Introduction to the OSD335x Reference Design Tutorial Series

The ***OSD335x family of System-In-Package (SiP) devices*** serves as a solid foundation to build advanced embedded systems quickly with reduced time to market. These SiPs tightly integrate Texas Instrument (TI)'s AM335x ARM® Cortex® A8 CPU, TPS65217C Power Management IC (PMIC), TL5209 LDO, up to 1GB of DDR3 and all associated passives. All of which is contained in a US quarter sized package as shown in Figure 1.

At Octavo Systems, we are committed to simplifying your design as much as possible so that you can quickly start designing and building your dream products. With this intent in mind, we have developed the ***OSD335x Reference Design Tutorial Series***, which will walk you through the OSD335x design process in a systematic manner.

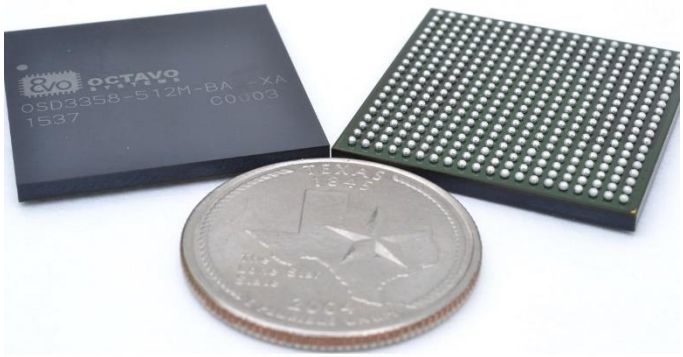


Figure 1 OSD335x in comparison with US quarter

The tutorial series is broken down into several lessons. Each lesson will address specific concepts and build upon the concepts of previous lessons. Each lesson will conclude with a Printed Circuit Board (PCB), which will implement and verify all of the concepts taught. The lessons will begin with the basics and then take you through advanced concepts.

Each lesson will cover:

Lesson 1: You will learn how to build the bare minimum circuitry required to boot the OSD335x without an operating system and all the concepts related to it. Lesson 1 tutorials are found in chapters 1-11.

Lesson 2: You will learn how to build the minimum circuitry required to Boot Linux on the OSD335x and all the concepts related to it. Lesson 2 tutorials are found in chapters 12-18.

Both Lesson 1 and Lesson 2 Tutorials can be applied easily to the OSD335x-SM device by following the additional information provided on the ***OSD335x-SM Design Tutorial webpage***.

Table of Contents

Revision History

Revision Number	Revision Date	Changes	Author
1	8/6/2017	Initial Release	Eshtaartha Basu
2	1/2/2018	LED series resistors (R10, R32 and R33) updated from 4.7k to 1k, Open - drain buffer TRM reference added to Reset Article	Eshtaartha Basu
3	6/11/2018	Added Lesson 2 Tutorial sections	Eshtaartha Basu
4	7/7/2018	Updated device tree figure 118 Ch 17	Eshtaartha Basu
5	11/14/2018	Updated Booting Linux 15.5 & UART Caveat in section 16.2.1	Eshtaartha Basu
6	12/18/2018	Replaced references to the XDS100v2 with the XDS110	Greg Sheridan
7	2/9/2019	Update to OSD335x Bare Minimum Board Boot Process section – Add JTAG programming step	Eshtaartha Basu
8	4/2/2019	Updated Non-Volatile Storage information (Ch 14)	Erik Welsh
9	4/11/2019	Updated steps to compile device tree in section 17.3	Eshtaartha Basu

1 Before You Begin

1.1 Fundamentals of the OSD335x

There are many parts within the OSD335x family. We will be focusing on the OSD3358-512M-BAS in this series. Therefore, all references to the OSD335x hereafter imply OSD3358-512M-BAS.

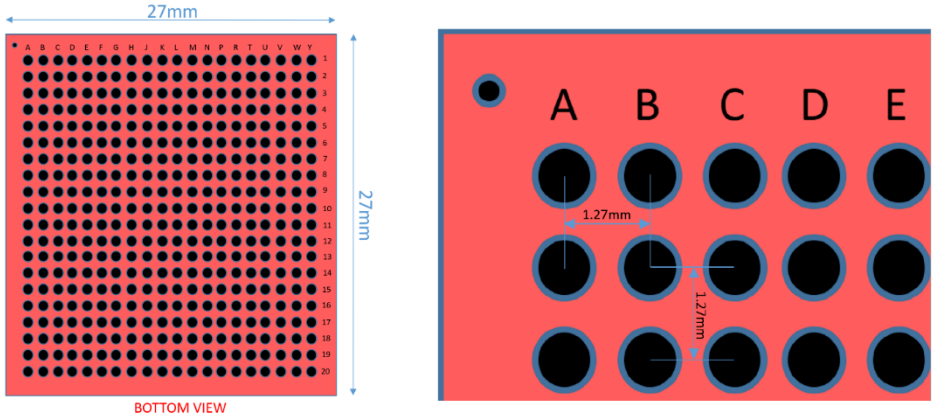


Figure 2 OSD335x BGA package

The OSD335x comes in a 27mm x 27mm Ball Grid Array (BGA) package with 400 balls and 1.27mm ball pitch as shown in Figure 2.

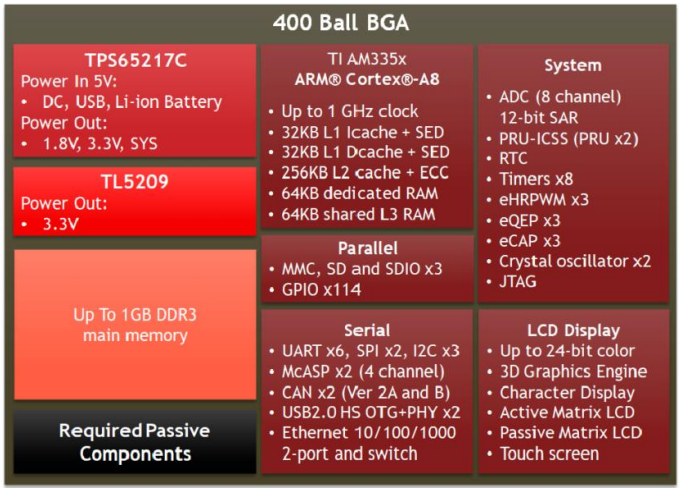


Figure 3 OSD335x Functional Diagram

It consists of four main components as shown in Figure 3. They are:

- Sitara® AM335x ARM® Cortex® A8 processor from TI - is the heart of the OSD335x.
- TPS65217 Power Management IC (PMIC) from TI - manages power distribution to various parts of the OSD335x and provides power to external devices.

- TL5209 Low Drop Out (LDO) regulator from TI - provides dedicated 3.3V rail to power external components of the OSD335x.
- Up to 1GB of DDR3 - is the main memory of the OSD335x.

All of the peripheral interfaces of the AM335x (except the DDR interface) are brought out to pins on the OSD335x. See the ***OSD335x datasheet*** for more information on the pinout.

