SNICKERDOODLE BOOK

krtkl
EMBEDDED SYSTEMS

Snickerdoodle Book

krtkl inc.
350 TOWNSEND STREET
SUITE 301A
SAN FRANCISCO, CA 94107
UNITED STATES

Contents

List of Figures

List of Tables

List of Code Listings

Preface

Who Should Read This?

Learning something new should not require some extraordinary bravery. Learning something new should be fun and exciting. This text has been written to give people, of all backgrounds (including n00bs), the tools they need to get started on the projects they always wanted to build.

> *I am always doing that which I cannot do, in order that I may learn how to do it.*
>
> *Pablo Picasso*

Bringing ideas to life, especially ideas that rely heavily on technology, often requires a mixture of inventing new technologies and integrating existing technologies. Often the integration is more cumbersome than the invention, forcing people to focus on the nuts and bolts of their project rather than the part of their project they are most passionate about. With Snickerdoodle, a great deal of the heavy-lifting in integration has been done. This can significantly reduce development time and effort and allows you to bring your projects to completion faster. More importantly, this allows you to keep your focus on the high-level design of your project and what makes it unique.

This book is intended to provide an overview of this hardware-centered design approach. From this text you will gain a basic understanding of the capabilities Snickerdoodle and the design workflow for realizing those capabilities in applications. This text does not go into detail about specific applications (although many examples are offered) to keep the survey to a high-level overview.

Snickerdoodle introduces a level of customization that is not typically available on prototyping and development platforms. It accomplishes this by shifting some of the hardware design into the software domain. This adds an element of design that most developers are not used to addressing. Normally, hardware design is rigid and developers need to adapt their designs to fit within the constraints introduced by the hardware. With Snickerdoodle, this is not the case. Hardware can be reconfigured side-by-side with software design to adapt to the needs of a particular system, rather than the other way around.

How to Read This Book

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

WARNING Warnings like this one indicate potential hazards. If care is not taken while following examples, failures and even damage to the board may occur.

CAUTION Notices like this show important details to consider when following examples and design workflows. Failures may occur if notices like this are not followed.

Important and useful information can be found in boxes like this one. These are helpful tips for working with Snickerdoodle and can help avoid issues during development.

Conventions Used

This text indicates navigation items from within development environments and menus through a ***Menu*** » ***Submenu*** » ***Selection*** format. This includes navigating and selecting items from menu and tool bars as well as UI elements such as drop-down menus.

Directory paths within a filesystem are indicated by **`Dir`** » **`Subdir`** » **`File`**. This includes paths on host machines as well as Snickerdoodle.

Linux commands are indicated by `Teletype` text format, when referenced from within the text. Command usage is noted using a code listing with a `$` to indicate a prompt. Code listings are also used to show snippets of code or file structures used for system configuration.

```
$ command arg1 [arg2 ...]
```

Additional Resources

This book is by no means a comprehensive look at the capabilities of Snickerdoodle and much less a comprehensive look at Linux, wireless or embedded control systems. See the list of references at the end of this book, as well as the links within the text for additional reading.

Hardware Architecture

1 Overview

	Snickerdoodle	upgrade options
Chipset	Xilinx® Zynq® -7010	Xilinx Zynq-7020
CPU	32-Bit dual-core ARM® Cortex™-A9 w/640kB cache and dual 128-bit NEON™ coprocessors	
Performance	3335 DMIPS/2668 MFLOPS @ 667 MHz	4330 DMIPS/3464 MFLOPS @ 866 MHz
Flash	16 MB XIP NOR and up to 200 GB SDIO NAND via captive microSD card cage	
DRAM	1 GB @ 25.6 Gbps	
SRAM	256 kB @ 28.4 Gbps	256 kB @ 36.9 Gbps
CPU to FPGA/Bandwidth	896-bit @ 150 MHz AXI3/134.4 Gbps	896-bit @ 200 MHz AXI3/179.2 Gbps
FGPA Programmable Logic	430k gates/17600 LUT-6	1.3M gates/53200 LUT-6
32-bit Performance	143150 MIPS @ 350 MHz	587575 MIPS @ 475 MHz
Distributed RAM	270kB/3354 Gbps	630kB/10275 Gbps
DSP Units/Performance	80 @ 464 MHz/74240 MMACs	220 @ 628 MHz/276320 MMACs
Total User GPIO	142	167
FPGA GPIO/performance	16 ADC/100 reconfigurable/46.2 Gbps	16 ADC/125 reconfigurable/75.7 Gbps
Fixed GPIO	28 GPIO, 4 I2S audio, 2 I2C, 5 1-Wire, 1 ADC, 2 DAC	
Wireless	150Mbps SISO 2.4GHz 802.11b/g/n	150Mbps MIMO 2.4GHz/5GHz 802.11a/b/g/n
	3Mbps dual-mode Bluetooth® 4.1 Classic+EDR/BLE, dual-band antenna, switched U.FL ports	
Serial Interfaces	2 gigabit ethernet, 2 CAN, 2 I2C, SPI, UART, USB 2.0 high-speed, microUSB console/JTAG	
Analog Interfaces	dual 1MSPS 12-bit ADCs w/16 channel multiplexer, dual 1MSPS 12-bit DACs	
Other Peripherals	5 LEDs, 2 pushbuttons, secure cryptographic key/certificate storage	
Software Support	krtkl iOS app, Linux, FreeRTOS, ROS, ArduPilot, Wiring, bare-metal C/C++, Vivado	
Power/Dimensions	+5V via microUSB or 3.7V-17V via power pins/3.5" x 2.0" (88.9mm x 50.8mm)	

Table 1.1: Snickerdoodle Features and Upgrade Options

Snickerdoodle is a compact, modular, highly-integrated embedded development platform that combines powerful processing capability with wireless connectivity and flexible configurability. Snickerdoodle utilizes a Xilinx Zynq-7000 series All-Programmable SoC which combines a dual-core ARM Cortex-A9 processor with a Xilinx 7-Series FPGA. The ARM core provides a well known architecture for application development and is an ideal target for an embedded Linux system. The tightly-integrated FPGA fabric allows for connection with a wide variety of peripheral devices as well as the realization of custom logic for high-speed application-specific processing. Snickerdoodle comes with either a Zynq Z-7010 or a more powerful Zynq Z-7020 available as an optional upgrade.

2

Connectors

The USB power input and serial console are both accessed using the USB micro-B receptacle, designated J1. Power input is described in more detail in Chapter 3 . The most notable connectors on Snickerdoodle are the 0.050 in. double row headers. There are two possible connector orientations:

TFM Top-facing shrouded male terminal header

SFM Bottom-facing female socket strip

J2 is a Samtec TFM-115-01-F-D-A/SFM-115-L1-F-D-A. J3, JA1, JA2, JB1, JB2 and JC1 are Samtec TFM-120-01-F-D-A/SFM-120-L1-F-D-A

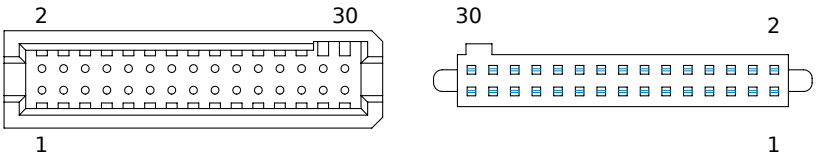


Figure 2.1: Snickerdoodle Connector Types and Pin Numbering (TFM left, SFM right)

CAUTION The pin numbering appears as a mirror image between the TFM and SFM connectors as they are intended to be mounted on opposite sides of the board. This pin numbering is reflected in all tables in this manual.

Figure 2.2 shows a male pin configuration mounted on the top-side of the board for easy access with wire-to-board connectors. Female connectors mounted on the bottom side of the board can be used for modular connection to baseboards. Table 2.1 shows the connectors and their descriptions.

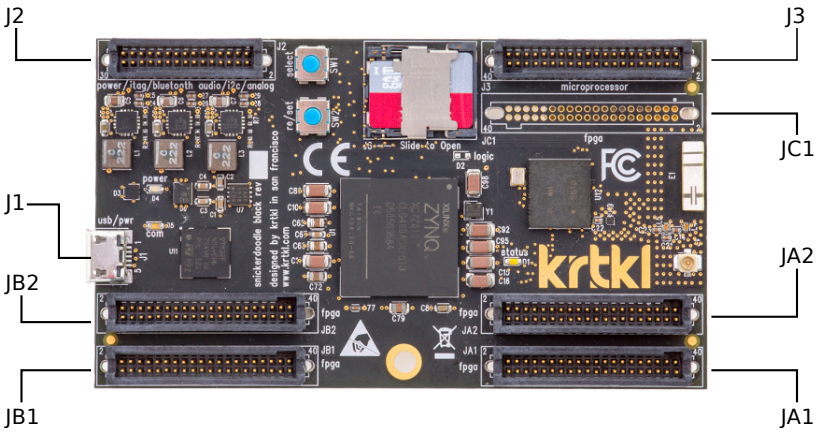


Figure 2.2: Snickerdoodle Connector Designation

Table 2.1: Snickerdoodle Connector Descriptions

Connector	Description
J1	USB and Power Input
J2	Power, JTAG, Bluetooth Audio, I^2C, Analog
J3	Microprocessor subsystem
JA1	FPGA
JA2	FPGA
JB1	FPGA
JB2	FPGA
JC1	FPGA

Processor Subsystem Connectivity

External connection to the processor subsystem (PS) is done through J3 and provides access to multiplexed input/output (MIO). By default, the MIO pins connected to J3 are configured as GPIO by the FSBL and the configuration is carried through U-Boot before being handed off to Linux at boot time. The configuration of these pins can be changed to utilize additional peripheral interfaces including Ethernet, USB host, UART, SPI, CAN and I^2C. The configuration of the PS and the pinout of the connector is illustrated in Figure B.6 .

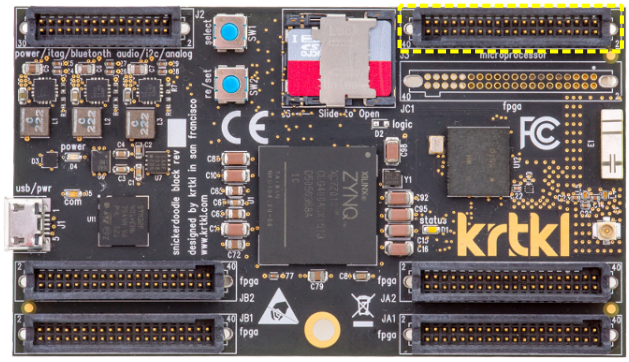


Figure 2.3: Processor Subsystem Connector

Figure 2.4 shows the unallocated MIO pin configuration for the Snickerdoodle default board preset. MIO[16:39] are connected to J3 and can be customized for application specific connections. Changes to the processing subsystem initialization (*i.e.*, FSBL, U-Boot) and the devicetree must be made for any customizations made to the MIO configuration.

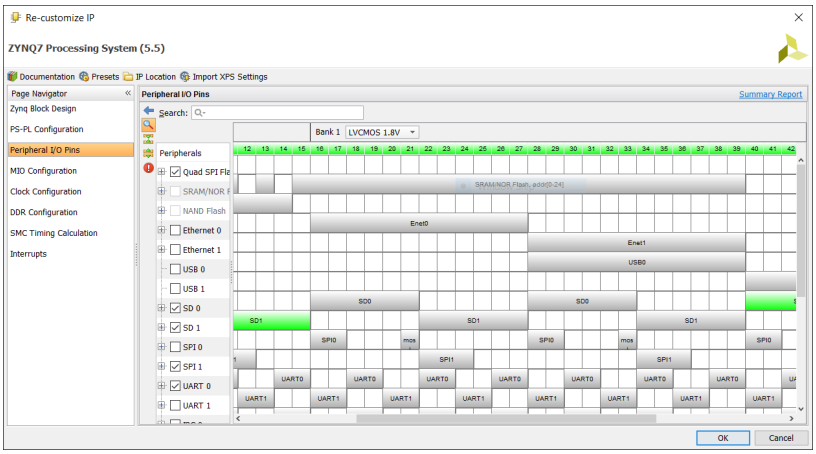


Figure 2.4: J3 Processing Subsystem Peripheral Configuration

FPGA Connectors

WARNING Care must be taken when connecting to the FPGA system as the pins can be reconfigured to a variety of input and output types.

The versatility and flexibility of Snickerdoodle is due to the integrated FPGA fabric built into the SoC. The power of the FPGA is made available on five dedicated connectors. Electrically, the connections available on these connectors have the ability to accept and generate a variety of signals; all specified in the Snickerdoodle configuration. The configuration of the FPGA pins can (and will often) include routing/multiplexing logic, logical processing and hardware acceleration to be done using the FPGA. Each connector has 25 reconfigurable I/O as well as 1 ADC input and one I^2C

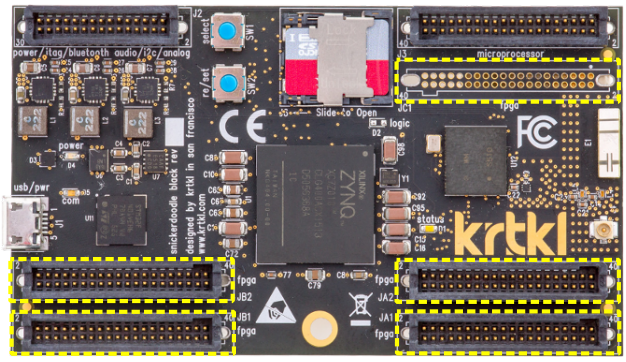
config port.

As described in Chapter 1 , the connectors can be configured as top-facing male headers (TFM) or bottom-facing female headers (SFM). Table Table 2.2 shows some important details of the connectors.

Table 2.2: FGPA Connector Specifications

	Top-Facing	Bottom-Facing
Part Number	TFM-120-01-F-D-A	SFM-120-L1-F-D-A
Contact System	Tiger Eye™	
Description	Terminal Strip	Socket Strip
Pitch	0.050in	
Pins	40	
Rows	2	
Current Rating	2.9A	
Mating Connectors	SFM (board-to-board) SFSD (discrete wire)	TFM (board-to-board) TFSD (discrete wire)

Figure 2.5: FPGA Connectors (JC1 shown unloaded)



3

Power Supplies

WARNING Plugging Snickerdoodle into a power source that does not conform to the electrical requirements can damage the Snickerdoodle.

Snickerdoodle can be powered directly through the USB micro-B connector (J1) or through power input pins located on J2. When plugging Snickerdoodle into USB power, the power source should provide adequate electrical current for the application. When connected through the power input pins on J2 (pins 29 and 30), the input source should supply between +3.7V and +17V.

Each FPGA connector has a +3.3V reference tied to pin 1 and the PS connector has a +1.8V signal reference tied to pin 1. Snickerdoodle can be configured to drive a variety of electronic circuits. Driving signals originating from Snickerdoodle should be connected to high impedance inputs and not used as a power source. The signals coming from Snickerdoodle can be used to control a variety of electronic circuits including motor controllers and charge circuits.

High performance applications should be connected through J2 with a maximum of 3.2A per power supply pin. A +12V supply is recommended for very high performance applications. The +1.8V source on J2 will supply a total of 0.5A to connected devices/accessories. The +3.3V outputs on the FGPA connectors (JA1, JA2, JB1, JB1 and JC1) are able to supply a total of 1A to connected devices/accessories.

Programmable Logic Power (`VCCO_xx`)

The programmable logic power should be provided on pin 3 on the FPGA connectors. The power supply is shared between connectors on the same bank (*i.e.*, JA1 and JA2, JB1 and JB2). For +1.8V interfaces, the on-board +1.8V source can be used as a supply to VCCO_xx. Similarly, the on-board +3.3V source can be used for +3.3V interfaces.

WARNING Do not connect different power supplies to the logic power pins on connectors on the same bank. For example, do not attempt to connect a +1.8V supply to JA1 and a +3.3V supply to JA2.

4

Zynq AP SoC Architecture

The Zynq AP SoC is divided into two distinct subsystems: The Processing System (PS), and the Programmable Logic (PL). Figure 3 shows an overview of the Zynq AP SoC architecture, with the PS colored light green and the PL in yellow. Note that the PCIe Gen2 controller and Multigigabit transceivers are not available on the Zynq7010 device.

Common specifications between the Z-7010 and the Z-7020:

Table 4.1: Zynq Z-7010 and Z-7020 Common Features

Processor Core	Dual ARM Cortex-A9 MPCore with CoreSight
L1 Cache	32 KB Instruction, 32 KB Data per processor
L2 Cache	512 KB
On-Chip Memory	256 KB
External Memory Support	2x Quad-SPI, NAND, NOR
DMA Channels	8 (4 dedicated to Programmable logic)
Peripherals	2x UART, 2x CAN 2.0B, 2x I^2C, 2x SPI, 4x 32b GPIO
Security	RSA Authentication, AES and SHA 256-bit Decryption and Authentication for Secure Boot

Table 4.2: Snickerdoodle Zynq SoC Features

	Z-7010-1	**Z-7020-3**
Maximum Frequency	667 MHz	866 MHz
Programmable Logic Cells	28K Logic Cells	85K Logic Cells

5

Zynq Configuration

The hardware architecture of the Zynq allows full control of the FPGA fabric through then processing subsystem. The processor acts as a master to the programmable logic so that the state of the SoC and the boot process is not dependent on the FPGA configuration. In fact, the FPGA bitstream does not need to be defined prior to booting the device. This makes the boot process more similar to a microcontroller than an FPGA. A boot image is loaded and executed upon startup which loads a First Stage Bootloader (FSBL), an optional bitstream for configuring the programmable logic and, finally, a user-defined processing subsystem application or operating system.

Boot Process

The boot process is typically defined by a primary boot loader such as U-Boot which is responsible for initiating each stage of the boot process. The stages of the boot process are as follows:

Step 1 Upon startup or reset, one of the processing cores executes a set of code from read-only memory (ROM) which initiates the boot process. The ROM is responsible for initiating processing of the first stage boot loader (FSBL) from a designated set of non-volatile memory (*e.g.,* microSD card, flash). The FSBL should be included in a Zynq Boot Image which contains the data for the remaining boot process stages.

Step 2 Once the boot process is handed off to the FSBL, it configures the processing system and loads a bitstream to configure the programmable logic (if a bitstream exists within the boot image). The FSBL then loads any user-defined application/system into memory and prepares to finish the boot process.

Step 3 The user application/system is loaded during the final stage of the boot process. During this stage a secondary boot loader can be used to load a operating system (most commonly Linux) or a user defined application can be directly loaded by the FSBL. For a more thorough explanation of the boot process, refer to Chapter 6 of the Zynq Technical

Reference Manual[1].

[1] Xilinx, Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*, 1.10 edition, February 2015

Table 5.1: Boot Mode Configuration Pins

MIO Bank 500						
2	3	4	5	6	7	8
BOOT_MODE						
Device				pll	V	

Snickerdoodle can be booted using three difference sources. Table 5.1 shows the boot mode configuration pins that can be used to specify the boot source. The specific configuration pins for determining the three available boot sources are shown in Table 5.2 (MIO [4:5]).

Table 5.2: MIO Boot Mode Configuration Pins

	MIO 4	MIO 5
JTAG	0	0
QSPI	0	1
SD Card	1	1

The boot mode can be configured wirelessly, through the krtkl app.

microSD Boot Mode

The most common way to boot Snickerdoodle is from a microSD card installed in J6 (described in more detail in Chapter 7). The microSD card slot provides a removable, upgradable, high-volume source of non-volatile storage from which a range of projects can be loaded. A microSD card can easily be loaded with a boot image that provides a small user application or a general purpose operating system such as Linux. A microSD card can be created and loaded with a bootable image by following a process described in Chapter 22 .

QSPI Boot Mode

Snickerdoodle has a 16 Mbyte quad-SPI NOR flash (Micron N25Q128A11ESE40F). The flash memory can be used to provide non-volatile storage of data and program code. The flash can be used as general storage for initialization of the processing subsystem as well as configuration of the FPGA by storing bitstream data that has been wrapped in a bootable image as described in Chapter 5 .

JTAG Boot Mode

A JTAG connection can be made through J2 where the signals have been made available on pins 17-26, as shown in Table B.1 . These connections can be accessed directly from J2 or broken out when Snickerdoodle is mounted on a baseboard. The JTAG connection can be used to boot Snickerdoodle

when connected to a host computer.

Because of the common JTAG interface the PL system can be configured using a JTAG connection without interfering with the processor.

6

DDR Memory

DDR Configuration

Micron® 8Gb mobile low-power DDR2 SDRAM (LPDDR2).

The following tables record the parameters specified in the DDR Configuration of the ZYNQ7 Processing System within Vivado.

Table 6.1: DDR Controller Configuration

Memory Type	LPDDR2
Memory Part	Custom
Effective DRAM Bus Width	32 bit
ECC	Disabled
Burst Length	8
DDR	400
Internal Vref	Disabled
Operating Temperature (C)	Normal (0-85)

Table 6.2: Memory Part Configuration

DRAM IC Bus Width	32 bits
DRAM Device Capacity	8192 MBits
Speed Bin	LPDDR2 1066
Bank Address Count (Bits)	3
Row Address Count (Bits)	14
Col Address Count (Bits)	11
CAS Latency (cycles)	6
CAS Write Latency (cycles)	NA
RAS to CAS Delay (cycles)	8
Precharge Time	9
tRC (ns)	63.0
tRASmin (ns)	42.0
tFAW (ns)	50.0

Table 6.3: Training/Board Details

DRAM Training	
Write leveling	0
Read gate	Enabled
Read data eye	Enabled
DQS to Clock Delay (ns)	
DQS0	0.004
DQS1	0.004
DQS2	0.004
DQS3	0.004
Board Delay (ns)	
DQ[7:0]	0.436
DQ[18:8]	0.436
DQ[23:16]	0.436
DQ[31:24]	0.436

7

microSD Slot

CAUTION Make sure the microSD card cage is securely latched before powering your Snickerdoodle

The microSD card cage (J6) on Snickerdoodle provides a compact, locking connector for non-volatile storage of up to 200 GB. The microSD interface uses an SDIO interface that is connected to MIO Bank 501 pins 40-45 (SDIO 0). The microSD slot provides a source for non-volatile storage. The Zynq processing subsystem can be booted from the microSD connection and, optionally, a logic bitstream can be defined. For information on creating a bootable microSD card, refer to Chapter 22 .

