Table of Contents

Introduction 1.1
Installation 1.2
- Linux 1.2.1
- Windows 1.2.2
- Docker 1.2.3
- AWS F1 1.2.4
- Licensing 1.2.5

Guides 1.3
- Concepts & Work Flow 1.3.1
- Quick Start 1.3.2
- Detailed Workflow 1.3.3
  - Creating a New Project 1.3.3.1
  - Software Context 1.3.3.2
  - Hardware Context 1.3.3.3
  - Context Connections 1.3.3.4
  - Device Driver 1.3.3.5
  - VSI Common Interface 1.3.3.6
  - Moving Blocks between Contexts 1.3.3.7
  - Remote Hardware Build 1.3.3.8
  - Setting up Trace 1.3.3.9
- User Guide 1.3.4
- Device Tree 1.3.5
- ZYNQ Tutorial 1.3.6
  - Creating a Zynq Platform 1.3.6.1
  - Creating the Sort Demo 1.3.6.2
- VC709 Tutorial 1.3.7
  - Creating a VC709 Platform for packet processing 1.3.7.1
  - Creating a Packet Processor 1.3.7.2
- Simulation 1.3.8
- System Interface 1.3.9
- Python and Java 1.3.10
- Interfacing with External Systems 1.3.11
  - Using API 1.3.11.1
  - Using TCP Ports 1.3.11.2
  - Using RESTful Endpoints 1.3.11.3

Features 1.4
Troubleshooting 1.5
Releases 1.6
Reference 1.7

Development Workflow 1.7.1
VSI Runtime 1.7.2
Runtime 1.7.2.1
Custom 1.7.2.2
Inout 1.7.2.3
Tcp Server 1.7.2.4
Tcp Client 1.7.2.5
Net 1.7.2.6
Interconnect 1.7.2.7
Connect 1.7.2.8
Log 1.7.2.9
Buffer 1.7.2.10
hls::stream 1.7.2.11
vsi::device 1.7.2.12
Logging 1.7.3
Command Line 1.7.4
Advanced 1.7.5
Templates 1.7.5.1

Introducing Visual System Integrator

Visual System Integrator is an unique tool for embedded/FPGA development which for the first time makes it possible to develop a full functioning system from prototyping to a working product within the same cockpit.

Visual System Integrator lets you define the topology (components, interfaces and data types) of the entire system using GUI or textually (C++).

An end system may contains multiple CPUs, FPGAs and DSPs spanning across physical machines from multiple vendors with varying toolsets. The generated system provides component/interface level instrumentation, trace and performance matrices out of the box. It enables the System Architects to make informed decisions with real performance and throughput matrices and enables the Product Owners to reduce time-to-market and faster development of new features.

Visual System Integrator has common software interfaces such as TCP/UDP/IP, PCIe, OpenAMP and Xilinx AXI builtin. It provides the entire Vivado IP FPGA catalog as well as supports Verilog/VHDL packaged IPs.

In addition any C/C++, Java or Python code can be effortlessly imported as a Software Component. If the imported code is synthesizable then it can be dragged and dropped between a CPU and an FPGA with zero manual wiring.

Because no system works perfectly the first time, VSI have integrated support for Simulators and Emulators. A component can be dragged inside a Simulator/Emulator, leaving the rest of the system as it is, enabling rapid debugging and tracing.

Visual System Integrator is blurring the line between prototyping in high level languages such as Python and months of effort porting the prototyping into the actual code for the end product. With VSI, the effort spent on prototyping is directly reduced from the end product development. It enables you to spend less time in repetitive tasks such as porting and test-benches and more time developing features and increasing robustness of your product.

Overview

Visual System Integrator installation is supported on Windows and Linux.

In addition an **Amazon AWS AMI** is also provided (with an alternative licensing model(pay per hour)) which can be used independently or in combination with AWS F1 FPGA Instances.

We also have a prebuilt **Docker image** (beta) which can be used on MacOS.

Requirements (excluding Amazon AWS AMI)

Before you start, make sure that your system meets the minimum hardware requirements as below:

Hardware Requirements:

- Dual core Intel Processor (quad-core or better recommended)
- 4GB of RAM (8GB or more recommended)
- 50GB of free disk space for Vivado installation (this does not apply if you already have a recent Vivado installation 2016.4 or above).

Software Requirements:

- Vivado HLx System Edition 2016.4 or newer: Download here (registration required)
- Xilinx SDK (Only required if it wasn't previously selected during Vivado Installation): Download here (registration required)

Before you begin

Download installation files for Visual System Integrator. Choose the version that corresponds to the Vivado version that you have installed.

- VSI installation files are available for:
  - Download link for Visual System Integrator 2016.4 or newer (works with corresponding Vivado HLs System Edition 2016.4 or newer)
- License for Visual System Integrator request one here.

Host specific instructions

- Windows
- Linux
- Amazon AWS
- Docker

This document explains how to install Visual System Integrator on Ubuntu Linux Host. The instructions might also work for other linux variants but we have only tested these instructions on Ubuntu Desktop 14.04 and 16.04.

Prerequisites

- Linux Kernel 3.x[1]
- Vivado HL System Edition 2016.4 or newer: Download here (registration required)
- Xilinx SDK (Only required if it wasn't previously selected during Vivado Installation): Download here (registration required)

Instructions

Options

- Use the bash script to download and install Visual System Integrator automatically.
- Manually follow steps to extract and install Visual System Integrator.

Install using the script

Coming soon

Manual Steps

1. Download the installation file from Here (Make sure to choose the major version that corresponds to the Vivado version that is installed on the host)[2]
2. Extract the compressed file in a directory of your choice (example: /opt/Systemview/VSI).
3. set environment variable `VSI_INSTALL` which should point to the directory where you extracted the file
   - bash: `export VSI_INSTALL=/opt/Systemview/VSI` (to make the change permanent, add this line to your ~/.bashrc)
4. Append $VSI_INSTALL/host/linux_x86_64/bin to your $PATH to be able to launch VSI from commandline
   - bash: `export PATH=$PATH:$VSI_INSTALL/host/linux.x86_64/bin` (to make the change permanent, add this line to your ~/.bashrc)
5. Optional: if you have already received your license, copy it to $VSI_INSTALL directory.
6. Open a terminal and issue the command `vsi`. If the installation was successful then you should see text similar to this:
   - `Visual System Integrator version: V2017.1_HEAD-29-gdc85abc, Compiler: "GNU - 5.4.0 20160609", Buildhost: "nanl - Linux-4.4.0-72-generic", Date: 2017-04-07T17:24:29`
     `loading Visual System Integrator...`
7. If you don't see similar text then see Troubleshooting

Footnotes

1. Typically a modern linux distribution released within the last 5 years should work; we recommend Ubuntu 14.04 LTS or newer (or Other Ubuntu derivatives Xubuntu, Kubuntu,...)
2. Certain VSI features are only available on newer releases. For a feature comparison matrix, look here

This document explains how to install Visual System Integrator on a Windows 10 x64 Pro Host. These instructions might work on other Windows variants however we have only tested them on Windows 10 x64.

Prerequisites

- Windows 10 x64 Home, Pro or Enterprise Edition[1]
- Vivado HL System Edition 2016.4 or newer: Download here (registration required)[2]
- Xilinx SDK : Download here (registration required)[3]

Instructions

Options

- Use the Powershell script to download and install Visual System Integrator automatically.
- Manually follow steps to extract and install Visual System Integrator

Use Powershell Script

Coming Soon

Manual Steps

1. Download the installation file from Here (Make sure to choose the major version that corresponds to the Vivado version that is installed on the host)[4]
2. Extract the compressed file in a directory of your choice (example: C:\Systemview\VSI).
3. set environment variable `VSI_INSTALL` which should point to the directory where you extracted the file
   - command prompt: `setx VSI_INSTALL C:/Systemview/VSI` (to make the change permanent, add it to system->advanced->environment variables)
4. Append $VSI_INSTALL/host/linux_x86_64/bin to your $PATH to be able to launch VSI from commandline
   - command prompt: `setx PATH %PATH%;%VSI_INSTALL%\host\windows.x86_64\bin` (to make the change permanent, add it to system->advanced->environment variables)
5. Optional: if you have already received a VSI license, copy it over to %VSI_INSTALL% directory.
6. Run VSI: Open a command prompt window and issue the command `vsi.exe`. If the installation was successful then you should see text similar to this:
   - `Visual System Integrator version: V2017.1_HEAD-29-gdc85abc, Compiler: "GNU - 5.4.0 20160609", Buildhost: "nanl - Linux-4.4.0-72-generic", Date: 2017-04-07T17:24:29`
     `loading Visual System Integrator...`
7. If you don't see similar text then see Troubleshooting
8. Windows HLS build fail, There is a walk around solution.
   - C:\Xilinx\Vivado\2017.4\win64\tools\clang\include\c++\4.5.2\exception*ptr.h add the below code before extern "C++": #ifdef _clang* class type_info; #endif
   - C:\Xilinx\Vivado\2017.4\win64\tools\clang\include\c++\4.5.2\nested_exception.h delete the following code: template inline void __throw_with_nested(_Ex&& __ex, const nested_exception* = 0) { throw __ex; }

Footnotes

1. Pathname longer than 256 characters are not supported on Windows 10 Home Edition.
2. Make sure to select SDK in installation options
3. Only required if Vivado was preinstalled without the Embedded SDK or if it was not selected during Vivado Installation.
4. Certain VSI features are only available on newer releases. For a feature comparison matrix, look here

About Docker

Docker is a tool that can be used to package up and sandbox applications to be run at any host for which docker runtime is available. As of now, docker runtime is available for Windows, Linux and MacOS.

Visual System Integrator + Docker

A docker container for VSI contains all the underlying dependencies required to run a VSI. It is a quicker method to test drive VSI or to programmatically install it on multiple hosts. In order to use the VSI container, you only need run `docker run gcr.io/systemviewinc/vsi:{version_number}` .

Requirements

- Docker: Download here[1]
- MacOS only: resize preallocated docker disk size to 110GB or more. instructions

Pull Visual System Integrator

Run `docker pull gcr.io/systemviewinc/vsi:{version_number}` and wait for the download to complete.

> Warning: This will download 50GB+ of data. Make sure that you are on a fast connection.

Docker + Visual System Integrator getting started

1. Run `docker run --privileged -dP -e 1920x1080 -v /home/{USER}/{some projects directory to map}:/projects gcr.io/systemviewinc/vsi:{version_number}` [2] [3] An instance of VSI docker container will start in detached mode. The returned hash is the container ID.
2. Run `docker port <container ID>` . This will return port numbers.
3. Use a VNC client to connect to VSI. The connection should be made to `localhost:<port mapped to 5901>` (MacOS comes with a preinstalled VNC client `Screen Sharing`).[4]
4. Optional: Another mapped port to ssh is available if commandline access is desired. Use the port mapped to 22 in previous steps. `ssh localhost -p <port mapped to 22>` [4]

Licensing

1. Open a command prompt inside the VSI container and use `ip address` to get the MAC Address.
2. Use the form here to request a license.
3. The license file, once received, can be used by `docker cp vsi.lic.<ethernet Address> <container id>:/opt/Systemview/VSI/{version_number}/` . This will copy the file to the VSI container.

Usage

1. Connect to the container using VNC.
2. Open a terminal and issue the command `vsi` . If the installation was successful then you should see text similar to this:
   - `Visual System Integrator version: V2017.1_HEAD-29-gdc85abc, Compiler: "GNU - 5.4.0 20160609", Buildhost: "nanl - Linux-4.4.0-72-generic", Date: 2017-04-07T17:24:29`
     `loading Visual System Integrator...`
3. If you don't see similar text then see Troubleshooting

Footnotes

1. Only required if docker wasn't previously installed.

2. 1920x1080 can be replaced with your native display resolution.
3. The `-v` switch can be used to map a local directory to a path inside the docker container. This can be used as the project directory or to save important user data that has to outlive the life of the docker container.
4. The password is `systemview`
5. VSI can be started without a valid license. All functionality works except the ability to generate projects.

Run Visual System Integrator on AWS EC2

Requirements

- An active AWS account with billing and the ability to launch paid instance.
- For Graphical User Interface, a Web Browser or a VNC Client
- While we set no restrictions on the type of instance that VSI runs on, for best use experience, we recommend using an instance with 8+ virtual cores and 30GB+ of RAM.
- In order to reduce User Interface lag, a fast connection to the internet is highly desirable.

Launching Visual System Integrator on AWS

1. Launch a VSI FPGA Developer Instance using AWS console or cli tool.[1, 2]
2. Once the the instance is running, connect to the instance using an ssh client [3].

   NOTE: The first login will take a while due to the AWS EC2 F1 HDK Shell initialization. Please be patient. When the initialization is complete)

Optional: Using the graphical session

1. Set the password for graphical session through VNC (hint: `vncpasswd`).
2. The machine has to be rebooted once for the graphical session to work. (hint: `sudo reboot` from the ssh client)

Through a Web-browser

To use VSI in graphical mode through a web browser, a local port has to be forwarded to the instance's port 80. The following syntax with the command line ssh client forwards local port 8080 to instances port 80:

```
ssh -L 8080:localhost:80 centos@<instance DNS name or IP Address>
```

In case if putty is being used on windows, use the plink.exe from command line (should be in the same directory as putty.exe). Similar syntax can be used to connect to the instance:

```
plink.exe -L 8080:localhost:80 centos@<instance DNS name or IP Address>
```

NOTE: The local port 8080 is used as an example. If another program on your computer is using it, use an alternative port that is free.

Once the ssh client has connected, issue the command `vncpasswd` to set a password for VNC session.

Open a web browser to the following URI http://localhost:8080. The password set earlier needs to be used to authenticate.

Through native VNC client:

To connect using a native VNC client, the following syntax can be used:

```
ssh -L 5901:localhost:5901 centos@<instance DNS name or IP Address>
```

or

```
plink.exe -L 5901:localhost:5901 centos@<instance DNS name or IP Address>
```

Once connected, use the `localhost:5901` through your VNC Client to connect to the Graphical session. The password set earlier needs to be used to authenticate.

For walk throughs and guides, consult the Visual System Integrator docs.

Notes on Security:

- The graphical session is tunneled through the SSH tunnel and is encrypted between your local computer and the instance.

- The graphical session password can be changed using `vncpasswd` from the ssh client.
- The VNC server listens to instance's local address only (127.0.0.1/localhost) in order to enforce encryption through a tunnel.