1.2 Lesson organization

Each lesson will consist of an ***Introduction*** section which describes the objective of the lesson followed by a series of articles, each of which will walk you through the design methodology. The lesson will conclude with an ***Expected outcome*** section which describes how the outcome of that lesson will look.

Furthermore, each lesson may have several ***Perks*** and ***Caveats***. The Perks will give additional information about the topic in discussion and the Caveats will warn you about conditions or situations which require more attention. The Perks and Caveats look like this:

> Perk:
>
> Perks will give you additional information about the topic in discussion.

> Caveat:
>
> Caveats will warn you about conditions or situations which require more attention.

Before starting lesson 1, we strongly encourage you to go through the OSD335x datasheet which can be found ***here***.

1.3 Pre-requisites

We will be using Autodesk Eagle for schematics and layout. Hence, we assume that you are familiar with Eagle schematics and layout. If not, please learn about them first. The following website is one of the many websites that can help you:

To learn more about schematics: ***https://learn.sparkfun.com/tutorials/using-eagle-schematic***

To learn more about layout: ***https://learn.sparkfun.com/tutorials/using-eagle-board-layout***

****

2 Introduction to Bare Minimum Circuitry to Boot OSD335x

2.1 Introduction

The objective of this lesson is to help you become familiar with the bare minimum setup required to boot the OSD335x and getting it ready to execute software. This lesson will consist of a series of articles which will walk you through every step of the design process. We start from specifications and guide you through every step till debugging the manufactured Printed Circuit Board (PCB). The lesson will conclude with a PCB that verifies the design by putting together everything that was taught.

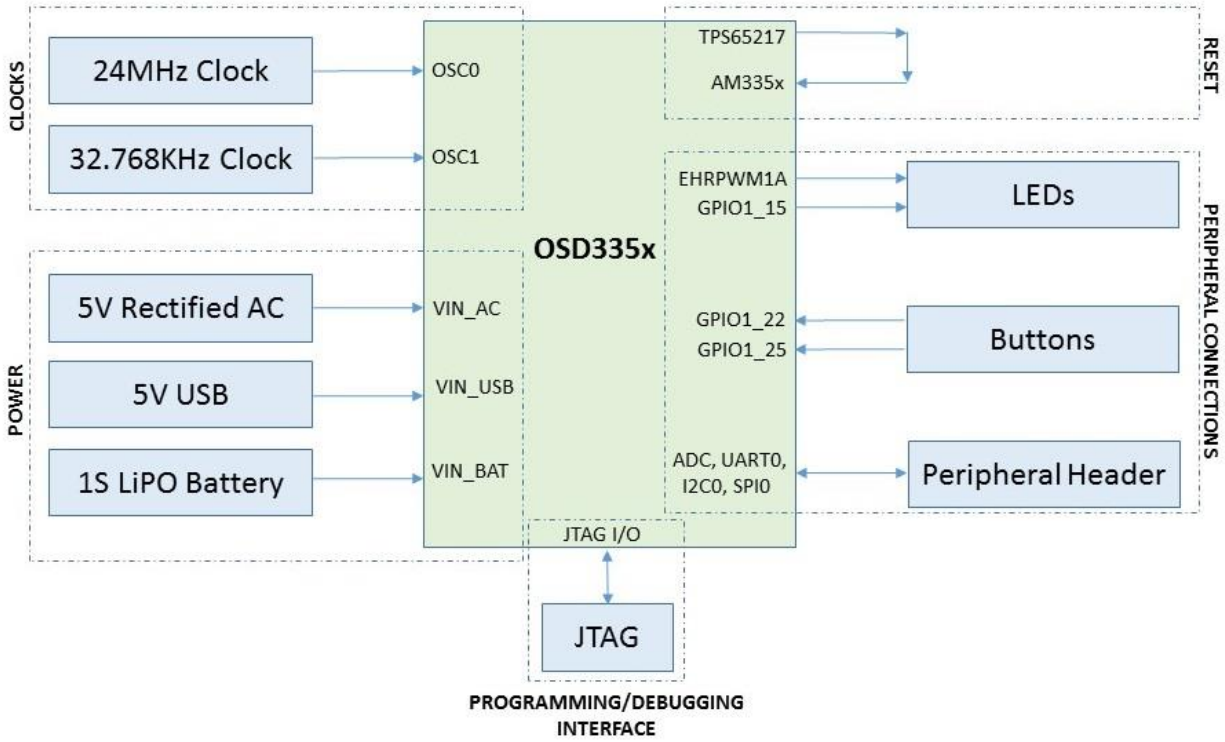


Figure 4 OSD335x Lesson 1 Block Diagram

To boot the OSD335x in its minimal configuration, we will need:

- Power circuitry
- Clock circuitry
- Reset circuitry
- Debugging/Programming interface (JTAG interface)

To make our design more functional, we will add a couple of buttons, LEDs and a peripheral header to allow us to better test our completed design.

A block diagram of this setup is shown in Figure 4. We will discuss each of these concerns in separate articles to have a better depth of understanding of each topic.

2.2 CAD Environment Setup

Before proceeding with the articles, let's understand the Autodesk Eagle design environment we will be using.

For this design, we are going to create a PCB with the following parameters:

- Board Size: 2500mil x 2000mil (2.5inch x 2inch)
- Number of layers: 4 layers.
- Trace width: 6mil (approx. 0.15mm). Since power traces generally carry more current, we will be using larger traces (at least 15mil or 0.4mm) for them.
- Trace spacing: 6mil.
- Minimum drill and via size: 12mil (approx. 0.30mm) drill and 24mil (approx. 0.60mm) finished via diameter (i.e., 6 mil annular ring).

Using these standard rules will help us reduce manufacturing cost. For your design, you are free to select the appropriate rules for your manufacturer and components that suit your design.

All design files for this lesson can be downloaded ***here***.

2.2.1 Library Setup

Octavo Systems provides an Eagle library, ***OSD3358_BAS_RefDesignParts.lbr***, that contains the schematic symbol and footprint for the device. This reference library can be downloaded ***here***.

- It is good practice to create your own Eagle Library for each design and copy all parts used in the design into it. For this design, we have created the library ***OSD3358_BAS_RefDesignParts.lbr***.
- Setup the Eagle library path to the location of the Octavo Systems library file and copy the device ***OSD3358-512M-BAS*** into your new library.

2.2.2 Schematic setup

- Open a new schematic file and name it appropriately.
- Make sure to use the ***OSD3358_BAS_RefDesignParts.lbr*** library in the schematics.
- From the library, add the ***OSD3358-512M-BAS*** symbols to the schematic as shown in Figure 5. The ***OSD3358-512M-BAS*** symbol will easily fit into A3 size sheets. In the design files, all four A3 ***sheets*** are consolidated into a single

Eagle sheet so that we can use the free version of the tool while still allowing the schematics to be easily viewable while printing.

