

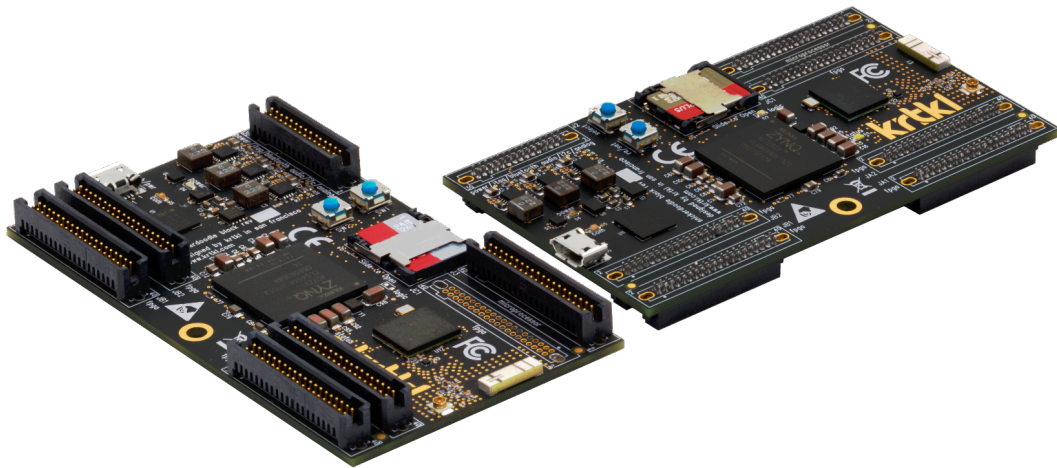
snickerdoodle

ARM
TechCon

krtkl[®]
EMBEDDED SYSTEMS

FROM THE I/O PIN TO THE INTERNET

OCTOBER 26, 2016



How to Read this Document

This document makes extensive use of links, references and notices in the page margins to detail additional information that can be useful while following the guide.



WARNING A warning notice indicates a potential hazard. If care is not taken to adhere to the safety precautions, damage may be done to snickerdoodle.

Warnings and cautions will be clearly visible in either the body of the text or in the margin and must be paid close attention while following the guided steps.



CAUTION A caution indication denotes a process that requires special attention. If the caution is not exercised and the process not adhered to, failure may result and/or potential damage to snickerdoodle.



Warning, caution and informational notices, such as this one, may also be found in the margin.

Keywords

Keywords and important terms are shown in *italicized* type. Additional important information can be found in the margins of text with superscript notation¹.

Navigation of menus and directories are shown using ***bold italicized*** type. Any hierarchical navigation is shown using an arrow to denote a ***Parent*** » ***child*** relationship.

Teletype text is used to highlight inputs, variables and system files within the host environment.

¹ Margin notes, such as this one, reference the body content and highlight technical details or references for further information.

1 Creating a New Project

Creating a new project can be done by selecting **File » New Project...** from the menubar or by selecting the *Create New Project* icon from the "Quick Start" menu, as shown in Figure 1 .

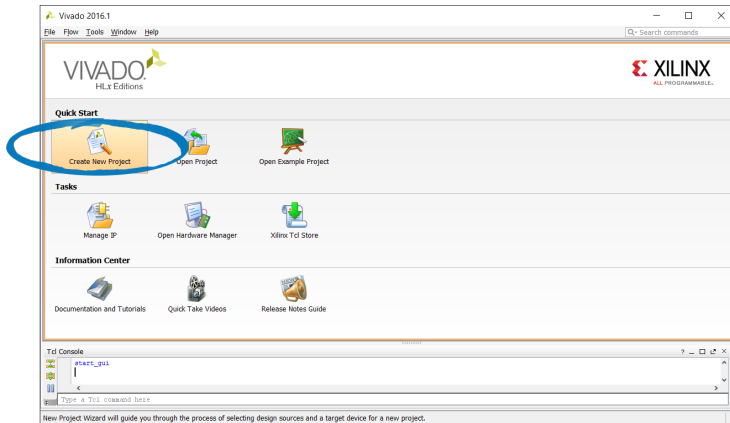


Figure 1: Vivado Startup Create New Project

The first step to creating a new Vivado project is to select the project name and parent directory. By default, Vivado will create a new subdirectory for which to store the project files. It is highly recommended that this setting be left unchanged to keep the project files and directories organized and easily accessible for development (*i.e.*, from the SDK). Figure 2 shows the input of the project name and parent directory when creating a new project.

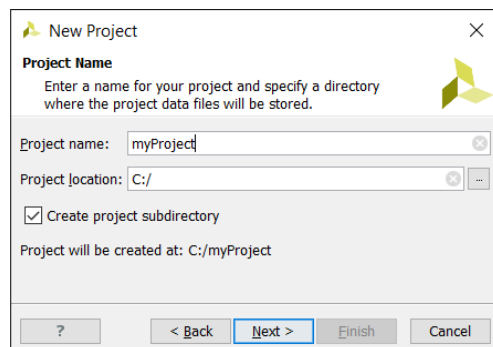


Figure 2: Selecting Project Name and Parent Directory

1.1 Project Type and Sources

At this point in the project creation process, IP and design sources can be added to the project. If you would like to include existing sources, you will

prompted to include those before choosing a project board. If you choose not to include any existing assets or constraints, the **Do no specify sources...** checkbox, shown in [Figure 3](#) , can be selected to skip this step. Sources can be added to the project at any time after creation as outlined [below](#).