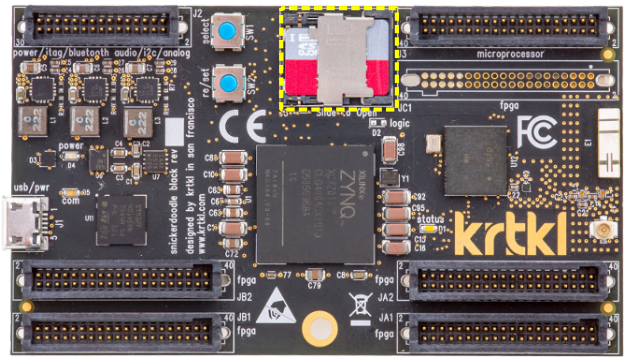


Figure 7.1: Loaded microSD Card Cage

The microSD card interface supports microSD cards of various speeds with a maximum clock frequency of 50 MHz. High speed cards (Class 10) are recommended when using the microSD card as the boot device.

8

Wireless

Snickerdoodle provides Wi-Fi and Bluetooth/BLE connectivity using a Texas Instruments WiLink™8 certified RF transceiver module.

	WL1831	**WL1837**
Radio	Single-Band (2.4 GHz)	Dual-Band (2.4 GHz & 5 GHz)
Technology	SISO	MIMO

Table 8.1: Snickerdoodle Wireless Module Features

Both modules support IEEE 802.11b/g/n standards, Bluetooth 4.0 and Bluetooth low energy; providing Snickerdoodle with a variety of connectivity options. Figure 8.1 shows the location of the UFL jack that can be used to connect an antenna to Snickerdoodle.

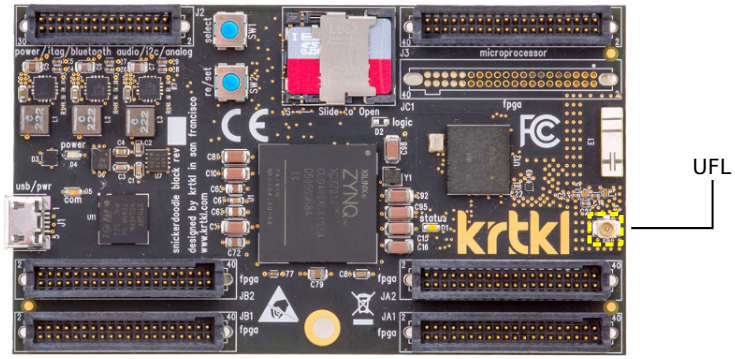


Figure 8.1: Wireless Radio Antenna UFL Connector

9

Platform Controller (STM32)

Snickerdoodle has an STM32F0 series 32-bit ARM®Cortex-A0®microcontroller connected to and in control of various signals. The platform controller connects to the

Zynq Interfaces

The STM32 is connected to the Zynq through multiple communication buses. Each communications bus provides a message channel to/from the platform controller and can be used to provide access (through firmware) to the various platform controller peripherals.

USB/UART Bridge

J1 carries USB signals to/from the platform controller. The platform controller firmware allows the UART interface between the Zynq and STM32 to be bridged with the USB interface to a host computer. This is most commonly used as an interface to the Linux console. The serial console can be set up to interact with any user-defined application/system; most commonly to display Linux console data. The firmware required to create the bridged connection is described in greater detail in Chapter 16 .

Table 9.1: SPI Connection

STM32	SPI2	PB[12:15]
Zynq	SPI1	MIO[46:49]

Table 9.2: UART Connection

STM32	UART1	PB[6:7]
Zynq	UART0	MIO[50:51]

SPI

The SPI interface between provides a high-speed interface between the Zynq and platform controller. The interface can be used to provide the Zynq with access to control over the various peripherals and connections on the platform controller (*e.g.*, LEDs, DACs, nINTs).

I²C Interfaces

Two I²C interfaces on the platform controller are configured. The first I²C interface (designated I²C1) is connected to a switch which has the ability to independently communicate over the I²C pins on each of the 7 in-board connectors. The second interface (designated I²C2) is connected to an on-board crypto-authentication device. The crypto-authentication device can be used for several purposes including wireless network security, IoT network authentication and IP/software security.

External I²C Switch Interface (I²C1)

The STM32 is connected to an I²C switch that provides distribution of I²C connectivity to the connectors.

Crypto-Authenticator Interface

The STM32 is connected to the crypto-authentication device through an I²C interface. The crypto-authentication device is described in more detail in Chapter 10 .

Bluetooth HCI

The STM32 is connected to the Bluetooth UART interface on the Wilink 8 module.

SPI Flash

The platform controller is connected to the SPI flash through an SPI interface. The SPI flash is normally used by the Zynq to boot Linux or store application data. Access to the SPI flash should be restricted to a single controller at a time. For example, the platform controller can be used to write a system to the SPI flash and then enable the Zynq to boot from the flash, after which the platform controller should not attempt to access the flash.

External Interrupt

Each FPGA connector has an external interrupt pin connected to the platform controller. These interrupt pins can be used to drive interrupt routines to monitor and handle device connections.

LEDs

Two timer peripherals on the platform controller are connected to the five on-board LEDs. The timers are configured as PWM outputs to give brightness control over the LEDs and allow a variety of LED patterns.

Power/USB Indicates input power and USB connectivity

Fault Indicates a board hardware fault

Application Can be used to indicate various software states

Wireless/Link Used to indicate wireless connection state

Bluetooth Indicates Bluetooth messages and states

User Buttons

The STM32 is connected to two user buttons that can be used to implement various functionality including boot configuration and interrupt based execution.

10

Crypto-Authentication Device

The crypto-authentication device provides mechanisms for securing the identity of a Snickerdoodle. The crypto-authenticator can be used for a number of applications including:

Network Protection Can create and support key agreements for message encryption between network nodes.

Checking User Password Can be used to securely validate user passwords and facilitate exchange of passwords between remote systems.

Media/IP Protection Can be used to validate the identity of a Snickerdoodle for licensing of media or IP with anti-cloning protection.

Hardware Device

The crypto-authentication device is an Atmel ATECC508A with the following features:

Key Length	256-bit
Storage Size	Up to 16 keys
Serial Number	Guaranteed unique 72-bit
EEPROM	10Kb memory
Hash Algoritm	SHA-256

Table 10.1: Crypto-Authentication Device Features

Development Environment

11

Application Development (SDK)

Xilinx provides a fully integrated Eclipse-based development environment for software development and deployment. The Xilinx SDK can be freely downloaded from Xilinx. The SDK uses Eclipse as it's backbone and adds some convenient utilities for building applications and systems for Snickerdoodle's application processor. Many aspects of the development cycle have been automated and integrated into the SDK. The SDK is hardware aware and is capable of automating significant portions of project creation, build and booting the application processor.

Figure 11.1: SDK Splash Screen

Project Workspace

To get started with application development, whether bare-metal or operating system based, a hardware aware workspace should be used. From within a hardware project directory, a `.sdk` directory is produced during the hardware export process. This directory contains the hardware specification to be used by the SDK for producing the required application board support packages. For more on developing producing custom hardware configurations, refer to Chapter **??** . For many applications, pre-defined hardware configurations can be used to get started with software layer development. Visit `https://github.com/krtkl` for example projects and

hardware configurations.

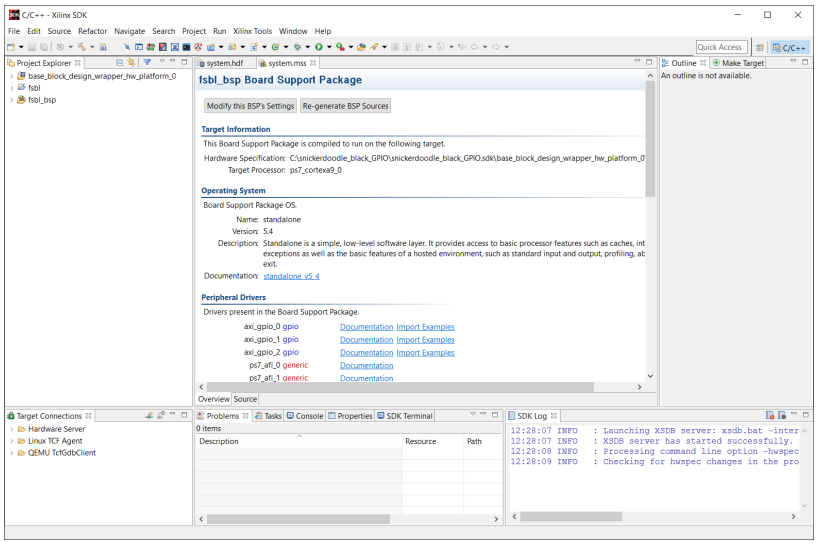


Figure 11.2: Exported Hardware SDK Workspace

Figure 11.2 shows an example SDK workspace for a particular hardware configuration with a FSBL project and BSP. The FSBL can be automatically generated by the SDK. To create a new project, select ***File*** » ***New*** » ***Application Project*** as shown in Figure 11.3 .

Create New Project

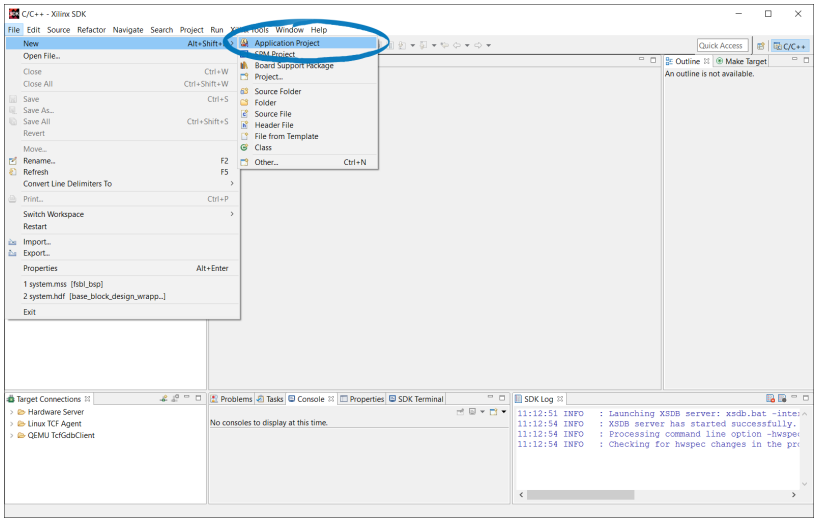


Figure 11.3: Create New Application Project Project From SDK Menu

12

Hardware Development (Vivado)

The versatility of Snickerdoodle is due to it's software definable and reconfigurable hardware interface. The on-board Zynq-7000's programmable logic allows for the hardware interface to be defined by a bitstream that can be generated in a way that is exceedingly similar to software and application development. The development environment, Vivado®, has been architected in such a way that the transition between developing for the programmable logic (PL) and processing subsystems (PS) is seamless.

This guide will get you started developing hardware definitions for Snickerdoodle. In this guide, it is assumed that you have an installation of Vivado (this guide uses the 2015.4 release) on a host computer. The process outlined in this guide was created using Windows 10 as the host computer operating system. While the process is similar for Windows users, those using Windows operating systems should consult the Windows version of this guide.

Portions of this guide have been excerpted and adapted from the Xilinx® Wiki found at `http://www.wiki.xilinx.com/`

Install Snickerdoodle Board Files

To begin working with programmable logic on Snickerdoodle, a set of board files need to be installed onto the host computer for access by Vivado. The board files are a set of text files that are read by Vivado on startup and provide a set of constraints that define the board hardware.

Download Board Files

The board files can be downloaded directly from GitHub. The files can be downloaded as a `.zip` file from a web browser, downloaded directly using `wget` or cloned using `git clone`.

The archive (`.zip`) can be downloaded directly from a Linux terminal by invoking `wget`. The following command will download the board files repository archive:

```
$ wget https://github.com/krtkl/snickerdoodle-board-files/archive/master.zip
```

[1] https://github.com/krtkl/snickerdoodle-board-files

On a Windows host, the board files can be downloaded from navigate to GitHub[1] by selecting ***Clone or download*** » ***download ZIP*** from the repository page.

Installing Board Files

CAUTION Moving or copying files into the Vivado install directory will require root or administrator permissions just as installing any program would.

After downloading the board files archive, the files will need to be installed in the proper location so that they may be read by Vivado. The directory that Vivado uses to load board files when starting is relative to it's install location. On a Windows host, typically Vivado is installed at `C:/Xilinx/Vivado/<release>`. On a Linux system, Vivado is usually installed into `/opt/Xilinx/Vivado/<release>`. The board files are located at `<vivado_install_dir>/data/boards/board_files`.

On a Linux system, the board files can be downloaded from the terminal using `wget` and extracting the archive to the board files directory.

The only files that should be installed are the `Snickerdoodle` and `Snickerdoodle_black` directories which contain the board files for the corresponding Snickerdoodle version. An exclusion file (`EXCLUDE`) is included in the archive to help with the copying process and prevent extraneous files (documentation, README, etc.) from being copied into the board files directory.

The following series of commands will extract the `.zip` archive and move the resulting board files into the `board_files` directory:

```
$ wget https://github.com/krtkl/snickerdoodle-board-files/archive/master.zip
$ cd snickerdoodle-board-files-master
$ rsync -av --exclude-from=EXCLUDE ./* /opt/Xilinx/Vivado/2015.4/data/boards/board_files/
```

Once the board files have been installed, Vivado will need to be restarted to load the files from the data directory.

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 12.1 .

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 12.2 shows the input of the project name and parent directory when creating a new project.

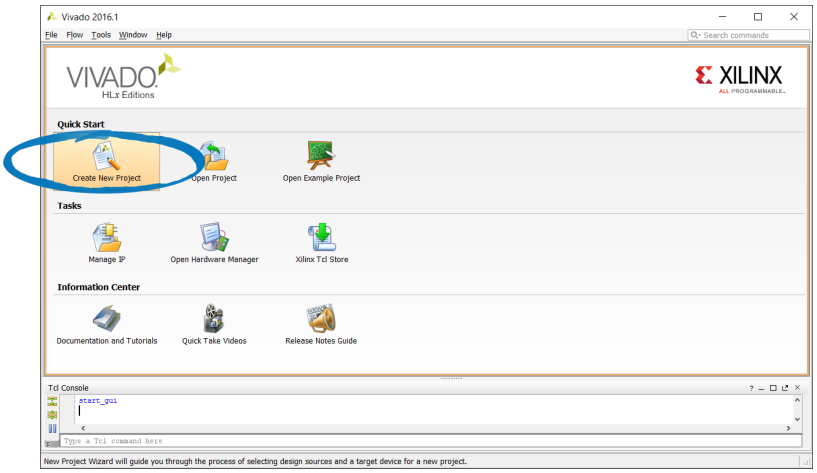


Figure 12.1: Vivado Startup Create New Project

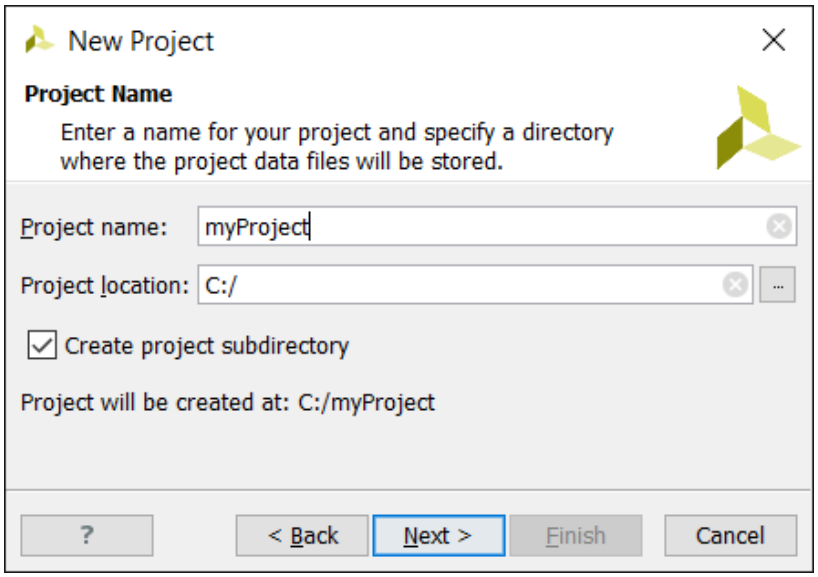


Figure 12.2: Selecting Project Name and Parent Directory

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 12.3 , can be selected to skip this step. Sources can be added to the project at any time after creation as outlined below.

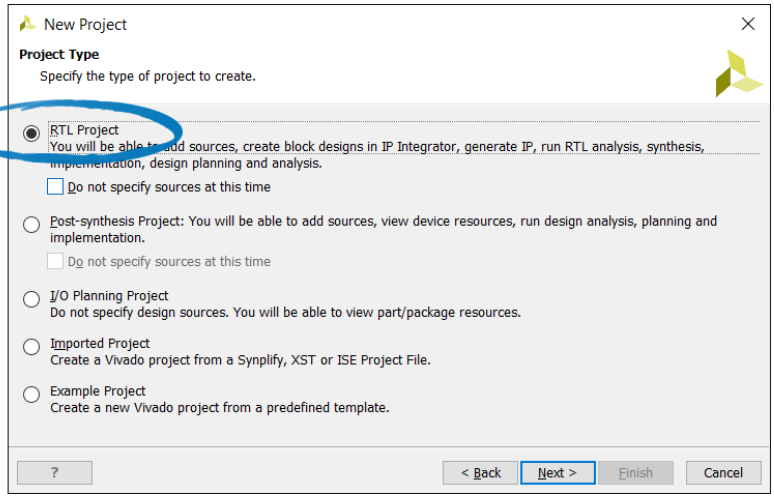


Figure 12.3: Project Type Selection Dialog

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 12.4 .

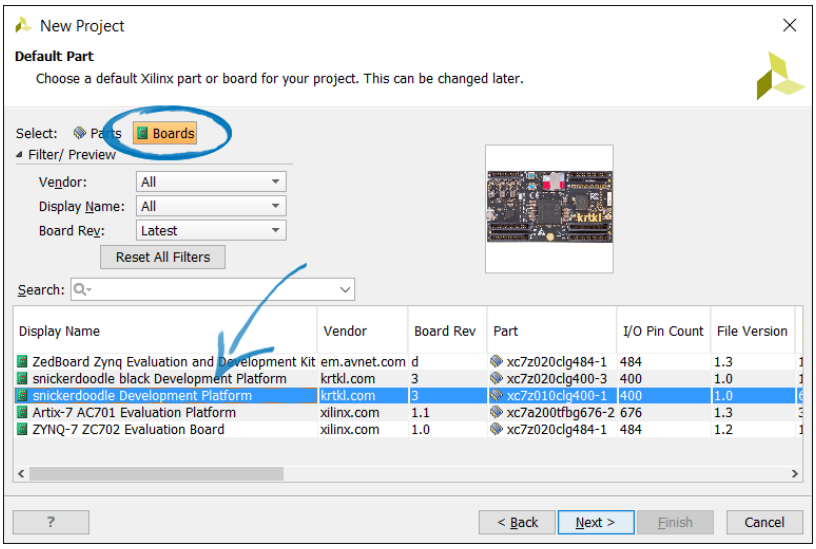


Figure 12.4: Selecting a Project Board

After selecting a board, the details of the project will be listed in the *New Project Summary* shown in Figure 12.5 . By clicking the "Finish" button, the project will be created and opened in the Vivado graphical user interface (GUI).

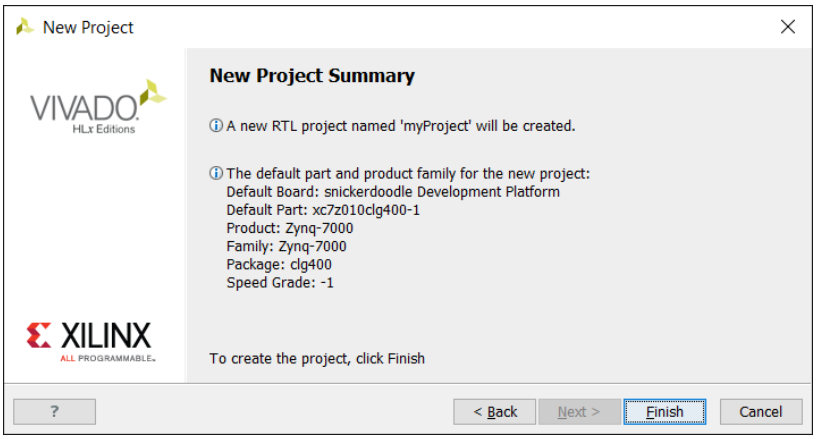
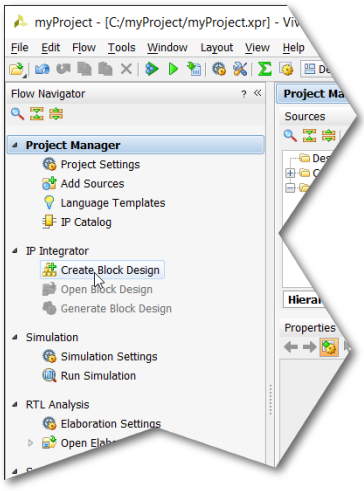


Figure 12.5: Review and Finish New Project Creation

Working with Vivado

Now that Vivado has been set up with the Snickerdoodle board files and a new project has been created, development can begin. The Vivado GUI is similar to many software development environments. Many project elements are contained within panes and editing of those elements can be

Additional information and resources on using Vivado to create and edit hardware designs can be found in the Vivado Design Suite User Guide: Getting Started or for a more in-depth guide at Vivado Design Suite User Guide: Using the Vivado IDE



done through the menubar or from the panes themselves. 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.