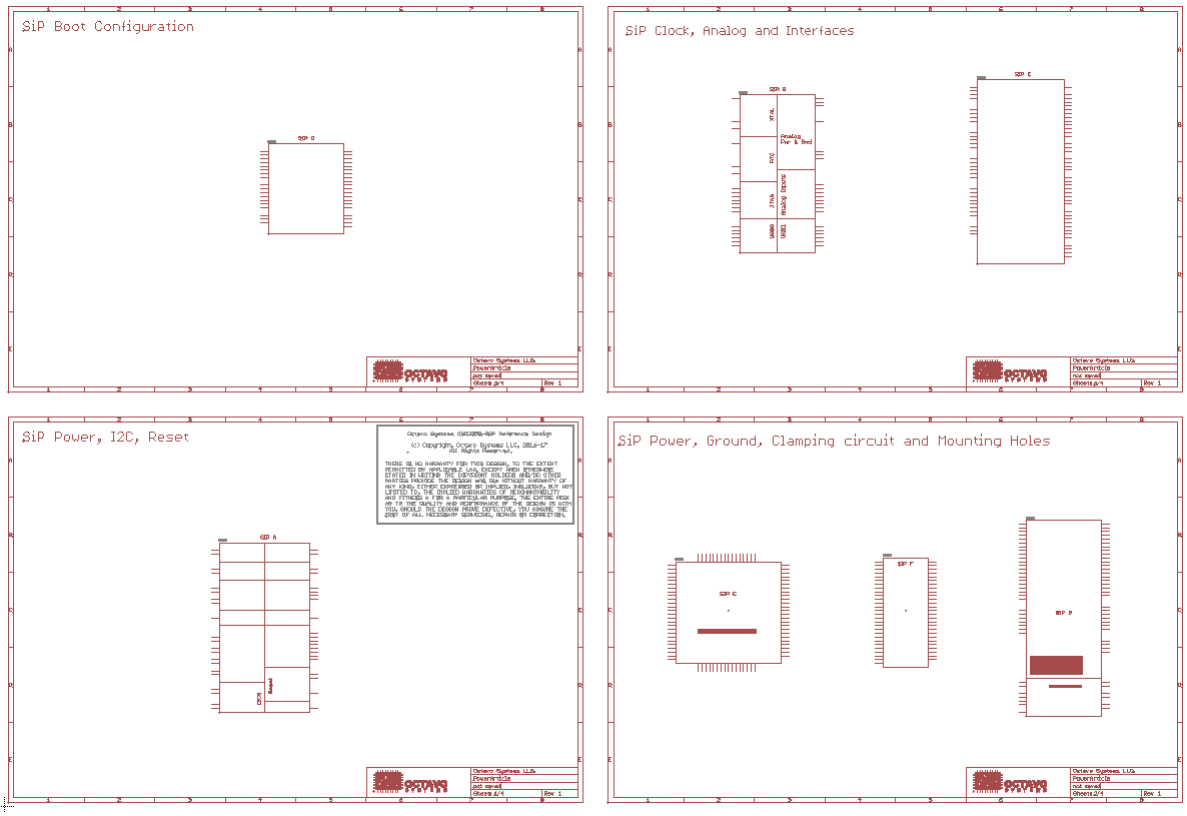


Figure 5 OSD335x symbol arrangement on schematic

2.2.3 Layout (Board) Setup:

- The layer stack up is shown in Figure 6. We'll be using ***Top*** and ***Bottom*** layers for signal routing. ***Route2*** layer acts as power plane (We will be connecting it to an appropriate power output pin of the OSD335x later). ***Route15*** layer acts as ground plane.

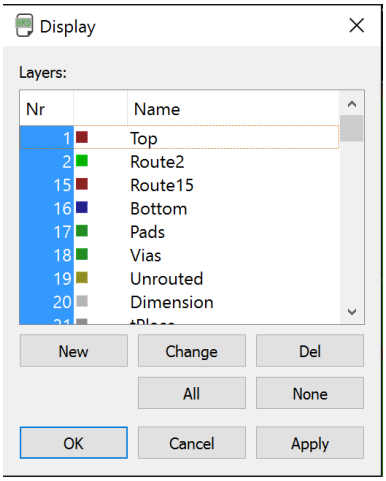


Figure 6 Layout Layer stack up

- Draw the outline of the board and place the OSD3358-512M-BAS footprint as shown in Figure 7.