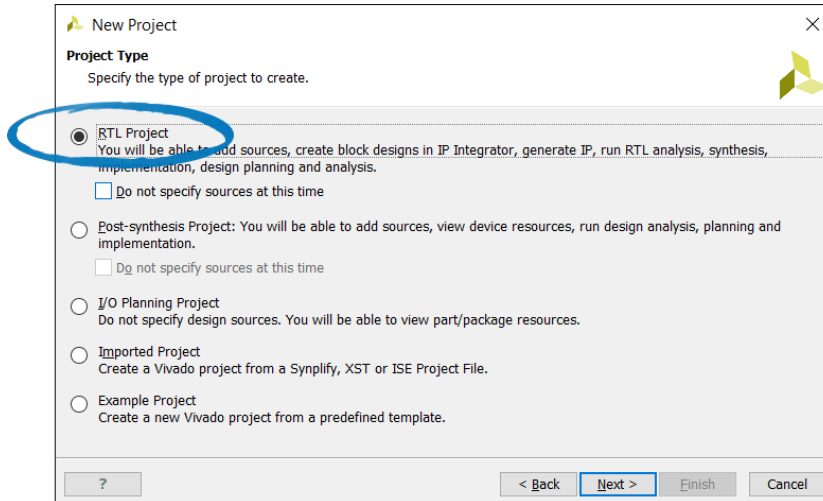


Figure 3: Project Type Selection Dialog

1.2 Choosing a Project Board

If the board files have been copied and loaded properly, they will be listed in the table of boards. To view the available boards, choose "Boards" rather than "Parts" at the top of the window as shown in [Figure 4](#) .

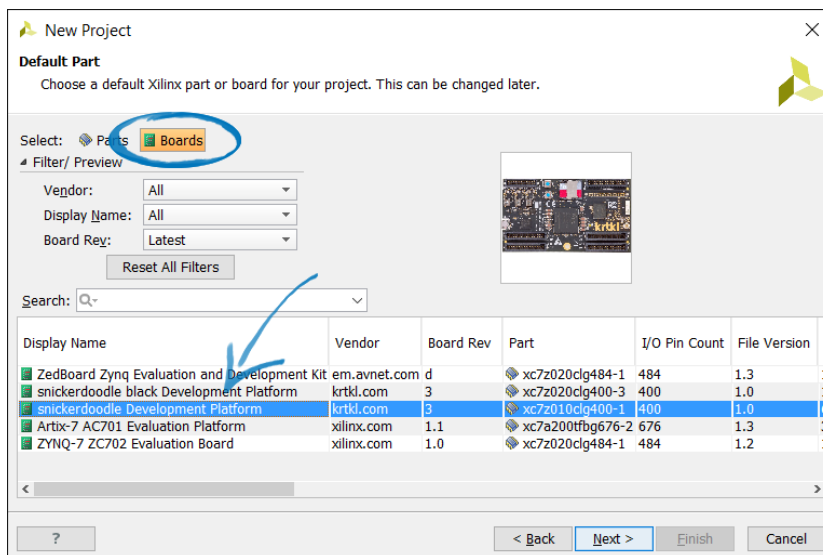


Figure 4: Selecting a Project Board

After selecting a board, the details of the project will be listed in the *New Project Summary* shown in [Figure 5](#) . By clicking the "Finish" button, the project will be created and opened in the Vivado graphical user interface (GUI).

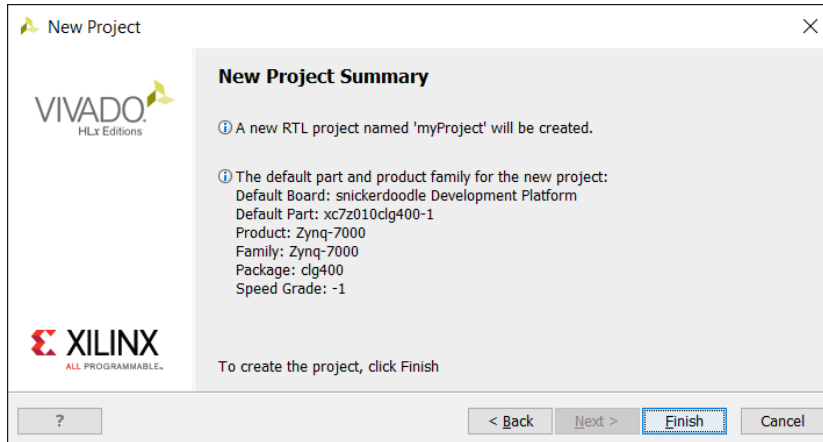


Figure 5: Review and Finish New Project Creation

2 Working with Vivado

After creating an empty project, you are free to import existing assets or develop sources from scratch. To get started with development, a block design needs to be created.

2.1 Create Block Design

A block design is a visual representation of the hardware configuration. Hardware configurations can be defined by including or creating sources and assigning hardware I/O. [Figure 7](#) shows an example block design with hardware definitions for fixed peripherals such as memory interfaces and a some GPIO ports defined in programmable logic.

To create a block design for a new project, select **Flow » Create Block Design** from the menubar or from the "IP Integrator" section of the *Flow Navigator* pane as shown in [Figure 6](#) .

Adding and integrating sources and IP into a block design can vary greatly depending on the design and application. For additional reading and information on integrating IP can be found at .