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 12.7 shows an example block design with hardware definitions for fixed peripherals such as memory interfaces and a some GPIO ports defined in programmable logic.

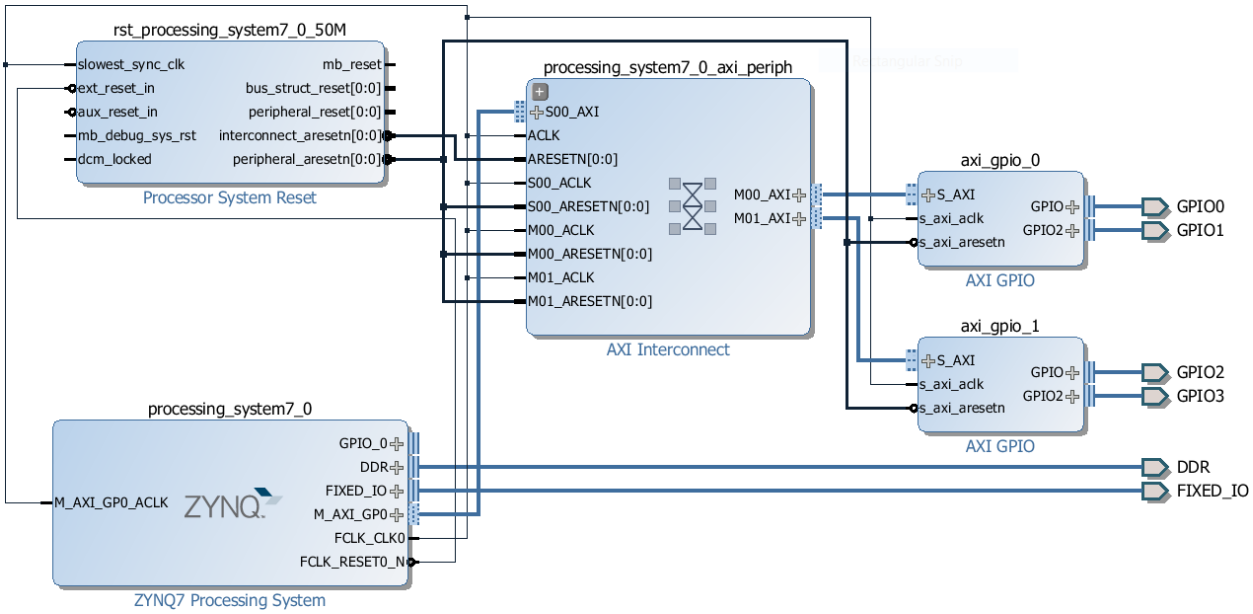


Figure 12.7: Example Block Design with Zynq Processing System and Microblaze Soft Core Processor

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 12.6 .

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 .

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 12.2 .

Code Listing 12.2: Error When Getting Ports Before Synthesizing and Implementing Design

```
> get_ports
```

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

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

Code Listing 12.3: Example TCL Console Output of `get_ports` Command

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.

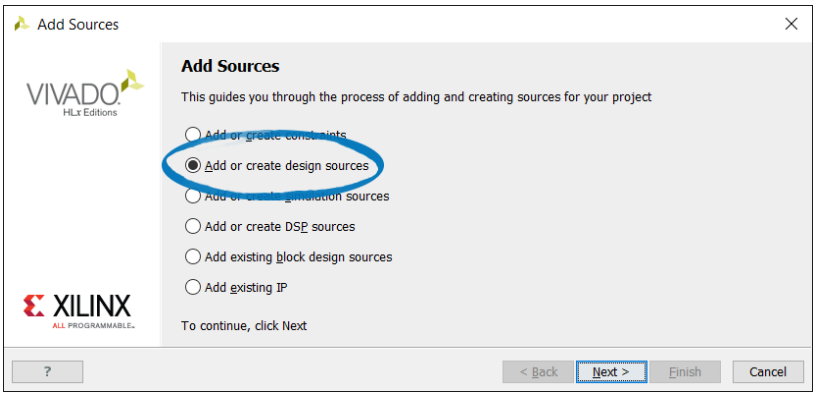


Figure 12.8: Adding New Sources to the Project with the Add Sources Wizard

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...".

Address Editor

The address editor can be used to configure and identify the register address values used to access the IP interfaces that have been used in the block design. These addresses will be used to specify configure the interfaces in the devicetree blob so that they are enumerated by Linux during the boot process and are made accessible from Linux. More information on devicetree usage, refer to Chapter 20 .

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*

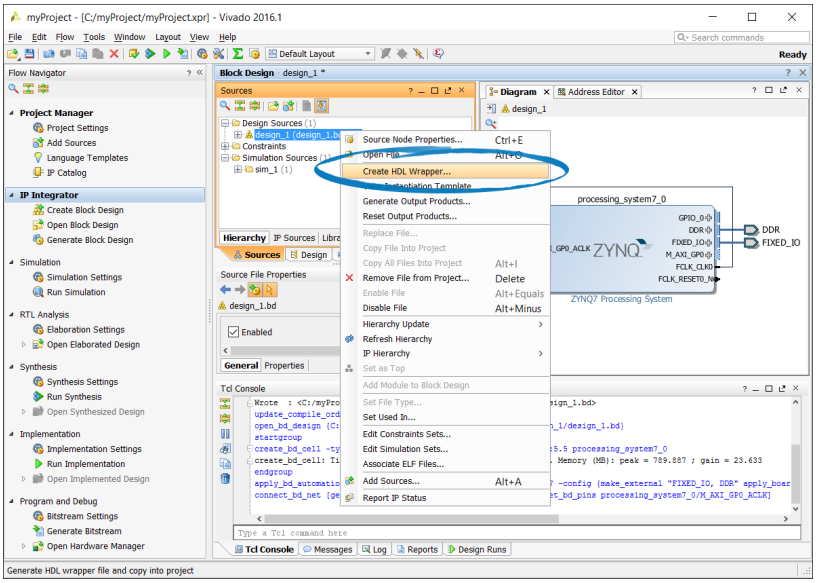


Figure 12.9: Creating an HDL Wrapper for the Design

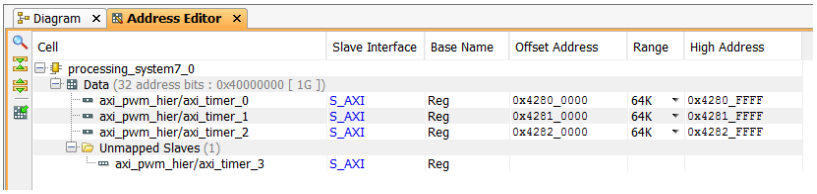


Figure 12.10: Unmapped Peripherals in Address Editor

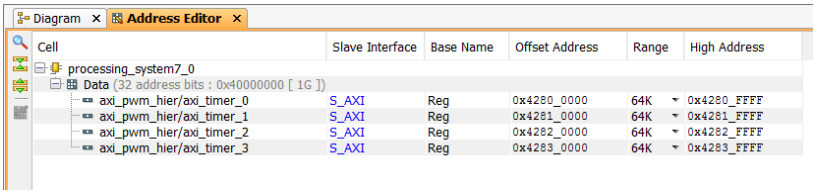


Figure 12.11: Address Editor

Navigator pane as shown in Figure 12.12 . The bitstream contains all the information necessary to define the programmable logic and the associated hardware peripherals/interfaces.

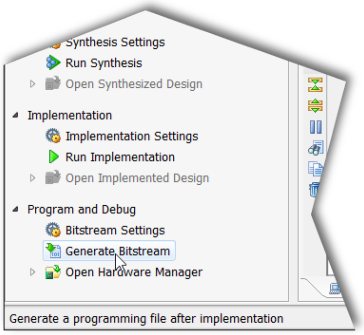
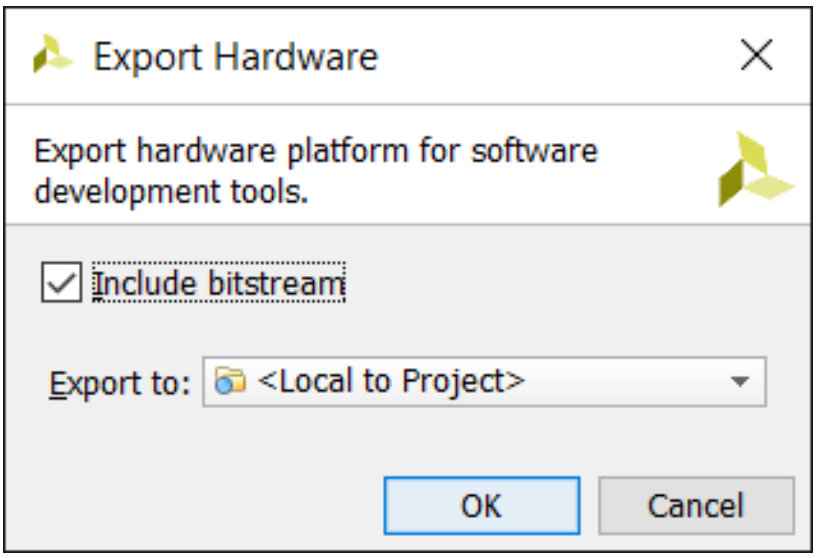


Figure 12.12: Generate Bitstream from Vivado Flow Navigator

Exporting Design with SDK

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 as shown in Figure 12.13 .



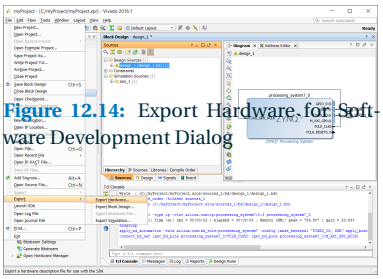


Figure 12.13: Export Hardware Configuration from Vivado

Figure 12.14: Export Hardware for Software Development Dialog

Figure 12.14 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.

> *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*

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, as shown in Figure 12.15 . 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

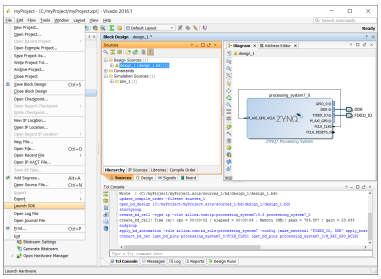


Figure 12.15: Launch SDK from Vivado

was exported to a different directory, you will need to change the workspace directory after the SDK has launched.

13

First Stage Boot Loader (FSBL)

The first stage boot loader 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 bitstream to the programmable logic before loading additional programming to the processing subsystem.

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 generated a Snickerdoodle boot image.

FSBL Application Project Platform and BSP

As shown in Figure 13.1 , the FSBL can be created in the same way as a bare-metal application project by navigating to ***File*** » ***New*** » ***Application Project*** as shown in Figure 11.3 . From the New Project dialog, select a ***standalone*** platform. A project name can be entered from this interface.

FSBL Application Project Template

By clicking ***Next>***, a project template option dialog (Figure 13.2) 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.

Project Outputs (`u-boot.elf`)

By default, the FSBL project will automatically build a `.elf` output. After generating and building the FSBL project, the SDK workspace will look similar to Figure 13.3 . Additional projects can be added to the workspace and worked on from within the same environment to reuse the hardware

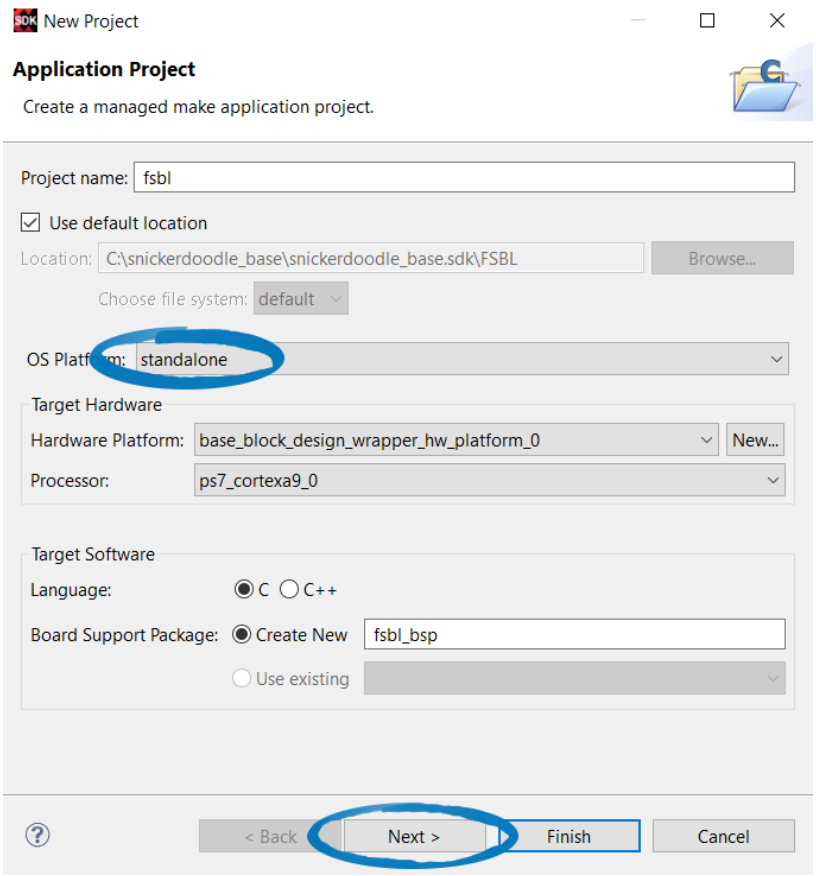


Figure 13.1: Create a New FSBL Project in SDK

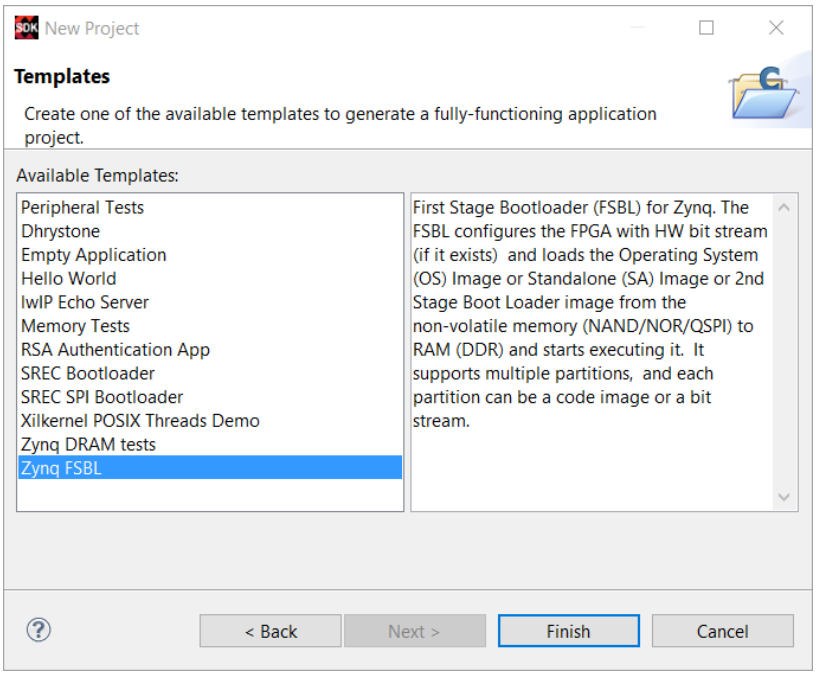


Figure 13.2: Create FSBL Project from Template in SDK

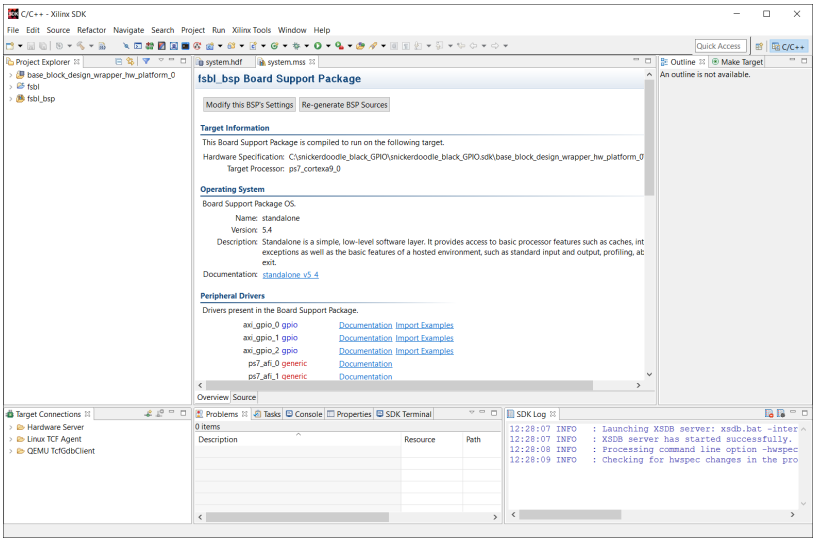


Figure 13.3: Exported Hardware Workspace with FSBL Project and BSP

configuration data.

The FSBL project output can be accessed from within the SDK GUI to build boot images using prebuilt binaries or outputs from other projects within the workspace. The build output will be generated to `<workspace_dir>/fsbl/Debug`, by default and can be changed from within the project preferences. The process for generating these images is described in the upcoming chapter.

14

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 specifics 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.

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:

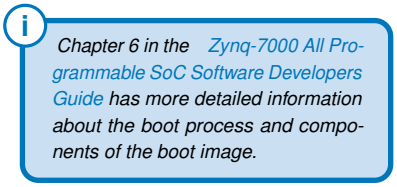
`FSBL` - First stage boot loader (FSBL). Responsible for loading the bitstream (if one exists), loading into memory and handing off the boot process to the second stage bootloader. The FSBL should always be specified with the *bootloader* tag in the `.bif`.

`U-Boot` - Second stage boot loader. Responsible for loading Linux system components (devicetree, uImage, 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.



Chapter 6 in the Zynq-7000 All Programmable SoC Software Developers Guide has more detailed information about the boot process and components of the boot image.

Xilinx, Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*, 1.10 edition, February 2015

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 it's own. An advantage of this approach is modularity and flexibility of the other system components and choice of boot medium. Code Listing 14.1 shows the *.bif* structure for producing a small boot image.

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

Code Listing 14.1: Minimal Boot Image with FSBL and U-Boot

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=0xXXXX` 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 14.2 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.

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

```
image : {
        [bootloader]fsbl.elf
        u-boot.elf
        [load=0x2a00000]devicetree.dtb
        [load=0x2000000]uramdisk.image.gz
        [load=0x3000000]uImage.bin
}
```

Code Listing 14.2: Load Linux Components Using FSBL

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=0xXXXX` 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).

Code Listing 14.3: Load Linux Components from Known Boot Image Offsets

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

Building Bitstream into *BOOT.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 (`BOOT.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).

Building Boot Images from SDK

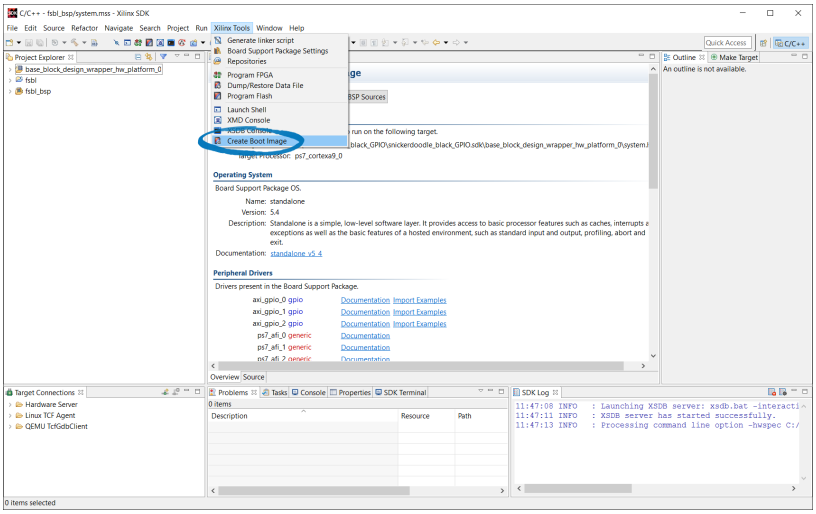


Figure 14.1: Create Boot Image SDK Menu Selection

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 14.2 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 14.1 .