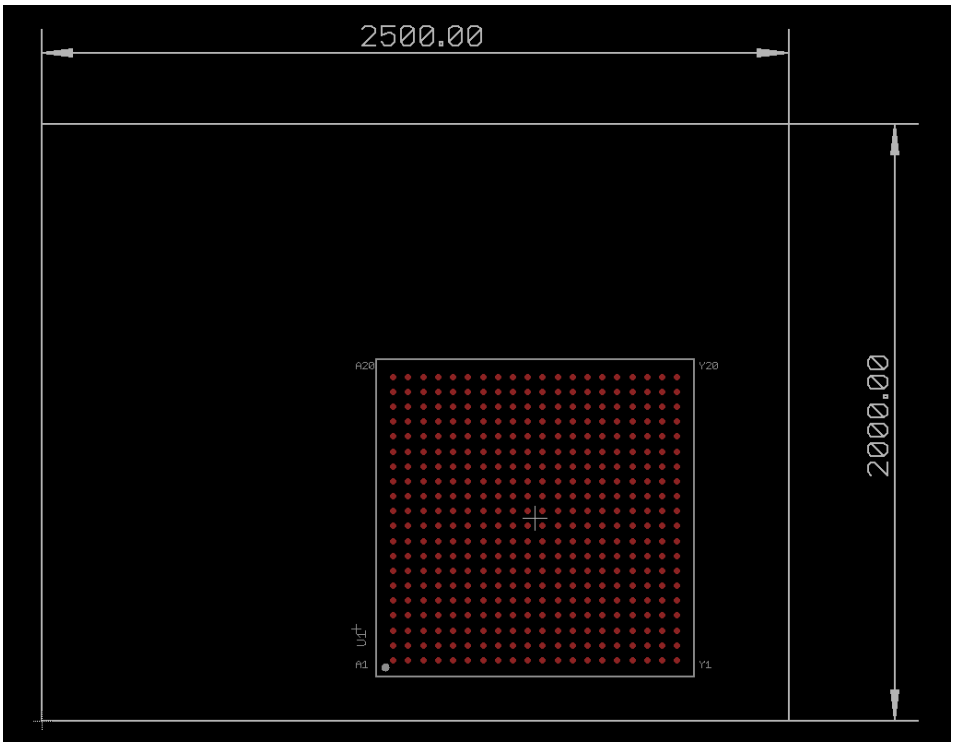


Figure 7 OSD335x Layout

2.3 OSD3358-512M-BAS pin distribution

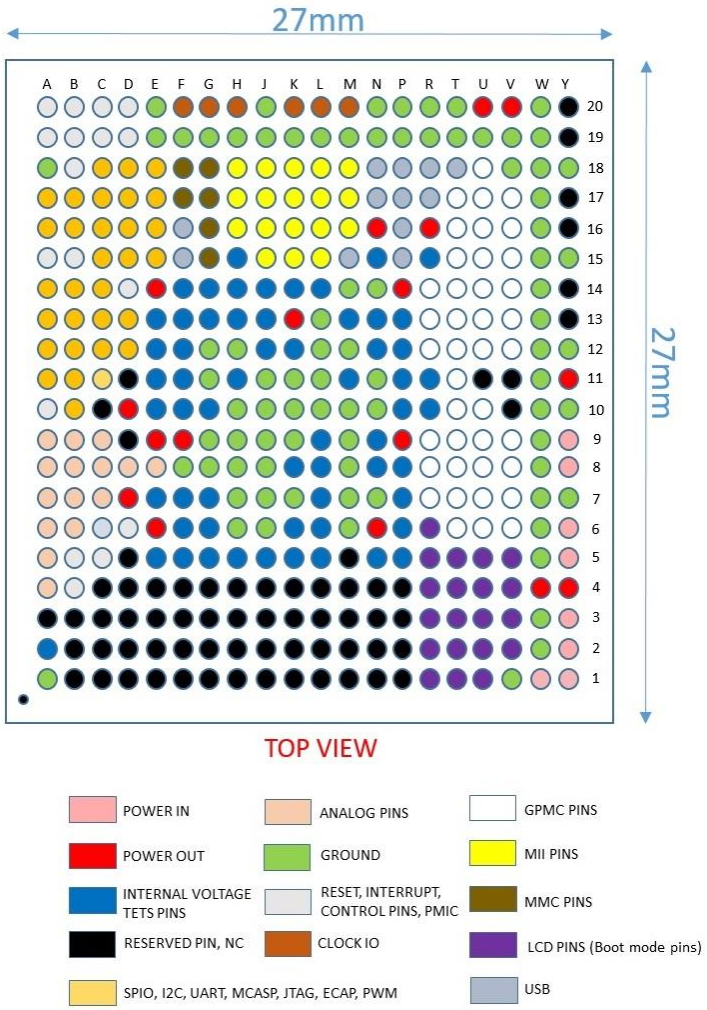


Figure 8 OSD335x BGA pin arrangement

Figure 8 gives visual representation of the OSD335x BGA pin arrangement. This will help us plan the placement of the SiP on our board.

*****

3 OSD335x Power Inputs and Outputs

3.1 Introduction

When it comes to booting a device, power circuitry is the first thing that comes to mind. To have stable and predictable operation from any electronic device, we need to make sure the power circuitry is well designed and is able to meet the power requirements of the device under normal as well as extreme operating conditions.

The Power Circuitry is a complex topic, so we have split the discussion into five parts. They are:

- **Power Inputs and Outputs**: This article will focus on input and output power considerations for the OSD335x.
- **Ground Connections**: This article will focus on ground connections and important layout pour considerations for the OSD335x.
- **Power Management**: This article will focus on topics related to control and management of power for the OSD335x.
- **Clamping Circuit**: This article will focus on clamping circuit which can be used to prevent certain power down issues that may arise with the OSD335x.
- **ESD Protection**: This article will focus on steps needed to provide Electro Static Discharge (ESD) protection to the OSD335x based Printed Circuit Boards (PCB).

This article is the first part of the OSD335x Reference Design Lesson 1 Power circuitry articles. We recommend you to read other parts of power circuitry article as well which are available further ahead in this article. As we discuss the power circuitry, we will build the schematic and layout the corresponding traces. All the components used in this article can be found in the ***provided library***. The components will be introduced after relevant discussion as we proceed with the article.

3.2 Power Input

The OSD335x can be powered from 5V DC power supply (generally sourced from an AC adapter), from a standard USB port (5V) or using a standard single cell (1S) Li-Ion/Li-Polymer (LiPo) battery.

3.2.1 VIN_AC

While this port is called VIN_AC, it is a DC power input generally powered from a 5V AC adapter. By default, this input has 2A current limit and can be used as primary power input. For our design, we will use a DC barrel connector to source this input since DC barrel connectors are used by many other electronic devices and adapters

compatible with this type of connector are easily available. The DC barrel connector device we're using can be found under the name ***PJ-102B_POWER_CON*** in the provided library. Let's connect this pin directly to the input pins Y5 and Y6 of the OSD335x.

> Caveat:
>
> While it is ok for us to connect the DC barrel connector directly to the OSD335x for this minimal design, you should add appropriate power input protection, such a ferrite beads, diodes and fuses, based on the needs of your application.

3.2.2 VIN_USB

This input of the OSD335x can be powered from the VBUS line of the USB client connector at 5V. By default, the input current limit of this pin is 500mA which is also the standard output current limit for a USB 2.0 host port. However, through software configuration of the power management IC (PMIC) inside the OSD335x, the current limit can be raised to 1.3A.

The USB client connector used in this design can be found under the name ***10118192-0001LF*** in the provided library. The VBUS pin of this connector should be connected to the pins Y8 and Y9 on the OSD335x.

> Caveat:
>
> Depending on your application, 500mA input current may not be sufficient. If you plan to increase the input current limit of this input through software, make sure the USB host can source the required additional current. For example, USB 3.0 host port can supply up to 0.9A output current at 5V.

3.2.3 VIN_BAT

VIN_BAT pin can act as either a battery input or output. It acts as a battery input when the OSD335x is running on battery power. It acts as a battery output when the OSD335x is charging the connected battery (more information on charging given in the below perk). This input should be powered by a single cell (1S) Li-Ion or Li-polymer battery with voltage range of 2.75V to 5.5V.

In this lesson, we will primarily use VIN_AC and VIN_USB to source power to the OSD335x. How-ever, in the future we may want to use a (1S) Li-ion or Li-Polymer

Caveat:

The OSD335x does not use VIN_BAT as an input event to power up the device from OFF state or SLEEP state. More information about this can be found under section 9.3.1.1 in the ***TPS65217 datasheet***.

Perk:

The OSD335x is also capable of charging Li-ion and Li-Po batteries using its built in linear charger. It has dedicated pins to monitor battery voltage (BAT_VOLT) and monitor battery temperature (BAT_TEMP). The VIN_BAT pin provides battery output while charging. Battery charging is managed by the TPS65217 PMIC present inside the OSD335x. More information about battery charging can be found in the ***TPS65217x Datasheet***.

battery to power our design. Hence, we will add thru-hole test points for battery power inputs so that we can connect a battery later if necessary. Thru-hole test points can be found under the name ***TESTPAD/W_HOLE_1X1*** in the provided library.

3.3 Input Power Schematics

Based on the description above, let's update our schematics with all the input power connections as shown in Figure 9 (Updates made to the schematics are shown using dotted lines).