Footnotes

1. Visual System Integrator at AWS Marketplace
2. AWS: Launch an AWS EC2 instance
3. AWS: Connecting to Your Linux Instance Using SSH

Licensing

Make sure that you have a working installation before following the steps in this section.

While designing a system using VSI drag and drop interface is free; generating a system requires a license. We provide a simple and automatic evaluation license for each new installation. The license is initially valid for 30 days and can be extended up-to 90 days.

In order to manage license, use the Visual System integrator launcher tool (`vsi` or `vsi.exe`)

The following commands are supported:

```
vsi -L, --license             query license status
vsi -r, --refresh-license     refreshes the existing license period. The license needs to be active before it can be
refreshed
vsi -R, --request-license     request an evaluation license for this node
```

After the installation on a new machine that doesn't have a license, use the `vsi -R` to request an evaluation license. Please fill the fields as accurately as possible as it will help our support staff to facilitate an extension of the evaluation license later, should you request it. The extension can be requested using this form (make sure that you use the same `Name` and `Company` fields. The existing license can be queried using `vsi -L`). Note that the initial 30 day license can only be requested for a machine that has never been issued a license before. (contact us using the license form if you need an extension)

Once the fields are filled, the information is used to automatically provision a license. Later, if the license was extended (either due to evaluation extension or paid conversion), the updated license can be refreshed using `vsi -r` . This will download the updated license and install it.

Guides

We have guides for the following platforms

- Zynq
- VC709

Visual System Integrator

Concepts

The Visual System Integrator (VSI) has two canvases:

- Platform Canvas
- System Canvas

The system design process in VSI begins by creating a platform. The platform consists of one or more “Execution context”. An “Execution Context” is defined as an entity in which a Hardware (RTL) or a Software (C,C++,Java) block can be placed for execution. The “Platform” defines the “Execution Contexts” that will be used in this System and how they are connected. VSI currently supports the following “Execution Contexts”:

- Software
  - X86
  - ARM-32 bit
  - ARM-64 bit
  - ARM-Cortex-R5
- Hardware
  - All Xilinx FPGAs supported by Vivado
  - Simulator (XSIM, Modelsim)

For Hardware Contexts blocks / interfaces that require timing/location constraints must be placed in the “Execution Context”.

Work Flow

Figure 1 shows the “Work Flow” when using VSI. The user starts by Creating a platform or Importing an existing platform. Once the Platform is Created/Imported the user then “Compiles” the platform. The next step is to create the “Application System” canvas and import the Platform definition. The application development involves importing RTL or C/C++/Java blocks of code with well-defined interfaces and connecting these interfaces to form the Dataflow graph of the Application System. The VSI System Compiler and the VSI Runtime will move the data between these blocks as they execute in their assigned “execution contexts”. With some restrictions, the blocks can be moved between “Execution Contexts”. Once the application system is developed the user can “Generate System” , the VSI System Compiler will generate Vivado IPI Projects (for Hardware Contexts) and CMake based projects (for Software Contexts); there are no restrictions to the number of “Execution Contexts” in the system.

Visual System Integrator: Work Flow
Describe Hardware Platform (Import Existing)
Compile Platform
Platform Meta data
Import Platform
Develop Application System
Compile Generate System
Software Projects
Eclipse, Qt, VC++, ...
Runtime , Drivers & OS Configurations ...
FPGA Projects
System View Inc. Confidential

Visual System Integrator: Quick Start

This Quick Start will describe how to create a simple system using Visual System Integrator.

Start the VSI software

To start the tool just type:

```
${VSI_INSTALL}/host/<HOST_TYPE>/bin/vsi
```

> ${VSI_INSTALL} – path to VSI installation
> HOST_TYPE – linux.x86_64 | windows.x86_64

Import Existing platform

In this guide we will import an existing platform.

1. ***Quick Start > Open Example Project***
2. Choose a name, location, and a default Xilinx board for your project.
3. Select **zynq** from platform_list. This will create a Platform with two “Execution Contexts” . A Software context `“zynq_ps”` and a Hardware context `“zynq_pl”`.
4. ***Click Flow > Compile Platform*** to compile the platform.

Application System Canvas

1. Clicking ***Flow > Create System*** to create an "Application Canvas". This will create a blank “System Canvas”.
2. Clicking ***Flow > Import Platform*** to import the platform definition. After importing the platform, the execution contexts show up as blank entities.

Develop Application

1. Import a Block From Library:

- Double click anywhere in the software context enter into it.
- Right-Click and select ***“Add IP“***.
- Select ***“VSI TCP Server v1.0”***.
  This will place the TCP/IP server in the System Canvas.

2. Import C/C++ code

We will use the “Software Import Wizard” to import a “sort” function as a block.

- Right-click and select ***“Add IP”***
- Select **“VSI Software Import Wizard”**.
- Double-Click on **vsi_gen_ip_0** to import the “sort” C function.
  - The “Source Directory” box either “Browse” to or enter the directory *${VSI_INSTALL}/target/common/hls_examples/sort*
    The “Software Import Wizard” will call a “built-in” C/C++ parser and present drop-down list of functions it finds (C/C++ Function Name).
  - Choose the sort function, and navigate to the “Arguments” section.

The "Arguments" section will show the two arguments `in_arr` and `out_arr` with some default values filled in.
- Mark `in_arr` as an *"Execution Trigger"*; this indicates that the VSI runtime will execute this function whenever data is available in this input interface.
- Mark the "Direction" of `out_arr` as *"output"*. The sort function puts the output into this buffer when it finishes execution.
- Change "Access Type" of `out_arr` to *"Random"*. The sort algorithm accesses the output array in a non-sequential fashion.
- Click OK. The interfaces of the block are now updated to reflect the name and direction specified in the "Arguments" section.

3. Make connection between graphical blocks

- Connect the `server_data_out` interface of the ***TCP/IP server*** to the `in_arr` interface of the ***vsi_gen_ip_0*** .
- Connect the `out_arr` interface of the ***vsi_gen_ip_0*** block to the `server_data_in` interface of the ***TCP/IP server*** .

> Some blocks can be moved between execution contexts.

Build Application

1. With the System Canvas open Click on ***Flow > Generate System***
2. Build the High-Level Synthesis blocks by clicking on ***Flow > Build HLS***
3. Build the executables for Software Contexts by clicking on ***Flow > Build Software Contexts*** .
4. Building the Hardware Project by clicking on ***Flow > Build Hardware***.
   - VSI also supports remote hardware build.

This completes the build process for the example project.

Execute Application

The following steps showing how to execute the generated project on a `Zynq-SoC` Based MicroZed board.

1. Prerequisites:
   - Build the Linux kernel and root file system for the Board you are using
   - Cloning repository https://github.com/systemviewinc/fpga_driver.git to your home directory
   - Source the xilinx environment settings:

     ```
     source /opt/Xilinx/Vivado/<vivado_version>/settings64.sh
     ```

   - Build the driver:

     ```
     make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm KERNEL_SOURCES=<home_dir>/build/driver_vsi/linux-xlnx
     ```

2. Login to your board and create a directory in the root-file-system, this is an optional step. It will help you isolate the files being copied.
3. Coppying the following files from the build directory to the target filesystem:
4. The bitstream file. This file can be found at */zynq_pl/zynq_pl.runs/impl_1/zynq_pl_wrapper.bin*.[1]

   > For `Yocto Linux` kernels. The generated Bitstream file is ***zynq_pl_wrapper.bit*** that can be found at */zynq_pl/zynq_pl.runs/impl_1/zynq_pl_wrapper.bit* .

5. The driver installation script found at */vsi_auto_gen/sw//driver.sh*

   > */vsi_auto_gen/sw/zynq_ps/driver.sh*

6. The Software Executable */vsi_auto_gen/sw/build//bin/*

   > */vsi_auto_gen/sw/build/zynq_ps/bin/zynq_ps*

7. The device driver ***"vsi_driver.ko"*** created in the previous step.

Visual System Integrator: Getting Started Guide

Introduction

This document will describe how to create a simple system using Visual System Integrator. The document assumes that VSI is installed, please follow instructions at http://docs.systemviewinc.com to install the product. In this document, we will describe how to build a Zynq 7000 based system.

Concepts

The Visual System Integrator (VSI) has two canvases:

- Platform Canvas
- System Canvas

The system design process in VSI begins by creating a platform. The platform consists of one or more “Execution context”. An “Execution Context” is defined as an entity in which a Hardware (RTL) or a Software (C,C++,Java) block can be placed for execution. The “Platform” defines the “Execution Contexts” that will be used in this System and how they are connected. VSI currently supports the following “Execution Contexts”:

- Software
  - X86
  - ARM-32 bit
  - ARM-64 bit
  - ARM-Cortex-R5
- Hardware
  - All Xilinx FPGAs supported by Vivado
  - Simulator (XSIM, Modelsim)

For Hardware Contexts blocks / interfaces that require timing/location constraints must be placed in the “Execution Context”.

Work Flow

Figure 1 shows the “Work Flow” when using VSI. The user starts by Creating a platform or Importing an existing platform. Once the Platform is Created/Imported the user then “Compiles” the platform. The next step is to create the “Application System” canvas and import the Platform definition. The application development involves importing RTL or C/C++/Java blocks of code with well-defined interfaces and connecting these interfaces to form the Dataflow graph of the Application System. The VSI System Compiler and the VSI Runtime will move the data between these blocks as they execute in their assigned “execution contexts”. With some restrictions, the blocks can be moved between “Execution Contexts”. Once the application system is developed the user can “Generate System” , the VSI System Compiler will generate Vivado IPI Projects (for Hardware Contexts) and CMake based projects (for Software Contexts); there are no restrictions to the number of “Execution Contexts” in the system.

start the vsi software

```
>${vsi_install}/host/<host_type>/bin/vsi
```

${VSI_INSTALL} – path to VSI installation

HOST_TYPE – linux.x86_64 | windows.x86_64

Now you can Create A New Project or continue with this guide to import an existing example.

Platform

In this guide we will import an existing platform.

Import Existing platform

Quick Start --> Open Example Project…

Choose a name, location, and a default Xilinx board for your project. Then choose zynq from platform_list.

This will create a Platform with two “Execution Contexts” . A Software context “zynq_ps” and a Hardware context “zynq_pl”.

Context connections

Expanding and zooming into each context we see that the "Device Driver" block in the "Software Context" is connected to the "VSI Common Interface" in the Hardware Context. This describes the connectivity between the two contexts. The connection represents a logical connection between the two contexts and allows the VSI system compiler to determine the connectivity.

Compile Platform

This completes the Platform Import process. The platform now needs to be "Compiled" . Click Flow --> Compile Platform to compile the platform.

Application System Canvas

Create Application System Canvas

The Application Canvas is created by Clicking Flow --> Create System …

This will create a blank “System Canvas”

Import Platform

We will “Import” the platform definition.

After we import the execution contexts show up as blank entities.

Develop Application

In VSI the application development process involves importing/creating blocks and connecting the interfaces of the blocks to represent the dataflow model of the entire system. The Blocks can be imported from a Library (these blocks have predefined interfaces). The user can also choose to import pre-existing C/C++/Java functions into the canvas using the “Software Import Wizard”, the arguments of the functions chosen to be imported become the interfaces of the imported block. See “VSI User Guide” for more details on the Software Import Wizard.

Import Block From Library:

We will import a TCP/IP server from the Library in the Software Context “zynq_ps”, this block will be used to send and receive data from the external host. Double click anywhere in the software context enter into it. Right-Click and select “Add IP … “.

Select “VSI TCP Server v1.0”.

This will place the TCP/IP server in the System Canvas.

Double click on the TCP/IP server block to check the configuration.

In the default configuration, the server will open the port 2020 and wait for connection from “any” host. When a connection request is received, it will accept the connection. There is no predefined protocol, any data received by the server is sent to the interface “server_data_out”. Any data received from the interface “server_data_in” is sent back to the client that is attached to the port. The TCP/IP server acts as a Data interface to the external host.

Import C/C++ code

We will use the “Software Import Wizard” to import a “sort” function as a block . Refer to the Section “Software Import Wizard” in “VSI User Guide” for more details on the Wizard. Right-click and select “Add IP …”

Select “VSI Software Import Wizard” .

Double-Click on “vsi_gen_ip_0” to import the “sort” C function. The “Source Directory” box either “Browse” to or enter the directory $(VSI_INSTALL)/target/common/hls_examples/sort (note you will need to enter the absolute PATH to the directory, environment variables are NOT allowed). The “Software Import Wizard” will call a “built-in” C/C++ parser and present drop-down list of functions it finds (C/C++ Function Name).

Choose the sort function , and navigate to the “Arguments” section. The “Arguments” section will show the two arguments “in_arr” and “out_arr” with some default values filled in.
Update the Fields marked in the screen shots.

- Mark “in_arr” as an “Execution Trigger”; this indicates that the VSI runtime will execute this function whenever data is available in this input interface.
- Mark the “Direction” of “out_arr” as “output”. The sort function puts the output into this buffer when it finishes execution.
- Change “Access Type” of “out_arr” to “Random”. The sort algorithm accesses the output array in a non-sequential fashion. The “Access Type” will ensure that a “Block Ram” interface is generated for this interface when placed in a “Hardware Context”.

Click OK. The interfaces of the block are now updated to reflect the name and direction specified in the “Arguments” section.

Connect the “server_data_out” interface of the TCP/IP server block to the “in_arr” interface of the “vsi_gen_ip_0” block. And connect the “out_arr” interface of the “vsi_gen_ip_0” block to the “server_data_in” interface of the TCP/IP block to complete the System Design.

The VSI runtime will now ensure that when data arrives on the “socket” of the TCP/IP it will be sent to the input of the “sort” function, and when the sort function finishes the VSI runtime will send the data from “out_arr” to the TCP/IP server which will send it back over the “socket” to the connected client. Some blocks can be moved between execution contexts.

Now the application development phase for this example, is complete.

Build Application

Generate Projects for Hardware & Software Contexts.

This step will generate complete projects for all Software & Hardware projects. With the System Canvas open Click on Flow --> Generate System …

All projects are generated. They are placed in the following location

- $< project_directory>/vsi_auto_gen/hls – directory contains all Vivado HLS projects for all Hardware Contexts. It contains a top level makefile. The user can cd to this directory and type “make” and all the HLS projects will be built.
- $< project_directory>/vsi_auto_gen/sw – directory contains the software projects for all the software contexts; each software context has its sub-directory . In this example, there will one directory “zynq_ps”.
- $< project_directory>/vsi_auto_gen/hw – directory contains scripts for creating Vivado IPI project for each Hardware Context.

Build the High-Level Synthesis blocks.