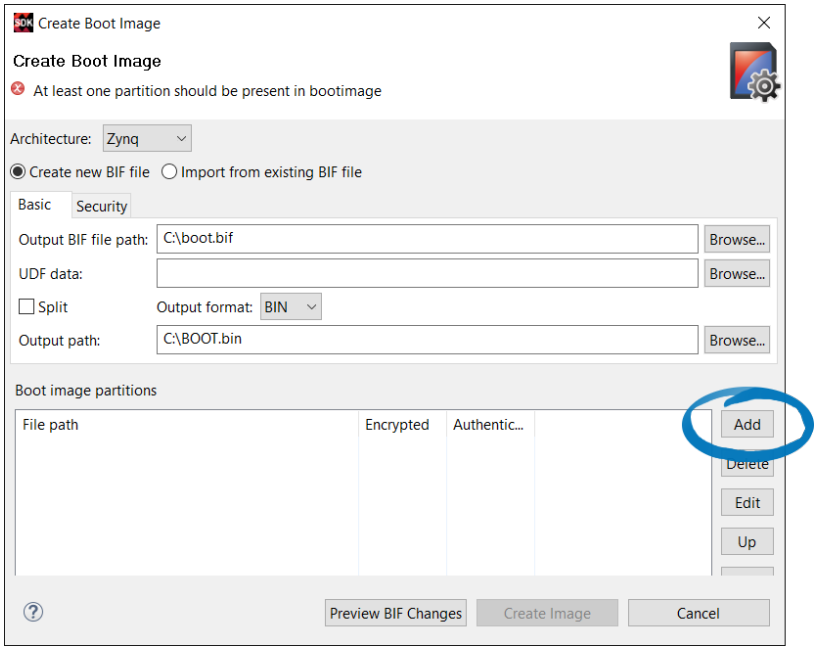


Figure 14.2: Create Boot Image Interface in Xilinx SDK

Prebuilt `.bif` files can be used by the SDK interface by selecting "Import from exisiting BIF file", shown at the top of the window in Figure 14.2 . 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 14.3) which will allow you to select and specify the partition file.

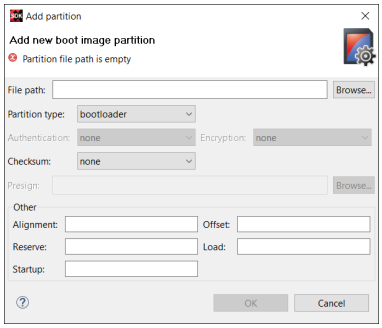


Figure 14.3: Create New Boot Image Partition

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 14.2 . FSBL should be specified as the bootloader by selecting *bootloader* as the parition type as shown in Figure 14.4 .

After adding the FSBL, add the bitstream and U-Boot images from the filesystem. Add the sources as a*datafile* as shown in Figure 14.5 . Prebuilt U-Boot executables can be downloaded from `https://github.com/krtkl` and copied to the workspace 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.

After importing the boot image sources, the boot image interface should look similar to Figure 14.6 .

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.

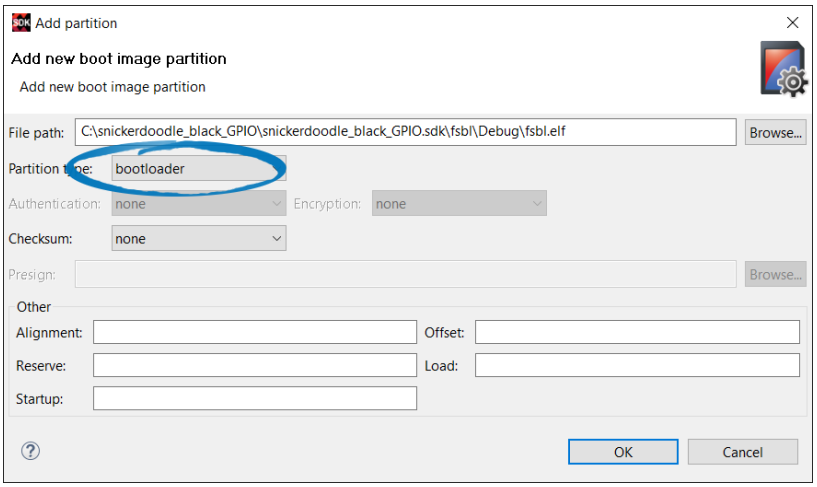


Figure 14.4: FSBL Bootloader Boot Image Partition

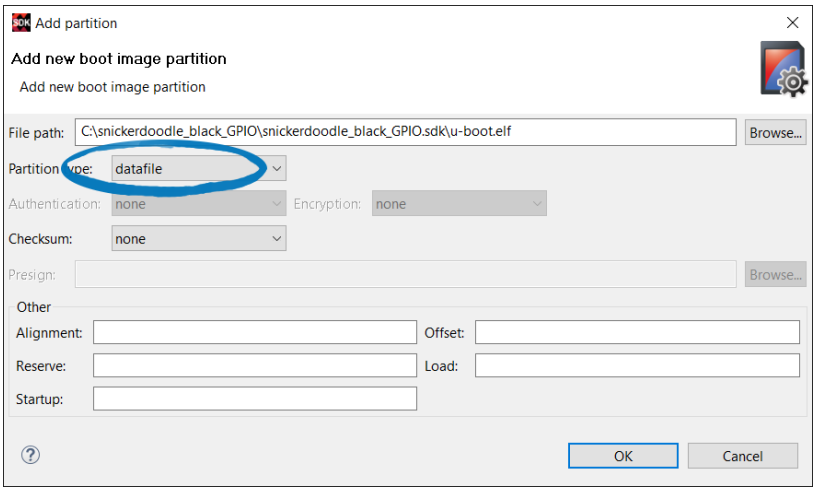


Figure 14.5: U-Boot Boot Image Partition Selection

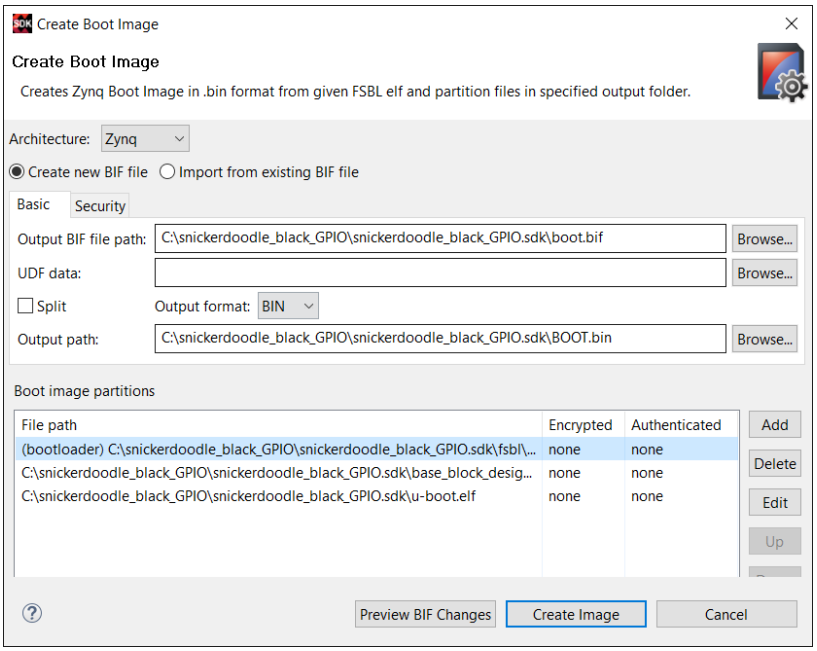


Figure 14.6: Boot Image Partition Table

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

Code Listing 14.4: SDK Create Boot Image Console Output

Building Boot Images with bootgen

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.

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:

Code Listing 14.5: Boot Information 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 named BOOT.bin using the boot image file boot.bif:

Code Listing 14.6: Generate BOOT.bin with bootgen

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

Platform Controller

Introduction

15

Peripherals

ADC

Several 12-bit analog to digital converters (ADC) on the platform controller are used to monitor the on-board systems. The power supply levels are monitored using the ADCs.

ADC_IN0	VIN_SENSE
ADC_IN1	VUSB_SENSE
ADC_IN2	MAIN_3V3_SENSE
ADC_IN3	MAIN_1V8_SENSE
ADC_IN6	VDDIO2_3V3_SENSE
ADC_IN8	ZYNQ_1V0_SENSE
ADC_IN9	ZYNQ_1V8_SENSE
ADC_IN12	ZYNQ_1V2_SENSE
ADC_IN13	ZYNQ_3V3_SENSE

Table 15.1: ADC Port Connection

I^2C

SPI

LEDs

The platform controller is connected to 5 LEDs that are used to indicate various states and events of the board. These LEDs are connected to peripherals that are configured as PWM timers for brightness control on the LEDs through the PWM duty cycle.

PA8	[D11]	TIM1 CH1	Red (Fault)
PC6	[E12]	TIM3 CH1	Green (USB/Power)
PC7	[E11]	TIM3 CH2	Blue (Bluetooth)
PC8	[E10]	TIM3 CH3	White (Application)
PC9	[D12]	TIM3 CH4	Orange (Wireless Link)

Table 15.2: LED PWM Timer Channels

USART

USART1

USART1 is connected to the Zynq processing subsystem on UART0 and is normally used to bridge the Zynq console output to the USB device peripheral. Normally the USB device is configured as a communications device class (CDC) interface.

Table 15.3: Platform Controller USART1 Configuration

Mode	Asynchronous
Hardware Flow Control (RS-232)	Disabled
Baud Rate	115 200
Word Length	8 bits
Parity	None
Stop Bits	1
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable

USART2

USART2 is connected to the Bluetooth HCI interface of the Wilink 8 module.

Table 15.4: Platform Controller USART2 Configuration

Mode	Asynchronous
Hardware Flow Control (RS-232)	CTS/RTS
Baud Rate	115 200
Word Length	8 bits
Parity	None
Stop Bits	1
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable

USB Device

The USB device peripheral is normally configured as a communications device class (CDC) interface to provide access to the Zynq console by bridging to USART1 which is connected to UART0 on the Zynq.

During certain boot modes, the USB device is configured for device firmware upgrade (DFU) and can be used to program the platform controller by writing directly to the embedded flash memory.

Basic Parameters

Table 15.5: USB Parameters

Speed	Full Speed 12MBit/s
Endpoint 0 Max Packet size	8 Bytes
Physical interface	Internal Phy

GPIO

PA9	D10	Output	ANT_SELECT_1
PA10	C12	Output	ANT_SELECT_2
PC1	J2	Output	ZYNQ_JTAG_NRST
PD1	B9	Output	ZYNQ_POWER_EN
PD2	C8	Output	WDI
PD7	A5	Output	WL_32KHZ_CLK_EN
PD8	K9	Output	ZYNQ_CLK_EN
PE2	B2	Output	MIO4_BOOT_SELECT
PE3	A1	Output	MIO5_BOOT_SELECT
PE4	B1	Input	SELECT_BUTTON
PE5	C2	Input	RESET_BUTTON
PE7	M7	Input	ZYNQ_POWER_GOOD
PF10	G2	Output	WL18xx_BT_EN
PD15	H10	Output	SMB_NRESET

Table 15.6: GPIO Pin Configuration

EXTI

PD10	J12	EXTI10	JA1_P2
PD11	J11	EXTI11	JA2_P2
PD12	J10	EXTI12	JB1_P2
PD13	H12	EXTI13	JB2_P2
PD14	H11	EXTI14	JC1_P2

Table 15.7: GPIO External Interrupt Pin Configuration

16

Firmware

The platform controller firmware provides the logic for handling the tasks of the...

Development Environment

The integrated development environment used to target the platform controller is the ARM® Keil microcontroller development kit (MDK). The MDK can be downloaded from http://www2.keil.com/stmicroelectronics-stm32/mdk. After registering and activating the MDK, custom platform controller firmware can be developed and compiled.

17

Device Firmware Upgrade (DFU)

For full customization of the Snickerdoodle system the firmware on the platform controller can be modified and upgraded.

Booting into DFU

The device firmware upgrade interface can be accessed by pressing and holding down both the SELECT and RESET buttons when powering the board. After holding both buttons for 4 seconds, the device will boot into DFU mode and allow access to program the platform controller with upgrade or even custom firmware.

Bootloader Pin Activation Pattern

The device system memory bootloader (which includes USB DFU firmware) can be entered when powering up the board by pulling the `BOOT0` pin high (+1.8V). This pin is normally pulled to ground through a 100kΩ resistor (R56) but can be connected to +1.8V by carefully shorting the pads for the normally unloaded R55 resistor shown in Figure 17.1 .

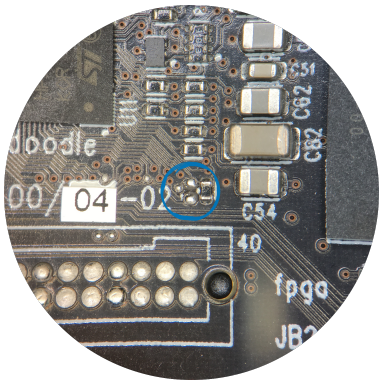


Figure 17.1: Unloaded R55 Pads

WARNING Extreme care should be taken when attempting to connect the pads on R55. Irreparable damage to the board may occur if an error is made while shorting the pads. DO NOT use this method for normal firmware upgrades unless and should only be used if the firmware DFU mechanism has been disabled.

Generate Loadable Images

HEX File Build Output

To produce a loadable DFU image, a `.hex` file is used and must be produced by the firmware build process. To enable `.hex` file production in the Keil μVision development environment, navigate to ***Project*** » ***Options for Target*** under the ***Output*** tab and check the box marked *Create HEX File* as shown in Figure 17.2 .

The generated `.hex` file can be used to create a firmware upgrade image.

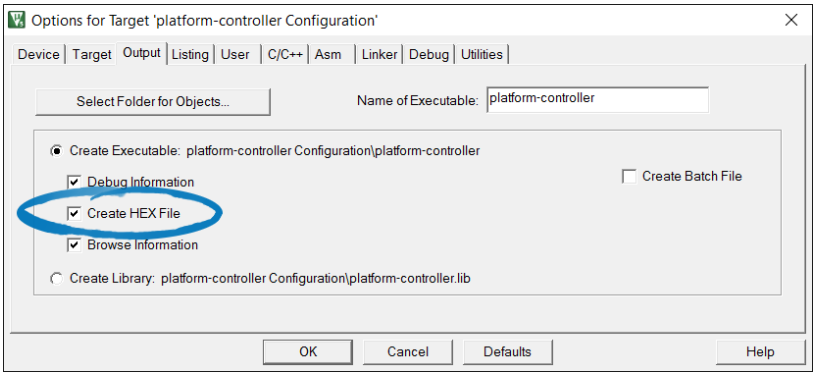


Figure 17.2: Create Firmware HEX File Build Output

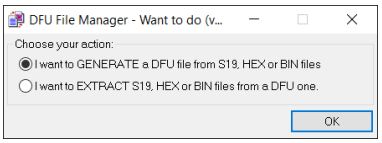


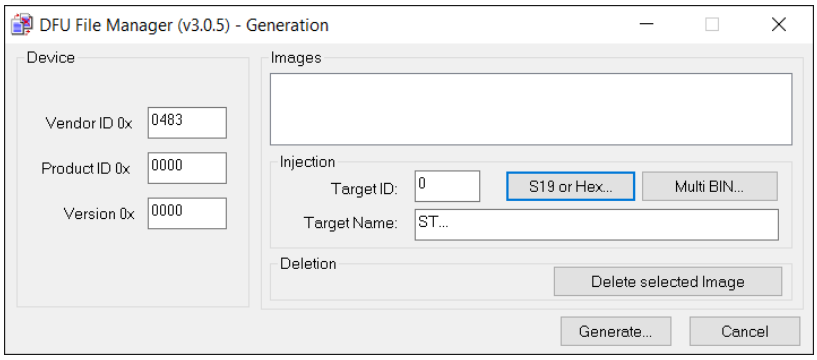
Figure 17.3: DFU File Manager Action Selection

Generate DFU Image

The DFU File Manager (DFUFM) allows users to generate DFU images from firmware build outputs (*i.e.*, S19, HEX or BIN files) and can also be used to extract firmware build outputs from DFU images. For a typical DFU, the DFUFM will be used to produce DFU images by selecting ***I want to GENERATE a DFU file...*** after opening the DFUFM.

With a `.hex` file produced from a firmware build, a usable DFU image can be produced. From the DFU File Manager, select ***S19 or Hex...*** from the *Injection* section of the interface and navigate to the `.hex` file you wish to use for the firmware upgrade and select ***Generate...***

Figure 17.4: Generate DFU Image File



Perform Firmware Upgrade

The DfuSe interface will detect any microcontrollers that are connected to the host USB in DFU mode.

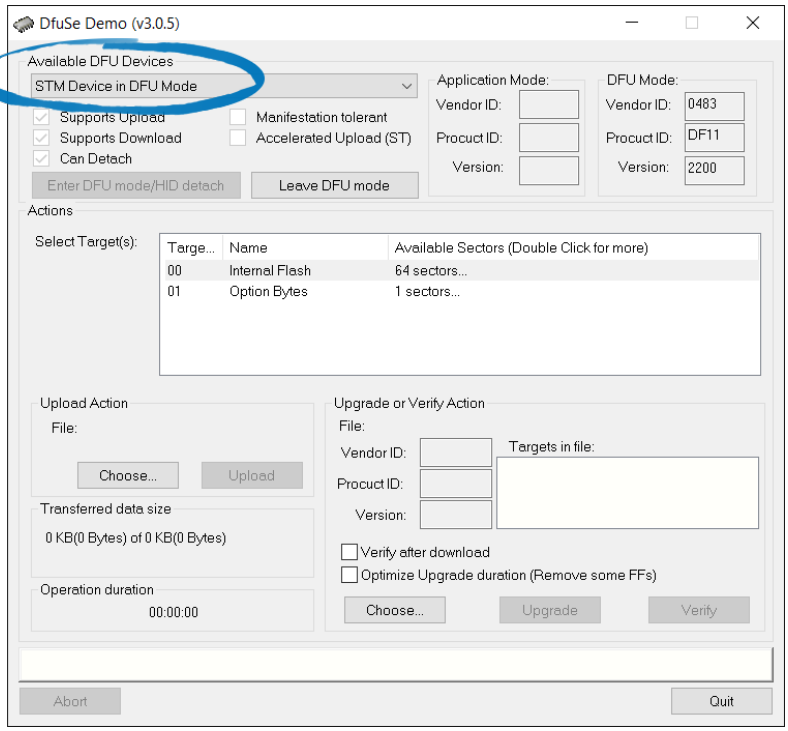


Figure 17.5: DfuSe Demo Interface

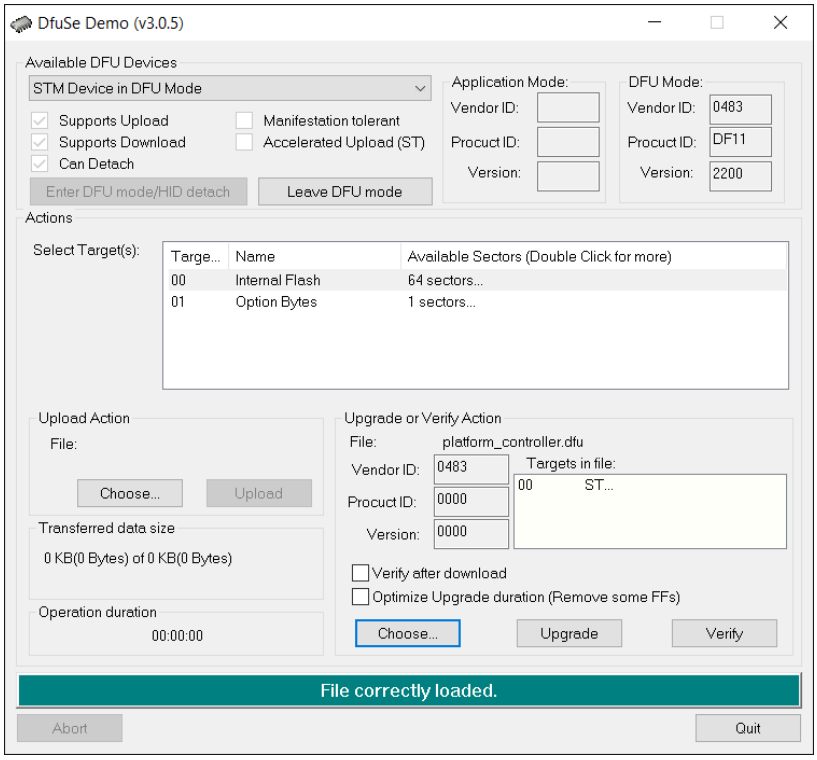


Figure 17.6: DFU Image File Loaded Into DfuSe Software

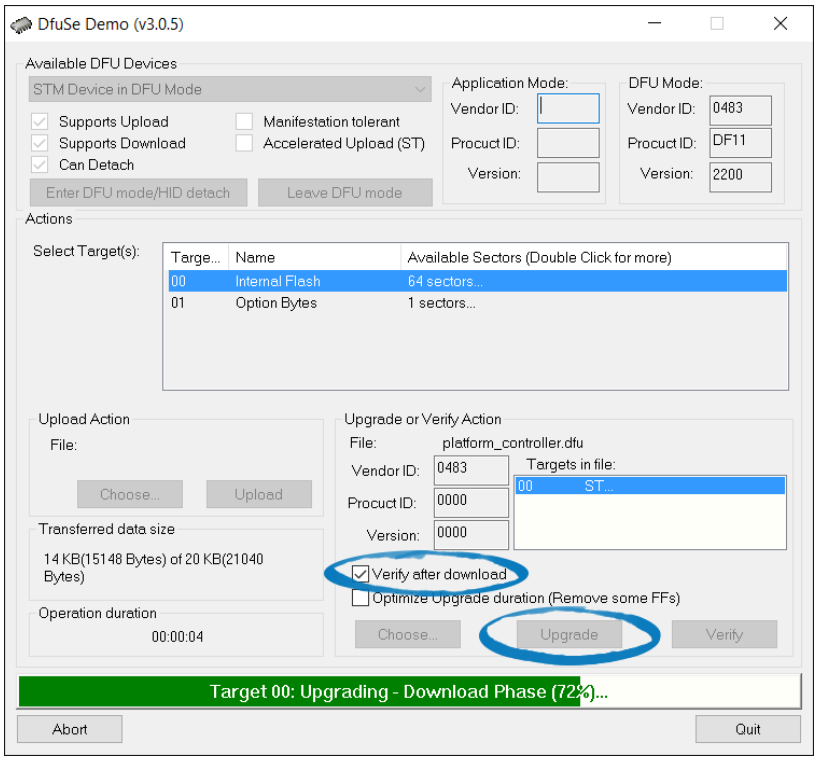


Figure 17.7: DFU Upgrade Download Progress

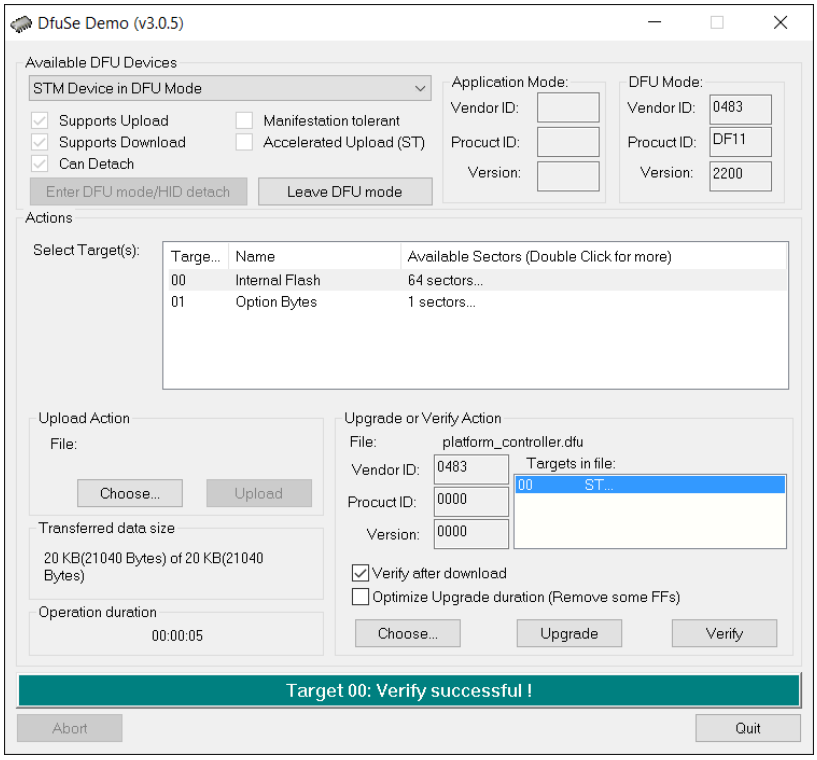


Figure 17.8: DFU Upgrade Success

18

USB Interface (Virtual COM Port)

The USB interface is most commonly specified as a communications class device and used to bridge the UART (RS-232) connection between the platform controller and the application processor.