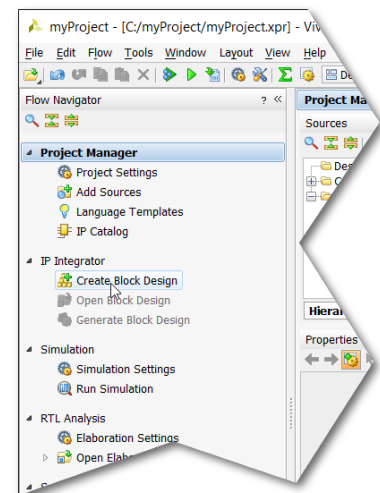


Figure 6: Create Block Design

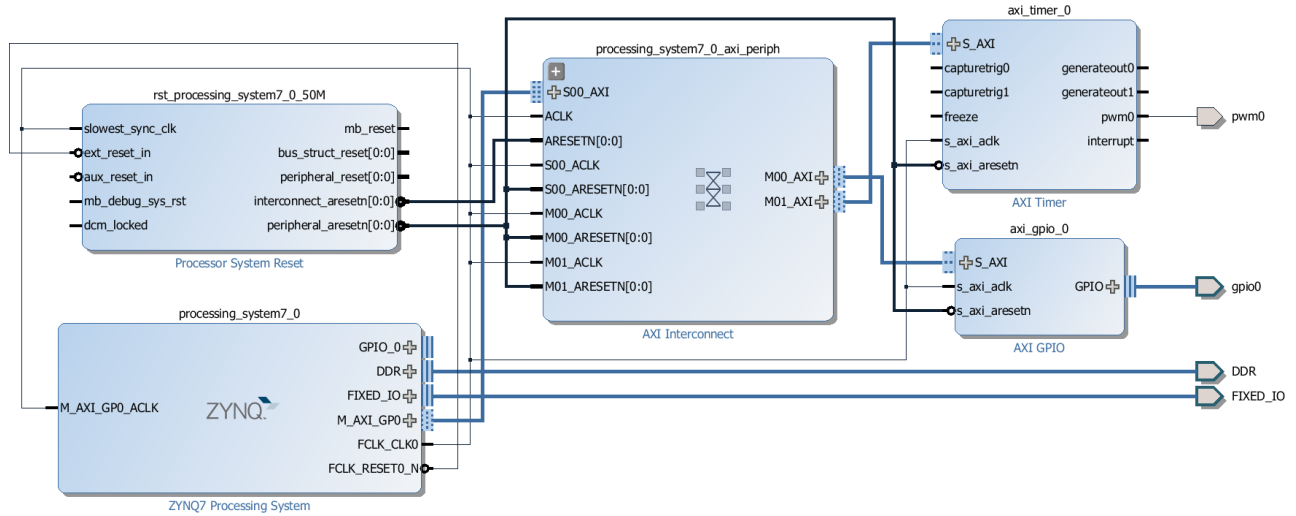


Figure 7: Example Block Design with Zynq Processing System and Microblaze Soft Core Processor

2.2 Create HDL Wrapper

Before generating a bitstream for the project, an HDL wrapper must be generated for the design. To do this, right click on the design (in this case "base_design") from within the *Sources* pane and select **Create HDL Wrapper...**

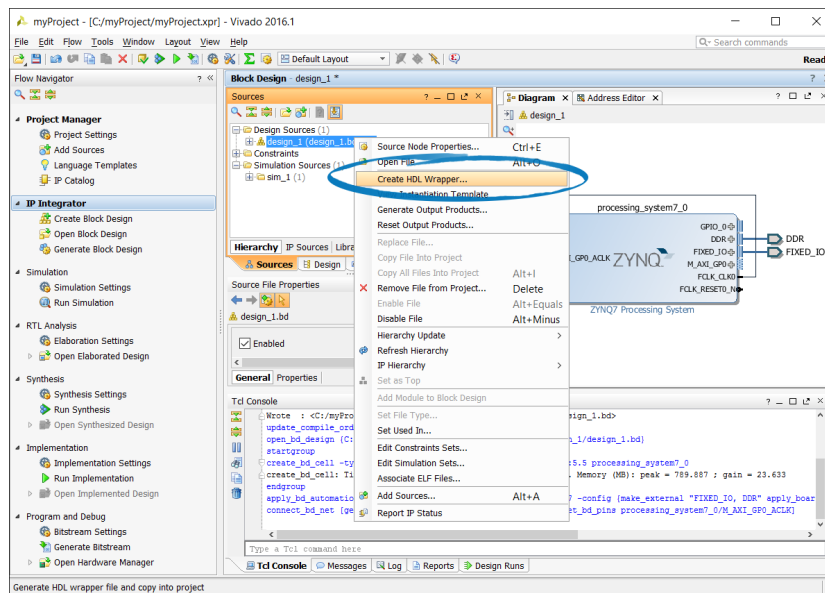


Figure 8: Creating an HDL Wrapper for the Design

2.3 Getting Ports (get_ports)

After synthesizing and implementing the block design, the `get_ports` command can be used in the TCL console to view the ports available to be constrained. Before the synthesis and implementation is complete, the `get_ports` command will produce an error message as shown in [Code Listing 1](#).

```
> get_ports
ERROR: [Common 17-53] User Exception: No open design. Please open
an elaborated, synthesized or implemented design before
executing this command.
```

Code Listing 1: Error When Getting Ports Before Synthesizing and Implementing Design

After running synthesis and implementation on the design, the ports can be viewed using the `get_ports` command inside the TCL console within Vivado. The port names returned by this command can be used as constraints to route the PL defined interfaces to specific pins.

```
> get_ports
...
gpio0_tri_io[0] gpio0_tri_io[10] gpio0_tri_io[11]
```

Code Listing 2: Example TCL Console Output of `get_ports` Command

2.4 Add Sources and IP

Adding sources can be done by right clicking inside the *Sources* pane or by selecting **File » Add Sources...** from the menubar. The *Add Sources Wizard* will appear and allow you to select existing sources, IP and constraints to add to the project.

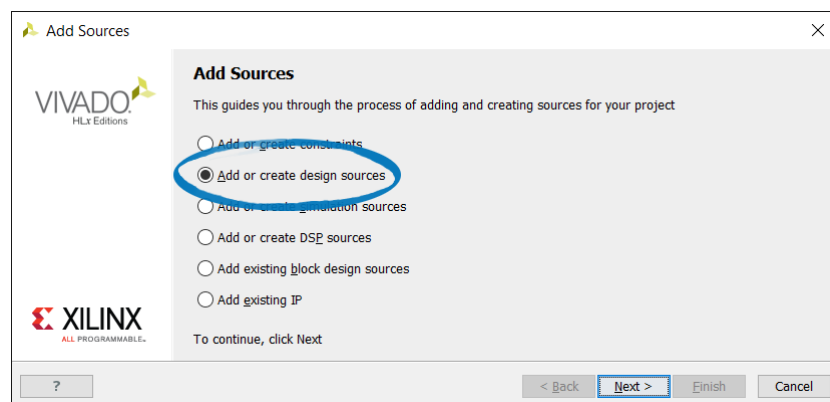


Figure 9: Adding New Sources to the Project with the Add Sources Wizard

2.5 Adding Constraints

Peripheral interface ports are mapped in the desired FPGA pin using the constraints file. Add a pair of lines to the constraints file for each pin to define the pin/port assignment and the logic level. The logic level should be the same for all pins on a single I/O bank. Connectors JA1 and JA2 share an I/O bank. JB1 and JB2 share an I/O bank. JC1 (on snickerdoodle black) has it's own I/O bank.