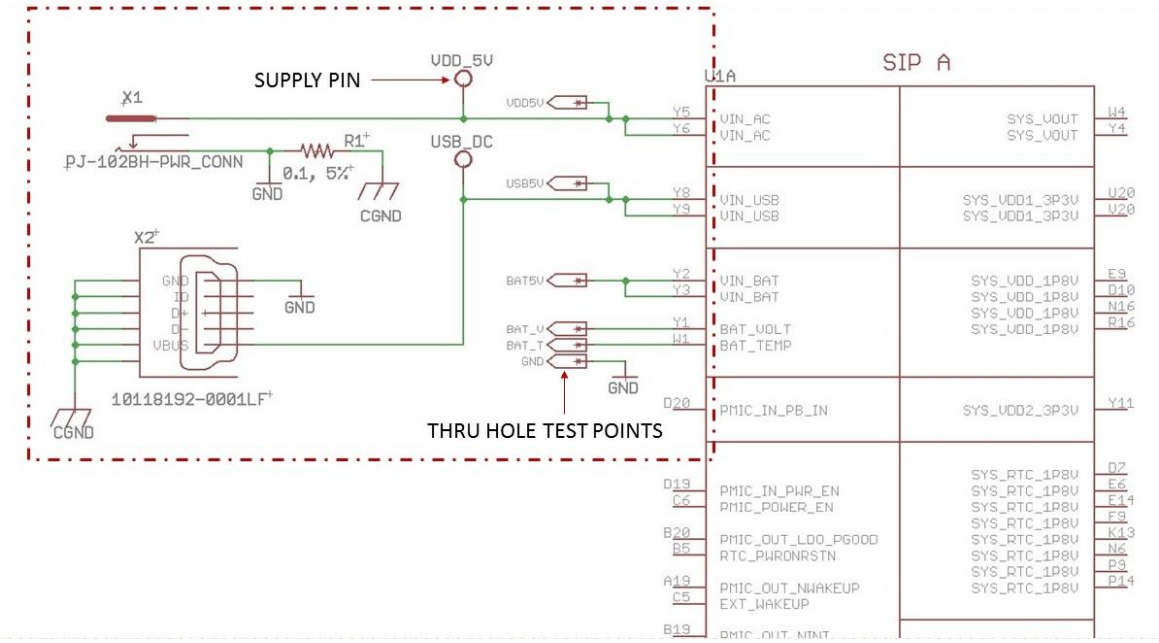


Figure 9 Power input connections

It is a good idea to add test points to all the input power rails so that we can easily test the voltages during debug. For this design, we have used thru-hole test points for all the input power rails. But, for your design, feel free to use surface test pads to save board area.

Supply pins are also added to input power rails so that we can see where these voltages are used elsewhere on the schematic.

The reason behind the presence of resistor R1 is explained under the ESD (Electro Static Discharge) protection chapter (chapter 6) of this document.

3.4 Input Power Layout

Now that we have completed the schematics for power input, let's begin the layout. As per the guidelines in the ***introduction article***, we will use 6mil (approx. 0.15mm) traces for signals and at least 15mil (approx. 0.40mm) traces for power traces and connect them to the OSD335x power pins using pours so that there is good connection to the BGA balls. While we could use pours for the entire power connection, in this design we do not have any peripherals with high current draw so we can use traces to make layout easier. The layout is shown in Figure 10.

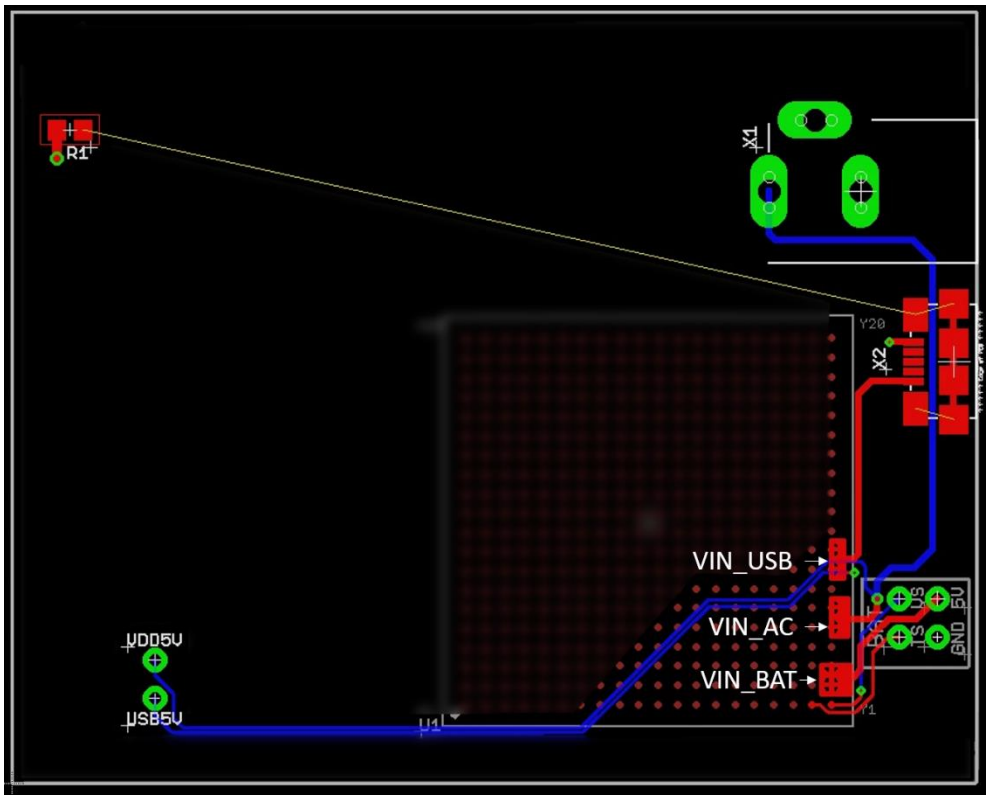


Figure 10 Power input layout

The components are placed and routed in a specific manner to accommodate future components and to facilitate easy routing as we go ahead with the articles.

3.5 Power output

The PMIC and LDO inside the OSD335x generate many different power outputs. Some of the outputs are only for internal use within the System in Package (SiP). However, others provide power that can be used by the systems external to the SiP. Please go through the ***power app note*** before budgeting power for your design. The maximum power output of each of these pins can be found in the ***OSD335x datasheet***. The following power outputs can be used for external devices:

- **SYS_VOUT:** Shared supply sourced by the PMIC. This rail also supplies power to the AM335x, DDR3 and TL5209 LDO inside the SiP. This output is not regulated. It merely reflects the voltage of the input power source that is being used to power the PMIC. Therefore, when using a battery, it is necessary to make sure that any components that use the SYS_VOUT power output can operate on a voltage between 3V and 5V since the PMIC will switch to a different power input when charging the battery.
- **SYS_VDD1_3P3V:** Dedicated 3.3V supply rail for external circuitry. Powered by the TL5209 LDO and enabled by LDO4 of the PMIC. This power output will be connected to the power plane of our layout.