Windows Console Connection

On Windows host systems a driver is required to use the USB interface to access the console on the application processor. The driver can be downloaded from STMicroelectronics.

Virtual COM Port Driver

To install the driver, first install the driver install package. The installer package will be installed into **`C:\`** » **`Program Files (x86)`** » **`STMicroelectronics`** » **`Software`** » **`Virtual comport driver`** » **`<windows_version>`**. Figure 18.1 shows running the 64-bit Windows 8 driver as administrator.

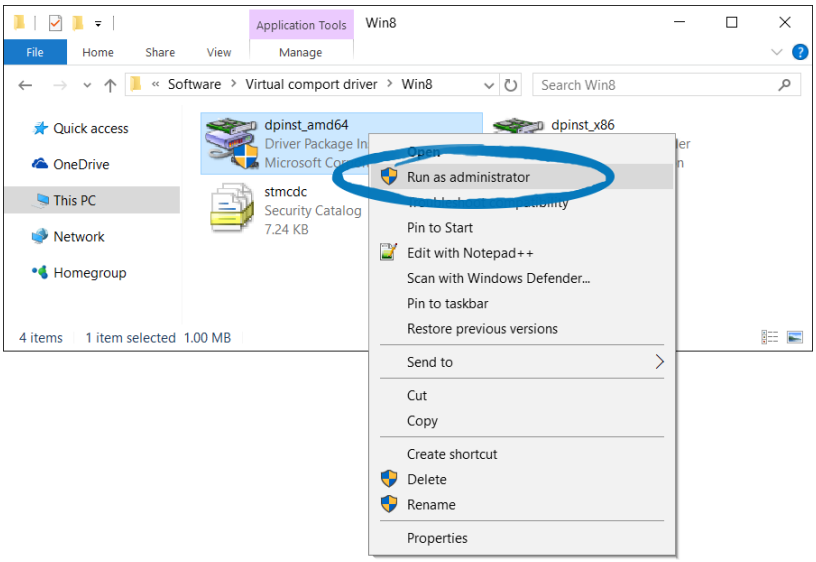


Figure 18.1: Run Virtual COM Port Driver Installer

After the install is complete, the driver should recognize the USB inter-

face and register it as a COM port on windows. Figure 18.2 shows the VCP device recognized and enumerated as 'COM6' which is viewed from the Devices and Printers interface of the Control Panel. The alias (in this case 'COM6') that is provided by Windows should be used to connect to the console using a terminal emulator.

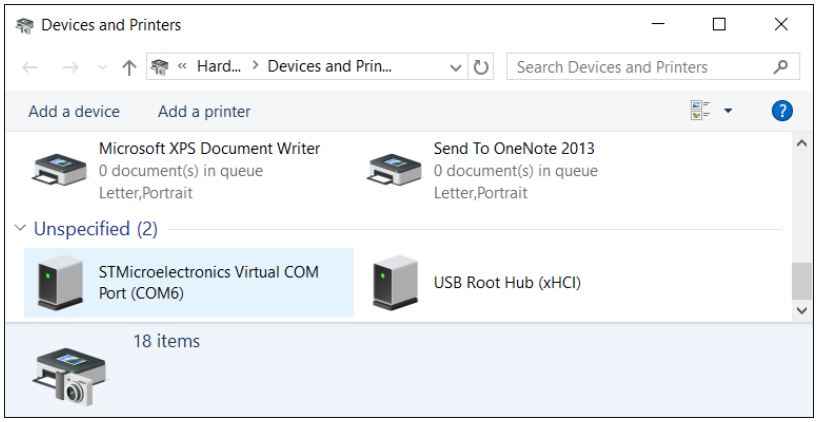


Figure 18.2: Snickerdoodle VCP Device Registered in Devices and Printers

Connecting to the Console

A variety of terminal emulators for Windows exist, such as PuTTY and Tera Term. Figure 18.3 shows the serial communication settings for connecting to the Snickerdoodle VCP interface.

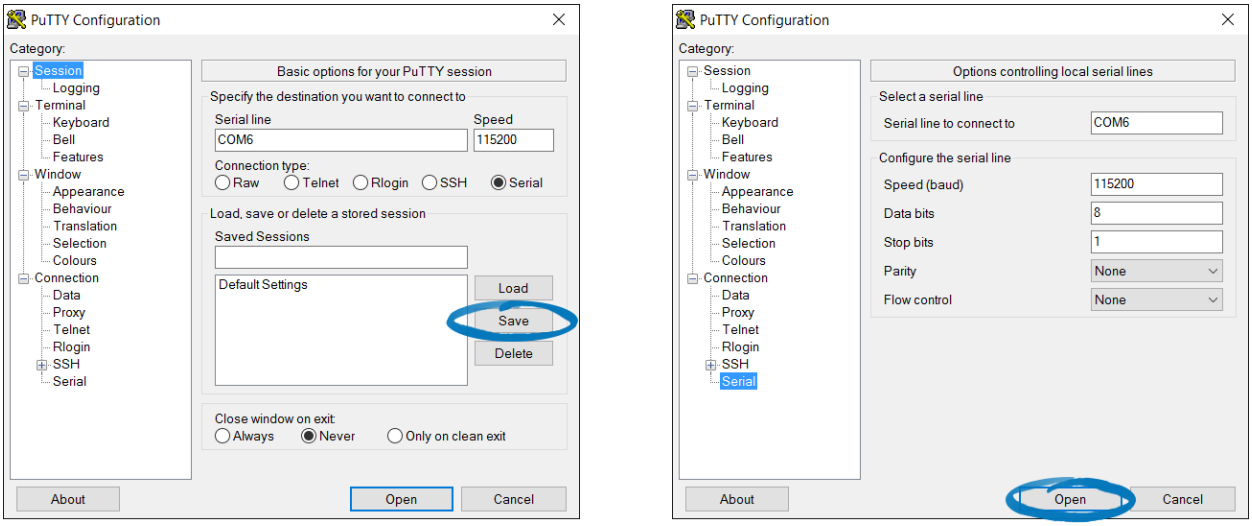


Figure 18.3: PuTTY Serial Connection Type

In PuTTY, the serial settings can be saved from the ***Session*** option. This is useful during development when the disconnecting and re-connecting the Snickerdoodle with the host machine. After selecting ***Open***, a terminal similar to the one shown in Figure 18.4 should appear.

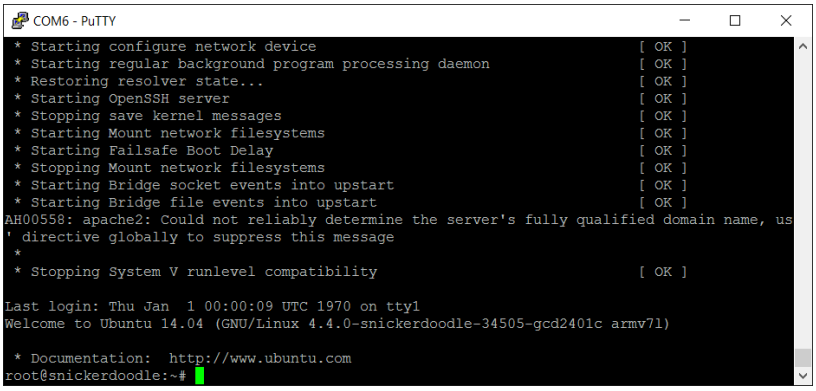


Figure 18.4: Connected PuTTY Serial Terminal

OS X and Linux Console Connection

OS X and Linux do not require any driver install to interact with the Snickerdoodle USB console. The USB connection will be enumerated by those systems as TTYs and are accessible using terminal emulators such as `screen` or `minicom`.

Getting Console Connection Location

The console TTY file location can be determined using `dmesg` in both OS X and Linux systems.

```
$ sudo dmesg | grep STM
025860.670322 STMicroelectronics Virtual COM Port@40133000:
    AppleUSBDevice::ResetDevice: <software attempt to RESET>
```

Code Listing 18.1: OS X Console Virtual COM Port Kernel Message

This output can be confirmed by checking the contents of the `/dev` directory for TTY devices using `ls`. In the example shown in Code Listing 18.2 the TTY device has been attached to `/dev/tty.usbmodem401331`.

```
$ ls /dev/tty.*
/dev/tty.Bluetooth-Incoming-Port        /dev/tty.usbmodem401331
```

Code Listing 18.2: Checking for TTY Devices in `/dev` in OS X

Code Listing 18.3 shows a truncated Kernel message printout with USB connection messages. The USB device is connected as `ttyACM0`, shown on the 8th line of the message printout. The location of the device on the system will be located at `/dev/ttyACM0` for use with a terminal emulator.

```
$ dmesg | tail
[  831.752314] usb 3-3.1: new full-speed USB device number 6 using xhci_hcd
[  831.847610] usb 3-3.1: New USB device found, idVendor=0483, idProduct=5740
[  831.847613] usb 3-3.1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  831.847614] usb 3-3.1: Product: STMicroelectronics Virtual COM Port
[  831.847615] usb 3-3.1: Manufacturer: STMMicroelectronics
[  831.847616] usb 3-3.1: SerialNumber: 00000000001A
[  831.877819] cdc_acm 3-3.1:1.0: This device cannot do calls on its own. It is not a modem.
[  831.877850] cdc_acm 3-3.1:1.0: ttyACM0: USB ACM device
[  831.878757] usbcore: registered new interface driver cdc_acm
[  831.878760] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
```

Connecting to the Console

`screen` is a simple and easy terminal emulator for use in 'nix systems. The simplicity of it's use provides a convenient method for interfacing to the console. In either an OS X or Linux terminal, the `/dev` location on the system is used as the only argument necessary.

Code Listing 18.4: Connecting to Snickerdoodle Console Using `screen`

```
$ screen /dev/ttyACM0
```

LINUX

Components

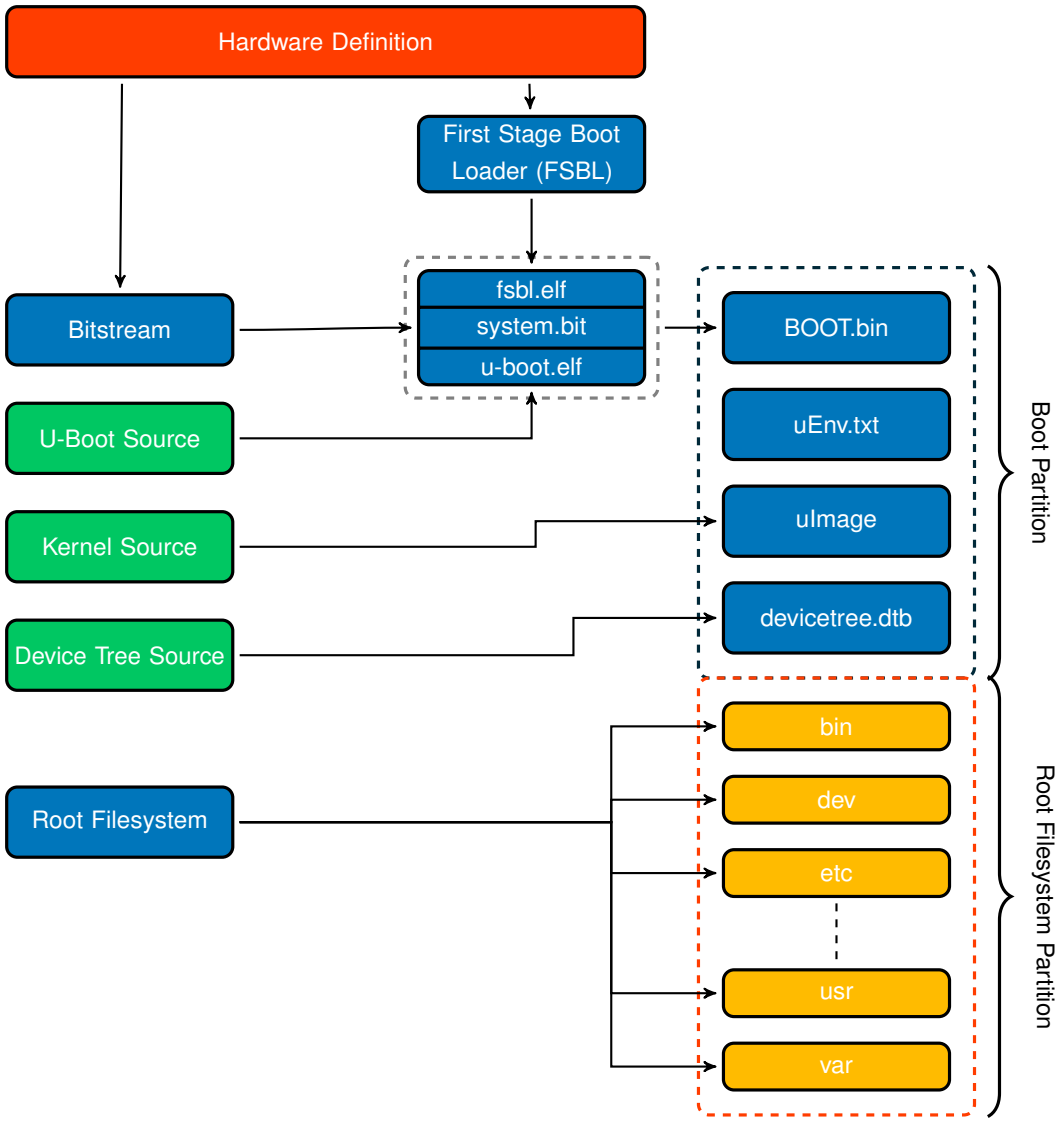


Figure 18.5: System Component Build Process and Resulting Boot Components and Root Filesystem

19

U-Boot

Build from Source

Download Sources

U-Boot is the second stage boot loader used to load the the Linux kernel and root filesystem. For U-Boot to load the kernel, the kernel image needs to be wrapped with a U-Boot header.

U-Boot is built with a `mkimage` utility which is used to wrap the kernel image. `mkimage` is also used to wrap file system images with U-Boot headers for loading. To build Linux images, U-Boot source needs to be downloaded and compiled to provide the `mkimage` utility binary.

To download the U-Boot source needed to compile Snickerdoodle Linux images, the following `git` command can be executed from the command line:

```
$ git clone https://github.com/krtkl/u-boot-xlnx.git
```

Configuration

Build U-Boot

Generate Build Script Utilities

U-Boot requires a utility that is normally built with the kernel. The device tree compiler (`dtc`) is used to compile device tree blobs from source and is called during the build process for U-Boot. Interestingly, building U-Boot requires a utility that is normally built with the kernel while the Linux kernel requires a utility that is normally built with U-Boot (`mkimage`). To avoid building a kernel image (can take quite a long time depending on host computer performance) just to access the `dtc` and still being required to recompile the kernel after U-Boot has been built, the utilities in the `scripts` directory (including `dtc`) can be compiled with the following invocation:

```
$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- scripts
```

Build U-Boot and `mkimage`

After the device tree compiler has been built, U-Boot can be built from the it's top-level directory by invoking the following command:

```
$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Prebuilt

20

Device Tree

The device tree contains a hardware description that is used by Linux to perform initialization of the hardware system and determine control mechanisms (*i. e.*, peripheral drivers, device drivers) to use. The device tree is a data structure which is populated by nodes representing the various hardware peripherals. Within the device tree nodes are properties describing the details of the peripheral.

Device Tree Nodes

The hardware abstraction, described by the device tree, is split into several parts. Each part is called a node and describes a separate part of the hardware. This includes the memory, CPUs and hardware peripherals such as serial (UART) ports and any devices synthesized in programmable logic. Figure 20.1 shows an example truncated device tree with PS and PL peripheral nodes `uart0` and `axi_timer_0`, respectively.

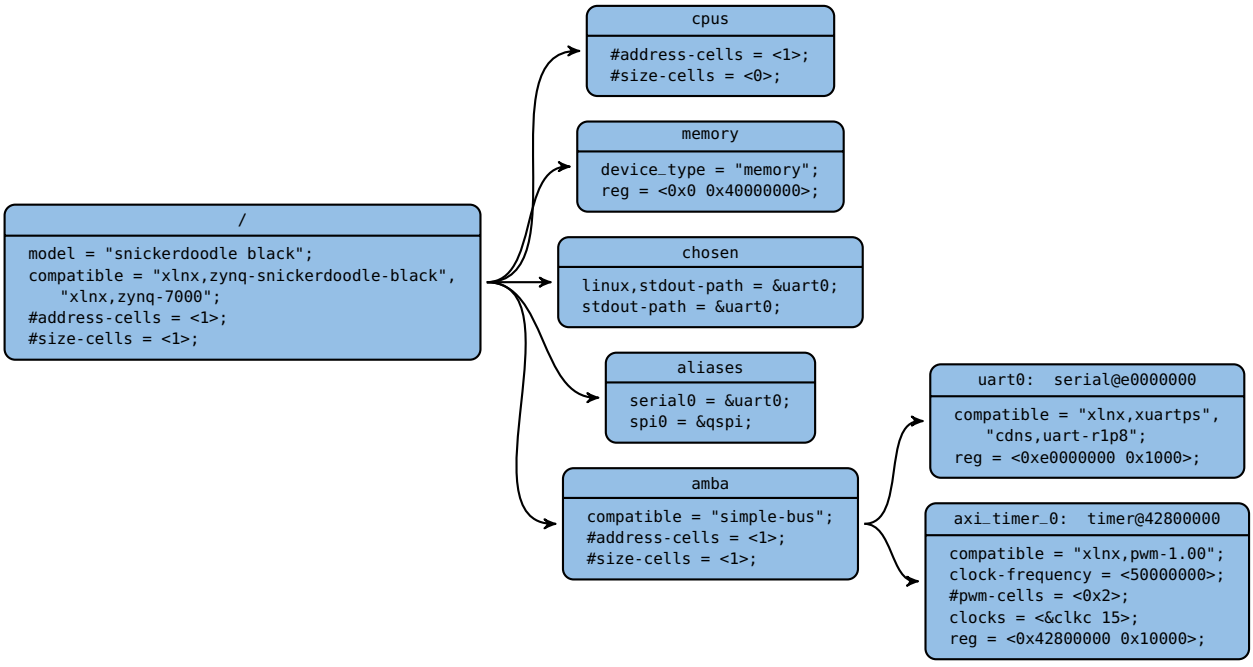


Figure 20.1: Devicetree Example

A devicetree describes device information in a system that cannot necessarily be dynamically detected by a client program[1].

[1] Linaro, Ltd., Harston Mill, Royston Road, Harston CB22 7GG. *Devicetree Specification*, 0.1 edition, 2016

Device Tree Node and Property Definitions

Code Listing 20.1: Device Tree Node and Property Definition Structure

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
}
```

Device Tree Bindings

Each node will be populated with a set of parameters which are used by the kernel and any drivers to configure and interact with the peripheral. Driver compatibility is declared using the *compatible* parameter and is used to tell the kernel which driver to use when using the peripheral described by the node.