The *Add Sources Wizard* can be used to create constraints files by selecting *Add or create constraints* and selecting *new file...* and selecting the constraints type and file name as shown in [Figure 10](#) . An XDC constraints file is essentially a tcl script that sets the system constraints by declaring a set of properties as shown below:

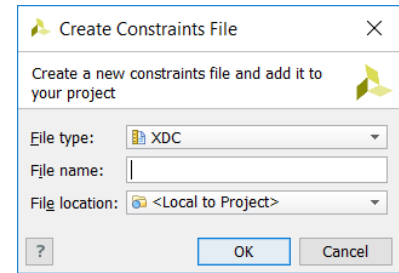


Figure 10: Create Constraints File Dialog

```
#####
#
# Constraints file for snickerdoodle black
#
# Copyright (c) 2016 krtkl inc.
#
#####
#
#-----
# Constraints for GPIO outputs
#-----
# JA1 Connector
#-----
### JA1.4 (IO_0_35)
set_property PACKAGE_PIN G14 [get_ports {gpio0_tri_io[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio0_tri_io[24]}]

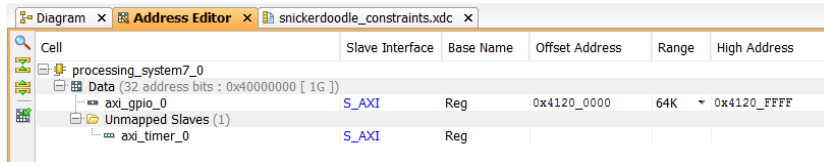
### JA1.5 (IO_L5P_T0_AD9P_35)
set_property PACKAGE_PIN E18 [get_ports {gpio0_tri_io[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio0_tri_io[8]}]

### JA1.6 (IO_L4N_T0_35)
set_property PACKAGE_PIN D20 [get_ports {gpio0_tri_io[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio0_tri_io[11]}]
```

3 Address Editor

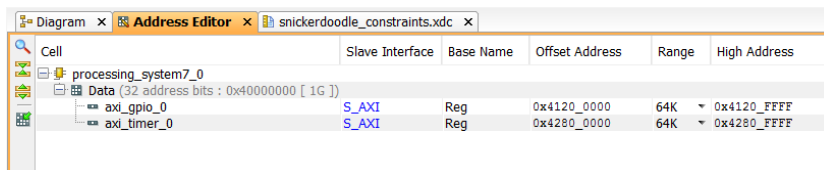
Hardware interfacing between the processing subsystem and programmable logic can be done through the register space of the hardware block. Each hardware block that requires it, can be assigned a register address space from which the processing subsystem expects to access the hardware. Assigning the register space of a hardware interface is done with the *Address Editor*. In

Linux, the offset address assigned to a hardware peripheral will need to be specified for any drivers that are required for control of the peripheral. The specification of the register space and assignment of a driver to the hardware peripheral is done using a node in the device tree. The declaration of the hardware peripheral within the device tree is covered later in this document.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
Unmapped Slaves (1)					
axi_timer_0	S_AXI	Reg			

Figure 11: Unmapped AXI Timer in Address Editor



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_timer_0	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF

Figure 12: Address Editor

3.1 Generating Bitstreams

Bitstreams can be generated by selecting **Flow » Generate Bitstream** from the menubar or from within the "Program and Debug" section of the *Flow Navigator* pane as shown in Figure 13. The bitstream contains all the information necessary to define the programmable logic and the associated hardware peripherals/interfaces.

4 Exporting Design with SDK

4.1 Export the Hardware Platform

Designs that are implemented using Vivado can be exported and opened in the SDK directly from Vivado. Before a design can be opened in the SDK, the hardware profile must be exported which can be done by selecting **File » Export » Export Hardware...** from the menubar.

Figure 14 shows the options for hardware export. Select the "Include bitstream" checkbox to include the bitstream with the hardware definition files for use with the SDK. If you would like to use the hardware platform in a workspace other than the hardware project directory, change the selection of the "Export to:" input. This will be the workspace for the SDK.

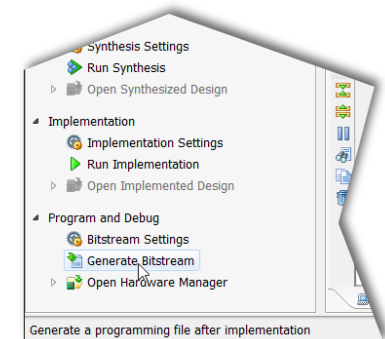


Figure 13: Generate Bitstream from Vivado Flow Navigator



If you choose an export location other than <Local to Project>, the SDK will need to be launched separately from Vivado to access the exported workspace location

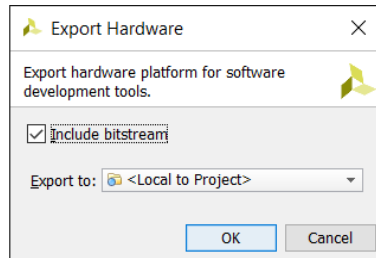


Figure 14: Export Hardware for Software Development Dialog

4.2 Launching the SDK

The SDK can be launched directly from Vivado to use the exported hardware profile for software development. To launch the SDK from Vivado, select **File** » **Launch SDK** from the menubar. If the hardware platform was exported to a directory within the Vivado project (by selecting <Local to Project> as export location), the hardware platform will be immediately available within the SDK workspace. If the hardware was exported to a different directory, you will need to change the workspace directory after the SDK has launched.

5 First Stage Boot Loader (FSBL)

The FSBL is used to initialize the processing subsystem and prepare it to load a user application or operating system. During the boot process, the BootROM loads the FSBL to the on chip memory. The FSBL can be made to load a bit-stream to the programmable logic before loading additional programming to the processing subsystem.