- **SYS_VDD2_3P3V:** Dedicated 3.3V supply for external circuitry directly supplied by LDO2 of the PMIC.
- **SYS_RTC_1P8V:** 1.8V output powered by LDO1 of the PMIC. It is also used internally to power the RTC of the AM335x.
- **SYS_VDD_1P8V:** 1.8V output powered by LDO3 of the PMIC.
- **SYS_ADC_1P8V:** 1.8V output powered by LDO3 of the PMIC and filtered for analog applications. It supplies power to the AM335x ADC. It can also be used to power external analog circuitry.

It is a good idea to add supply pins and test points to all the power output pins as shown in Figure 11 so that we can probe the voltages during debug and board bring-up.

3.6 Test points on internal power rails

The OSD335x provides external access to critical internal power rails. These pins should be used for testing/monitoring purposes only. ***They shouldn't be used to power external circuitry***. Test points need to be added to these power rails so that internal voltages can be looked up in case of power-up issues. You can use either thru-hole test points or test pads, whichever makes your routing easier. For this design, we will be using thru-hole test points.

The OSD335x pins that provide access to internal power rails are **VDDSHV_3P3V, VDDS_DDR, VDD_MPU, VDD_CORE and VDDS_PLL.**

Perk:

If you're curious about how the internal power rails are used within the SiP, you can find more information below:

- **VDDSHV_3P3V:** Dedicated 3.3VDC to power the AM335x I/O. It is supplied by the TPS65217 LDO4.
- **VDDS_DDR:** Dedicated 1.5VDC supply to power the AM335x DDR3 interface and DDR3 device.
- **VDD_MPU:** Dedicated 1.1VDC supply to power the AM335x MPU domain.
- **VDD_CORE:** Dedicated 1.1VDC supply to power the AM335x CORE domain.
- **VDDS_PLL:** Filtered 1.8VDC to supply power to the AM335x PLLs and oscillators.

You can also refer ***Texas Instruments (TI) Power Hookup Application Note*** for in depth information.

3.7 Schematics for power output pins and test points on internal power rails

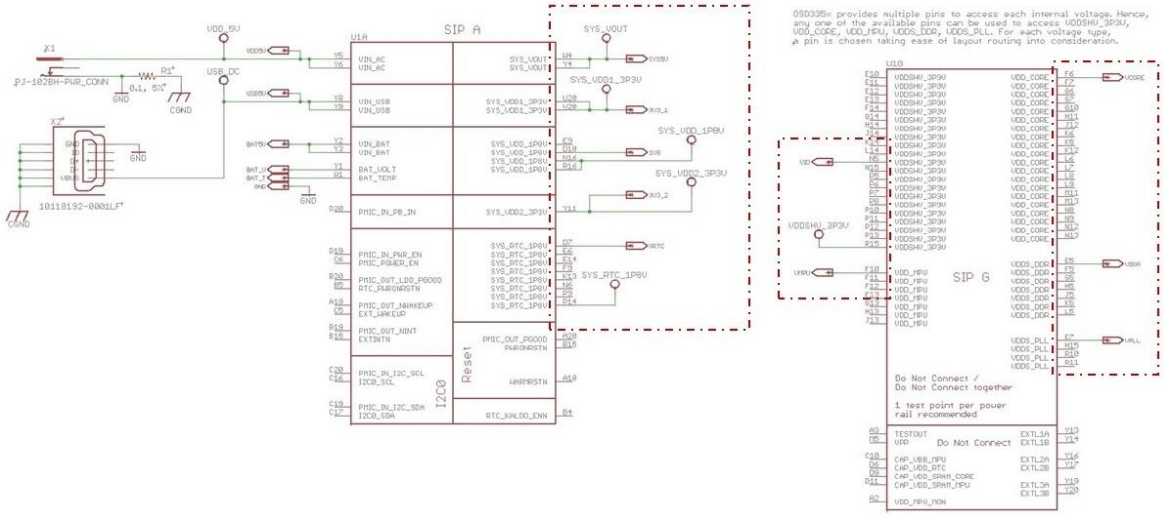


Figure 11 OSD335x schematic for test points on internal power rails and power output pins

Let's add test points and supply pins suitably as shown in Figure 11 (Updates made to the schematics are shown using dotted lines). Thru-hole test points can be found under the device name ***TESTPAD/W_HOLE_1X1*** in the provided library.

3.8 Layout for test points on internal power rails

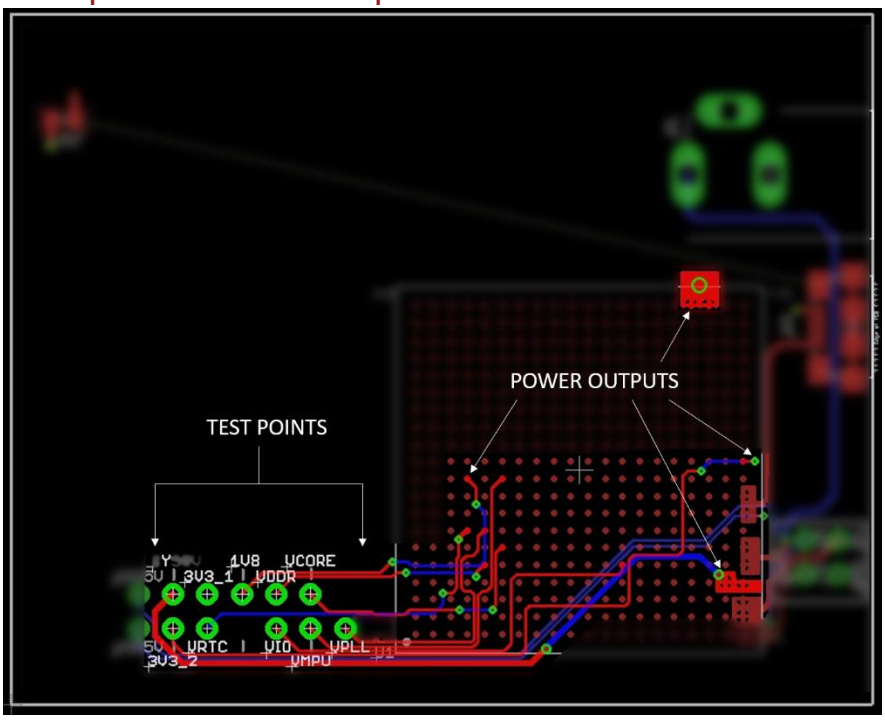


Figure 12 OSD335x layout for power output pins and test points on internal power rails

The test points were placed and routed to accommodate future components and facilitate easy routing as shown in Figure 12.