Code Listing 20.2: Programmable Logic Device Tree Source Nodes with Driver Bindings

```
&amba {
        axi_timer_0: timer@42800000 {
                compatible = "xlnx,pwm-1.00";
                clock-frequency = <50000000>;
                #pwm-cells = <0x2>;
                clocks = <&clkc 15>;
                reg = <0x42800000 0x10000>;
        };

        axi_gpio_0: gpio@41200000 {
                compatible = "xlnx,xps-gpio-1.00.a";
                gpio-controller;
                #gpio-cells = <0x2>;
                interrupt-parent = <&intc>;
                reg = <0x41200000 0x10000>;
                xlnx,is-dual = <0x1>;
                xlnx,all-inputs = <0x0>;
                xlnx,tri-default = <0xFFFFFFFF>;
                xlnx,gpio-width= <0x19>;
                xlnx,all-inputs-2 = <0x0>;
                xlnx,tri-default-2 = <0xFFFFFFFF>;
                xlnx,gpio-width-2 = <0x19>;
        };
};
```

21

Kernel

Linux Source

The source required to configure and compile the Linux kernel for Snickerdoodle can be downloaded using the following `git` command:

```
$ git clone https://github.com/krtkl/snickerdoodle-linux.git
```

Checkout Stable Branch

At the time of this document's publication, the current stable and supported kernel release for Snickerdoodle is 3.14. The `sd-linux/3.14` branch of the Snickerdoodle Linux kernel has been updated to include the necessary driver source to run the Snickerdoodle peripherals (wireless, etc.). To switch the local repository to this branch, the following `checkout` command can be used:

```
$ git checkout sd-linux/3.14
```

Snickerdoodle Default Kernel Configuration

Before the kernel or any of the executables that are built, the kernel must be configured to match the hardware configuration. This includes architectural and driver support.

```
$ make ARCH=arm zynq_snickerdoodle_defconfig
```

Code Listing 21.1: Apply Snickerdoodle Default Kernel Configuration

Build U-Boot

After building the scripts within the Linux source repository, the path to the device tree compiler must be added to the `$PATH` variable so that it is available to `make` when building U-Boot. After navigating to **`snickerdoodle-linux`**

» **scripts** » **dtc**, the $PATH variable can be updated as shown below:

```
$ cd scripts/dtc
$ export PATH=$PATH:$(pwd)
```

Just as with the device tree compiler, for mkimage to be callable from make during the kernel build process, it's parent directory must be added to $PATH. The mkimage executable is built in the tools directory within the U-Boot source directory. By navigating to **snickerdoodle-u-boot** » **tools**, the $PATH variable can be appended with the location of the mkimage utility.

```
$ cd tools
$ export PATH=$PATH:$(pwd)
```

To test that mkimage has been made available to be called by the system (and to debug possible conflicting installations of mkimage), which can be used to output the parent directory of the mkimage binary:

```
$ which mkimage
/home/snickerdoodle/snickerdoodle-u-boot/tools/mkimage
```

The above output shows the mkimage executable located in the directory that was added to $PATH in the previous step.

Configure Linux

After the mkimage utility has been generated, the Linux kernel can be built for U-Boot. Before building, a configuration file (.config) must be produced to define the kernel build options. The default Snickerdoodle configuration can be specified using make:

```
$ make ARCH=arm zynq_snickerdoodle_defconfig
```

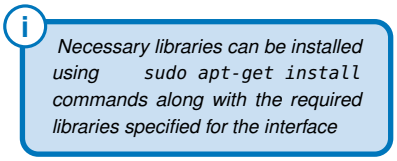
If additional configuration options are needed, the configuration can be edited using a variety of interfaces. make can be called to initiate the configuration interface using the following command:

```
$ make ARCH=arm <config_interface>
```

Options for <config_interface> are:

config - Text-based interface

menuconfig - Text-based interface with hierarchical menus, radiolists and interactive assistance. This option allows incremental saving of



changes. (ncurses must be installed: `libncurses5-dev`)

nconfig - Text-based menus (curses must be installed: `libcdk5-dev`)

xconfig - QT/X-windows interface (QT is required: `qt4-dev-tools`)

gconfig - Gtk/X-windows interface (GTK is required)

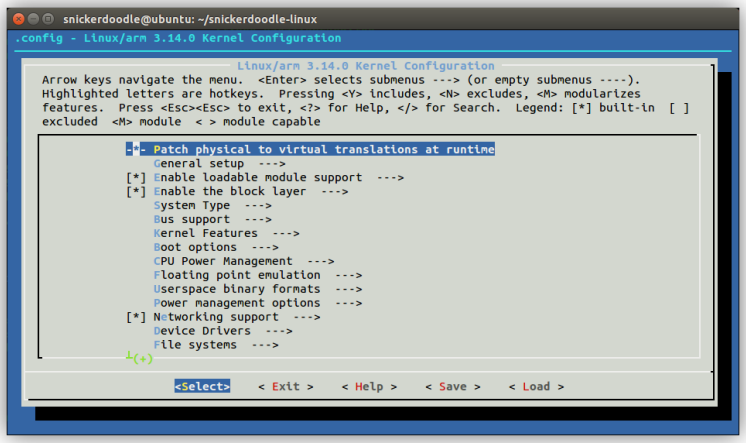


Figure 21.1: Menu Configuration Interface (`menuconfig`)

Figure 21.1 shows the QT based configuration interface that can be used to configure Linux build options.

Building Linux

The next and final step for building the Linux kernel is the build process itself. While building, the build components will be output to the console. A clean build can take quite a while. To build the Linux kernel with the necessary U-Boot header, the `uImage` argument should be specified. The following command will begin the build process:

```
$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- LOADADDR=0x8000 uImage
```

The build process will output messages for each compilation step. After the final build step is complete, a set of messages similar to the following will appear (note the output image path on the last line):

```
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.14.0-Snickerdoodle-00004
Created:      Fri Dec 11 16:34:56 2015
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4145568 Bytes = 4048.41 kB = 3.95 MB
Load Address: 00008000
```

```
Entry Point:  00008000
  Image arch/arm/boot/uImage is ready
```

Build and Install kernel Modules

The kernel modules, if any have been specified by the Linux configuration, can be built by invoking:

Code Listing 21.3: Compile kernel Modules

```
$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- modules
```

After building the kernel modules, they can be installed into a root file system by calling:

Code Listing 21.4: Install kernel Modules into Root Filesystem

```
$ make ARCH=arm INSTALL_MOD_PATH=<path_to_rootfs> modules_install
```

Loading Bitstream from Linux

Bitstreams can be loaded to the FPGA from a booted Linux system. This allows bitstreams to be loaded on a system without needing to regenerate a boot image or reboot the system. This can be useful for testing and development.

Preparing Bitstream for Loading (bit-reversing)

The bitstreams must be bit-reversed before they can be loaded from Linux. The `-split` argument can be supplied to the `bootgen` utility to tell it to create a bit-reversed copy of the bitstream that can be loaded from Linux.

```
$ bootgen -image boot.bif -split bin -o i BOOT.bin
```

Using `-split bin` will append a `.bin` suffix to the input files specified in the boot image file (`boot.bif`). In the case of the `.bif` example above, this will output a new, bit-reversed bitstream named `bitstream.bit.bin`.

Writing Bitstream to Device

CAUTION Attempting to write bitstreams that have not been reformatted to bit-reversed will not successfully load to `xdevcfg` and the `prog_done` flag will fail to be asserted.

Now that the bitstream has been bit-reversed, it can be written to the FPGA using the `xdevcfg` driver. To load the bitstream, write the contents of the bitstream file to the `xdevcfg` device using the following command:

```
$ cat bitstream.bit.bin > /dev/xdevcfg
```

Writing the bitstream to the device can take some time. The `prog_done` flag can be read to check whether the bitstream has finished loading:

```
$ cat /sys/devices/amba.0/f8007000.ps7-dev-cfg/prog_done
1
```

This process allows bitstreams to be loaded, and thus the FPGA to be reconfigured, without needing to alter the *BOOT* partition of the SD card (*i.e.*, `BOOT.bin`) or reboot the system.

22

Boot Medium

Load microSD Card Using Windows

Because Windows does not support reading/writing of Linux filesystems (ext2, etc.), the partitioning and formatting of the microSD card must be done with disk images. The disk image contains all of the information required to define the partition scheme, disk format and file system contents. This includes the boot partition components (*`BOOT.bin`*, *`devicetree.dtb`*, etc.) as well as the root filesystem partition and it's components.

Format the microSD Card

To load a bootable microSD card with a Snickerdoodle Linux system, the card must first be formatted. A tool for formatting the microSD card is produced by the SD Association and can be found at `https://www.sdcard.org/downloads/formatter_4`. After downloading and installing the SDFormatter software, connect the microSD card to the host machine and run the SDFormatter.

CAUTION Be sure to check that the drive letter in the SDFormatter is that of the inserted microSD card to avoid formatting (and erasing) any drives unintentionally.

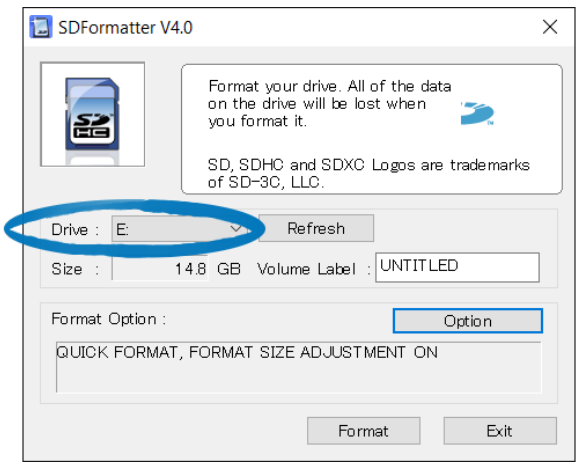


Figure 22.1: SDFormatter V4.0 Interface

Before formatting the card, set the *FORMAT SIZE ADJUSTMENT* to ***ON*** in the Option Settings as shown in Figure 22.2 . The *FORMAT TYPE* can be left as ***QUICK***.

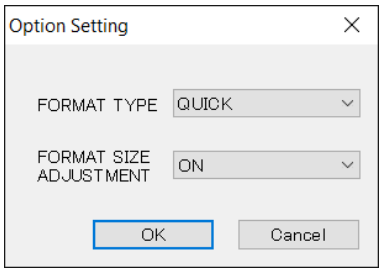


Figure 22.2: Set Format Options in SDFormatter

After setting the format options, select ***Format*** from the SDFormatter interface to start formatting the disk.

Load the microSD Card Image

Before loading the Linux system on the microSD card, first download the system image. SD card images, as well as Linux system components and filesystems can be downloaded from `http://krtkl.com/downloads/`. The SD card image represents a 4GB SD card but can be loaded onto any size microSD card of 4GB or larger size. The compressed image is much smaller than 4GB as the system does not use nearly the full size of a 4GB card. After downloading, extract the system image to a convenient directory that will be accessed when writing the image to the card.

The Win32 Disk Imager utility is used to write the disk image to the freshly formatted microSD card. Download the utility from `http://sourceforge.net/projects/win32diskimager` and install it on the Windows host.

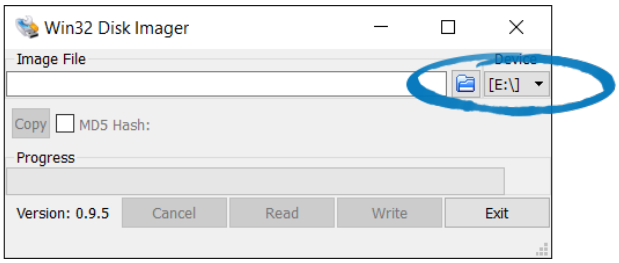


Figure 22.3: Win32 Disk Imager Disk Letter Selection

After opening the Win32 Disk Imager, verify the drive letter matches that of the newly formatted card. By selecting the folder button next to the image file path, you will be able to navigate to and select the extracted system image file. After selecting the system image, select ***Write*** to begin writing the image to the microSD card, as shown in Figure 22.4 .

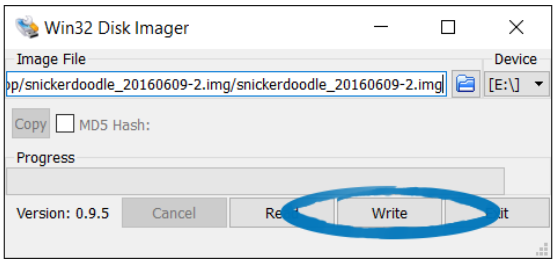


Figure 22.4: Write Linux System Disk Image with Win32 Disk Imager

When the Win32 Disk Imager is finished writing the Linux system to the microSD card it can be ejected from the Windows host and loaded on Snickerdoodle. After mounting the microSD, Snickerdoodle can be powered

on and booted from the microSD card. For information on connecting to the Snickerdoodle Linux console, refer to Chapter 18 .

Load microSD Card Using OS X

OS X does not have native file system support for Linux filesystems. For this reason, the microSD card creation process parallels the process for Windows hosts.

Format the microSD Card

The SD Association supports an SD card formatting utility for OS X machines. The utility can be downloaded from `https://www.sdcard.org/downloads/formatter_4`. After downloading and installing the SDFormatter, insert the microSD card and run the SDFormatter program.

Load the microSD Card Image

Before attempting to copy the Linux system image to the microSD card, verify the disk location on the OS X host. `mount` can be used from a terminal to check the mount point of the newly formatted card. Code Listing 22.1 shows the truncated output of the `mount` command, showing the disk label and `/dev` tree mount point. In this example, the disk labeled 'UNTITLED' is the first and only partition on `/dev/disk4` as it was created during the card formatting step (with it's own file path `/dev/disk4s1`).

Code Listing 22.1: Verify microSD Card Location in OS X

```
$ mount
...
/dev/disk4s1 on /Volumes/UNTITLED (msdos, local, nodev, nosuid,
    noowners)
```

The formatted partition should be unmounted from the OS X host, before copying the Linux system image. Rather than using `umount`, the particularities of OS X require `diskutil unmount` be used to unmount the card's partition.

Code Listing 22.2: Unmount microSD Card Partition in OS X

```
$ diskutil unmount /dev/disk4s1
Volume UNTITLED on disk4s1 unmounted
```

CAUTION When loading the system image to the microSD card, do not include the partition in the output file name. In this example, `/dev/disk4s1` is the newly formatted FAT32 partition on `/dev/disk4` and so `/dev/disk4` is used as the output file for `dd`.

Code Listing 22.3: Load Linux System Image to microSD Card from OS X

```
$ sudo dd if=<image_path> of=/dev/disk4 bs=1m
```

After loading the image to the microSD card, the card can be ejected using `diskutil eject` and is ready to be loaded into Snickerdoodle.

```
$ sudo diskutil eject /dev/disk4
Disk /dev/disk4 ejected
```

After loading the microSD card, Snickerdoodle is ready to be powered on boot from the microSD card. For information on connecting to the Snickerdoodle Linux console, refer to Chapter 18 .

Create microSD Card Using Linux

From a Linux environment, the microSD card can be created from the individual system components. This allows for greater flexibility in the configuration of the system and allows an opportunity to pre-load custom system components rather than replacing the components in the default configuration image.

Connecting and Locating the microSD Card

The `mount` command can be used to locate the SD card device, once it has been connected to the host computer. In the example below, an SD card has been connected on `/dev/sdb1` and mounted at `/media/user/UNTITLED`.

```
$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/cgroup type tmpfs (rw)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
...
/dev/sdb1 on /media/user/UNTITLED type vfat
    (rw,nosuid,nodev,uid=1000,gid=1000,shortname=mixed,dmask=0077,
    utf8=1,showexec,flush,uhelper=udisks2)
```

Partitioning the microSD Card (`fdisk`)

Before partitioning the SD card, any partitions that have been mounted on the system must be unmounted. This can be done using the `umount` command.

```
$ umount /dev/sdb1
```

Once the SD card has been located, we can partition it for the Linux system (*BOOT* partition) and root filesystem (*ROOTFS* partition) using `fdisk`[1]. `fdisk` must be run with root permissions (`sudo`) using the disk parent as the argument (do not use the parition number in the argument).

```
$ fdisk /dev/sdb
```

[1] Additional information on `fdisk` can be found at: http://tldp.org/HOWTO/Partition/fdisk_partitioning.html

CAUTION Do NOT include any partition number (in the example case '1') when running `fdisk` on the SD card. '`/dev/sdb`' NOT '`/dev/sdb1`'

From within the `fdisk` interface, we can view the parition table at any time using the 'p' command. In this example, an 8GB SD card with a single FAT32 partition is being used and will be re-partitioned for snickerdoodle.

```
Command (m for help): p

Disk /dev/sdb: 7969 MB, 7969177600 bytes
255 heads, 63 sectors/track, 968 cylinders, total 15564800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            8192    15564799     7778304    b  W95 FAT32
```

BOOT Partition

First, a partition must be allocated for the Linux system binaries and files. This includes *BOOT.bin* (FSBL, bitstream, U-Boot), *uEnv.txt*, *devicetree.dtb*, and the Linux kernel *uImage*. The partition size for these files is recommended to be 128MB in size which translates to an additional 262144, 512 byte sectors.

```
Command (m for help): d
Selected partition 1

Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-15564799, default 2048): <RETURN>
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-15564799, default
    15564799): +262144
```

The default partition type for `fdisk` is **Linux** (type ID 83). The *BOOT* partition needs to be formatted as **FAT32** (type ID 'C'). Do do this, the 't' command is used:

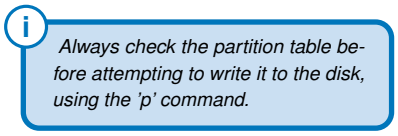
```
Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
```

ROOTFS Partition

Second, a partition for the root filesystem must be created. This partition will be formatted as a **Linux** type using type ID 83. This is the default partition type.

```
Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partition number (1-4, default 2): <RETURN>
Using default value 2
First sector (264193-15564799, default 264193): <RETURN>
Using default value 264193
Last sector, +sectors or +size{K,M,G} (264193-15564799, default
    15564799): <RETURN>
Using default value 15564799
```

Before writing the parition table, you should verify the partition layout by printing it with the 'p' command. In this example, an 8GB SD card has been partitioned with a 128MB FAT32 *BOOT* partition and the rest allocated for a Linux *ROOTFS* partition.




```
Command (m for help): p

Disk /dev/sdb: 7969 MB, 7969177600 bytes
255 heads, 63 sectors/track, 968 cylinders, total 15564800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            2048      264192      131072+   c  W95 FAT32
    (LBA)
/dev/sdb2          264193    15564799     7650303+  83  Linux
```

Once the partition table has been verified, the 'w' command can be used to write the table to the disk:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

Formatting Partitions (`mkfs/mke2fs`)

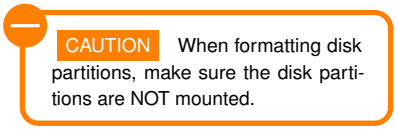
With a partitioned SD card, the partitions need to be formatted with the necessary filesystem type. For the *BOOT* partition, the filesystem type is

[2] Additional information on `mkfs/mke2fs` and it's front end tools can be found at: http://www.tldp.org/HOWTO/Partition/formatting.html

VFAT. Formatting the *BOOT* partition can be done using the `mkfs.vfat`[2]. To format a FAT32 filesystem on `/dev/sdb1` with a 'BOOT' disk label, the following command can be used:

```
$ mkfs.vfat -n BOOT /dev/sdb1
```

The format for the *ROOTFS* partition can be done with `mke2fs` which will format a Linux partition with an ext2/ext3/ext4 filesystem. To format an ext4 filesystem on `/dev/sdb2` with a block size of 1k (1024) and a 'ROOTFS' disk label, the following command can be used:



CAUTION When formatting disk partitions, make sure the disk partitions are NOT mounted.

```
$ mke2fs -b 1024 -t ext4 -L ROOTFS /dev/sdb2
```

If the formatting is successful, the following output with be written to the console (writing superblocks and filesystem accounting information can take some time depending on the size and speed of the SD card):

```
mke2fs 1.42.9 (4-Feb-2014)
Filesystem label=ROOTFS
OS type: Linux
Block size=4096 (log=0)
Fragment size=4096 (log=0)
Stride=0 blocks, Stripe width=0 blocks
478208 inodes, 7650300 blocks
382515 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=74973184
934 block groups
8192 blocks per group, 8192 fragments per group
512 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409,
            663553,
        1024001, 1990657, 2809857, 5120001, 5971969

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

After the partitions have been properly formatted, the SD card must be ejected and re-connected before moving the Linux boot components and root filesystem contents to the disk.

```
$ eject /dev/sdb
```

BOOT Partition Components

The latest *BOOT* partition Linux components for Snickerdoodle and Snickerdoodle Black can be downloaded using `git`.

```
$ git clone https://github.com/krtkl/snickerdoodle-linux-prebuilt.git
```

Alternatively, the sources can be downloaded directly from a web browser from the krtkl GitHub page as a `.ZIP` file.

Copy Files to SD Card

After downloading the SD card components
The *BOOT* components can be installed onto the SD card using the `cp` command, starting with `BOOT.bin`. In this example, the files are copied to the *BOOT* partition that has been mounted at `/media/user/BOOT`. As stated above, the `mount` command can be used to locate the mount point of the SD card partitions.