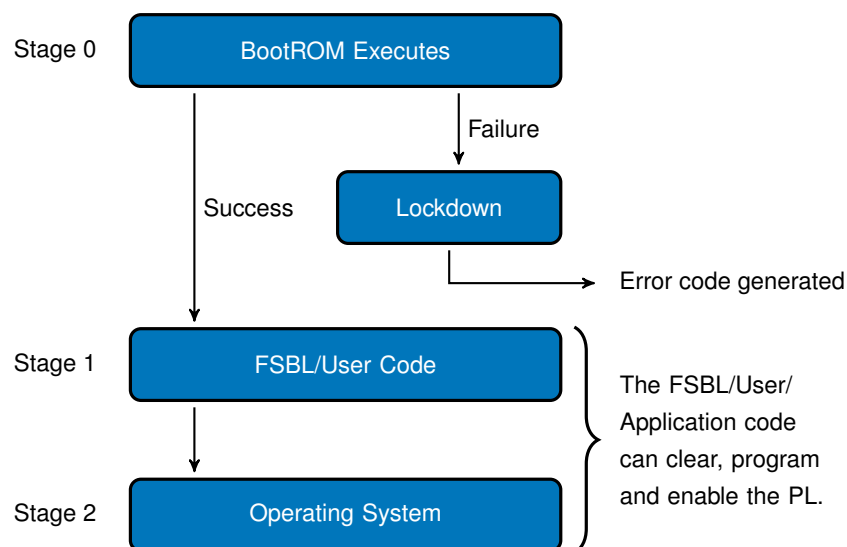


Figure 15: PS/PL Boot Process for Hardware and Software(? , p. 151)

6 Create FSBL Project

Within an exported hardware SDK workspace, an FSBL project can be automatically generated. The SDK uses the hardware definition files to determine the configuration data for the FSBL. The hardware definition files also contain initialization code that can be used to build make custom builds of U-Boot which, combined with the FSBL, can be used to generate a Snickerdoodle boot image.

6.1 FSBL Application Project Platform and BSP

As shown in [Figure 16](#), the FSBL can be created in the same way as a bare-metal application project by navigating to **File » New » Application Project**. From the New Project dialog, select a **standalone** platform. A project name can be entered from this interface.

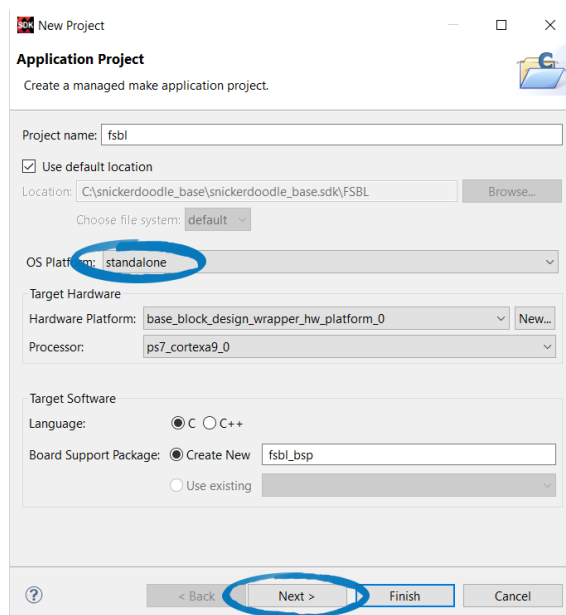


Figure 16: Create a New FSBL Project in SDK

6.2 FSBL Application Project Template

By clicking **Next>**, a project template option dialog ([Figure 17](#)) will appear. Within this dialog window, an FSBL project template can be selected, which will use the hardware configuration to generate the initialization code for the FSBL. Clicking **Finish** will complete the project generation and build the FSBL code from the workspace hardware configuration.

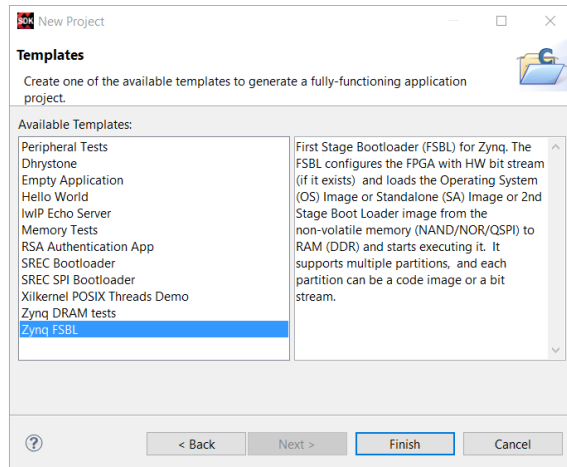


Figure 17: Create FSBL Project from Template in SDK

The FSBL template will read the hardware definition that was exported by Vivado and generate code required to configure the processing subsystem. A set of initialization files will be generated which is used by the FSBL to configure the processing subsystem before handing off execution to a second stage bootloader or user application. These initialization files can also be used to initialize the processing subsystem for bare-metal applications or custom operating systems.

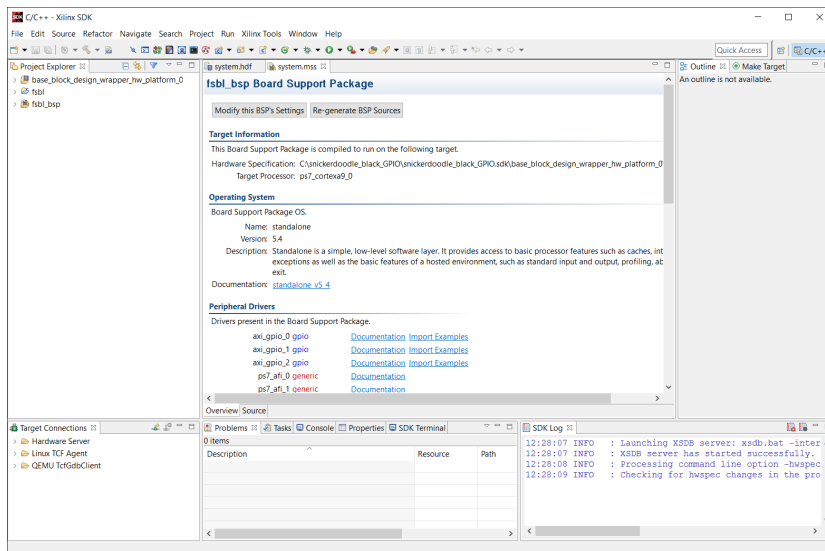


Figure 18: Exported Hardware Workspace with FSBL Project and BSP

7 Boot Image (*BOOT.bin*)

Whether using a pre-built hardware configuration or working with a custom solution, a bootable image must be produced which initializes the processing subsystem and (optionally) the programmable logic. The specific components that are included in a boot image depend heavily on the requirements of the application. While a single boot image configuration cannot be made to suit all applications, a generic boot configuration can be used as a starting point for customization.