This step will call Vivado HLS under the hood to create RTL blocks for all synthesizable C/C++ blocks that are in Hardware Contexts. Please note this step might take a few minutes. With the System Canvas open click on Flow Build HLS …

Build the executables for Software Contexts.

In this step, we will create the executables for all the Software Execution contexts. With the system canvas, open click Flow --> Build Software Contexts….

This call the compiler or cross-compiler specified in the context definition and generate the executables for each of the software contexts in the project. The executable is placed the directory $< project_directory>/vsi_auto_gen/sw/< system_directory>/build/< software_context_name>/bin/< software_context_name>. In this example, the executable will be $< project_directory>/vsi_auto_gen/sw/system_1/build/zynq_ps/bin/zynq_ps .

Building the Hardware Project:

To build the hardware, select Flow --> Build Hardware….

This completes the build process for the example project.

Execute Application

In this section, we will go through the steps of executing the generated project on a Zynq-SoC Based MicroZed board.

Prerequisites:

- Build the Linux kernel and root file system for the Board you are using
- Cloning repository https://github.com/systemviewinc/fpga_driver.git to your home directory.
- Source the xilinx environment settings: source /opt/Xilinx/Vivado/< version>/settings64.sh
- Build the driver with make:make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm KERNEL_SOURCES=/< home_dir>/build/driver_vsi/linux-xlnx

Copying Files:

Login to your board and create a directory in the root-file-system, this is an optional step. It will help you isolate the files being

copied. Four files need to be copied from the build directory to the target filesystem.

- The bitstream file
  - For Non-Yocto linux kernels. The generated Bitstream file is zynq_pl_wrapper.bin. This file can be found at < hw_project_directory>/zynq_pl/zynq_pl.runs/impl_1/zynq_pl_wrapper.bin.
  - For Yocto Linux kernels. The generated Bitstream file is zynq_pl_wrapper.bit. This file can be found at < hw_project_directory>/zynq_pl/zynq_pl.runs/impl_1/zynq_pl_wrapper.bit.
- The driver installation script found at $< project_dir>/vsi_auto_gen/sw//driver.sh ($< project_dir>/vsi_auto_gen/sw/zynq_ps/driver.sh)
- The Software Executable $< project_dir>/vsi_auto_gen/sw/build//bin/ ($< project_dir>/vsi_auto_gen/sw/build/zynq_ps/bin/zynq_ps)
- The device driver “vsi_driver.ko” created in the previous step.

Start Application on Target

- The following step will configure the FPGA with the bitstream copied in the previous step
  - cat zynq_pl_wrapper.bin > /dev/xdevcfg
- Load the driver , you might need to change the execute permissions “chmod +x ./driver.sh”
  - ./driver.sh
- Start the application
  - ./zynq_ps

Send Data to Application

At this point the application is up and running and is waiting for data on the TCP/IP Socket . System View provides two applications to send data to the target. The source code for both can be downloaded from http://release.systemviewinc.com : TCP Clients [Folder]

Python client

echo_client.py is a simple python program that will connect and send data to a TCP/IP server to run this client on the host from which you want to send the data. python echo_client.py < zed ip address> 2020 < number of times to send data>

` ex: python echo_client.py 192.168.2.21 2020 1

Output should look like this: 0 connecting to 192.168.2.21 port 2020 sending "4096" total receiving "4096" total closing socket *Checking Data** TX data length: 1024 RX data length: 1024 *RX data matches TX sorted data*`

Socket C Client

This client will read from a file and send the data to a TCP/IP server , the source code is provided “sock_send_recv_file.c”, it needs to be compiled on the host that will send the data to the target.

```
Compile: gcc –o sock_send_recv_file sock_send_recv_file.c –lpthread Execute : sock_send_recv_file <ip_address> <port_no> <input_file> <output_file> <packet_size> sock_send_recv_file 192.168.2.21 2020 sort.in sort.out 4096
```

Running Trace

If any interface in the system design is annotated for “trace” then a trace server is automatically built-into the application. Download the latest trace application from http://release.systemviewinc.com : Trace Application [folder] for the appropriate host. Once the application is running in the previous step, the trace application can be connected to the application to view realtime transaction trace information for the complete system.

Enter the IP Address of the target and press connect to start trace data collection & display.

Create a New Project...

Choose a Name & Location

Choose The “Board”

And finish creating the project.

Software Context Details

Expand the Software Context “zynq_ps” by clicking on the

. Then double-click on the "vsi_context_ARM" icon

to inspect the context details.

- Context Type : Software
- Hostname : can be replaced by the ip address of the context
- Include Directories : comma separated list of directories that will be used to search for header files.

And to inspect the software parameters.

- CPU Type : ARM, X86, ARM-64, ARM-R5
- Operating System : Linux/Freertos (currently FreeRtos is supported for ARM-R5 only)
- C++ Compiler Options to use when the project is created for this Software Context.
- C Compiler Options to be used when the project is created for this Context
- Use Cross Compiler : this is the Cross Compiler Prefix prepended to gcc or g++ when the project is generated for this context

Hardware Context Details

Expand the Hardware Context “zynq_pl” by clicking on the

. Then clicking on the "vsi_context_zynq_1" icon

.

- Context Type : Hardware
- Hostname : is replaced by the ip address if this Hardware context is to be run the simulator , otherwise ignored.

- FPGA Part Number : for this hardware context
- FPGA Board : being used for this project.
- Simulator to use.
- Simulation/Synthesis Design Top : Name of the module to be used as “Top” . The design generated by VSI need not be the top-level module. Top level module must instantiate the design generated by VSI, if this field is left blank then it is assumed that the VSI generated design will be the top-level.
- Simulation/Synthesis Defines & Parameters : is a comma separated list parameter value pairs these will be passed to the Synthesis and Simulation projects. E.g. INX_SIMULATOR=1,DDR4_4G_X16=1
- Synthesis Source Directories : Comma separated list of directories. All sources from these directories will be added to the Synthesis project, in addition to the sources generated by the VSI compiler. Can have environment variables for e.g. $(VSI_INSTALL)/syn_src
- Synthesis Include Directories : Comma separated list of directories. These include directories are added to the search path of the synthesis project. Can contain environment variables.
- Simulation Source Directories : Comma separated list of directories. All sources from these directories will be added to the Simulation project, in addition to the sources generated by the VSI compiler. Can have environment variables for e.g. $(VSI_INSTALL)/sim_src
- Simulation Include Directories : Comma separated list of directories. These include directories are added to the search path of the Simulation project. Can contain environment variables.
- Simulation file list : is a file that contains a list of files . The file must contain only one filename per line. When this file is provided the “automatic” source code ordering is disabled for this project.

Context connections

Expanding and zooming into each context we see that the "Device Driver" block in the "Software Context" is connected to the "VSI Common Interface" in the Hardware Context. This describes the connectivity between the two contexts. The connection represents a logical connection between the two contexts and allows the VSI system compiler to determine the connectivity.

Device Driver

The user can configure various parameters of the device driver using the Device Driver configuration panel.

- Driver Library Name : Is the kernel Image name which will be installed when the application program for this context is started.
- Major Number : For Driver Type “Zynq”, “PCIe” & “OpenAMP” the driver uses a “File” based mechanism to communicate with the hardware. The Major number species the Major number to use for this driver in the linux kernel, default value is 241 can be changed if this major number is already being used by another driver in the system.
- Size of DMA Buffer: The DMA during initialization allocates a contiguous memory buffer to communicate with the DMA. This parameter determines the total memory allocated for this buffer.
- Buffer Size for each file : Each file is allocated a certain amount of buffer space from the DMA buffer mentioned before; This parameter determines the amount of buffer per file.
- Driver Type : Lets the VSI System Level compiler decide the driver specific mechanism to use when communicating with this driver.
    - ZYNQ (Connected to FPGA Over AXI Interfaces)
    - PCI/E (Connected to FPGA Over a PCIe Link)
    - OpenAMP (Connected to Another Software Context Using Open APM)
    - Simulator (Connected To A simulator using Remote Procedure Call [RPC])
- Interrupt controller Address : Base address of the interrupt controller in the VSI Common interface (typically should not change).

- Number of CDMAs – Choose the number of CDMAs in the VSI Common interface.
- CDMA N^{th} address – Address of the Nth CDMA's control port in the VSI Common Interface (typically should not change).

- PCIe Device ID : For Driver Type PCI/e this parameter should match the hardware PCI/e core configuration.
- PCIe Control Port Address: The driver internally uses this value to determine if a DMA will be needed to transfer the data , or a “direct” read write operation can be performed.

VSI Common Interface

VSI Common Interface is the Hardware (FPGA) side of the connectivity. It is a collection of DMAs and an Interrupt controller and can operate with ALL versions of ZynQ & over PCI/e. The following screenshot shows the configuration dialog for the VSI Common Interface.

The current version of the VSI Common Interface can be configured with upto 2 DMAs; future versions will have more DMAs. The DMAs communicate with the System Memory using the DMA*[n] interfaces , these interfaces should be connected to PCI/e Slave Port, or the Zynq HP[n] ports. The M_AXI*[n] ports are left open and used by the VSI System Compiler to connect processing blocks that are placed in the Hardware execution Context. The CTRL is used by the compiler to connect the AXI – Lite control interfaces. Interrupts from the blocks are connected by the System Compiler to the open interrupt input irq_i_[n]. The output interrupt from the VSI Common Interface is connected to the input of the Zynq processing block or the PCI/e block's interrupt input.

Moving Blocks between Contexts:

Some blocks can be moved between execution contexts. In this example the “sort” function code is synthesizable to hardware using the Vivado HLS C RTL compiler and hence can be moved to Hardware Context (“zynq_pl”).

Go back to the system level diagram by clicking the “system_1” tab in the Diagram view.

Expand both the contexts by clicking on the . Select the “vsi_gen_ip_0” block and drag it inside the Hardware Context (“zynq_pl”) and release it. Not all blocks can be moved between contexts, an attempt to move the “TCP/IP” server block will result in error.

When the VSI System Compiler detects an interface crossing between two “Execution Contexts”, it will configure the runtime environment on the Software Context (“zynq_ps”) to communicate with the “Device Driver”. On the Hardware Context it will connect the interfaces to the “vsi_common_interface” and setup the driver for the communication channels. Each interface that crosses the boundary is treated a separate channel. In this example, we have two interfaces that are crossing the boundary, hence two communication channels will be setup for the driver. There are no limits to the number of interfaces that can cross between contexts.

Remote Hardware Build

Launching Hardware Builds on Remote Linux Hosts

VSI supports simultaneous remote hardware builds on multiple Linux hosts.

Prerequisites

On the remote host:

- VSI installation is assumed to be available from the login shell
- The mailutils package needs to be installed to enable sending email after each job is done.

On the local host:

- Configure SSH so that the host does not require a password each time you launch a remote run.
  - Copy the public key to remote-host using: `ssh-copy-id -i ~/.ssh/id_rsa.pub <remote-host_name>`
- Install expect
  - On Ubuntu `apt install expect`

Launch Build

To launch a build, simply add the hostname and the email address to the vsi_build_hardware command as follow: `vsi_build_hardware {<system_name>} {<remote_host_name>} {<user@example.com>}`

When the remote build is completed or failed, an email will be sent to the email address. If the build was successful, the email contains the bitstream file and the location of it at the remote host.

Setting up Trace

Any interface in the VSI system diagram can be instrumented for “trace”. To enable trace on an interface right-click on the interface connection and click “Toggle Trace”.

A special icon will appear on all the interfaces annotated for trace.

arg_2_seq_o

SOFTWARE IMPORT WIZARD

Introduction

VSI software import Wizard, allows users to import C/C++ functions into the VSI dataflow environment. The user can specify multiple source directories, a built-in C/C++ parser will parse all source files in these directories and create a list of functions the user can choose from. Once the user chooses the function associated with the block, the wizard will prefill most of the information “Arguments” section. Figure below shows the screen shot of the Software Import Wizard. The following sections will describe the wizard and each field in greater detail.

Top Level Tab

Source Directory:

The Source directories can be selected by “Browsing” or by entering a comma separated list of directories. A built-in “clang” parser is invoked every time the directory field changes. All files C/C++ files in the directory list are parsed by the built-in parser and a function list is created. The label changes to red (“Top Level”) if the parser encounters an error, correct the parse error to continue. The built-in parser can be run from the command line to determine the cause of the error. To run the parser from the command line, use the following command *$(VSI_INSTALL)//bin/vsi_clang \< source_files> [-I \< include_dir>]

\< HOST> = linux.x86_64 | windows.x86_64

--help* will list the available options.

By default the parser will add the “Vivado HLS Include” path to the include directory path; all other include files are assumed to be present in the directory list provided. The source files from these directories will be copied to the project directory, unless **vsi::src_link_files** is called from the TCL console, in which case a “soft link” will be created from the project directory to the source directories. The option set by **vsi::src_link_files** is persistent for a given project it can be switched back by the command **vsi::src_copy_files**.

C/C++ Function Name:

This dropdown list is created by the built-in parser. Choose the function associated with the block. By default the function is expected to be **non-blocking**. The VSI compiler will generate a wrapper function which will call the chosen function; the input arguments are updated by the **vsi::runtime** environment before the function is called, the output arguments are read and send to the input of the connected function by the **vsi::runtime** when the function returns. The vsi::runtime supports other execution models as well. Complex systems sometimes require functions to have feedback/dependency between each other; i.e. a function can only proceed depending on the state of another function. This can be achieved by using Alt-***hls::stream*** or ***vsi::device*** as argument types. Please refer to section hls::stream and vsi::device for more details on these execution models.

Number of Parameters:

This field is automatically updated when the function is chosen from the drop-down list and represents the number of arguments the function has. The maximum number of arguments allowed is 16 (8 for versions 2016.4 and earlier).

Allocate on Local Stack:

The vsi:runtime will create a wrapper function which calls the C/C++ function specified in the “**C/C++ Function Name**” field. The wrapper function allocates some local variables to process the inputs and outputs of the functions. By default these variables will be allocated on the stack, on some target architectures (such as Cortex-R5) running real-time operating systems, the thread stack is limited; by unchecking this box the user can force the runtime to a declare the local variable as “***static***” which will cause them to be allocated on the global memory area.

Thread Allocation:

The **vsi::runtime** will execute a function on its caller’s thread by default. It will create a separate thread for a function block, if

- “Dedicated Thread” is chosen
- No “Arguments” are marked as “Trigger” in the Argument Tab

Execution Criterion:

vsi::runtime will execute a function if any ***input*** argument marked as “trigger” changes. The runtime can also execute a block based on a timer by choosing “Timer Expiry”. The runtime will create a timer which will execute the function every “Execution Timer” nano-seconds. The runtime will keep track of the amount of time spent by the function itself and will adjust the timer to make sure the function is executed as close to the “Execution Timer” value as possible.