3.9 Analog reference input and ground

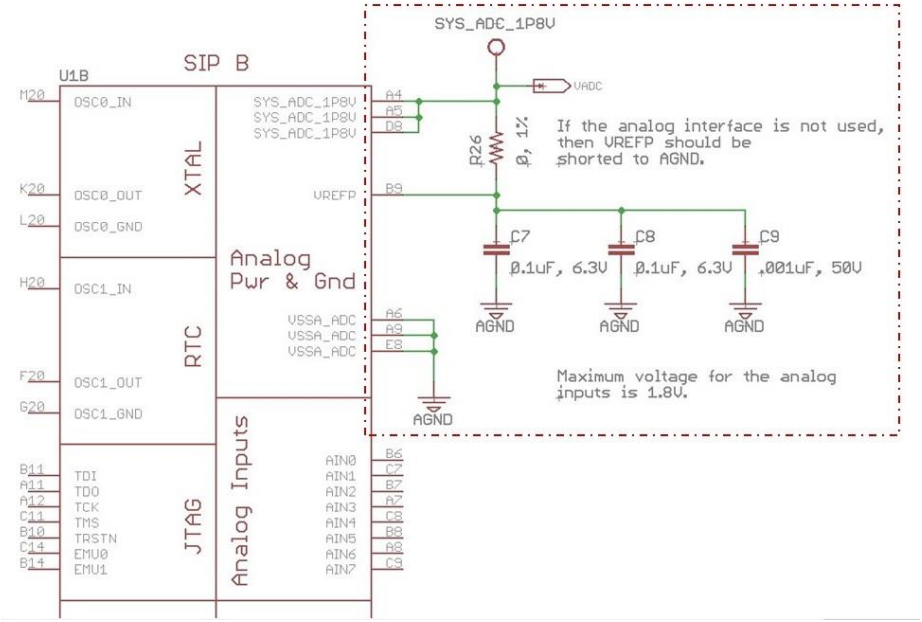


Figure 13 Schematic for Analog Reference input and Ground

The OSD335x has an Analog-to-Digital Converter (ADC) interface that can be used for things like monitoring voltages and interfacing with analog sensors. To use the ADCs, the Analog Power and Ground must be connected appropriately. The interface can tolerate inputs up to 1.8V depending on the analog voltage reference VREFP. Internally, the OSD335x connects the VREFN pin of the AM335x to analog ground, so the range of ADC is analog ground to VREFP. Generally, VREFP is connected to SYS_ADC_1P8V but it can be set to a lower voltage using a voltage divider.

Since the voltage reference, VREFP, needs to be a clean as possible, we want to put a resistor footprint between VREFP and the power connection. This gives the option of putting a resistor or ferrite bead in at a latter place if we need to suppress noise. Also, we want to put bypass capacitors between VREFP and analog ground to help suppress noise. These connections can be seen in Figure 13 (Updates made to the schematics are shown using dotted lines).

If you do not need to use the ADC interface in your application, then VREFP should be shorted with AGND.

Layout for Analog Connections can be made as shown in Figure 14.

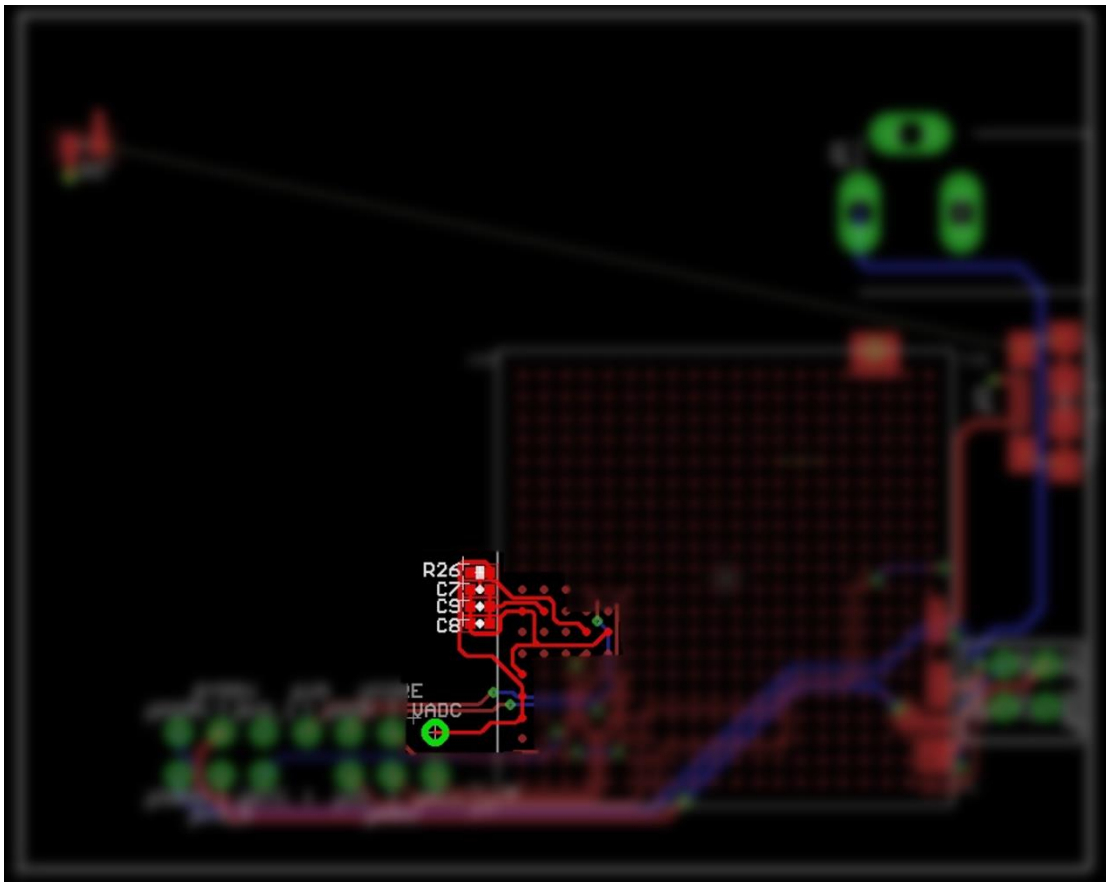


Figure 14 Layout for Analog Connections

*****

4 OSD335x Ground Connections

4.1 Introduction

This article is the second part of the OSD335x Reference Design Lesson1 Power circuitry articles. It will focus on ground connections and pours required for reliable performance from the OSD335x. As we discuss the power circuitry, we will build the schematic and layout the corresponding traces.

4.2 Ground Connections

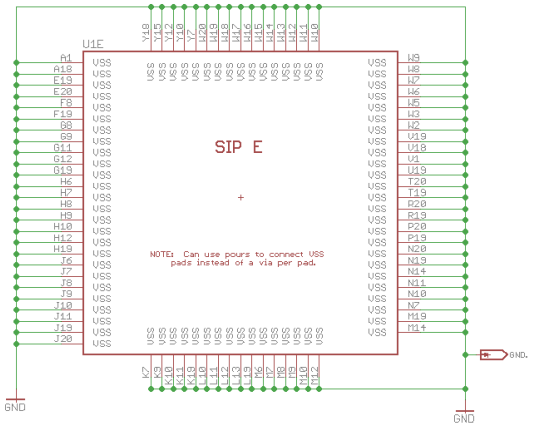


Figure 15 OSD335x Ground connections

The OSD335x has several ground pins. All the pins on the ***SIP E*** symbol should be grounded, as shown above in Figure 15, even though they are connected together within the SiP. This is to make sure all components within the OSD335x are uniformly grounded and ensure the shortest return current paths for all the components inside the OSD335x. For our reference design, a thru-hole test point was added to help us measure voltage during bring-up.