The boot image should contain, at least, all of the information required to initialize the processing subsystem, typically through a combination of FSBL and U-Boot boot loaders. A bitstream which defines the programmable logic configuration can be loaded upon boot by building the bitstream into the boot image or a minimal boot image can be used if the bitstream is to be loaded from user/application space.

8 Boot Information File (*.bif*)

The components of a boot image are defined by a boot information file (*.bif*). The structure of the *.bif* will depend on the specific application details of the system. Typical components for creating a boot image are as follows:

FSBL - First stage boot loader (FSBL). Responsible for loading the bitstream (if one exists), loading into memory and handing off the boot process to the second stage bootloader. The FSBL should always be specified with the *bootloader* tag in the *.bif*.

U-Boot - Second stage boot loader. Responsible for loading Linux system components (devicetree, ulmage, file system)

Bitstream - The bitstream to be built into the boot image.

Device Tree - The Linux devicetree to be loaded by FSBL or U-Boot.

Kernel - Linux kernel image to be loaded by FSBL or U-Boot.

8.1 Small Initialization Boot Image

A very small boot image can be built using FSBL and U-Boot. An image with this configuration can be used to load application or operating system images from a specified boot medium (*i.e.*, microSD, QSPI). This boot image provides no application programming, on its own. An advantage of this approach is modularity and flexibility of the other system components and choice of boot medium. [Code Listing 3](#) shows the *.bif* structure for producing a small boot image.

```
image : {
    [bootloader]fsbl.elf
    u-boot.elf
}
```

Code Listing 3: Minimal Boot Image with FSBL and U-Boot

8.2 Load Images Using FSBL

An entire standalone image can be produced with all of the components required to boot Linux. The first configuration of such a boot image is specified to load the system components into memory using FSBL. With this specification, the Linux components will be loaded into memory before FSBL hands off execution to the second stage boot loader (U-Boot). The `load=0xFFFF` tag is added to any images that should be loaded by the FSBL during the boot process. This will place the image in the specified memory location, from which the system can be booted using `bootm` from U-Boot. [Code Listing 4](#) shows an example `.bif` specification with the memory addresses to load the Linux components. With this configuration, the images are pre-loaded and thus the boot process is well-defined and inflexible but simple.

i The `ulmage.bin` file specified in the `.bif` is simply a `ulmage` that has been renamed with `.bin` for compatibility with `bootgen`

```
image : {
    [bootloader]fsbl.elf
    u-boot.elf
    [load=0x2a000000]devicetree.dtb
    [load=0x20000000]uramdisk.image.gz
    [load=0x30000000]uImage.bin
}
```

Code Listing 4: Load Linux Components Using FSBL

8.3 Load Images Using U-Boot

The third configuration, while not nearly exhausting the possible configurations, to consider is building a boot image with the system components at known offsets within the image. The `offset=0xFFFF` tag is added to the components to control their location within the boot image. This is a convenient architecture for building standalone images not dependent on outside sources while not pre-loading the components into memory. With this configuration, the system remains flexible to loading outside components from boot medium while keeping a known bootable set of components within a singular boot image.

CAUTION To avoid creating a boot image of unnecessarily large size, the offsets for the system components should be carefully calculated using the file sizes of the components (including FSBL and U-Boot).

```

image : {
    [bootloader]fsbl.elf
    u-boot.elf
    [offset=<dt-offset>]devicetree.dtb
    [offset=<ramdisk_offset>]uramdisk.image.gz
    [offset=<uimage_offset>]uImage.bin
}

```

Code Listing 5: Load Linux Components from Known Boot Image Offsets

9 Building Bitstream into *B00T.bin*

Bitstreams can be built into the boot image and specified to be loaded when the boot image starts the boot process. This method can be useful for bitstreams that need to be loaded *before* the operating system is booted, especially in cases where the operating system expects PL devices to be available during or immediately after the boot process which includes cases where PL devices are defined in the devicetree blob. There are two methods for building the boot image (B00T.bin): a GUI based method executed from the SDK environment and a command line method from which the GUI process is derived. The boot image can be used to load bare-metal applications as well as general purpose operating system (e.g., Linux, FreeRTOS).

9.1 Building Boot Images from SDK

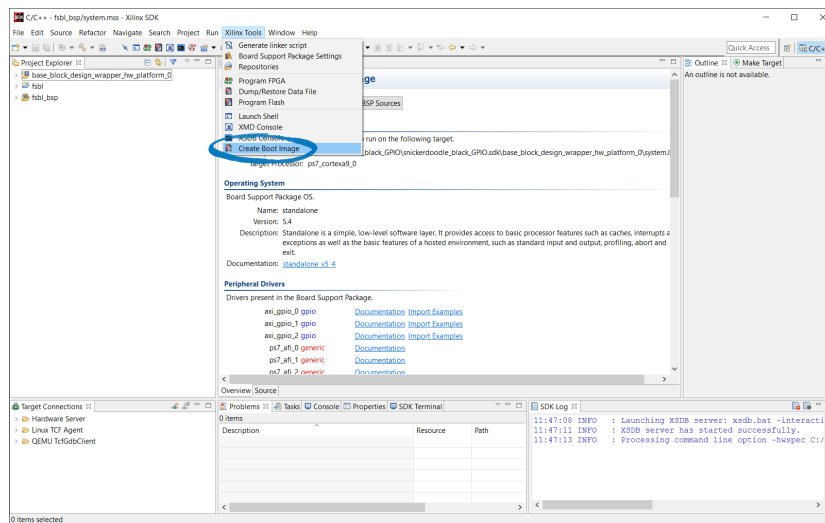


Figure 19: Create Boot Image SDK Menu Selection

The SDK provides a graphical way to select the components/partitions for the boot image and generate a reusable boot image file (.bif). The graphical front-end provides the necessary interface for selecting existing boot partitions

(typically pre-compiled and/or provided by Vivado project) and will generate the .bif file along with the boot image. Figure 20 shows the graphical interface for creating Zynq boot images from the SDK. To access the interface and begin generating images, select **Xilinx Tools » Create Boot Image** from the menubar in the SDK as shown in Figure 19 .