Execution Timer (nS):

This field is appeared when the “**Execution Criterion**” is specified as “Timer”, and specifies the interval for executing the given function.

Arguments Section

Each argument of the function chosen in “C/C++ Function name” will be exposed as an interface of the “block” . This “section” allows the user to specify details about the interface that are not inferred by the built-in parser. Some fields have default values populated by the parser which the user can override. Figure below shows the Screen shot of the Argument section, the number of rows in the argument section is determined by the “**Number of Parameters**” in the **Selected Function**.

Arguments

#	C/C++ HLS name	C/C++ HLS Type	Type	Direction	Access Type	Array Max Packet Size (Bytes)	Execution Trigger	Variable Length	Buffer Depth	Sideband attributes	Output Delay (ns)
Arg 1	in_arr	int	array	input	Sequential	4096	✓		0		0
Arg 2	out_arr	int	array	output	Random	4096			0		0

C/C++ HLS Name:

This field is automatically updated when the function is picked in the “Selected Function” and is the name of the argument.

C/C++ HLS Type:

This filed is automatically updated when the function is picked in the “Selected Function” and is the type of the argument. The two type ***hls::stream*** and ***vsi::device*** have special handling .

Execution Trigger:

The function is executed whenever data arrives on this “Input” interface. See “Type" dropdown list for more details.

Direction:

The choices as a) ***Input*** b) ***Output*** or c) ***Inout*** .

Variable Length:

Software Context : This is used ***only*** when the direction is ***Input and Type is an Array***. ***Hardware Context***: This is used ***only*** when the direction is ***Input and Access Type is Sequential***.

Access Type:

The access type will generate different interfaces depending on the type of ***Context***. The following table shows the interfaces generated for the different access types.

Context	Access Type	Interface Generated
Software	Sequential	Unidirectional Streaming
	Random	Unidirectional Streaming
	Memory	AXI Memory Mapped (can only be connected to memory interfaces in Hardware Context). See vsi::device class for more details.
	Control	AXI Lite Memory Mapped can only be connected to AXI Lite memory interfaces in Hardware Context). See vsi::device class for more details.
Hardware	Sequential	Unidirectional Streaming
	Random	Block Ram Interface
	Memory	AXI Memory Mapped (can only be connected to memory interfaces in Hardware Context).
	Control	AXI Lite Memory Mapped can only be connected to AXI Lite memory interfaces in Hardware Context).

The Sequential Vs Random access is important when the block resides in the ***Hardware Context***. Sequential access promises the Hardware Synthesis (Vivado HLS) that the function will always access the argument (Array / Pointer) in a sequential manner and is converted to an AXI Streaming interface.

Array / Max packet size (Bytes):

This field is auto filled when the function is picked in the “Selected function” and represents the size of the buffer that will be allocated in the wrapper function for this argument. For “Pointer” type this field is initialized to -1, since the compiler cannot determine the amount of storage to allocate for this argument. It is **important** to specify the maximum size of the buffer to be allocated, for “pointers” this value will be used to allocate dynamic storage in the runtime, a value of -1 will cause a runtime failure. This field is not used when the “Type” is “Reference”

Type:

This field is auto filled by the parser when the function is picked in the “Selected function” and represents the storage allocation attribute of the “C/C++ Type” . The choices are a) Array b) Pointer c) Reference d)Interupt e)Value. This attribute along with the C/C++ Type is used by the VSI system compiler to generate the wrapper function for both Software & Hardware . The following table describes the behavior during runtime in a Software Context depending on the Type chosen.

Type	Direction	Wrapper Action
Array	Input/ Inout [Execution Trigger && !Variable Length]	A local array of C/C++ Type will be allocated , the wrapper will accumulate input into the array . The user function will be ***called only when*** enough data has been received to fill the full array
	Input / Inout [Execution Trigger & Variable Length]	A local array of C/C++ Type . When new data arrives on this input, the wrapper function will copy the data into the local variable and call the user function.
	Input [!Execution Trigger]	A local array C/C++ Type, When new data arrives on this interface the data is copied into the array. If more data arrives before the function is executed, the old data will be overwritten.
	Output / Inout	Data is copied from the output Array and sent to the input of the block that it is connected to.
Pointer		It is chosen when the argument is a pointer in C/C++ code. Behaves the same way as Array. Note it is ***important*** to specify the Max Size of the buffer to be allocated.
Reference		It is chosen when the argument is a reference in C/C++ code. Behaves the same as a pointer
Interupt		Currently not supported
Value	Input / Output	It is chosen when the argument is basic type in C/C++ code such as int

Buffer Depth:

This field is used when the block is placed in the Hardware Context and the access type is “Sequential”. The VSI System Compiler will generate a FIFO of the given depth for each of the interfaces that has value > 0 associated with it.

Sideband Attributes:

The hls::stream Class can be used to describe an AXI Streaming interface in software.

```
 template<int D> struct ap_axis_dkt { ap_uint<D> data; ap_uint<1> last; ap_uint<D/8> keep; ap_uint<1> id; };
hls::stream<ap_axis_dkt<32> > in;
```

The above example shows an AXI Streaming interface with sideband signals of ID, LAST & KEEP. The sideband signals will be listed in this entry. When such an interface crosses the boundary between a Hardware & Software context, the VSI system compiler generates code to try to preserve the sideband signals across the interface, with some restrictions.

- ID – is recognized in the first “beat” of the transfer, the ID will remain the same for the rest of the packet.
- KEEP – attribute is preserved only on the last beat of the transfer. Both these attributes require the “LAST” attribute to be present.

HLS Parameters

Information in this block is used to generate the Vivado HLS project when the block is placed in a Hardware Context.

Clock Frequency in HZ:

This parameter will be passed to the Vivado HLS project as the requested timing.

Bind Effort:

This parameter specifies the effort that the Vivado HLS compiler will make to share resources, “low” means larger design generated, “high” smaller design generated.

Scheduling Effort:

“Low” will generate slower hardware design, “High” will generate faster hardware design.

hls::stream Class

Introduction

This class is an API compatible class with Xilinx provides **hls::stream** class in Vivado HLS. In the **vsi::runtime** implementation the **hls::stream** is thread safe . This document describes the **vsi::runtime** implementation of the hls::stream class, please refer to the Section “Using HLS Streams” in the “Vivado High Level Synthesis : User Guide (UG902)” for details on the hardware implementation of this class. Care has been taken to match the hardware behavior.

Class instance.

The hls::stream is a template class and requires a “type” to be instantiated. The data in the hls::stream class can only be accessed sequentially; the data is stored and retrieved using the First In First Out method. The following example shows an instance of hls::stream. `typedef unsigned int uint; hls::stream<uint> uistrm;`

Public APIs.

void hls::stream< T>.write(T&);

Is blocking operation , it pushes a value into the stream . `uint w_uint =32; uistrm.write(w_uint);`

The C++ “<<” operator can also be used to write data into a stream.

```
uistrm << w_uint;
```

T hls::stream< T>.read();

Is a blocking read operation , it will wait till data is available on the stream and returns the data; in the vsi::runtime implementation the calling thread is blocked from execution and will wait till some other thread puts data into the stream using hls::stream< T>.write(T&); The read operation will pop the first element from the hls::stream FIFO and return to the user.

```
uint r_uint = uistrm.read();
```

The C++ “>>” operator can also be used to pop data from the stream FIFO.

```
uistrm >> r_uint;
```

bool hls::stream< T>.write_nb(T&);

Is a non-blocking write operation, will return “true” if the write was successful else it will return “false”.

```
uint w_uint = 32; if (uistrm.write_nb(w_uint)) { // write success code } else { // write fail code }
```

bool hls::stream< T>.read_nb(T&);

Is the non-blocking version of the read, if data is available the function will update the reference in the argument with the values and return “true” , it will return false if the data is not available. `uint r_uint; if (uistrm.read_bd(r_uint)) { // read successful } else { // read failed no data in the stream }`

bool hls::stream< T>.full();

Return “true” if there is no more space in the stream to write data, return “false” if there is space.

bool hls::stream< T>.empty();

Returns “true” if the stream is empty() , this can be used in conjunction with the read() method to implement non-blocking reads.

Examples:

Example 1:

The hls::stream class can used to implement dependencies between blocks in a complete system. In the following code example, the function vsi_mem_ctl reads an input array , and sends the data to an output stream (out_stream); then waits for response from the another function on the stream “resp” before continuing to execute.

`void vsi_memory_ctl(int in_arr[1024], hls::stream > &out_stream, hls::stream > &start, hls::stream > &resp) { static int count = 0 ; ap_axis_d<32> e; int i;

```
printf("%s started %d\n",__FUNCTION__,count);
// perform operation
for (i = 0 ; i < 1024; i++) {
    e.data = in_arr[i];
    e.last = (i == 1023);
    out_stream.write(e); // write to output stream
    if (e.last) printf("%s last detected %d\n",__FUNCTION__,i);
}
ap_axis_d<32> w ;
w.data = 1;
w.last = 1;
// tell next function to start
start.write(w);
printf("%s sent start waiting for response\n",__FUNCTION__);

// wait for response
ap_axis_d<32> r = resp.read();
printf("%s done %d\n",__FUNCTION__,count++);
```

}`

Example 2:

This functions reads from two AXI streams and converts them into a single stream with different IDs. `void stream_mux (hls::stream<ap_axis_dkt<DATA_WIDTH> > &in1, hls::stream<ap_axis_dkt<DATA_WIDTH> > &in2, hls::stream<ap_axis_dkt<DATA_WIDTH> > &outp) { while (!in1.empty() || !in2.empty()) { // non blocking wait ap_axis_dkt<DATA_WIDTH> out; ap_axis_dkt<DATA_WIDTH> in; ap_uint<1> tid; bool set = false; if (!in1.empty()) { in = in1.read(); tid = 0; set = true; } else if (!in2.empty()) { in = in2.read(); tid = 1; set = true; } out.data = in.data; out.keep = in.keep; out.last = in.last; out.id = tid; if (set) outp.write(out); }`

More examples can be found in the source $(VSI_INSTALL)/target/common/hls_examples/stream_mux/stream_mux.cc

vsi::device Class

Introduction

Available from version 2017.1, this class allows blocks residing in the Software Context, to directly communicate with AXI Memory mapped devices in the Hardware Context. The class exposes a “posix” compliant API which allows easy access to the devices . The class would typically be used as an argument to a function, and should be connected to an AXI Memory Mapped interface in an Hardware Context. If the Argument is a **vsi::device** the Access Type must be either “Memory” or “Control”.

Public APIs

int vsi::device.write (void *buff, size_t count);

Will write count bytes from buff to the device. It reqturns the number of bytes actually written to the device. Write is a blocking operation.

```
void write_mem(vsi::device &mem) { char buff[12] = “hello world”; mem.write(buff,sizeof(buff)); }
```

int vsi::device.read(void *buff, size_t count);

This is will read the number of count number of bytes from the device and place it into buff, it returns the number of bytes actually read. Read is a blocking operation.

```
void read_mem(vsi::device &mem) { char buff[1024]; if (mem.read(buff,sizeof(buff)) != sizeof(buff)) { printf(“Not Enough Data\n”); } }
```

int vsi::device.pwrite(void *buff, size_t count, size_t offset);

The function will write count number of bytes from the buff at offset specified by offset. This function is blocking and can be used to write values at a specific offset such as a register in a device. It returns the number of bytes actually written.

```
void set_timer (vsi::device &t_dev) { unsigned int TLR = 0; t_dev.pwrite(&TLR,sizeof(TLR),4); // write to 0 register at offset 4 }
```

int vsi::device.pread(void *buff,size_t count, size_t offset);

The function will read count number of bytes into buff from offset specified by *offset*. This function is blocking and can be used to read values from a register in a device. It returns the number of bytes actually read.

```
int get_timer (vsi::device &t_dev) { unsigned int TLR =0; if (t_dev.pread(&TLR,sizeof(TLR),4) != sizeof(TLR)) // read from offset 4 printf(“ Read Failed\n”); return TLR; }
```

int vsi::device.poll(int timeout);

Waits for interrupt / data to become available to read, for devices with no interrupts this function will return immediately. The timeout parameter is in milliseconds.

int vsi::device.device_fd();

Returns the File Descriptor associated with this device. Note that the file descriptor is NOT available for devices running on a simulator context or a non-hardware context, the function call will cause an **assertion** if called for a device which is not associated with a file descriptor.

Device Tree

Building Device Tree for the board you are using (This demo is for board zcu102).

Prerequisites

This section assumes that the following prerequisites have been satisfied:

- PetaLinux Tools Installation is completed.
- PetaLinux Working Environment Setup is completed.

Petalinux Flow

BSP package was used to built the petalinux project. However users, can also use the HDF exported from Vivado to build the project. (BSP can be downloaded from the Xilinx website here.)

1. Change to the directory under which you want PetaLinux projects to be created.
2. Run petalinux-create command on the command console:

   ```
   petalinux-create -t project -s <path-to-bsp>
   ```

3. Modify the system-user.dtsi file in the following path:

   ```
   <petalinux-project-path>/project-spec/meta-user/recipes-bsp/device-tree/files/
   ```

   As follow:

   ```
   /include/ "system-conf.dtsi"
   /{
    amba {
            vsi_driver_plat@80000000 {
                compatible = "xlnx,sv_driver_plat";
                interrupt-parent = <0x4>;
                interrupts = <0x0 0x59 0x4>;
                reg = <0x0 0x80000000 0x0 0x000C000 0x0 0x88000000 0x0 0x08000000>;
            };
        };
   };
   ```

   The interrupt number is the default interrupt number when you connect to interrupt number zero. If you connect to other interrupt number, you have to modify it.

4. Configure Petalinux:

   ```
   petalinux-config
   ```

   - Subsystem auto hardware -> advanced bootable images storage settings -> dtb image settings ->IMAGE STORAGE MEDIA -> primary sd
   - Image Packaging Configuration -> Root FS type -> sd card
   - Image Packaging Configuration -> Turn off copy final image to tftp boot
   - yocto settings -> disable network sstate feeds

5. Build the Device Tree:

```
petalinux-build -c device-tree
```

This will create the image/linux directory.

6. Use the files in pre_built directory to create the BOOT.BIN as follow:

```
petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit --pmufw pmufw.elf --u-boot
```

Copy image.ub and BOOT.BIN from image/linux to your SD card.

