mover HW IP type used to transfer an array argument. Typically, the compiler chooses the type of the data automatically by analyzing the code. This pragma can be used to override the compiler inference rules.

There are some additional requirements for using AXIDMA\_SIMPLE and AXIDMA\_2D. The first requirement is that the corresponding array must be allocated using `sds_alloc()`.

- For AXIDMA\_2D, the pragma `SDS data dim` must be present to specify the 2D array's size of each dimension. The `SDS data copy` pragma is also needed to specify a rectangular sub-region of the 2D array to be transferred. The array second dimension size, sub-region offset and column size must all result in addresses aligned to 64-bit boundaries (number of bytes divisible by 8).
- In the example shown below, `NUMCOLS`, `row_offset`, `col_offset` and `cols` must be multiples of 8 (each char bitwidth is 8) for AXIDMA\_2D to work properly.

```
#pragma SDS data data_mover(y_lap_in:AXIDMA_SIMPLE, y_lap_out:AXIDMA_2D)
#pragma SDS data dim(y_lap_out[NUMROWS][NUMCOLS])
#pragma SDS data copy(y_lap_out[row_offset:rows][col_offset:cols])
void laplacian_filter(unsigned char y_lap_in[NUMROWS*NUMCOLS],
    unsigned char y_lap_out[NUMROWS*NUMCOLS],
    int rows, int cols, int row_offset, int col_offset);
```



X14707\_060515



## SDSoC Platform Interfaces to External Memory



**IMPORTANT:** *The syntax and implementation of this pragma might be revised in a future release.*

The syntax for this pragma is:

```
#pragma SDS data sys_port(ArrayName:port)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- `ArrayName` must be one of the formal arguments of the function.
- `port` must be ACP or AFI or MIG. The Zynq-7000 All Programmable SoC provides a cache coherent interface between programmable logic and external memory (S\_AXI\_ACP) and high-performance ports (S\_AXI\_HP) for non-cache coherent access (AFI). If no `sys_port` pragma is specified for an array argument, the interface to external memory is determined automatically by the SDSoC system compilers, based on array memory attributes (cacheable or non-cacheable), array size, data mover used, etc. This pragma overrides the SDSoC compiler choice of memory port. MIG is valid only for the `zc706_mem` platform.
- Multiple arrays can be specified in one pragma, separated by commas.

## Hardware Buffer Depth

The syntax of this pragma is:

```
#pragma SDS data buffer_depth(ArrayName:BufferDepth)
```



**IMPORTANT:** *The hardware interpretation of this pragma might be revised in a future release.*

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- Multiple arrays can be specified in one pragma, separated by a comma(.). For example:

```
#pragma SDS buffer_depth(ArrayName:BufferDepth, ArrayName:BufferDepth)
```

- `ArrayName` must be one of the formal parameters of the function.
- `BufferDepth` must be a compile-time constant value.
- This pragma applies only to arrays that map to BRAM or FIFO interfaces, and specifies the number of hardware buffers to allocate for the array argument, for example, to support pipelining. For a hardware buffer the following must hold:
  - BRAM:  $1 \leq \text{BufferDepth} \leq 4$ , and  $2 \leq \text{ArraySize} \leq 16384$  with data width  $\leq 64$
  - FIFO:  $2 \leq \text{BufferDepth} * \text{ArraySize} \leq 16384$  with data width  $\leq 64$

## Asynchronous Function Execution

These pragmas are paired to support asynchronous invocation of a hardware function.

The syntax of these pragmas is:

```
#pragma SDS async(ID)
#pragma SDS wait(ID)
```

The `async` pragma is specified immediately preceding a call to a hardware function, directing the compiler to return control to the CPU immediately after setting up the hardware function and its data transfers.

The `wait` pragma must be inserted at an appropriate point in the program to direct the CPU to wait until the associated `async` function (and data transfers) have completed.

- The `ID` must be a compile time constant unsigned integer, and represents a unique identifier for the hardware function. That is, using a different `ID` for the same hardware function results in a different hardware instance for the function. Consequently, these pragmas can be used to force the creation of multiple hardware instances.
- In the presence of an `async` pragma, the SDSoC system compiler does not generate an `sds_wait()` in the stub function for the associated call. The program must contain the matching `sds_wait(ID)` or `#pragma SDS wait(ID)` at an appropriate point to synchronize the controlling thread running on the CPU with the hardware function thread. An advantage of using the `#pragma SDS wait(ID)` over the `sds_wait(ID)` function call is that the source code can then be compiled by compilers other than `sdscc` (that do not interpret either `async` or `wait` pragmas).

## Partition Specification

The SDSoC system compilers `sdscc/sds++` can automatically generate multiple bitstreams for a single application that is loaded dynamically at run-time. Each bitstream has a corresponding partition identifier. A platform might not support bitstream reloading, for example, due to platform peripherals that cannot be shut down and then brought back up after reloading.

The syntax of this pragma is:

```
#pragma SDS partition(ID)
```

The `partition` pragma is specified immediately preceding a call to a hardware function, directing the compiler to assign the implementation of the hardware function to the partition `ID`.

- In the absence of a `partition` pragma, a hardware function is implemented in partition 0.
- `ID` must be a positive integer. Partition `ID` 0 is reserved.
- The following example shows an example of using this pragma:

```
foo(a, b, c);  
#pragma SDS partition (1)  
bar(c, d);  
#pragma SDS partition (2)  
bar(d, e);
```

In this example, hardware function `foo` has no partition pragma, so it is implemented in the partition 0. The first call to `bar` is implemented in the partition1 and the second `bar` is implemented in the partition 2.

A complete example showing the usage of this pragma can be found in `samples/file_io_manr_sobel_partitions`.

## SDSoC Environment API

This chapter describes functions in `sds_lib` available for applications developed in the SDSoC environment.

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

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

**void sds\_wait(unsigned int id)**

Wait for the first accelerator in the queue identified by `id`, to complete. The recommended alternative is the use `#pragma SDS wait(id)`, as described in [Asynchronous Function Execution](#).

**void \*sds\_alloc(size\_t size)**

Allocate a physically contiguous array of `size` bytes.

**void \*sds\_alloc\_non\_cacheable(size\_t size)**

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

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

**void sds\_free(void \*memptr)**

Free an array allocated through `sds_alloc()`

**void \*sds\_mmap(void \*physical\_addr, size\_t size, void \*virtual\_addr)**

Create a virtual address mapping to access a memory of `size` bytes located at physical address `physical_addr`.

- `physical_addr`: physical address to be mapped.
- `size`: size of physical address to be mapped.
- `virtual_addr`:
  - If not null, it is considered to be the virtual-address already mapped to the `physical_addr`, and `sds_mmap` keeps track of the mapping.
  - If null, `sds_mmap` invokes `mmap()` to generate the virtual address, and `virtual_addr` is assigned this value.

**void \*sds\_munmap(void \*virtual\_addr)**

Unmaps a virtual address associated with a physical address created using `sds_mmap()`.

**unsigned long long sds\_clock\_counter(void)**

Returns the value associated with a free-running counter used for fine-grain time-interval measurements. The counter counts ARM CPU clock cycles, and wraps to zero.

# SDSCC/SDS++ Compiler Commands and Options

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

---

## Name

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

---

## Command Synopsis

```
sdscc | sds++ [hardware_function_options] [system_options]
[performance_estimation_options] [options_passed_through_to_cross_compiler]
[-sds-pf platform_name] [-sds-pf-info platform_name] [-sds-pf-list] [-target-os os_name]
[-verbose] [ --help] [-version] [files]
```

## Hardware Function Options

```
[-sds-hw function_name file [-files file_list] [-hls-tcl hls_tcl_directives_file]
[-clkid clock_id_number] -sds-end]*
```

## Performance Estimation Options

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

## System Options