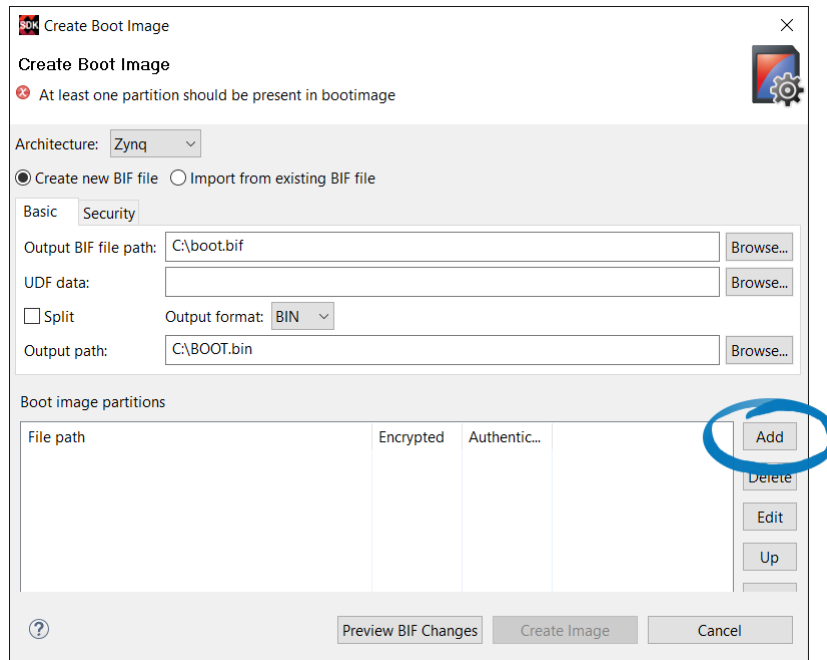


Figure 20: Create Boot Image Interface in Xilinx SDK

Prebuilt .bif files can be used by the SDK interface by selecting "Import from existing BIF file", shown at the top of the window in Figure 20 . Boot partitions are listed in the "Boot image partitions" table within the interface. To add boot partitions from the interface, selecting "Add" will open an "Add partition" dialog (shown in Figure 21) which will allow you to select and specify the partition file.

9.2 Import and Specify Boot Images Sources

The most typical boot image for a Linux based system includes a FSBL, bitstream and U-Boot which will be used to load Linux. The image sources should be added one-by-one by clicking *Add* from the *Create Boot Image* interface as shown in Figure 20 . FSBL should be specified as the bootloader by selecting *bootloader* as the partition type as shown in Figure 22 .

After adding the FSBL, add the bitstream and U-Boot images from the filesystem. Add the sources as *adatafile* as shown in Figure 23 . Prebuilt U-Boot executables can be downloaded from [GitHub](https://github.com/krtkl)² and copied to the workspace

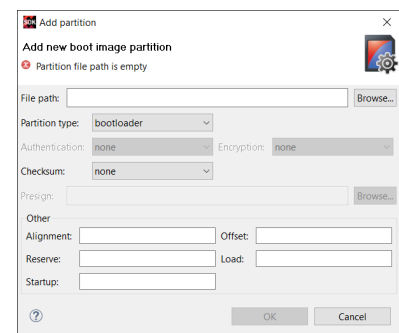


Figure 21: Create New Boot Image Partition

² <https://github.com/krtkl>

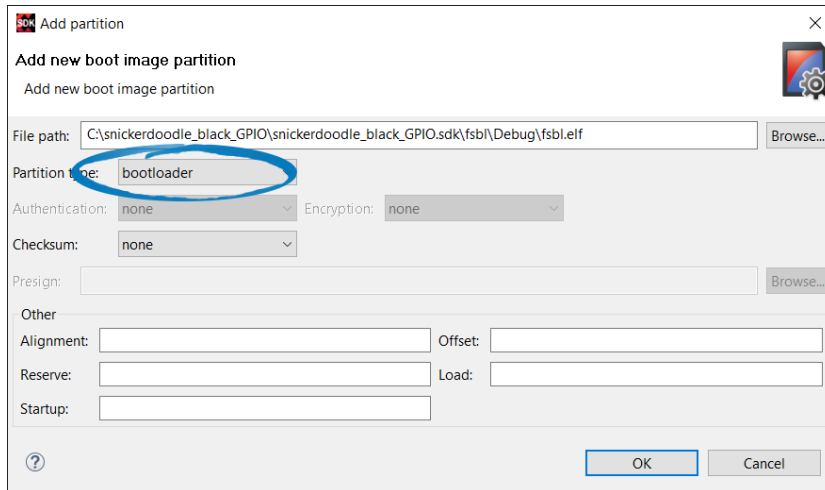


Figure 22: FSBL Bootloader Boot Image Partition

directory for import into boot image generation. Additional system components can be added in the same way and *Offset* or *Load* locations can be specified in the *Add partition* dialog.

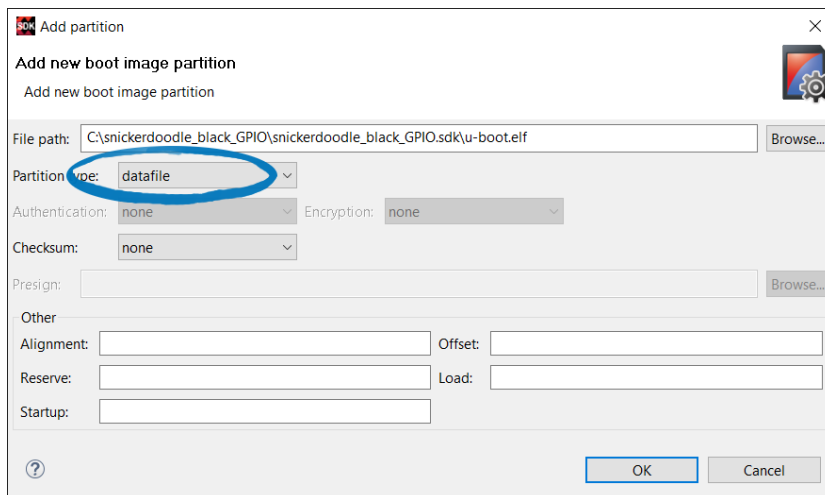


Figure 23: U-Boot Boot Image Partition Selection

After importing the boot image sources, the boot image interface should look similar to [Figure 24](#) .