Creating a Zynq Platform

- Create New project , using a Zynq Board of your choice.

- Point IP catalog to VSI repository

- Create the predefined Zynq Platform

- This will create a platform Canvas with Zynq
- Compile this platform

```
* Flow --> Compile Platform ...
```

- Create a System Canvas

```
* Flow --> Create System ...
```

- Import compiled platform into System Canvas

```
* Flow --> Import Platform ...
```

- After importing the platform the System canvas should have two execution contexts

```
* Zynq_ps - is the software execution context (Representing the ARM)
```

  - Zynq_pl is the Hardware execution context (Representing the FPGA Fabric)

Creating the Sort Demo

- Import C/C++ code into VSI system Canvas (Using sort.cc which is can be found at $VSI_INSTALL/target/common/sort)

- Add the TCP Server software IP block to the Software Context from IP Library

- At this point you have a complete software project to perform sort in software
  - Flow --> Generate System ...

- Next we move the sort block from software to hardware

- Add Trace to interfaces of interest
  - Right Click "Toggle Trace.."

- Generate the complete software , hardware & HLS projects
  - Flow --> Generate System ...

- Use HLS to convert sort function to verilog

- Compile Software Project created for Zynq_PS

Creating a VC709(Virtex) Platform - for packet processing

- Create New project , using a VC709 Board.

- Create the predefined VC709 Platform

- This will create a platform Canvas with VC709 + X86 connected over PCIe (Gen3)
- Compile this platform using TCL command

```
* vsi_compile_platform
```

- Create a System Canvas using TCL command

```
* vsi_create_system system
```

- Import compiled platform into System Canvas

```
* vsi_import_platform
```

- After importing the platform the System canvas should have two execution contexts
- X86 - is the software execution context & VC709 is the Hardware execution context

Creating a Packet Processing application using the VC709(Virtex) platform

- Import packet matching engine to the FPGA (vc709) context. Source found in $(VSI_INSTALL)/target/common/hls_examples/regex_packet/match/regex_exec.cc

```
Packet Matching Engine" into FPGA context.    Source found $(VSI_INSTALL)/target/common/hls_examples/regex_pack
```

- Import Match Compiler to the Software (X86) context. Source found in $(VSI_INSTALL)/target/common/hls_exmples/regex_packet/compile/regex_comp.cc

```
Compiler" into Software context.    Source found $(VSI_INSTALL)/target/common/hls_examples/regex_packet/compile/
```

- Add a TCP server to Softwar (X86) Context that will feed the pattern to be compiled and sent the matching engine. Set the server port to 2020

- Add another TCP server in X86 that will send the packets to the line and recive matched packets back. Set the server port to 2021

- Annotate connections to be traced

- Generate the complete system & hardware

- Synthesize the pattern matching engine using Vivado HLS

- Compile the Software project

- Generate FPGA bitstream for the FPGA project

VSI Simulation

This section describes how to build simulation in Visual System Integrator. Current example reads file from the host machine handled by simulation system and writes back the result.

Create VSI project

From menu bar, select ***File*** -> ***New project***

Specify Project name and location as shown in figure below.

"Default Part" window: select "Boards", ZedBoard and press Next -> Finish.

Create Platform

menu bar -> Flow -> Create Plaform, press OK in the appeared window.

Open Diagram that has been created at the previous step with your platform (default name: platform_1).

Software Context

menu bar -> Flow -> Create Context Select Type: Software, and setup name of new context, for example: soft_context

Double click on software context and add path to Xilinx header Include Directory. In exempel it is: /opt/Xilinx/Vivado_HLS/2017.1/include

Open Software context (soft_context). Add “VSI Driver Setup Wizard” IP.

Set Driver Type to SIMULATOR

Hardware Context (Simulation)

***menu bar -> Flow -> Create Context* Select Type: Hardware, and setup name of new context, for example: hardware_simulation

In “context parameters” select the following parameters: Target platform: Simulator Simulator: Vivado Simulator

Open Hardware context (hardware_simulation). Add “VSI Xilinx AXI BFM” IP, make next connections: gen_clk <-> m_axi_aclk, gen_reset <-> m_axi_aresetn

Contexts cross connection

Open platform diagram to connect “VSI Driver Setup Wizard” M_AXI and “VSI Xilinx AXI BFM” plat_axi ports.

Save all your changes.

Create System

menu bar -> Flow -> Create System, press OK in the appeared window.

Open system (default name system_1) and import platform: ***menu bar -> Flow -> Import Platform***

Open Software Context (double click at the empty space in software context)

Add "VSI Software Import Wizard" IP component. Open Re-customize IP window (double click) at the source directory, select path to vsi/staging/target/common/hls_examples/memory and press Refresh. Set the following parameters:

- Function : read_file_send
- Arg1 Direction : output

Create VSI Software IP component with the following parameters:

- Function : recv_write_file
- Arg1 Direction : input

Create VSI Software IP component with the following parameters:

- Function : process_data
- Arg1 (in_s) Direction : input
- Arg2 (out_s) Direction : output

Make connection between components:

Read_file {out_s} <-> {in_s} process {out_s} <-> {in_s} writefile

Save your changes.

From system diagram drag and drop process ip block to the simulation context space.

Now process is located at the hardware simulation context.

System generation

- Compile platform : ***menu bar -> Flow -> Compile Platform***
- Generate system : ***menu bar -> Flow -> Import Platform***
- Build HLS system : ***menu bar -> Flow -> Build HLS***
- Build Software : ***menu bar -> Flow -> Build HLS***

Run Simulation

Open Xilinx Vivado and execute script /vsi_auto_gen/hw/hardware_simulation_platform.tcl that will generate Vivado project.

Go to /vsi_auto_gen/sw/system_1/build/soft_context

Generate test file

```
dd if=/dev/random of=infile.txt bs=1 count=64
```

Launch simulation from Vivado. ***menu bar -> Flow -> Run Simulation -> Run Behavioral Simulation***

Run simulation ***menu bar*** -> ***Run*** -> ***Run All***

Start program from the command line

Go to project_location/vsi_auto_gen/sw/system_1/build/soft_context and run the program: `./bin/soft_context` In this example input file (infile.txt) has to be located in the directory where generic program is executed.

Break simulation

menu bar -> ***Run*** -> ***Break***

Output.txt file (similar to input.txt) is created as a result of the simulation.

System Interface
Watch later
Share
Connect AXI BRAM controlers

Java and Python support

Visual System Integrator 2017.1 and onwards provides builtin bindings for Python and Java. A user can use Python and/or Java to call FPGA core directly or go through an intermediary C/C++ block. In addition, a Python/Java callback can be initiated directly from FPGA. Moreover, a mix/match between Python, Java and C/C++ can be used in a single system[1]

This enables the user to easily integrate their existing workflow, utilize existing language expertise and third-party libraries effortlessly. It can also be used to integrate the VSI output with another system using one of the external language.

For experimentation and/or research, it allows a system architect to keep well-tested and established parts of the system in C/C++/FPGA while allowing rapid prototyping using Python or Java.

Enabling External Language Support

Any version beyond VSI 2017.1 already has external language support.

Walkthrough for Python

- For the purpose of demonstration in this walkthrough, we will use a simple python function included in vsi_examples repository. Using git, clone the vsi_examples repository: `https://github.com/systemviewinc/vsi-examples.git`.
- Create a new project and work through the wizard to the end. Select any part/board as it won't matter since we're only going to be using the software for the walkthrough.
- Click `menu->flow->Create Platform`. Accept the prefilled default options and Press okay.
- Click `menu->flow->Create Context`. Change the dropdown to `Software` and enter name as `python_test`. Press okay.
- Open Context block properties by double clicking inside `python test` and scroll down to `Software Parameters->language`
- Select the target language you intend to use for the context. For this walkthrough, we will use `Python`
- Ensure that `-static` flag is NOT included in `C` or `C++` compile options. Remove it if you see it included.
- Set the python version in `Python Options` section to 2.7x. Press okay to close the properties dialog.
- Select `menu->Flow->Compile Platform`.
- Then create a system canvas by selecting `flow->create system`. Accept the prefilled defaults.
- Open the newly created system canvas and select `menu->Flow->Import Platform`
- Double click the `python_test` context to enter hierarchy.
- (Optionally) Verify that the correct language is available in the desired context in System Canvas.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Change the `Source Language` in `Source Parameters` to `Python`.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1`.
- The `Python Function` dropdown box should be populated with multiple entries. Open and select `process_stream`.
- Change the direction of `Arg 2` in Arguments section to `output`. Press okay to close properties.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1`.
- The `C/C++ Function` dropdown box should be populated with multiple entries. Open and select `process_tcp1`.
- Change the direction of `Arg 2` in Arguments section to `output`. Press okay to close properties.
- Drag and connect each input to output so that two blocks are connected together. i.e sbOut of python block should be connected to in1 of C/C++ block and sbIn of python block should be connected to out1 of C/C++ block.
- Select `flow->Generate System`.
- Select `flow->Build Software Contexts` and then Select `Build`.
- Wait for the build to successfully complete.
- Browse to the directory `<project_dir>/vsi_auto_gen/sw/system_1` in a terminal.
- Run the system by using the following command `python python_test/python/python_test_main.py`
- The example will send 256 bytes of data through each block. A statement on console is printed every 100,000 iteration.

Walkthrough for Java

- For the purpose of demonstration in this walkthrough, we will use a simple java function included in vsi_examples repository. Using git, clone the vsi_examples repository: `https://github.com/systemviewinc/vsi-examples.git`.
- Create a new project and work through the wizard to the end. Select any part/board as it won't matter since we're only going to be using the software for the walkthrough.
- Click `menu->flow->Create Platform`. Accept the prefilled default options and Press okay.
- Click `menu->flow->Create Context`. Change the dropdown to `Software` and enter name as `java_test`. Press okay.
- Open Context block properties by double clicking inside `java test` and scroll down to `Software Parameters->language`
- Select the target language you intend to use for the context. For this walkthrough, we will use `java`
- Ensure that `-static` flag is NOT included in `C` or `C++` compile options. Remove it if you see it included.
- Set the java version in `java Options` section to 2.7x. Press okay to close the properties dialog.
- Select `menu->Flow->Compile Platform`.
- Then create a system canvas by selecting `flow->create system`. Accept the prefilled defaults.
- Open the newly created system canvas and select `menu->Flow->Import Platform`
- Double click the `java_test` context to enter hierarchy.
- (Optionally) Verify that the correct language is available in the desired context in System Canvas.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Change the `Source Language` in `Source Parameters` to `java`.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1`.
- The `Java Function` dropdown box should be populated with multiple entries. Open and select `process_stream`.
- Change the direction of `Arg 2` in Arguments section to `output`.
- Change size for each `arg 1` and `arg 2` to be 256 bytes. Press okay to close properties.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1`.
- The `C/C++ Function` dropdown box should be populated with multiple entries. Open and select `test_java`.
- Change the direction of `Arg 2` in Arguments section to `output`. Press okay to close properties.
- Drag and connect each input to output so that two blocks are connected together. i.e sbOut of java block should be connected to inc of C/C++ block and sbIn of java block should be connected to out of C/C++ block.
- Select `flow->Generate System`.
- Select `flow->Build Software Contexts` and then Select `Build`.
- Wait for the build to successfully complete.
- Browse to the directory `<project_dir>/vsi_auto_gen/sw/system_1` in a terminal.
- Run the system by using the following command `java -classpath build/java_test/java/java_test.jar java_test_main`
- The example will send 256 bytes of data through each block. A statement on console is printed every 100,000 iteration.

Exercise

- Try modifying the test string in example code and verify that the printed statement changes.
- Modify the iteration cycle when a statement is printed to 1,000,000.
- Change the design to have another python/java block.

Footnotes

1. A single context can only support a mix of one external language and C/C++. For example Java and C/C++ or Python and C/C++. In order to use both Java and C++, a logical partitioning of contexts, each supporting an external language has to be used. VSI Software Interconnect can be used to seamlessly connect the two context without any handwritten code.

Interfacing with an External Systems

External Interfaces

Visual System Integrator generated binaries can be used as subsystem of a larger system. During design, any unconnected interface can be marked as `external interface` which exposes it to an external system. In order to interface to the `external interface`, VSI provides various options suitable for any type of external system. Here we will list down the options that are available out of the box:

- API
- TCP Ports
- RESTful API

API

This provides the lowest level access to an interface marked as external. Each of the external interface is exported as a raw stream_buffer and can be written to from a C/C++ Api. In order to use it, the context has to be generated as a shared library. The resulting shared library can be loaded by an external system using `[dlopen](http://man7.org/linux/man-pages/man3/dlopen.3.html)` explicitly or linked against using the `api.h` file that is generated in the public folder.

Walkthrough

- For the purpose of demonstration in this walkthrough, we will use a simple function included in vsi_examples repository. Using git, clone the vsi_examples repository: `https://github.com/systemviewinc/vsi-examples.git`.
- Create a new project and work through the wizard to the end. Select any part/board as it won't matter since we're only going to be using the software for the walkthrough.
- Click `menu->flow->Create Platform`. Accept the prefilled default options and Press okay.
- Click `menu->flow->Create Context`. Change the dropdown to `Software` and enter name as `api_walkthrough`. Press okay.
- Open Context block properties by double clicking inside `api_walkthrough` and scroll down to `Software Parameters->map exposed interfaces as`. Make sure that `Api` is selected.
- Scroll down to `C/C++ options` and ensure that `-static` flag is NOT included in `C` or `C++` compile options. Remove it if you see it included.
- Open the `Builds` dropdown and select Shared Library.
- Press okay to close the properties dialog.
- Select `menu->Flow->Compile Platform`.
- Then create a system canvas by selecting `flow->create system`. Accept the prefilled defaults.
- Open the newly created system canvas and select `menu->Flow->Import Platform`
- Double click the `api_walkthrough` context to enter hierarchy.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1`.
- The `C/C++ Function` dropdown box should be populated with multiple entries. Open and select `process_tcp1`.
- Enable `Execution Trigger` for `Arg 1` in Arguments section.
- Change the direction of `Arg 2` in Arguments section to `output`. Press okay to close properties.
- Right click each interface and select `Mark External`.
- Select `flow->Generate System`.
- Select `flow->Build Software Contexts` and then Select `Build`.
- Wait for the build to successfully complete.
- Browse to the directory `<project_dir>/vsi_auto_gen/sw/system_1` in a terminal.
- VSI automatically generate an example executable to demonstrate the api usage. Run the example by using the following command `./build/api_walkthrough/bin/api_walkthrough_example`
- The example will write to each input interface marked external and then read from each output marked external A statement on console is printed for each read and write.