```
$ cp BOOT.bin /media/user/BOOT
$ cp uEnv.txt /media/user/BOOT
$ cp devicetree.dtb /media/user/BOOT
$ cp uImage /media/user/BOOT
```

After copying the *BOOT* components, the `sync` command should be used to make sure the system buffers have been flushed and the process of writing the files to the SD card is complete.

```
$ sync
```

ROOTFS Sources

The root filesystem can be extracted directly into the *ROOTFS* partition using the '-C' argument when extracting the archive contents. An Ubuntu 14.04 filesystem can be downloaded from `http://krtkl.com/downloads/`. In this example, the *ROOTFS* partition is mounted at `/media/user/ROOTFS`, which should be checked before attempting to extract the root filesystem. The root filesystem contains a lot of large packages (ROS, python, etc.) and may take several minutes to complete the process of writing to the SD card.

```
$ tar -C /media/user/ROOTFS -xvzf
    snickerdoodle-ubuntu-14.04.tar.gz
```

After extracting the root filesystem to the SD card, use the `sync` command to flush the system buffers and ensure the write process is complete. Ad-

ditionally, making sure to unmount the SD card partitions before ejecting will make certain that any lingering write processes have completed before the SD card is removed.

```
$ sync
$ umount /dev/sdb1
$ umount /dev/sdb2
$ eject /dev/sdb
```

23

Booting from QSPI

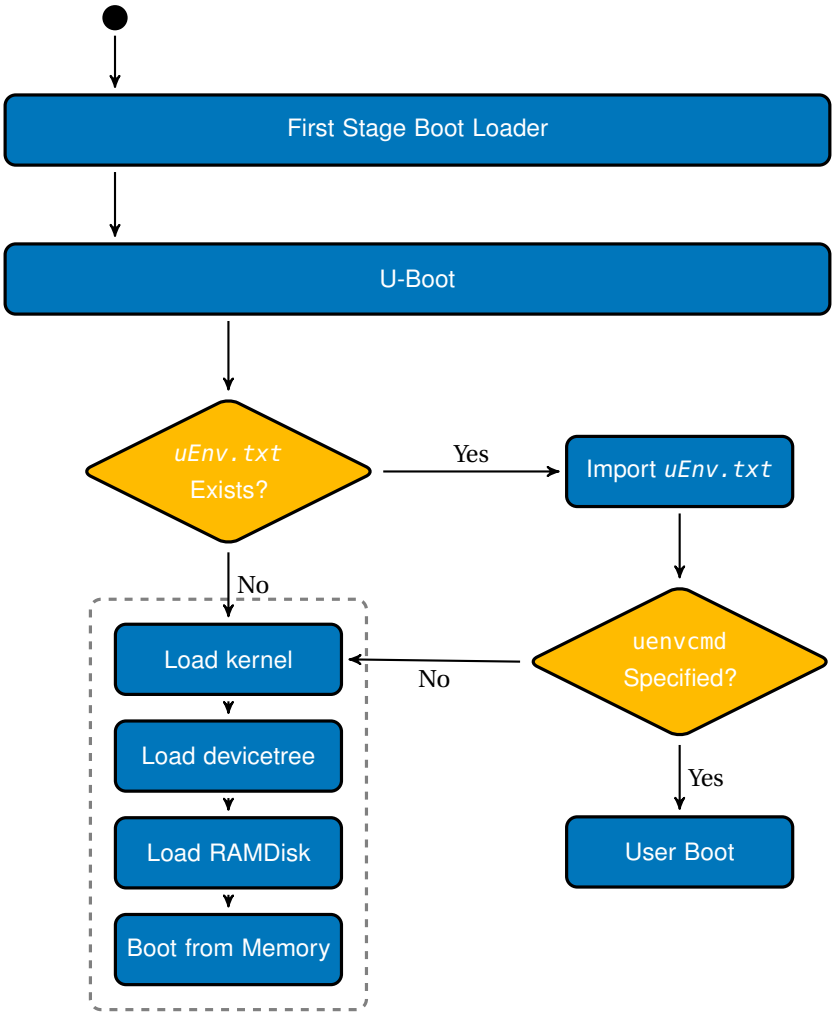


Figure 23.1: microSD Card Boot Process

SD Card Preparation

Fetching Boot Sources

Formatting SD card from Windows Computer Management

Updating QSPI Images

Snickerdoodle has several boot source options.

Boot media can be combined with a variety of system arrangements. Some common arrangements include booting Linux images from the QSPI and mounting an SD card for mass storage, and booting Linux images and filesystem from a partitioned SD card and using the QSPI flash for secure storage and backups.

Update QSPI Images from U-Boot

SD Card Preparation

When updating the SD card from U-Boot, only the components required to load U-Boot along with the update components are needed. All of these components should be written to a FAT filesystem partition on the SD card and then...

Code Listing 23.1: Bring Up the Wireless Interface Using `ifconfig`

```
$ ifconfig wlan0 up
[  136.536102] wlcore: PHY firmware version: Rev 8.2.0.0.224
[  136.644809] wlcore: firmware booted (Rev 8.9.0.0.31)
[  136.660641] IPv6: ADDRCONF(NETDEV_UP): wlan0: link is not ready
```

Code Listing 23.2: Starting `wpa_supplicant` in QSPI Linux

```
$ wpa_supplicant -Dnl80211 -c/etc/wpa_supplicant.conf -iwlan0 -B
Successfully initialized wpa_supplicant
[  284.698468] wlan0: authenticate with 94:10:3e:91:f5:ba
[  284.706618] wlan0: send auth to 94:10:3e:91:f5:ba (try 1/3)
[  284.739128] wlan0: authenticated
[  284.751229] wlan0: associate with 94:10:3e:91:f5:ba (try 1/3)
[  284.785542] wlan0: RX AssocResp from 94:10:3e:91:f5:ba
    (capab=0x411 status=0 aid=19)
[  284.800398] IPv6: ADDRCONF(NETDEV_CHANGE): wlan0: link becomes
    ready
[  284.806702] wlan0: associated
[  284.912502] wlcore: Association completed.
```

Code Listing 23.3: Checking Network Connection With `iw`

```
$ iw wlan0 link
Connected to 94:10:3e:91:f5:ba (on wlan0)
        SSID: KX
        freq: 2437
        RX: 11256 bytes (44 packets)
        TX: 1189 bytes (11 packets)
        signal: -46 dBm
        tx bitrate: 150.0 MBit/s MCS 7 40MHz short GI

        bss flags:      short-preamble short-slot-time
        dtim period:    1
        beacon int:     100
```

Code Listing 23.4: Obtaining an IP Address Using `udhcpc`

```
$ udhcpc -iwlan0
udhcpc (v1.20.1) started
```

```
Sending discover...
Sending select for 10.0.111.180...
Lease of 10.0.111.180 obtained, lease time 600
deleting routers
route: SIOCDELRT: No such process
adding dns 10.0.111.1
```

```
$ ping google.com
PING google.com (216.58.194.174): 56 data bytes
64 bytes from 216.58.194.174: seq=0 ttl=54 time=31.311 ms
64 bytes from 216.58.194.174: seq=1 ttl=54 time=21.386 ms
64 bytes from 216.58.194.174: seq=2 ttl=54 time=52.124 ms
^C
--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 21.386/34.940/52.124 ms
```

Code Listing 23.5: Testing Network/Internet Connectivity with `ping`

Partitioning the SD card from Snickerdoodle

```
Disk /dev/mmcblk0: 15.9 GB, 15931539456 bytes
32 heads, 32 sectors/track, 30387 cylinders, total 31116288
    sectors
Units = sectors of 1 * 512 = 512 bytes

        Device Boot      Start         End      Blocks  Id System
/dev/mmcblk0p1              32      262143      131056   c Win95
    FAT32 (LBA)
/dev/mmcblk0p2          262144    31116287    15427072  83 Linux
```

```
Disk /dev/mmcblk0: 15.9 GB, 15931539456 bytes
32 heads, 32 sectors/track, 30387 cylinders
Units = cylinders of 1024 * 512 = 524288 bytes

        Device Boot      Start         End      Blocks  Id System
/dev/mmcblk0p1               1         256      131056   c Win95
    FAT32 (LBA)
/dev/mmcblk0p2             257       30387    15427072  83 Linux
```

```
$ mkdosfs -n BOOT /dev/mmcblk0p1
mkdosfs 3.0.12 (29 Oct 2011)
```

```
$ mke2fs -b 4096 -L ROOTFS /dev/mmcblk0p2
Filesystem label=ROOTFS
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
964768 inodes, 3856511 blocks
192825 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
118 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
```

```
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736,
            1605632, 2654208
```

Double check the partition table

```
$ fdisk -l /dev/mmcblk0
Disk /dev/mmcblk0: 15.9 GB, 15931539456 bytes
32 heads, 32 sectors/track, 30387 cylinders
Units = cylinders of 1024 * 512 = 524288 bytes

        Device Boot      Start         End      Blocks  Id System
/dev/mmcblk0p1               1         256      131056   c Win95
    FAT32 (LBA)
/dev/mmcblk0p2             257       30387    15427072  83 Linux
```

Mount ROOTFS Partition

```
$ mount /dev/mmcblk0p2 /card/ROOTFS
```

Download Filesystem Tarball

```
$ wget ftp://10.0.111.179/rootfs/snickerdoodle-ubuntu-14.04.tar.gz
Connecting to 10.0.111.179 (10.0.111.179:21)
snickerdoodle-ubuntu   9% |**                            | 41014k  0:05:06 ETA
```

```
$ ftpget -v 10.0.111.179 rootfs/snickerdoodle-ubuntu-14.04.tar.gz
Connecting to 10.0.111.179 (10.0.111.179:21)
ftpget: cmd (null) (null)
ftpget: cmd USER anonymous
ftpget: cmd PASS busybox@
ftpget: cmd TYPE I (null)
ftpget: cmd PASV (null)
ftpget: cmd SIZE snickerdoodle-ubuntu-14.04.tar.gz
ftpget: cmd RETR snickerdoodle-ubuntu-14.04.tar.gz
ftpget: cmd (null) (null)
ftpget: cmd QUIT (null)
```

Extract and Install Filesystem

```
$ tar -xvzf snickerdoodle-ubuntu-14.04.tar.gz
```

24

Wireless

This chapter details the process and methods used to connect Snickerdoodle to a local wireless network to establish network and internet connectivity. The files and commands outlined in this chapter are found accessed directly from a booted Snickerdoodle using a terminal interface established through connection with the snickerdoodle serial port located at J1 or J2.

The following information contains details on how to connect to a variety of network security modes using a combination of automated and manual processes.

Portions of this chapter have been adapted from the TI WL18xx documentation found at http://processors.wiki.ti.com/index.php/Connect_to_Secure_AP_using_WPA_Supplicant

WPA Supplicant (`wpa_supplicant`)

The `wpa_supplicant` is a program that initializes and establishes network connectivity based on a set of configurations specified by a combination of a configuration file and (optionally) command line input.

`wpa_supplicant.conf`

Network configuration can be stored and additionally recalled automatically when booting snickerdoodle to connect to known and trusted networks. These configurations are typically stored in `/etc/wpa_supplicant.conf`[1]. Multiple network configurations can be stored in a single `wpa_supplicant.conf`.

[1] Additional information on the structure of `wpa_supplicant.conf` can be found at http://linux.die.net/man/5/wpa_supplicant.conf

```
$ wpa_supplicant -d -Dnl80211 -c/etc/wpa_supplicant.conf -iwlan0 -B
```

Configuration Structure

An example network configuration for an open (unsecured) network can be found below. The first three lines of the file specify the configuration for the front-end (`wpa_cli`). Additional configuration examples for various network security types can be found at the end of this document.

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
```

Code Listing 24.1: Example *wpa_supplicant.conf* File

```
update_config=1

network={
        auth_alg=OPEN
        key_mgmt=NONE
        ssid="my_Network"
        mode=0
}
```

`wpa_supplicant` is daemon and therefore only one instance of it may be run on a snickerdoodle, all other modifications and configuration of settings must be made with the frontend application, `wpa_cli`.

WPA Command Line Interface (`wpa_cli`)

wpa_cli is a text-based frontend program for interacting with wpa_supplicant. It is used to query current status, change configuration, trigger events, and request interactive user input.

```
wpa_cli -i interface [-hv] [command ...]
```

- *`-h`* prints help information
- *`-v`* verbose output
- *`-i interface`* interface name, typically wlan0

In all commands used in the examples the first argument is the interface that is being configured, preceded by the `-i` flag (`-i wlan0`).

`wpa_cli` Commands

Setting the attributes of a network is done through a set of commands that are specified as arguments to `wpa_cli`. Many of these commands require additional arguments to specify the attributes that are being set.

CAUTION Pay careful attention to the order and number of arguments used for each command. Some commands (especially the `set_network` command) utilize a variable number of arguments to set various network parameters.

`disconnect` - Puts the interface into a disconnected state and waits for a reassociate command before connecting.

```
wpa_cli -i wlan0 disconnect
```

`add_network` - Adds and assigns an index number to a new network for the specified interface. All necessary configuration of the newly added network is done with `set_network` command.

```
wpa_cli -i wlan0 add_network
```

`select_network` - Selects network with the specified succeeding index argument and disables others.

```
wpa_cli -i wlan0 select_network 0
```

enable_network - Enables network with specified succeeding index argument.

```
wpa_cli -i wlan0 enable_network 0
```

reassociate - Forces reassociation of the currently selected network.

```
wpa_cli -i wlan0 reassociate
```

status - Gets current WPA/EAPOL/EAP status of the currently selected network.

```
wpa_cli -i wlan0 status
```

list_networks - Lists configured networks for the specified interface.

```
wpa_cli -i wlan0 list_networks
```

remove_network - Removes network with the specified succeeding SSID argument from the list of configured networks.

```
wpa_cli -i wlan0 remove_network '"Bandipur"'
```

set_network Command

The following set of commands are used with the `set_network` command to specify network attributes. All of the following commands require the argument following "`set_network`" to be the network index, followed by any parameter commands and arguments.

auth_alg - Configures network authentication algorithm. Typically, the complimentary argument is `OPEN`.

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
```

key_mgmt - Sets authenticated key management protocol for the network. Possible protocols are `NONE`, `WPA-EAP` or `WPA-PSK`.

```
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
```

psk - Sets WPA passphrase or pre-shared key (PSK) to be used with the network.

```
wpa_cli -i wlan0 set_network 0 psk '"12345678"'
```

wep_key0 - Sets WEP key passphrase to be used by the network.

```
wpa_cli -i wlan0 set_network 0 wep_key0
    ABCdef1234567890abcDEF3333
```

pairwise - Sets CCMP algorithm as pairwise (unicast) cipher for WPA.

```
wpa_cli -i wlan0 set_network 0 pairwise CCMP
```

group - Sets CCMP algorithm as group (broadcast/multicast) cipher for WPA.

```
wpa_cli -i wlan0 set_network 0 group CCMP
```

If an error is made while setting a network parameter, simply re-input the corresponding `set_network` command with the correct arguments to override the parameter setting. To un-set parameters (for different configuration types) the network must be removed from the list.

proto - Sets security protocol as WPA2.

```
wpa_cli -i wlan0 set_network 0 proto WPA2
```

eap - Sets PEAP as extensible authentication protocol's (EAP) method.

```
wpa_cli -i wlan0 set_network 0 eap PEAP
```

identity - Sets identity string for EAP.

```
wpa_cli -i wlan0 set_network 0 identity '"test"'
```

password - Sets password string for EAP.

```
wpa_cli -i wlan0 set_network 0 password '"test"'
```

phase1 - Sets outer authentication method version as PEAPv0.

```
wpa_cli -i wlan0 set_network 0 phase1 '"peapver=0"'
```

phase2 - Sets inner authentication method as EAP-MSCHAPv2.

```
wpa_cli -i wlan0 set_network 0 phase2 '"MSCHAPV2"'
```

mode - Sets IEEE 802.11 operation mode as infrastructure (Managed) mode, i.e., associate with an AP.

```
wpa_cli -i wlan0 set_network 0 mode 0
```

ssid - Sets network service set identifier (SSID) of the specified network index.

```
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

`wpa_cli` Invocation

Configuring wireless networks using `wpa_cli` is an excellent way to test network connectivity and settings before applying those configurations to automatically establish connections. When using `wpa_cli`, it is important to invoke `wpa_supplicant` exactly ONCE before attempting to use any `wpa_cli` commands. `wpa_supplicant` should be used with the `-B` flag to daemonize the supplicant. An example call to `wpa_supplicant`:

```
wpa_supplicant -d -D nl80211 -c /etc/wpa_supplicant.conf -iwlan0
    -B
```

This call uses the `nl80211` driver (specified by `-D`) and a configuration file specified by `-c`. The `-d` flag is used to output verbose debugging messages.

Removing Existing Networks

Removing all of the networks from the network list can be done using a combination of invocations of `wpa_cli`. This can be useful when wanting to work with only one network or testing network configurations manu-

ally. An example script/command which will remove all of the networks currently stored in the `wpa_cli` list is as follows:

```
for i in ‘wpa_cli -iwlan0 list_networks | grep ^[0-9] | cut -f1‘;
    do wpa_cli -iwlan0 remove_network $i; done
```

After removing all networks, a single network can be added by using the `add_network` command.

```
$ wpa_cli -i wlan0 add_network
```

Setting Network Parameters

After adding a network, it can be configured using a ’0’ index for any `set_network` commands that follow. Configuring the network parameters is done through a series of `set_network` commands that specify the characteristics of the network based on it’s authentication/security type and the credentials specific to the network. Below is an example of an unsecured (open) network. The `set_network` commands for other network configuration types can be found at the end of this document

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt NONE
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid ’"my_Network"’
```

After configuring the network using a series of `set_network` commands, the network can be enabled using the following series of commands:

```
wpa_cli -i wlan0 select_network 0
wpa_cli -i wlan0 enable_network 0
wpa_cli -i wlan0 reassociate
```

To verify connection status with the WPA supplicant, the `status` command can be used as shown below:

```
$ wpa_cli -i wlan0 status
bssid=ff:ee:dd:cc:bb:aa
ssid=my_Network
id=0
mode=station
pairwise_cipher=CCMP
group_cipher=CCMP
key_mgmt=WPA2-PSK
wpa_state=COMPLETED
```

```
p2p_device_address=12:34:56:78:90:ab
address=cd:ef:11:22:33:44
```

Additional information about the wireless connection can be viewed using the `iw` command.

```
$ iw wlan0 link
Connected to ff:ee:dd:cc:bb:aa (on wlan0)
        SSID: my_Network
        freq: 2462
        RX: 216135 bytes (735 packets)
        TX: 1345 bytes (13 packets)
        signal: -39 dBm
        tx bitrate: 72.2 MBit/s MCS 7 short GI

        bss flags:      short-preamble short-slot-time
        dtim period:    1
        beacon int:     100
```

For networks where a DHCP server is already running and administering IP addresses to clients, the command can be used to obtain an IP address and finalize connection to the network. Using `ifconfig` will confirm the snickerdoodle IP address.