After selecting **Create Image**, the SDK console should output the status of the bootgen command and the generated boot image location. The generated boot image can then be loaded onto a microSD card and used as a boot source.

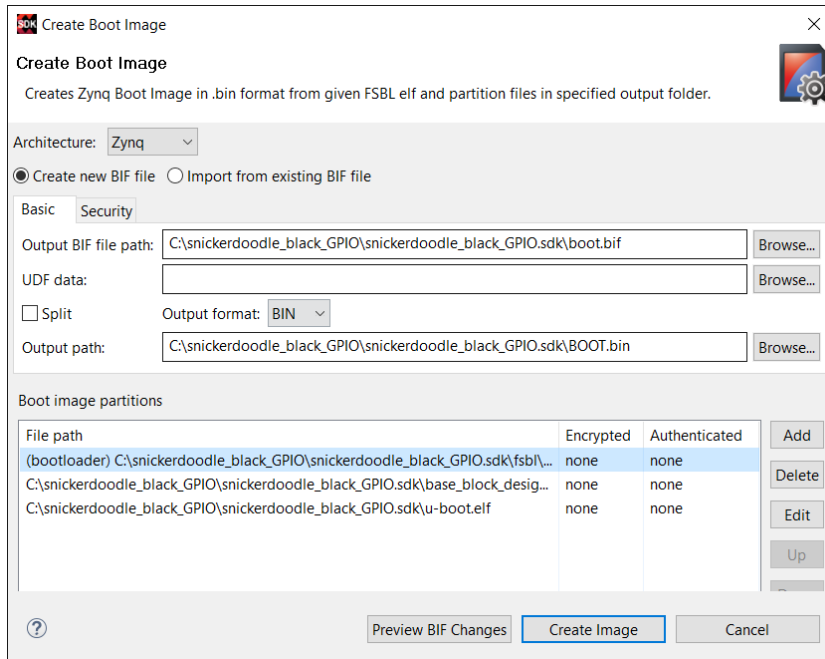


Figure 24: Boot Image Partition Table

```
cmd /C bootgen -image boot.bif -arch zynq -o \
C:\snickerdoodle_black_GPIO\snickerdoodle_black_GPIO.sdk\BOOT.bin
```

Code Listing 6: SDK Create Boot Image Console Output

9.3 Building Boot Images with bootgen

The GUI interface for building boot images from within the SDK is simply a wrapper for the command line executable that serves the same purpose. `bootgen` can be provided with a `.bif` file to specify the boot image partitions. Below is an example `.bif` file:

```
image : {
    [bootloader]fsbl.elf
    bitstream.bit
    u-boot.elf
}
```

After copying the boot partition file and the `.bif` file to a local working directory, the `bootgen` utility can be executed from that directory to output the boot image. Invoking the following command will create a boot image (`BOOT.bin`) using the boot image file `boot.bif`:

CAUTION When building boot images with `bootgen` the boot partition files and `.bif` file should be isolated in a working directory from which `bootgen` is executed.

Code Listing 7: Boot Information File

Code Listing 8: Generate `BOOT.bin` with `bootgen`

```
$ bootgen -image boot.bif -o i B00T.bin
```

10 Linux

10.1 Device Tree

Declaration of FPGA hardware peripherals in the device tree allows them to be configured and controlled by the Linux kernel using device drivers. This allows the control of the hardware to be abstracted to user-space using any of a number of interfaces including sysfs. The device tree node for a hardware peripheral specifies the device driver to use for hardware interfacing in the compatible parameter. The hardware register address space is declared in the reg parameter.

```
gpio@41200000 {
    compatible = "xlnx,xps-gpio-1.00.a";
    gpio-controller;
    #gpio-cells = <0x2>;
    interrupt-parent = <0x3>;
    reg = <0x41200000 0x10000>;
    xlnx,is-dual = <0x0>;
    xlnx,all-inputs = <0x0>;
    xlnx,tri-default = <0xffffffff>;
    xlnx,gpio-width = <0x19>;
};
```

Code Listing 9: AXI GPIO IP Peripheral Device Tree Node

```
timer@42800000 {
    clock-frequency = <500000000>;
    #pwm-cells = <0x2>;
    clocks = <0x1 15>;
    compatible = "xlnx,pwm-1.00";
    reg = <0x42800000 0x10000>;
};
```

Code Listing 10: AXI Timer (PWM) IP Peripheral Device Tree Node

The device driver will determine how the hardware peripheral is abstracted to user-space (if at all). In the case of the GPIO and AXI timer peripherals, the user-space abstraction is done through sysfs.

10.2 Hardware Control and Observation in User Space

Many hardware blocks can be controlled using built-in Linux drivers that abstract control of the hardware to file I/O. This allows interfacing with the hard-

were using a familiar Linux interface. The following code snippet shows an example routine for setting the value of a GPIO output pin by opening, writing to, and closing the value file of that pin in the GPIO subsystem of sysfs.

```
/**
 * @brief Set GPIO Value
 *
 * @param pin: Pin ID to write value
 * @param val: Value to write to the output pin
 * This parameter can have the following values:
 * @arg GPIO_HIGH: Set the gpio output high
 * @arg GPIO_LOW: Set the gpio output low
 * @retval 0 on success, error status otherwise
 */
int gpio_write(int pin, int val)
{
    int len, fid, status;
    char pin_path[BUFFER_LENGTH];
    char pin_val[2];

    len = snprintf(pin_path, BUFFER_LENGTH, "/sys/class/gpio/gpio%d/value", pin);

    fid = open(pin_path, O_WRONLY);

    if (fid < 0) {
        fprintf(stderr, "Failed to open GPIO %d value for writing\n", pin);
        return -1;
    }

    len = snprintf(pin_val, 2, "%d", val ? 1 : 0);

    status = write(fid, pin_val, len);

    if (status < 0) {
        fprintf(stderr, "Failed to set value for GPIO %d\n", pin);
        return -2;
    }

    close(fid);

    return 0;
}
```