Exercise

- Analyze the example code for API walkthrough in `<project_dir>/vsi_auto_gen/sw/system_1/api_walkthrough/example`. Modify it to read data from a file instead of a variable and write to the input interface.

TCP Ports

This provides the simplest way to interface to a system that is not on the same machine. Each of the external interface is mapped to a TCP port and can be read from or written to.

Walkthrough

- For the purpose of demonstration in this walkthrough, we will use a simple function included in vsi_examples repository. Using git, clone the vsi_examples repository: `https://github.com/systemviewinc/vsi-examples.git` .
- Create a new project and work through the wizard to the end. Select any part/board as it won't matter since we're only going to be using the software for the walkthrough.
- Click `menu->flow->Create Platform` . Accept the prefilled default options and Press okay.
- Click `menu->flow->Create Context` . Change the dropdown to `Software` and enter name as `tcp_walkthrough` . Press okay.
- Open Context block properties by double clicking inside `tcp_walkthrough` and scroll down to `Software Parameters->map exposed interfaces as` . Make sure that `TCP Ports` is selected.
- Press okay to close the properties dialog.
- Select `menu->Flow->Compile Platform` .
- Then create a system canvas by selecting `flow->create system` . Accept the prefilled defaults.
- Open the newly created system canvas and select `menu->Flow->Import Platform`
- Double click the `tcp_walkthrough` context to enter hierarchy.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard` .
- Open newly added block properties by double-clicking it.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/example_1` .
- The `C/C++ Function` dropdown box should be populated with multiple entries. Open and select `process_tcp1` .
- Enable `Execution Trigger` for `Arg 1` in Arguments section.
- Change the direction of `Arg 2` in Arguments section to `output` . Press okay to close properties.
- Right click each interface and select `Mark External` .
- Select `flow->Generate System` .
- Select `flow->Build Software Contexts` and then Select `Build` .
- Wait for the build to successfully complete.
- Browse to the directory `<project_dir>/vsi_auto_gen/sw/system_1` in a terminal.
- Run the system by using the following command `./build/tcp_walkthrough/bin/tcp_walkthrough`
- Use a tcp client to connect to the input interface marked external. For this walkthrough, we'll use `netcat` as a TCP client. Enter the following command to run `netcat localhost 1999` .
- Open another terminal and enter `netcat localhost 2000` . This will connect to the output where you will see the processed data being emitted.
- Enter a blob of data < 256 Bytes and press enter.
- You should see the processed data in the second terminal.

RESTful API

A much more high level API can be used through RESTful interfaces. RESTful is a semi-official standard used throughout cloud/web infrastructure and Visual System Integrator provides a simple way to plug an FPGA/Embedded workflow into the rich ecosystem effortlessly while retaining the high performance promised by VSI.

Walkthrough

- For the purpose of demonstration in this walkthrough, we will use a simple function included in vsi_examples repository. Using git, clone the vsi_examples repository: `https://github.com/systemviewinc/vsi-examples.git`.
- We will also need a RESTful server in order to display the processed data. For this walkthrough, we will use a third- party library mongoose. Using git, clone the repository `https://github.com/cesanta/mongoose.git` locally.
- In a terminal, enter the `examples` directory in cloned repository and run `make`. This will build the example projects.
- Enter the `restful_server` directory and run `./restful_server`. This will startup the restful server on port 8000 locally.
- Create a new project and work through the wizard to the end. Select any part/board as it won't matter since we're only going to be using the software for the walkthrough.
- Click `menu->flow->Create Platform`. Accept the prefilled default options and Press okay.
- Click `menu->flow->Create Context`. Change the dropdown to `Software` and enter name as `rest_walkthrough`. Press okay.
- Open Context block properties by double clicking inside `rest_walkthrough` and scroll down to `Software Parameters->map exposed interfaces as`. Make sure that `RESTful Endpoints` is selected.
- Press okay to close the properties dialog.
- Select `menu->Flow->Compile Platform`.
- Then create a system canvas by selecting `flow->create system`. Accept the prefilled defaults.
- Open the newly created system canvas and select `menu->Flow->Import Platform`
- Double click the `rest_walkthrough` context to enter hierarchy.
- Add a new `VSI Software Wizard` block by right clicking and selecting `VSI Software Wizard`.
- Open newly added block properties by double-clicking it.
- Click browse beside `Source Directory` and browse to select source code directory `<vsi_examples>/sort`.
- The `C/C++ Function` dropdown box should be populated with multiple entries. Open and select `sort` function.
- Enable `Execution Trigger` for `Arg 1` in Arguments section.
- Change the direction of `Arg 2` in Arguments section to `output` and access type to `Random`. Press okay to close properties.
- Right click each interface and select `Mark External`.
- Select `flow->Generate System`.
- Select `flow->Build Software Contexts` and then Select `Build`.
- Wait for the build to successfully complete.
- Browse to the directory `<project_dir>/vsi_auto_gen/sw/system_1` in a terminal.
- Run the system by using the following command `./build/rest_walkthrough/bin/rest_walkthrough`
- Open another terminal and enter the following command: `curl -H "Content-Type: application/json" -X POST -d '[{ "name": "rest_walkthrough::vsi_gen_ip_0::arg_2_seq_o", "type": "webhook", "target": "http://localhost:8000/printcontent"}]' http://localhost:1999/connect`. This sets the REST endpoint for the output interface which will be used to emit the processed data.
- Enter `printf "" > ~/sort.txt; for i in {1..256}; do printf "ABCDEFGH12345678" >> ~/sort.txt; done`. This will create a file with unsorted blob of alphanumeric data.
- Enter `curl -H "Content-Type: application/octet-stream" -X POST -d @/home/$USER/sort.txt http://localhost:1999/rest_walkthrough::vsi_gen_ip_0::arg_1_seq_i`. This will read the sort.txt file and push it to the input interface marked external.
- You should see part of processed data in the first terminal with the content sorted.

Feature comparison

Features	2016.4	2017.1
Drag and drop to design full system	Yes	Yes
TCP (Server/Client) IP	Yes	Yes
C/C++ Software Wizard IP	Yes	Yes
Automatically trace interfaces	Yes	Yes
Collect Live Trace data	Yes	Yes
Generate driver scripts	Yes	Yes
Compile software from Within VSI	Yes	Yes
Synthesize HLs from within VSI	Yes	Yes
Python Software Wizard IP	No	Yes
Java Software Wizard IP	No	Yes
API Interface	No	Yes
RESTful Interface	No	Yes

Troubleshooting

General

- The Context parameters in platform are greyed out and cannot be changed
  - Run vsi::__::fix_platform_context in VSI TCL console (the platform canvas must be open)
- When importing the platform file into an existing system context tab, encountered with the error message "*failed import. The design "platform" is not a valid vsi_system design*".Follow the below steps
  - Select the "**Sources**" tab.
  - Select the < **system_context** > from the list in the Sources tab.
  - `Import Platform` .
- Modifying TCP server settings for data type and package size is not editable in input and output blocks
  - uncheck and re-check the check box for the input and output, then the fields will be editable.
- when creating platform for a particular speed grade part using the create_zync_platform.tcl script the default part value is set to xc7z020clg400-2.
  - Edit vsi_create_zynq_platform.tcl file to the corresponding part of the speed grade.
- User constraints is not been applied due to the default user constraints in Vivado.
  - After encountering with the error edit the file mentioned in the error according to the required constraints to the user.
  - For example: When a Project with x86 and PCIe is created user constraints. Disable the 4 constraints regarding the pcie lane location and add the required ones.
- Crashing of the software executable due to insufficient FIFO size.
  - For example: The system consists of a x86 PC and a FPGA connected over PCIe. The system consists 2 tcp Servers and a customized VHDL IP, see image. When more than 4096Bytes are sent to the hardware at ones, the Software executable crashed.

  - In the generated vivado Project, open the Blockdesign, search for the vc707 block and open. There is a FIFO Block located near the input and output of the custom VHDL IP. Adjust the FIFO size to be larger then the max. possible input packet size.

Runtime / Generated Context

- The first step should be to enable logging for runtime. VSI Runtime supports a modular logging system that can be turned on by setting the environment variables `VSI_LOG_LVL=info and VSI_LOG_COMPS=all`. For the full set of valid values, see logging section

Java Specific

- Swig Director Error

Python Specific

- Swig Director Error
  - Check the data size is properly set inside each software wizard. Sometimes, an IP upgrade can null out the values.
  - Check that the data type is correctly set. For valid types that are supported, check here
- Cannot import
  - Make sure that all the sources are present in the current directory. In addition, a `_<context_name>.so` should be present.
- Build errors on `Flow->Build Software Contexts`
  - When switching between a cross compile and host compile for a context, even when "clean and rebuild" is selected, the switched context still has a toolchain file and CMake may attempt to use it when compiling. The fix is to delete the context directory under `<project>/vsi_auto_gen/sw` and regenerate the context using `generate system`. After which the build should use the correct compiler.

Files for Manual Install (follow the instructions for your host type in installation):

System View Download Area

Development Workflow

In progress

VSI Runtime

Visual System Integrator includes a lightweight library that provides implementation of foundation classes used by the generated code. Here we will go into details on what each class does and how is it used by the generated system.

- Runtime
  - Responsible for initialization and shutdown routines.
- Custom
  - Provides routines to run external functions. It is responsible for data streams and also convert them into the parameter types required by the function. Also see related IP `vsi_gen_ip`.
- TcpServer
  - Provides a TCP interface that listens for incoming connection on a given port for the duration until the context is active. Also see related IP `tcp_server`
- TcpClient
  - Provides a TCP interface that attempts to connect to a given port. It will optimistically continue to retry if disconnected until the context is active. Also see related IP `tcp_client`;
- Interconnect
  - Provides connection interface between contexts connected through ethernet. Also see Related IP `platform_interconnect_sw`.
- Net
  - Base class that provides TCP and UDP connectivity.
- Device
  - Base class provides connectivity between contexts connected through kernel driver interface.
- Inout
  - Base class that provides routine for data movement.
- Connect
  - Base class that defines connection parameters.
- Trace
  - Utility class that provides functionality related to VSI Trace. It is responsible for receiving trace events, compressing trace data and routing it to the main context.
- TraceDevice
  - Utility class that polls for raw trace events from contexts connected through kernel driver/shared memory interfaces.

vsi::runtime::Runtime

Runtime class is responsible for initialization and shutdown routines. It is a singleton class and cannot be created or initialized using `new`. To call any of the members, always use vsi::runtime::Runtime::Get() to get the singleton instance.

Function Reference

- `vsi::runtime::Runtime::Get()` : Returns the singleton instance.
  - return: the singleton instance of Runtime. It creates the singleton instance if it didn't previously exist.
- `void init(const char* name, unsigned int flags)` : Initializes the Runtime Library and starts executing.
  - `name` : name of the context. Used for logging and trace.
  - `flags` : a list of flags that control the behavior of the runtime. Supported flags are `RUNTIME_MAIN | RUNTIME_NONBLOCKING`
    - `RUNTIME_MAIN` : Sets this context as main context. This behavior ensures that all the trace data is channelled to this context and can be collected by connecting to this context.
    - `RUNTIME_NONBLOCKING` : Modifies the behavior of this function to be nonblocking and not block the thread that called it. Useful for integration with thirdparty libraries or when VSI is generating part of the system, with other part being handwritten and main thread is performing other tasks.
    - `RUNTIME_NONE` : Default flag indicating that no flags are provided. Omitting the parameter altogether has the same affect.
  - `return` : void
- `void wait_ready()` : Utility function that blocks until runtime has initialized. Calling this at the beginning of a new thread ensures that it doesn't attempt to use a runtime resource that has not been initialized yet.
- `const char *ctx_name()` : Returns the name set by init()
  - `return` : name set by init()
- `void shutdown()` : Signal the Runtime to close down owned resources, close device descriptors and stop threads.

vsi::runtime::Custom

Provides routines to run external functions. It is responsible for incoming and outgoing data streams and also converts it into the parameter types required by the function. Also see related IP `vsi_gen_ip`. Usage of this class outside the runtime library is not supported. The function reference here is merely to provide information into runtime internal functionality and to better understand the generated code.

Utility Class: vsi::runtime::CustomCB

A wrapper class for cross language callbacks. Only has a single method `void run(Custom *cb)`.

Function Reference

- `void init()` : Initializes the component
- `void event()` : Callback triggered when there is data available on one of the inputs configured as trigger; alternatively, when the IP is configured to trigger on an interval.
- `void set_callback(CustomCB *cb)`
  - `cb` : pointer to a wrapped function. A non-c++ function needs to be wrapped using CustomCB before it can be passed to this function.
- `Inout *get_arg_<type>_<num>()` : Returns the inout instance attached to a parameter. Available data stream can be retrieved from using inout->stream(). Supported types are mem, ctrl and seq. The maximum number of parameters is 16 [1]
  - return: pointer to Inout attached to a function parameter.

Footnotes

1. Inout usage reference

vsi::runtime::Inout

This is a utility class that serves as a binding between runtime components. It is responsible for moving data in/out and to provide handles to device and memory.