```
[[[-apm] [-dmclkid clock_id_number] [-mno-bitstream] [-mno-boot-files]
[-rebuild-hardware] [-poll-mode <0|1>] [-instrument-stub]]
```

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

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

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

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

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

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

---

## General Options

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

### **-sds-pf platform\_name**

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

### **-sds-pf-info platform\_name**

Display general information about a platform and exit (no other options are specified). Use the `-sds-pf-list` option to list available platforms.

### **-sds-pf-list**

Display a list of available platforms and exit (no other options are specified); for example, `sdscc -sds-pf-list`.

## **-target-os os\_name**

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

- `linux` : for the Linux OS. This is the default if the command line contains no `-target-os` option
- `standalone` : for standalone or bare-metal applications
- `freertos` : for FreeRTOS

## **-verbose**

Print verbose output to STDOUT.

## **-version**

Print the `sdscc` version information to STDOUT.

## **--help**

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



## Hardware Function Options

Hardware function options provide a means to consolidate `sdscc` options within a Makefile to simplify command line calls and make minimal modifications to a pre-existing Makefile. The Makefile fragment below illustrates the use of `-sds-hw` blocks to collect all options in the `SDSFLAGS` Makefile variable and to replace an original definition of `CC` with `sdscc` `${SDSFLAGS}` or `sds++` `${SDSFLAGS}`. Thus the original Makefile for an application can be converted to an `sdscc/sds++` compiler Makefile with minimal changes.

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

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

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

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

.PHONY: all

all: ${EXECUTABLE}

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

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

### **-sds-hw function\_name file [-files file\_list] [-hls-tcl hls\_tcl\_directives\_file] [-clkid <n>]] -sds-end**

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

## -files file\_list

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

## -hls-tcl hls\_tcl\_directives\_file

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

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

If the `-hls-tcl` option is used, the user-defined Tcl file is sourced instead of the Tcl file generated by the SDSoc environment. Ensure that the specified Tcl file contains commands that result in a functionally correct directives file. The clock period is platform-specific and reset levels are required to be active-Low.

```
# synthesis directives
create_clock -period <clock_period>
config_rtl -reset_level low
# user-defined synthesis directives
source <user_hls_tcl_directives_file>
# end user-defined synthesis directives
# end synthesis directives
```

## -clkid <n>

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

Platform	Value of <n>
zc702	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zc702_hdmi	1 – 142 MHz
	2 – 100 MHz
	3 – 166 MHz
zc706	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zed and microzed	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zybo	0 – 25 MHz
	1 – 100 MHz
	2 – 125 MHz
	3 – 50 MHz

## Compiler Macros

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

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

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

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

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

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

## System Options

### -apm

Insert an AXI Performance Monitor (APM) IP block to monitor all generated hardware/software interfaces. Within the SDSoC IDE, in the Debug Perspective, you can activate the APM prior to running your application by clicking the **Start** button within the Performance Counters View. For more information on the SDSoC IDE, see the [SDSoC Environment User Guide: Getting Started \(UG1028\)](#).

### -dmclkid <n>

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

Platform	Value of <n>
zc702 platform	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zc702_hdmi platform	1 – 142 MHz
	2 – 100 MHz
	3 – 166 MHz
zc706 platform	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zed and microzed platforms	0 – 166 MHz
	1 – 142 MHz
	2 – 100 MHz
	3 – 200 MHz
zybo platform	0 – 25 MHz
	1 – 100 MHz
	2 – 125 MHz
	3 – 50 MHz

## **-mno-bitstream**

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

## **-mno-boot-files**

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

## **-rebuild-hardware**

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

## **-poll-mode <0|1>**

The `-poll-mode <0|1>` option enables DMA polling mode when set to 1 or interrupt mode when set to 0 (default). For example, to specify DMA polling mode, add the `sdsc -poll-mode 1` option.

## **-instrument-stub**

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

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

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

# Additional Resources and Legal Notices

---

## Xilinx Resources

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

---

## Solution Centers

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

---

## References

These documents provide supplemental material useful with this guide:

1. *SDSoC Environment User Guide: Getting Started* ([UG1028](#)), also available in the docs folder of the SDSoC environment.
2. *SDSoC Environment User Guide* ([UG1027](#)), also available in the docs folder of the SDSoC environment.
3. *SDSoC Environment User Guide: Platforms and Libraries* ([UG1146](#)), also available in the docs folder of the SDSoC environment.
4. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
5. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* ([UG850](#))
6. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
7. *PetaLinux Tools Documentation Workflow Tutorial* ([UG1156](#))
8. [Vivado® Design Suite Documentation](#)
9. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))



---

## 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 [www.xilinx.com/legal.htm#tos](http://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 [www.xilinx.com/legal.htm#tos](http://www.xilinx.com/legal.htm#tos).

© Copyright 2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.