```
$ dhclient wlan0
$ ifconfig
...
wlan0     Link encap:Ethernet  HWaddr 12:34:56:78:90:ab
          inet addr:10.0.111.112  Bcast:10.0.111.255
              Mask:255.255.255.0
          inet6 addr: fe80::36b1:f7ff:fedf:6f93/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1130 errors:0 dropped:1 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:305537 (305.5 KB)  TX bytes:2134 (2.1 KB)
```

BASEBOARDS

Introduction

An extremely convenient way to work with Snickerdoodle is through a baseboard. Baseboards can be used for everything from exposing the I/O to easy to access pin headers (*i.e.*, the breakyBreaky breakyout board) to providing fully integrated hardware interfaces (*e.g.*, HDMI, Ethernet, motor controllers/drivers). The connectors down configuration, referenced in Chapter 2 .

25

breakyBreaky

Breakout Pin Headers

Describe the overall connector layout.

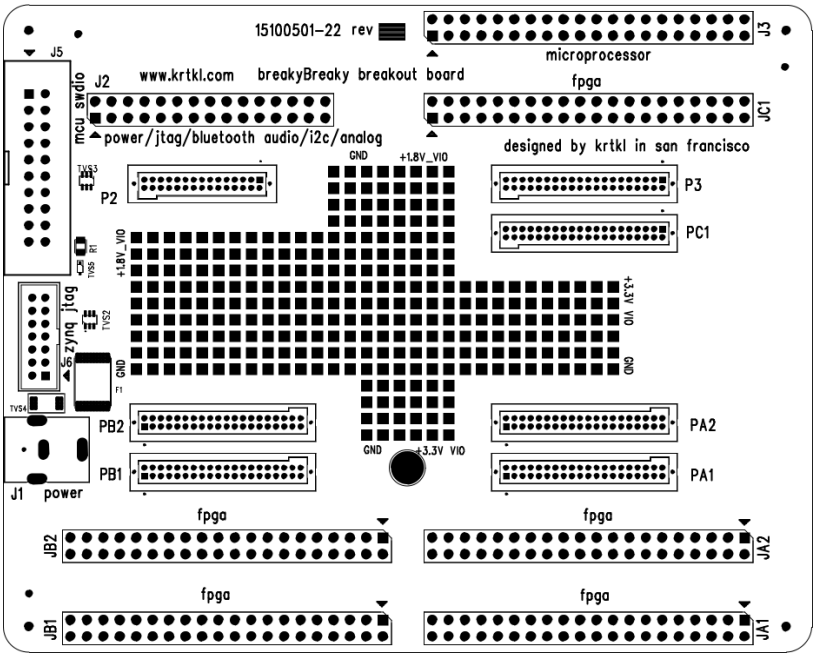


Figure 25.1: breakyBreaky Connector Layout

CAUTION The breakout pin headers are rotated 180° from the corresponding Snickerdoodle connectors. Pin 1 on the pin headers are indicated by arrows in the breakyBreaky silkscreen.

Power Connector (J1)

The J1 connector is a 5.5mm x 2.5mm barrel jack which provides an external power source to the Snickerdoodle. Power can also be independently provided through the micro-USB connector on the Snickerdoodle which also provides the console.

FPGA Connectors

The FPGA connectors provide access to the user designated FPGA I/O.

JTAG Connectors (J5 and J6)

Programming of the Snickerdoodle platform controller and the application processor can both be achieved (if desired) using the JTAG connectors on breakyBreaky. The pins that provide the programming interface are also connected to J2 which (as with the other connectors) mirrors the corresponding connector on the Snickerdoodle. The JTAG connectors provide a standardized connection for ease of access with software flashloaders/debuggers.

APPENDICES

A

High-Level Block Diagram

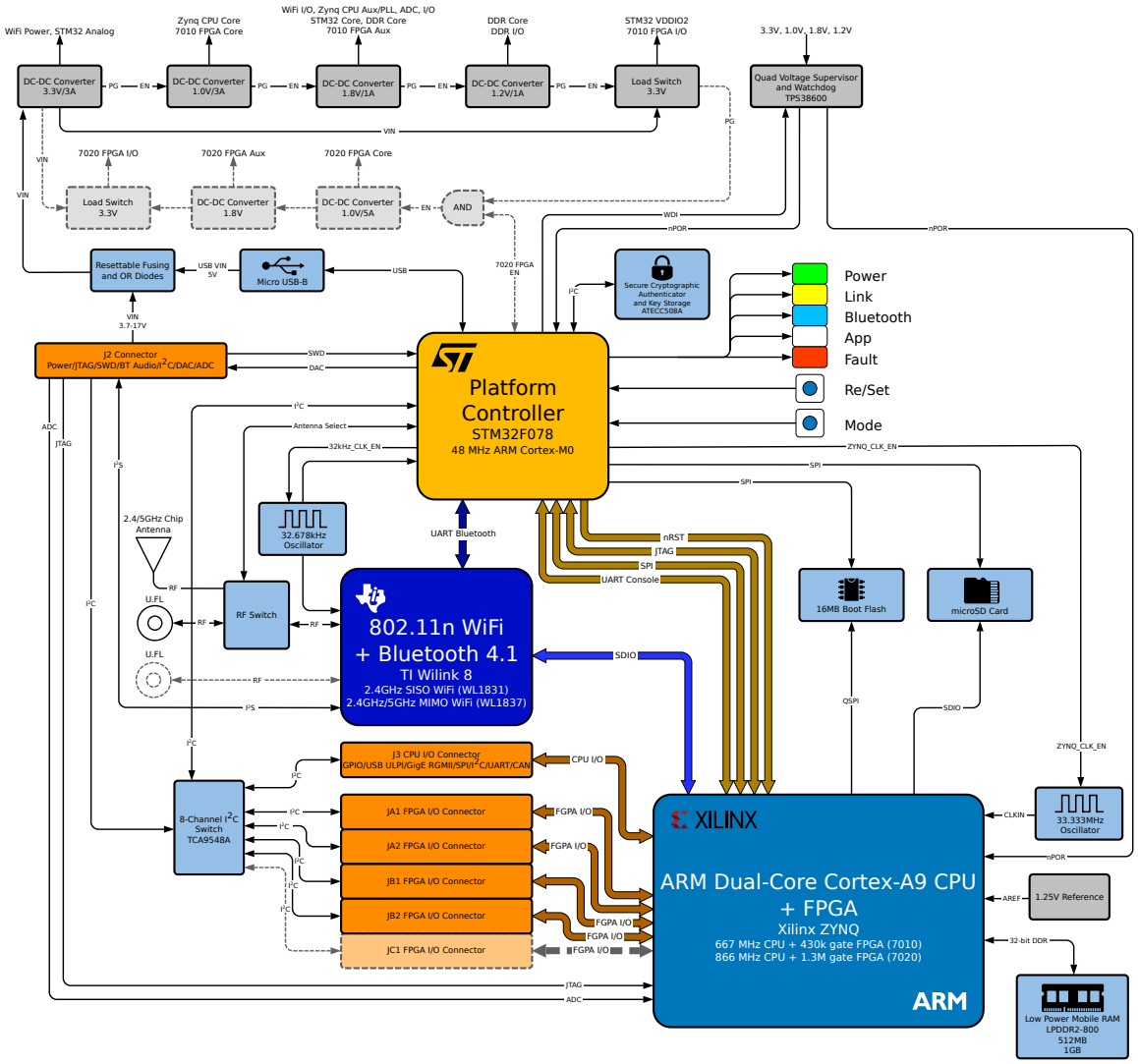
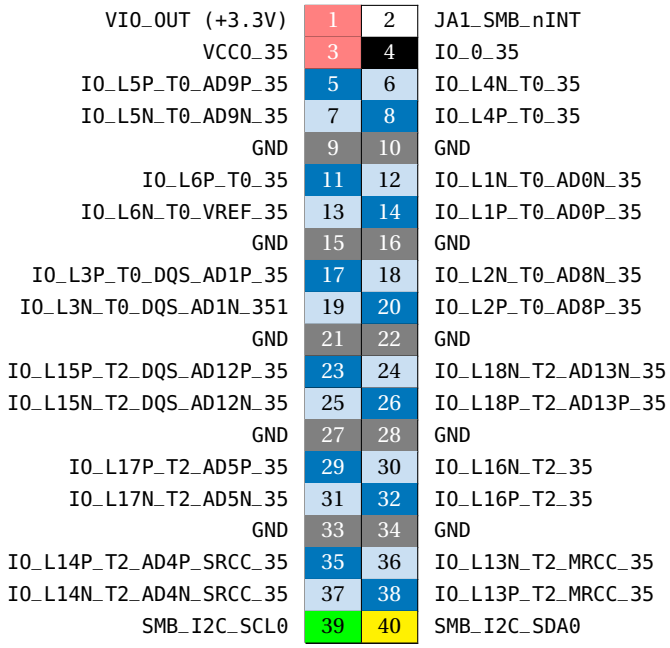


Figure A.1: Snickerdoodle High-Level Block Diagram

B

Connectors

FPGA Connectors



VIO_OUT (+3.3V)	1	2	JA1_SMB_nINT
VCCO_35	3	4	IO_0_35
IO_L5P_T0_AD9P_35	5	6	IO_L4N_T0_35
IO_L5N_T0_AD9N_35	7	8	IO_L4P_T0_35
GND	9	10	GND
IO_L6P_T0_35	11	12	IO_L1N_T0_AD0N_35
IO_L6N_T0_VREF_35	13	14	IO_L1P_T0_AD0P_35
GND	15	16	GND
IO_L3P_T0_DQS_AD1P_35	17	18	IO_L2N_T0_AD8N_35
IO_L3N_T0_DQS_AD1N_351	19	20	IO_L2P_T0_AD8P_35
GND	21	22	GND
IO_L15P_T2_DQS_AD12P_35	23	24	IO_L18N_T2_AD13N_35
IO_L15N_T2_DQS_AD12N_35	25	26	IO_L18P_T2_AD13P_35
GND	27	28	GND
IO_L17P_T2_AD5P_35	29	30	IO_L16N_T2_35
IO_L17N_T2_AD5N_35	31	32	IO_L16P_T2_35
GND	33	34	GND
IO_L14P_T2_AD4P_SRCC_35	35	36	IO_L13N_T2_MRCC_35
IO_L14N_T2_AD4N_SRCC_35	37	38	IO_L13P_T2_MRCC_35
SMB_I2C_SCL0	39	40	SMB_I2C_SDA0

Figure B.1: FPGA Connector JA1

VIO_OUT (+3.3V)	1	2	JA2_SMB_nINT
VCCO_35	3	4	IO_25_35
IO_L22P_T3_AD7P_35	5	6	IO_L24N_T3_AD15N_35
IO_L22N_T3_AD7N_35	7	8	IO_L24P_T3_AD15P_35
GND	9	10	GND
IO_L23P_T3_35	11	12	IO_L19N_T3_VREF_35
IO_L23N_T3_35	13	14	IO_L19P_T3_35
GND	15	16	GND
IO_L21P_T3_DQS_AD14P_35	17	18	IO_L20N_T3_AD6N_35
IO_L21N_T3_DQS_AD14N_35	19	20	IO_L20P_T3_AD6P_35
GND	21	22	GND
IO_L9P_T1_DQS_AD3P_35	23	24	IO_L10N_T1_AD11N_35
IO_L9N_T1_DQS_AD3N_35	25	26	IO_L10P_T1_AD11P_35
GND	27	28	GND
IO_L8P_T2_AD10P_35	29	30	IO_L7N_T1_AD2N_35
IO_L8N_T2_AD10N_35	31	32	IO_L7P_T1_AD2P_35
GND	33	34	GND
IO_L11P_T1_SRCC_35	35	36	IO_L12N_T1_MRCC_35
IO_L11N_T1_SRCC_35	37	38	IO_L12P_T1_MRCC_35
SMB_I2C_SCL1	39	40	SMB_I2C_SDA1

Figure B.2: FPGA Connector JA2

VIO_OUT (+3.3V)	1	2	JB1_SMB_nINT
VCCO_34	3	4	IO_25_34
IO_L1P_T0_34	5	6	IO_L2N_T0_34
IO_L1N_T0_34	7	8	IO_L2P_T0_34
GND	9	10	GND
IO_L6P_T0_34	11	12	IO_L4N_T0_34
IO_L6N_T0_VREF_34	13	14	IO_L4P_T0_34
GND	15	16	GND
IO_L3P_T0_DQS_PUDC_B_34	17	18	IO_L5N_T0_34
IO_L3N_T0_DQS_34	19	20	IO_L5P_T0_34
GND	21	22	GND
IO_L9P_T1_DQS_34	23	24	IO_L7N_T1_34
IO_L9N_T1_DQS_34	25	26	IO_L7P_T1_34
GND	27	28	GND
IO_L8P_T1_34	29	30	IO_L10N_T1_34
IO_L8N_T1_34	31	32	IO_L10P_T1_34
GND	33	34	GND
IO_L11P_T1_SRCC_34	35	36	IO_L12N_T1_MRCC_34
IO_L11N_T1_SRCC_34	37	38	IO_L12P_T1_MRCC_34
SMB_I2C_SCL2	39	40	SMB_I2C_SDA2

Figure B.3: FPGA Connector JB1

Figure B.4: FPGA Connector JB2

VIO_OUT (+3.3V)	1	2	JB2_SMB_nINT
VCCO_34	3	4	IO_0_34
IO_L23P_T3_34	5	6	IO_L24N_T3_34
IO_L23N_T3_34	7	8	IO_L24P_T3_34
GND	9	10	GND
IO_L20P_T3_34	11	12	IO_L19N_T3_VREF_34
IO_L20N_T3_34	13	14	IO_L19P_T3_34
GND	15	16	GND
IO_L21P_T3_DQS_34	17	18	IO_L22N_T3_34
IO_L21N_T3_DQS_34	19	20	IO_L22P_T3_34
GND	21	22	GND
IO_L15P_T2_DQS_34	23	24	IO_L18N_T2_34
IO_L15N_T2_DQS_34	25	26	IO_L18P_T2_34
GND	27	28	GND
IO_L16P_T2_34	29	30	IO_L17N_T2_34
IO_L16N_T2_34	31	32	IO_L17P_T2_34
GND	33	34	GND
IO_L14P_T2_SRCC_34	35	36	IO_L13N_T2_MRCC_34
IO_L14N_T2_SRCC_34	37	38	IO_L13P_T2_MRCC_34
SMB_I2C_SCL3	39	40	SMB_I2C_SDA3

Figure B.5: FPGA Connector JC1

VIO_OUT (+3.3V)	1	2	JC1_SMB_nINT
VCCO_13	3	4	IO_L6N_T0_VREF_13
IO_L11P_T1_SRCC_13	5	6	IO_L12N_T1_MRCC_13
IO_L11N_T1_SRCC_13	7	8	IO_L12P_T1_MRCC_13
GND	9	10	GND
IO_L19P_T3_13	11	12	IO_L20N_T3_13
IO_L19N_T3_VREF_13	13	14	IO_L20P_T3_13
GND	15	16	GND
IO_L21P_T3_DQS_13	17	18	IO_L22N_T3_13
IO_L21N_T3_DQS_13	19	20	IO_L22P_T3_13
GND	21	22	GND
IO_L15P_T2_DQS_13	23	24	IO_L18N_T2_13
IO_L15N_T2_DQS_13	25	26	IO_L18P_T2_13
GND	27	28	GND
IO_L17P_T2_13	29	30	IO_L16N_T2_13
IO_L17N_T2_13	31	32	IO_L16P_T2_13
GND	33	34	GND
IO_L14P_T2_SRCC_13	35	36	IO_L13N_T2_MRCC_13
IO_L14N_T2_SRCC_13	37	38	IO_L13P_T2_MRCC_13
SMB_I2C_SCL4	39	40	SMB_I2C_SDA4

Processor Subsystem Connector

VCCO_MIO1_501	1	2	PS_MIO52_501
NC	3	4	PS_MIO53_501
PS_MIO16_501	5	6	PS_MIO19_501
PS_MIO17_501	7	8	PS_MIO18_501
GND	9	10	GND
PS_MIO20_501	11	12	PS_MIO23_501
PS_MIO21_501	13	14	PS_MIO22_501
GND	15	16	GND
PS_MIO24_501	17	18	PS_MIO27_501
PS_MIO25_501	19	20	PS_MIO26_501
GND	21	22	GND
PS_MIO28_501	23	24	PS_MIO31_501
PS_MIO29_501	25	26	PS_MIO30_501
GND	27	28	GND
PS_MIO32_501	29	30	PS_MIO35_501
PS_MIO33_501	31	32	PS_MIO34_501
GND	33	34	GND
PS_MIO36_501	35	36	PS_MIO39_501
PS_MIO37_501	37	38	PS_MIO38_501
SMB_I2C_SCL5	39	40	SMB_I2C_SDA5

Figure B.6: Processor Subsystem Multiplexed Input/Output (MIO) on J3

Power and Programming Connector

VP_0	1	2	VN_0
DAC_OUT1/ADC_IN4	3	4	DAC_OUT2/ADC_IN5
GND	5	6	GND
SMD_I2C_SCL7	7	8	SMB_I2C_SDA7
GND	9	10	GND
BT_AUD_CLK	11	12	BT_AUD_FSYNC
BT_AUD_OUT	13	14	BT_AUD_IN
MCU_SWCLK/PA13	15	16	MCU_SWDIO/PA14
MCU_nRST	17	18	MCU_VDDIO/VCCO_0 (+3.3V)
Zynq nRST (PS_SRST_B_501)	19	20	GND
TCK_0	21	22	TDI_0
TMS_0	23	24	TDO_0
MCU_VDDIO2/VCCO_0 (+3.3V)	25	26	PGND
GND	27	28	GND
VIN (+3.7-17V)	29	30	VIN (+3.7-17V)

Table B.1: Connector J2

C

wpa_supplicant.conf Network Configurations

This appendix contains a set of example network configurations for `wpa_supplicant.conf` with various security types. Common network security types have been chosen to be included here, however, they are not the only security schemes that are possible. The `wpa_supplicant` man page contains some additional examples and helpful resources for further reading.

WPA

Code Listing C.1: WPA *wpa_supplicant.conf* Network Configuration

```
network={
        auth_alg=OPEN
        key_mgmt=WPA-PSK
        psk="1234abcdWXYZ"
        ssid="MYNETWORK"
        mode=0
}
```

WPA2

Code Listing C.2: WPA2 *wpa_supplicant.conf* Network Configuration

```
network={
        auth_alg=OPEN
        key_mgmt=WPA-PSK
        psk '"1234abcdWXYX"
        ssid="my_Network"
        proto=RSN
        mode=0
}
```

WEP 40 Open

Code Listing C.3: WEP 40 Open *wpa_supplicant.conf* Network Configuration

```
network={
        auth_alg=OPEN
        wep_key0=1234567890
        key_mgmt=NONE
```

```
        ssid="my_Network"
        mode=0
}
```

WEP 128 Open

```
network={
        auth_alg=OPEN
        wep_key0=1234abcdWXYZ5678efghSTUV90
        key_mgmt=NONE
        ssid="my_Network"
        mode=0
}
```

Code Listing C.4: WEP 128 Open *wpa_supplicant.conf* Network Configuration

WPA PSK

```
network={
        auth_alg=OPEN
        key_mgmt=WPA-PSK
        psk="1234abcdWXYZ"
        pairwise=CCMP TKIP
        group=CCMP TKIP
        ssid="my_Network"
        mode=0
}
```

Code Listing C.5: WPA PSK *wpa_supplicant.conf* Network Configuration

D

wpa_cli Network Configurations

This appendix contains sets of commands used to configure network settings using `wpa_cli`. Each of the commands are variants of the `set_network` command and should be preceded by commands required to remove or add networks to the network list and followed by commands used to check the status of and finalize the connection, as described in the previous sections.

WPA

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-PSK
wpa_cli -i wlan0 set_network 0 psk '"12345678"'
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA2

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-PSK
wpa_cli -i wlan0 set_network 0 psk '"12345678"'
wpa_cli -i wlan0 set_network 0 proto RSN
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA/WPA2 PSK

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-PSK
wpa_cli -i wlan0 set_network 0 psk '"12345678"'
wpa_cli -i wlan0 set_network 0 pairwise CCMP TKIP
wpa_cli -i wlan0 set_network 0 group CCMP TKIP
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WEP 40 Open

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 wep_key0 1234567890
wpa_cli -i wlan0 set_network 0 key_mgmt NONE
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WEP 128 Open

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 wep_key0 1234abcdWXYZ5678efghSTUV90
wpa_cli -i wlan0 set_network 0 key_mgmt NONE
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA Enterprise (EAP TLS)

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
wpa_cli -i wlan0 set_network 0 pairwise TKIP
wpa_cli -i wlan0 set_network 0 group TKIP
wpa_cli -i wlan0 set_network 0 proto WPA
wpa_cli -i wlan0 set_network 0 eap TLS
wpa_cli -i wlan0 set_network 0 identity '"test"'
wpa_cli -i wlan0 set_network 0 client_cert '"/etc/certs/cert.pem"'
wpa_cli -i wlan0 set_network 0 private_key '"/etc/certs/key.pem"'
wpa_cli -i wlan0 set_network 0 private_key_passwd '"test"'
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA Enterprise (EAP PEAP0)

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
wpa_cli -i wlan0 set_network 0 pairwise TKIP
wpa_cli -i wlan0 set_network 0 group TKIP
wpa_cli -i wlan0 set_network 0 proto WPA
wpa_cli -i wlan0 set_network 0 eap PEAP
wpa_cli -i wlan0 set_network 0 identity "test"
wpa_cli -i wlan0 set_network 0 password "test"
wpa_cli -i wlan0 set_network 0 phase1 "peapver=0"
wpa_cli -i wlan0 set_network 0 phase2 "MSCHAPV2"
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA2 Enterprise (EAP TLS)

```
wpa_cli -i wlan0 set_network 0 proactive_key_caching 1
```

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
wpa_cli -i wlan0 set_network 0 pairwise CCMP
wpa_cli -i wlan0 set_network 0 group CCMP
wpa_cli -i wlan0 set_network 0 proto WPA2
wpa_cli -i wlan0 set_network 0 eap TLS
wpa_cli -i wlan0 set_network 0 identity '"test"'
wpa_cli -i wlan0 set_network 0 client_cert '"/etc/certs/cert.pem"'
wpa_cli -i wlan0 set_network 0 private_key '"/etc/certs/key.pem"'
wpa_cli -i wlan0 set_network 0 private_key_passwd '"test"'
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA2 Enterprise (EAP PEAP0)

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
wpa_cli -i wlan0 set_network 0 pairwise CCMP
wpa_cli -i wlan0 set_network 0 group CCMP
wpa_cli -i wlan0 set_network 0 proto WPA2
wpa_cli -i wlan0 set_network 0 eap PEAP
wpa_cli -i wlan0 set_network 0 identity '"test"'
wpa_cli -i wlan0 set_network 0 password '"test"'
wpa_cli -i wlan0 set_network 0 phase1 '"peapver=0"'
wpa_cli -i wlan0 set_network 0 phase2 '"MSCHAPV2"'
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

WPA/WPA2 Enterprise (EAP TLS)

```
wpa_cli -i wlan0 set_network 0 auth_alg OPEN
wpa_cli -i wlan0 set_network 0 key_mgmt WPA-EAP
wpa_cli -i wlan0 set_network 0 proto WPA2
wpa_cli -i wlan0 set_network 0 eap TLS
wpa_cli -i wlan0 set_network 0 identity '"test"'
wpa_cli -i wlan0 set_network 0 client_cert '"/etc/certs/cert.pem"'
wpa_cli -i wlan0 set_network 0 private_key '"/etc/certs/key.pem"'
wpa_cli -i wlan0 set_network 0 private_key_passwd '"test"'
wpa_cli -i wlan0 set_network 0 mode 0
wpa_cli -i wlan0 set_network 0 ssid '"my_Network"'
```

References

[1] Linaro, Ltd., Harston Mill, Royston Road, Harston CB22 7GG. *Devicetree Specification*, 0.1 edition, 2016.

[2] Xilinx, Inc. *Vivado Design Suite User Guide: Designing with IP*, November 2015.

[3] Xilinx, Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*, 1.10 edition, February 2015.

Index