Function Reference

- `Inout(INOUT_TYPE type)` : Constructs an inout. The newly constructed inout is set as an input, an output or an inout.
  - `type` : The type of the newly constructed inout. Valid types are INPUT, OUTPUT and INOUT.
- `Inout(INOUT_TYPE t, const char *name)` : Alternative constructor that takes a name as a parameter. Used when the parent component is on PL's side.
  - `type` : The type of the newly constructed inout. Valid types are INPUT, OUTPUT and INOUT.
  - `name` : The name of the logical component that this inout represents.
- `void init(Log *log)` : Initializes the inout. Once initialized, the inout can be written to and/or read from depending on the configuration.
  - `log` : The logger component to use. An inout is typically attached to a component and shares an instance of the logger component.
- `void init(Log *_log, std::function<void(spBuffer)> _write_data)` : Alternative initialization that allows to pass a function definition. Used by the components when initializing the `Inout` as a type `input` or `inout` in order to process the incoming data.
  - `log` : The logger component to use. An inout is typically attached to a component and shares an instance of the logger component.
  - `_write_data` : A function definition passed that will be called each time data is written to the `Inout`. The data pointer is passed as a spBuffer (`std::shared_ptr<Buffer>` [1])
- `void read_connection(spBuffer buf)` : Read a chunk of data from the opposite endpoint of the attached connection.[2]
  - `buf` : Buffer pointing to the data.[1]
- `void write_connection(spBuffer buf)` : Write a chunk of data to the opposite endpoint of the attached connection.[2]
  - `buf` : Buffer pointing to the data.[1]
- `void attach(class Connect *conn)` : Attached this `Inout` to a connection.[2]
  - `conn` : A connection class that this inout will attach itself to.[2]
- `const char *name()` : Return the name of the logical component that this `inout` is attached with.
  - return: A constant pointer to the name of the component.
- `void set_device(const char *device_name)` : Takes a device path as a parameter, opening it and attaching itself to it. If the `Inout` is set as `INPUT` then the device is treated as a write only. Conversely, if the `Inout` is set as `OUTPUT`, the device is set as read only. If the `Inout` is set as `INOUT`, both read and write is available. In case of `OUTPUT` or `INOUT`, the device will be polled for data.[3]
  - `device_name` : The device path to use. Needs to be a valid device created with mknod.
- `void set_device_rpc(const char *host_name, int m)` : Binds to a host a an RPC client. The behavior is similar to set_device except that the connection is made over the network.
  - `host_name` : The hostname or IP address of the RPC server.
  - `m` : A unique number that is used to demux the RPC data belonging to a device.
- `void read_data(spBuffer buf)` : Attempts to read a chunk of data from the attached device. Not valid if the attached component is not a device.[3]
  - `buf` : A Buffer pointer contains the data or null if no data was available.[1]
- `hls::stream_base *stream()` : Returns a `hls::stream_buffer` pointer. The stream buffer can be written and read from, depending on how the `Inout` was configured.
  - return: Pointer to a `hls::stream_buffer`.
- `void init_device()` : Initializes an attached device. This is an alternative to `init()` and only works if `set_device` or `set_device_rpc` was called beforehand.
- `void set_address(int64_t _ba, unsigned int _r)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device`.[3]
  - `_ba` : Base address for the device

  - `_r` : Read size for the device.
- `void set_control_address(int64_t _ba, unsigned int _r, int _sp)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` .[3]
  - `_ba` : Base address for the device control block.
  - `_r` : Maximum read size for the device control block.
- `void set_interrupt(unsigned int interrupt_number)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` .[3]
  - `interrupt_number` : Interrupt number bound to the device descriptor.
- `void set_device_size(unsigned int _size)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` .[3]
  - `_size` : Read size for the device. This is the same as the `_r` parameter in `set_address` .
- `int poll(int timeout)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` .[3]
  - `timeout` : How long before the poll timesout.
- `int poll_connection(int timeout)` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` at the opposite endpoint of the connection.[3]
  - `timeout` : How long before the poll timesout.
- `int device_fd()` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` .[3]
  - return: returns the underlying device descriptor number.
- `int device_fd_connection()` : Wrapper function that takes device specific parameters as part of device setup process. The parameters as not kept and are passed directly to the attached `Device` at the opposite endpoint of the connection.[3]
  - return: returns the underlying device descriptor number.
- `INOUT_TYPE type() {return _type;` : Returns the type of the connection earlier set while `Inout` was constructed.

Footnotes

1. Buffer reference
2. Connect reference
3. Device reference

vsi::runtime::TcpServer

Function Reference

- void init(): Starts a statically initialized server. The port and allowed address are determined by protected member `port_no` and `allow_host`.
- ~TcpServer(): Closes all connection and shuts down the server.

Protected Members

- `port_no` : Port number that the server listens on.
- `allow_host` : Allows connections from this host only.
- `INPUT_ENABLE` : TCP Server's input is enabled.
- `OUTPUT_ENABLE` : TCP Server's output is enabled.
- `use_udp` : Uses UDP protocol instead of TCP.
- `server_data_in` : Represents input `Inout` attached to this server.
- `tcp_server_connection` : Represents socket that the TCP Server is listening on.
- `server_data_out` : Represents output `Inout` attached to this server.

vsi::runtime::TcpClient

Function Reference

- void init(): Starts a statically initialized TCP client. The port and host address are determined by protected member `port_no` and `Host`.
- ~TcpClient(): Closes all connection and shuts down the client.

Protected Members

- `port_no` : Port number that the client listens on.
- `Host` : Attempt to connect to this host name or address. Host names are resolved using OS Apis.
- `INPUT_ENABLE` : TCP Client's input is enabled.
- `OUTPUT_ENABLE` : TCP Client's output is enabled.
- `use_udp` : Uses UDP protocol instead of TCP.
- `client_data_in` : Represents input `Inout` attached to this client.
- `client_data_out` : Represents output `Inout` attached to this client.
- `tcp_client_connection` : Represents socket interface that the TCP Client is using to connect with.

vsi::runtime::Net

This Utility class is used by TcpServer, TcpClient and Interconnect to provide data move over TCP.

Function Reference

- `Net(int socktype, unsigned short port, Log *log, const char *server_name = "localhost", std::function<void(spBuffer)> _callback = [] (spBuffer) {})` : Constructs a `Net` class.
  - `socktype` : Sets the socket type. Supported types are `SOCK_STREAM` for TCP and `SOCK_DGRAM` for UDP.
  - `port` : Sets the port. It is a shared number that can be used to listen on or to connect to, depending on which method is called afterwards.
  - `log` : A log object belonging to the parent component.
  - `server_name` : A host name. It will be resolved to an IP and used as a host to connect to if `connect()` is used and as a host to allow connection from if `listen()` is used.
- `connect()` : Attempts to connect to a server on the shared port number.
- `listen()` : Binds to the shared port and starts listening for incoming connections.
- `end()` : Disconnect from clients and/or server.
- `connected()` : Returns true if a connection is established.

vsi::runtime::Interconnect

A platform only component that is used to establish connectivity between two software contexts using TCP or UDP.

Function Reference

- `Interconnect(const char* comp_name, const char* host_name, const int port, unsigned int flags)` : Constructs an interconnect class.
  - `comp_name` : name of the interconnect. Used for logging and trace purpose.
  - `host_name` : Host name to establish the connection with.
  - `port` : Incoming or outgoing port of the interconnect.
  - `flags` : Modify the behavior of the interconnect. Valid flags are `INTERCONNECT_MASTER` and `INTERCONNECT_DEFAULTROUTE` .
    - `INTERCONNECT_MASTER` : Sets the interconnect as a TCP server.
    - `INTERCONNECT_DEFAULTROUTE` : The interconnect will be used as a default route to route all data that has no address through itself. The primary type of such data is trace data. Functionally similar to TCP Default Gateway.
- `init()` : Starts the interconnect.
- `Inout &default_link()` : Returns the default `Inout` of the interconnect. Primarily used by Trace component.
- `Inout &link(int port)` : Add a new link to the interconnect. This starts up a new tcp server or client and attempts to connect to or listens to the host specified in the constructor.

vsi::runtime::Connect

A class used to store state of a connection between components.

Function Reference

- `Connect(const Inout& _io1,const Inout& _io2, bool _plat_intf, int _trace, bool _sys_intf)` : Constructs the class.
  - `_io1` : An `Inout` that defines one endpoint of the connection.
  - `_io2` : An `Inout` that defines one endpoint of the connection.
  - `_plat_intf` : Defines whether this connection is a platform interface.
  - `_trace` : Specifics whether trace data will be collected from this connection.
  - `_sys_interface` : Marks this connection as a system interface. A system interface is an exposed platform interface that maps platform IO to a system.

vsi::runtime::Log

The runtime provides a modular logging interface that can be enable when the executable is started.[1]

Function Reference

- `Log(const char *name)` : Construct the class.
  - `name` : Name is used both as part of the logging entry and as a filter when `VSI_LOG_COMPS` is used.[1]
- `print(LOG_LVL lvl, const char *__restrict fmt, ...)` : Print a log entry to cout.
  - `lvl` : Specify a lvl. If the level is lower than the level that environment variable VSI_LOG_LVL is set to, the method will be set to void. Valid levels are `LOG_CRIT` , `LOG_ERR` , `LOG_DBG` , `LOG_INFO` and `LOG_VERBOSE` .
  - `fmt` : Format string. Similar to printf format string.
  - `...` : Variadic parameters. Similar to printf.
- `print(LOG_LVL lvl, unsigned int opts, const char *__restrict fmt, ...)` : Print a log entry to cout.
  - `lvl` : Specify a lvl. If the level is lower than the level that environment variable VSI_LOG_LVL is set to, the method will be set to void. Valid levels are `LOG_CRIT` , `LOG_ERR` , `LOG_DBG` , `LOG_INFO` and `LOG_VERBOSE` .
  - `opts` : Used options to modify the behavior of formatting. Valid opts are `OPT_NOCOMP` , `OPT_NONEWLINE` , `OPT_SYSERR` .
    - `OPT_NOCOMP` : Don't append the component name to the log entry.
    - `OPT_NONEWLINE` : Don't append a newline to the log entry.
    - `OPT_SYSERR` : Appends the string retrieved by `strerror(errno)` .
  - `fmt` : Format string. Similar to printf format string.
  - `...` : Variadic parameters. Similar to printf.
- `print_if(bool cond, LOG_LVL lvl, const char *__restrict fmt, ...)` :
  - `cond` : Print this entry if the expression evaluate to true.
  - `lvl` : Specify a lvl. If the level is lower than the level that environment variable VSI_LOG_LVL is set to, the method will be set to void. Valid levels are `LOG_CRIT` , `LOG_ERR` , `LOG_DBG` , `LOG_INFO` and `LOG_VERBOSE` .
  - `fmt` : Format string. Similar to printf format string.
  - `...` : Variadic parameters. Similar to printf.
- `print_if(bool cond, LOG_LVL lvl, unsigned int opts, const char *__restrict fmt, ...)` :
  - `cond` : Print this entry if the expression evaluate to true.
  - `lvl` : Specify a lvl. If the level is lower than the level that environment variable VSI_LOG_LVL is set to, the method will be set to void. Valid levels are `LOG_CRIT` , `LOG_ERR` , `LOG_DBG` , `LOG_INFO` and `LOG_VERBOSE` .
  - `opts` : Used options to modify the behavior of formatting. Valid opts are `OPT_NOCOMP` , `OPT_NONEWLINE` , `OPT_SYSERR` , `OPT_EXIT` .
    - `OPT_NOCOMP` : Don't append the component name to the log entry.
    - `OPT_NONEWLINE` : Don't append a newline to the log entry.
    - `OPT_SYSERR` : Appends the string retrieved by `strerror(errno)` .
    - `OPT_EXIT` : Exit the execution in case of error.
  - `fmt` : Format string. Similar to printf format string.
  - `...` : Variadic parameters. Similar to printf.
- `char *name()` : Return the name of the component that this log is constructed for.
  - return: Name of the component.

Footnotes

1. Logging

Buffer

This utility class provides method to store and retrieve binary data.

Function Reference

`Buffer()`

Construct an empty buffer.

C/C++

```
vsi::runtime::Buffer();
```

Python

```
vsi_runtime.Buffer()
```

`Buffer(size_t len)`

Construct a buffer of specific size.

Parameters

- len: Size of the buffer to construct.

C/C++

```
vsi::runtime::Buffer(256);
```

Python

```
vsi_runtime.Buffer(256)
```

`Buffer(const char *string)`

Construct a buffer using a null terminated string. The length and content are same as string's.

Parameters

- `string` : The input string.

C/C++

```
vsi::runtime::Buffer("abcd");
```

Python

```
vsi_runtime.Buffer('abcd')
```

`Buffer(void *_data, size_t len, bool direct = false)`

Construct a buffer using a data pointer and given length.

Parameters

- `_data` : Pointer of the data to copy from.
- `len` : Size of the buffer to construct.
- `direct` : Set to true to avoid allocating a new buffer such as when using it with language binding.

C/C++

```
void copy_buff(void *data, size_t d_size) {
    vsi::runtime::Buffer buf(data, d_size);
    // Do something with the buffer
}
```

Python

N/A

`put(const unsigned char _byte, int _offset)`

Copy a single byte into the buffer at the given offset.

Parameters

- `_byte` : Byte to copy into the buffer.
- `_offset` : This will be used as the target offset into the buffer for the copy operation.

C/C++

Python

`put(Buffer *src, int _offset = 0)`

Copies another Buffer's contents into this buffer at the given offset.

Parameters

- `src` : Pointer to the source buffer.
- `_offset` : This will be used as the target offset into the buffer for the copy operation.

C/C++

Python

`getInt()`

Returns an integer value and forwards the current offset by it's size.

Parameters

```
return: The integer value stored at current offset.
```

C/C++

Python

`compare(Buffer &buf)`

Compares the content of this buffer with another Buffer.

Parameters

- `buf` : Pointer to another `Buffer` to compare against.
- return: true if the contents are identical. False otherwise.

C/C++

Python

`putInt(int i)`

Stores an integer at the current offset and forwards the offset by integer's size.

Parameters

- `i` : Integer value to store in the buffer.

C/C++

Python

`fill(const char *pattern)`

Fills the buffer with the given pattern. The pattern is repeated to fill the whole buffer. If the pattern is bigger than the buffer then only the portion that it is clipped.

Parameters

- `pattern` : A pattern to fill in the buffer. A null terminated string is required.

C/C++

Python

`getShort()`

Return a short value at current offset and forward the offset by short size.

Parameters

- return: The short value that is at current offset.

C/C++

Python

`putShort(short i)`

Write the short value at current offset and forward the offset by short size.

Parameters

- `i` : Short value to write.

C/C++

Python

`getChar()`

Read a char value at current offset and forward the offset by char size.

Parameters

- return: The char value that is at current offset.

C/C++

Python

`putChar(char i)`

Write the char value at current offset and forward the offset by char size.

Parameters

- `i` : Char value to write.

C/C++

Python

`size()`

Returns the size of the buffer minus current offset. Note that internal size might be larger than the value return if offset is > 0;

- return: Readable size of the buffer (size - offset).

C/C++

Python

`get()`

Returns the underlying raw pointer of stored data at current offset.

Parameters

- return: Pointer to the data.

C/C++

Python

`rewind()`

Rewinds the buffer, setting the offset at zero.

C/C++

Python

`data()`

Returns the underlying raw pointer of stored data at current offset.

C/C++

Python

`base_size()`

returns the total size of the underlying buffer. Note that this is the overall memory size that is used by the buffer, not affected by the current offset.

- return: total size of the underlying buffer.

C/C++

Python

`base_data()`

Returns the raw pointer of the underlying buffer. The pointer is to the base and is not affected by current offset.

- return: raw pointer to the underlying buffer.

C/C++

Python

`substr(size_t size)`

Returns a part of buffer clipped to the specified size.

Parameters

- `size` : Only return these many bytes.
- return: std::string containing the clipped buffer. If buffer size was larger than passed size, it is appended by `...` to indicate clipping status.

C/C++

Python

`bytes(size_t size, const char *separator)`

Returns a string formatted as hexadecimal and separated by the provided separator character.

Parameters

- `size` : limit the return string to this size.
- `separator` : user this character as the separator.
- return: std::string formated as hexadecimal string.

C/C++

Python

`offset()`

Returns the current offset of the buffer.

- return: integer specifying the current offset.

C/C++

Python

`set(const void *_data, size_t len)`

Sets the buffer contents. The underlying buffer will resize to accommodate the new size.

Parameters

- `_data` : Pointer to raw content to copy.
- `len` : Size of the content to copy. Also used as the new size for this buffer.

C/C++

Python

`set(const char *string)`

Sets the buffer contents to a null terminated string. The underlying buffer will resize to accommodate the new size.

Parameters

- `string` : the null terminated string to use as the content and to calculate the new size of the buffer.

C/C++

Python

`direct_set(void *_data, size_t _size)`

Change the buffer to use the provided raw buffer as the underlying buffer. The provided raw buffer should be allocated using malloc or new and will be freed by this `Buffer` when it's destructor is called.

Parameters

- `_data` : Pointer to the raw buffer.
- `_size` : Size of the raw buffer;

C/C++

Python

`add(const void *_data, size_t len)`

Appends data to the current buffer. The underlying buffer will be resized to accommodate the new size.

Parameters

- `_data` : Pointer to the data.
- `_size` : Size of the raw buffer;

C/C++

Python

`add(Buffer *buf)`

Appends another Buffer's contents to the current buffer. The underlying buffer will be resized to accommodate the new size.

Parameters

- `buf` : Pointer to the `Buffer` to copy the data from.

C/C++

Python

`offset(size_t _offset)`

Sets the current offset to provided value.

Parameters

- `_offset` : New offset value. It should not be larger than the underlying buffer's size.

C/C++

Python

`dev_offset(size_t _offset)`

An auxiliary piece of integer that is used to specify the target offset to write the data once it reaches it's destination.

Parameters

- `_offset` : Offset to write the data to. The offset is only enforced for memory devices.

C/C++

Python

`dev_offset()`

Returns the memory offset where the buffer should be written to.

Parameters

- return: Returns the auxiliary offset.

C/C++

Python

`reduce(size_t size)`

Truncates the buffer by provided size. The truncated bytes are permanently lost.

Parameters

- `size` : Truncate the underlying buffer by this size. The overall size of the buffer must be larger or equal to the provided size.

C/C++

Python

`alloc(size_t len)`

Allocate the given size for the underlying buffer. If there was an existing buffer than it is freed.

Parameters

- `len` : The new size of the underlying buffer.

C/C++

Python

hls::stream and hls::stream_buffer

This provides a hls::stream compatible implementation.

When used with C/C++, the code is synthesizable to FPGA.

Function Reference

`stream_buffer::stream_buffer()`

construct a stream_buffer.

C/C++

Python

`read(void *out, size_t size)`

read specified number of bytes from the stream_buffer. If the size provided is lesser than the size of first chunk, then only the specified number of bytes are read and truncated from the chunk. If the size of larger than the available chunk then the chunk is popped off and is returned as it is. The returned size is guaranteed to be less or equal to the requested size.

Parameters

- `out` : Target raw buffer.
- `size` : size of the data to read.
- return: the size of actual bytes read.

C/C++

Python

`length()`

Return the size of the all the data available in the stream pipeline.

Parameters

- return: The total bytes of data that can be read. Note that these might be divided between multiple chunks and may requite multiple reads.

C/C++

Python

`read(Buffer buf)`

Pop off the first chunk of data from the stream and return it.

Parameters

- `Buffer` : A buffer element is returned. The element is automatically freed once it is no longer in use or the function/scope exits.

C/C++

Python

vsi::device

This provides direct access to a hardware device or software shared memory using posix compliant calls.

Function Reference

`int write(const void *buff, size_t count)`

Parameters

- `buff` : Raw buffer containing the data to write.
- `count` : Size of bytes to write. Should be equal to or smaller than the buff size.
- return: the size of actual bytes written. Write number of bytes to the device.

C/C++

Python

```
# dev = VsiDevice
buf_in = vsi_runtime.Buffer(256)
buf_in.fill("abcd0123")
dev.write(buf_in)
```

`int read (void *buff, size_t count)`

read specified number of bytes from the vsi::device. Returns the number of bytes read.

Parameters

- `buff` : Target raw buffer.
- `count` : size of the data to read.
- return: the size of actual bytes read.

C/C++

Python

```
# dev = VsiDevice
buf_out = vsi_runtime.Buffer(256)
dev.read(buf_out)
```

`int pwrite(const void *buff, size_t count, int offset) :`

Parameters

- `buff` : Raw buffer containing the data to write.
- `count` : Size of bytes to write. Should be equal to or smaller than the buff size.
- `offset` : Specify the offset to write.
- return: the size of actual bytes written.

C/C++

Python

```
# dev = VsiDevice
OFFSET = 0
buf_in = vsi_runtime.Buffer(256)
buf_in.fill("abcd0123")
dev.pwrite(buf_in, OFFSET)
```

`int pread(void *buff, size_t count, int offset)` :

Parameters

```
- `buff`: Target raw buffer.
- `count`: size of the data to read.
- `offset`: Specify the offset to read.
- return: the size of actual bytes read.
```

C/C++

Python

```
# dev = VsiDevice
OFFSET = 0
buf_out = vsi_runtime.Buffer(256)
dev.pread(buf_out, OFFSET)
```

`int poll(int timeout)`

Blocks until an interrupt or till timeout expires.

Parameters

```
- `timeout`: Timeout to wait for an interrupt.
```

C/C++

Python

```
# dev = VsiDevice
dev.poll()
```

`int device_fd()`

Returns the underlying operating system device descriptor.

C/C++

Python

```
# dev = VsiDevice
FD = dev.device_fd()
```

Language Specific notes and code snippets

C/C++

Python

Since python has no `void*` type, in order to use read, write, pread and pwrite from Python, the pointer and size needs to be wrapped into a VsiDevice type.

A typical example of it is as following:

```
def process_device(dev):
    global COUNT, OFFSET
    COUNT += 1
    buf_in = vsi_runtime.Buffer(256)
    buf_out = vsi_runtime.Buffer(256)
    buf_in.fill("abcd0123")
    dev.pwrite(buf_in, OFFSET)
    dev.pread(buf_out, OFFSET)
    if  (COUNT % 20) == 0:
        print 'Matched:{}, COUNT:{}, OFFSET:{}'.format(buf_in.compare(buf_out), COUNT, OFFSET)
    OFFSET += 1
```

When reading from a file, the alternative method of Buffer can be used to reuse the same memory:

```
def process_device_file(dev):
    global COUNT, OFFSET, file_name
    create_file(file_name)
    buf_in = vsi_runtime.Buffer(256)
    buf_out = vsi_runtime.Buffer(256)
    file_size = os.stat(file_name).st_size
    f = open(file_name, "rb")
    try:
        buf_in.set(f.read(256))
        while file_size > (256 * COUNT):
            dev.pwrite(buf_in, OFFSET)
            dev.pread(buf_out, OFFSET)
            # if  (COUNT % 10) == 0:
            print 'Matched:{}, COUNT:{}, OFFSET:{}'.format(buf_in.compare(buf_out), COUNT, OFFSET)
            OFFSET += 1
            COUNT += 1
            buf_in.set(f.read(256))
    finally:
        f.close()
```

Logging and printf

VSI runtime has a modular logging system which can be turned on using the following environment variables

Environment variables

- VSI_LOG_LVL

  ```
  can be one of the following: CRIT, ERR, DBG, INFO (case insensitive)
  ```

- VSI_LOG_COMPS

  ```
  can be either ALL or a list of components (case insensitive)
  ```

Examples

- Enable logging for all components and maximum logging

  ```
  env VSI_LOG_LVL=info VSI_LOG_COMPS=all ./executable
  ```

- Enable logging for all components and error logging only

  ```
  env VSI_LOG_LVL=err VSI_LOG_COMPS=all ./executable
  ```

- Enable logging for inputs and outputs components and debug logging only

  ```
  env VSI_LOG_LVL=dbg VSI_LOG_COMPS=inputoutput ./executable
  ```

- Enable logging for tcp_server_0_1 and Trace components and info logging

  ```
  env VSI_LOG_LVL=info VSI_LOG_COMPS=tcp_server_0_1trace ./executable
  ```

printf and puts

Often, printf and puts is used to debug C/C++ functions. In a multithreaded application with parallels paths, especially when debugging timing issues and parallel execution paradigms, it can become challenging to figure out which thread is printing.

VSI Runtime adds a helper template to printf and puts. `{{tid}}` inside the formatting argument will be replaced by a unique ID for the current thread. For example,

```
printf("write to memory {{{tid}}}");
```

will be printed as

```
write to memory {12AE}
write to memory {A92F}
```

where 12AE and A92F are unique ids for two threads accessing the same function.

Project build Commands

- vsi::build_projects:

```
Build generated software projects. Output is saved to build.log file
```

- vsi::clean_projects:

```
Cleans built projects.
```

Command line options

- vsi::src_copy_files:

```
 Copies source files to vsi_auto_gen directory. Source files are all the *.cxx and *.h files in the directory se
lected in VSI Software Import Wizard.
```

- vsi::src_link_files

```
Create a symbolic link to  each source file instead of copying.
```

VSI Templating Language

Overview

Why

Visual System Integrator ilang compiler takes the visual representation of a system and generated several projects from it, one per execution context or subsystem. A project comprises of:

- Generated code in the supported language for that context
- Surrounding build system to generate executable for that context

In order to keep ilang compiler agnostic of the underlying language and build systems for each context, it works through an intermediary language.

What

VSI Templating language is a mixed syntax language which can traverse through a data-tree and convert arbitrary text files, filling in the missing values. It can also convert a single file into multiple files, each file generated using a sub-tree from the system data-tree. It does it while being agnostic to what those text files are. The Templating syntax is denoted by curly braces i.e `{{name}}` .

Rules

The secondary syntax for templating is always surrounded by double curly braces.

Anything not surrounded by curly braces is treated as primary syntax of the target language and is not processed.

For instance, consider template A:

```
const int i = 0;
const char *a = "{{name}}";
```

will be transformed to:

```
const int i = 0;
const char *a = "context name";
```

The secondary syntax of templating language refers to a node by it's name. In order to point to a child node of a root node, a period is used i.e `{{parent.child}}`

- ### Replacement

  In it's most simplest form, a transformation rule is a replacement operation which replaces a given rule by the compile time value of the system. i.e. {{name}} would be replaced with the context name.

- ### If

  If rule is a section rule and is defined in pair. To do an if check on any node, use the operator `if` and to close the section, use `end` . For instance, consider the following template:

  ```
  {{if declareint}}
  const int i = 0;
  {{end}}
  ```

will be transformed to:

```
const int i = 0;
```

Only if `declareint` node is set. In case of it not being set, the generated code will not contain this section. An optional `{{else}}` or `{{else if condition}}` can be used to include alternative code.

```
{{if declareint}}
const int i = {{declareint}};
{{else if declarestr}}
const char *str = "{{declarestr}}";
{{else}}
const int j = 0;
{{end}}
```

Optionally, each condition can be a comparison such as `==` or `!=` i.e `{{if declareint != 10}}`

- ## If Not

If Not rule is a section rule and is defined in pair. To do an if-not check on any node, use the operator `if !` and to close the section, use `end`. For instance, consider the following template:

```
{{if !declareint}}
const int i = 0;
{{end}}
```

will be transformed to:

```
const int i = 0;
```

Only if `declareint` node is NOT set. In case of it being set, the generated code will not contain this section.

- ## If equals

If equals compares a given value or two variables and the block is only Not rule is a section rule and is defined in pair. To do an if-not check on any node, use the operator `if !` and to close the section, use `end`. For instance, consider the following template:

```
{{if declareint == 10}}
const int i = 0;
{{end}}
```

will be transformed to:

```
const int i = 0;
```

Only if `declareint` node is type int and has a value 10. If the declareint is not set or has value other than 10, the block will not be included.

- ## Contains

Contains rule is a section rule and is defined in pair. To do an Contains check on any node, use the operator `contains` and to close the section, use `end`. For instance, consider the following template:

```
{{contains ctx.TEMPLATE}}
const int i = 0;
{{end}}
```

will be transformed to:

```
const int i = 0;
```

Only if ctx contains a child node of string type AND the child node contains 'TEMPLATE' i.e. having the value 'TEMPLATE | CONST'. If the child node doesn't contain 'TEMPLATE', the generated code will not have the section.

- ## For

For rule is a section iteration rule and is defined in pair. It will do a for loop for the immediate child nodes and will generate 0 to n amount of section. To do a For loop on a node, use the operator `for` and to close the section, use `end`. For instance, consider the following template:

```
{{for ctx.blocks}}
const int {{name}} = 0;
{{end}}
```

will be transformed to:

```
const int block1 = 0;
const int block2 = 0;
const int block3 = 0;
```

If ctx contains a blocks child node with child nodes of type list or type object with a name node, each with value block1, block2 and block3. If no child nodes exist then For acts like an If and the generated code will not have the section.

- Optional syntax:

  ```
  {{for block: ctx.blocks}}
  const int {{name}} = 0;
  {{end}}
  ```

  The `{{block}}` contains the key if the block is an object and can be used in combination with the `[]` accessor to access the current value i.e `ctx.blocks[block]`.

  ### loop variables:

  Following variables are automatically set when iterating using `for`.

- `{{loop.value}}` : Additionally, to access the current child node, a special syntax of `{{.}}` is used.
- `{{loop.key}}` : In the case if the child node is of type object, the key for the object can be accessed by using `{{loop.key}}`
- `{{..}}` : In the case if a For block is inside another For block, the parent node value can be accessed by using the `{{..}}` syntax. Also, each upper level can be accessed by denoting number of prefix dots i.e `{{....}}` will access the value of the parent node three level up. The same rule apply to Replacement rules as well as `{{loop.key}}` key rule i.e `{{.loop.key}}` will access the key of the parent For loop, `{{..name}}` will access a node of type string or int in the object two level up.

- ## [] accessor

similar to a key-value pair map such as std::map, the value of a node in the tree can be accessed using `[]` accessor. i.e. `blocks["zynq_ps"]`. Multiple levels accessors are supported such as `blocks["zynq_ps"]["arg_1_mem"]` etc.