It is good to have test points to ground on the design for testing during bring-up, but it is not necessary as long as there is access to ground somewhere in the design. In compact designs, you can use surface test points to save space.

4.3 Ground pour layout discussion:

When grounding the OSD335x, you are free to use one via for each ground pin. However, vias occupy a lot of routable space and a lot of vias can cause routing problems (this does not necessarily apply to blind and buried vias since they occupy much less route-able board area). Routable space can be saved by instead using ground pours and minimizing the number of ground vias. For this design, since the ground pins are clustered in two areas, we have placed two copper pours on the top layer for ground. Then we have at least one via per two ground pins so there is a good connection to the ground plane. When using copper pours, the following things should be considered:

- Have at least one via for every two to four ground pins.
- Vias should be placed wisely keeping current return path also in mind.
- Care should be taken not to flood the entire area below the BGA of the OSD335x with a ground pour. If the area of the ground pour is large, it may sink too much heat from the BGA and may result in bad solder joints during reflow. Similarly, make sure thermals are used for the pours to prevent the pours from sinking too much heat and causing bad solder joints.

Perk:

A ground pour is nothing but a copper pour connected to GND or GND plane of the PCB. More information about copper pours can be found ***here***.

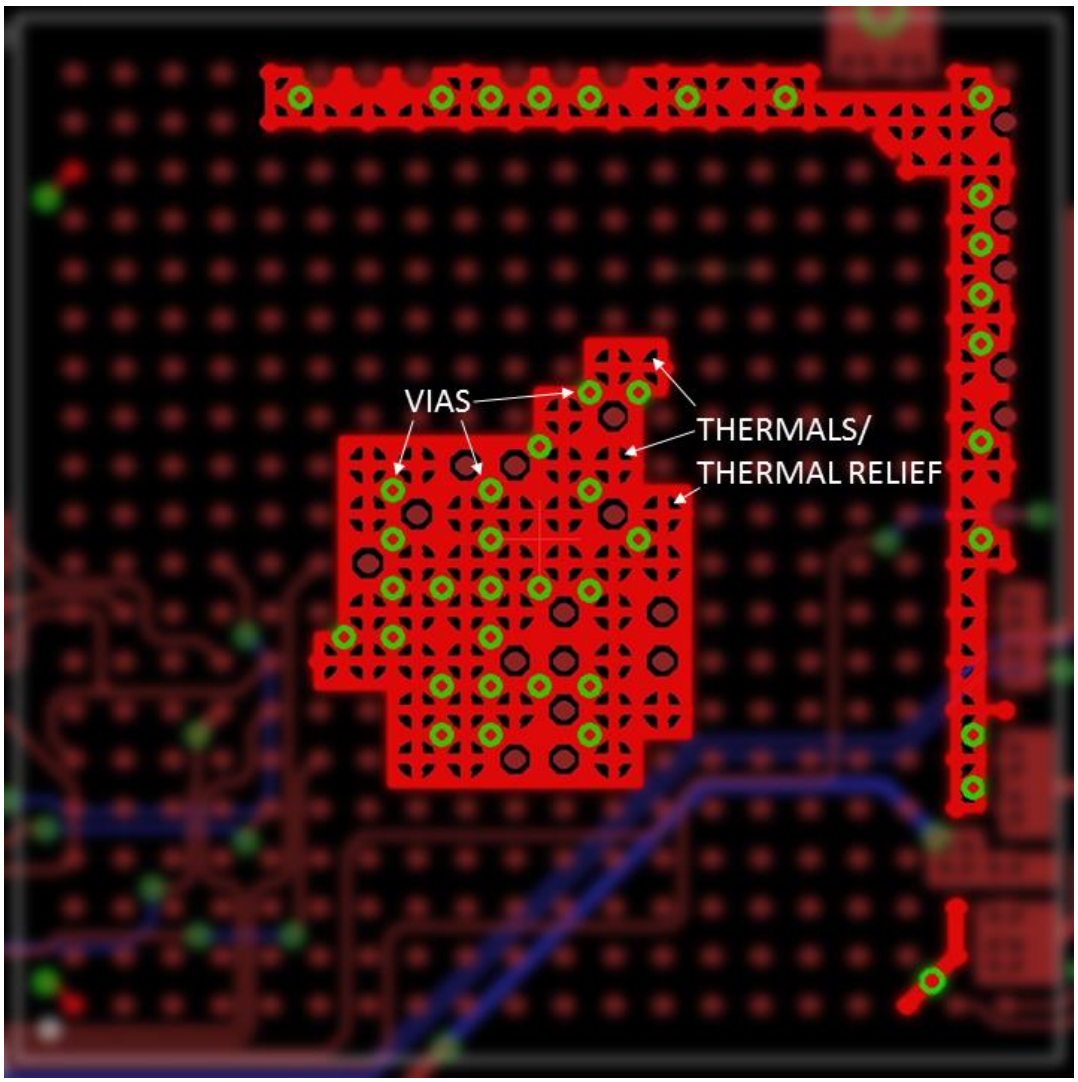


Figure 16 OSD335x Ground Pour

As shown in Figure 16, we have used two ground pours for this design to minimize the size of each pour. Each pour uses thermals (thermal relief) and there are plenty of vias to ensure good connection to ground.

Perk:

A ***thermal (thermal relief)*** in PCB jargon refers to a particular way in which a pad is connected to a pour/plane to make good electrical connection but poor thermal connection. Copper is a good conductor of heat. Hence, if good thermal connection were to be made, the plane/pour would draw all the heat away from the pad leading to a bad solder joint. More information about thermals and their uses can be found ***here***.

4.4 Power and Ground Planes

As we already stated, our design uses a 4 layer PCB. The top and the bottom layers are used for signal routing whereas the layer beneath the top layer is used as a power plane and the layer below the power plane is used as a ground plane. Use of power and ground planes that are closely spaced to each other reduces crosstalk and interference between the top and bottom signal layers.

Most of the components in this design will operate on 3.3V. Hence, we will be connecting SYS_VDD_3P3V to the power plane. The power plane will help those components directly receive power through a via with minimal resistance since the power plane is spread across the board.

The ground plane is used to connect all components to a common ground reference. It will:

- provide a low resistance return path for currents to reduce noise.
- will prevent current loops.
- will act as EMI shield protecting the PCB from external noise and prevent radiation of high frequency noise from the PCB.
- provide uniform impedance plane for traces carrying high frequency signals with high fidelity against reflections.

Power and Ground planes can be created in the layout by drawing one complete polygon along the edge of the board in layer 2 and connecting it to SYS_VDD_3P3V and drawing the other complete polygon in Layer 15 and connecting it to GND using the ***NAME*** tool of Eagle. The POWER and GND polygons are shown in Figure 17. When the ***RATSNEST*** button is pressed on the layout, the power and ground planes occupy the entire area of the board as shown in Figure 18 and Figure 19.

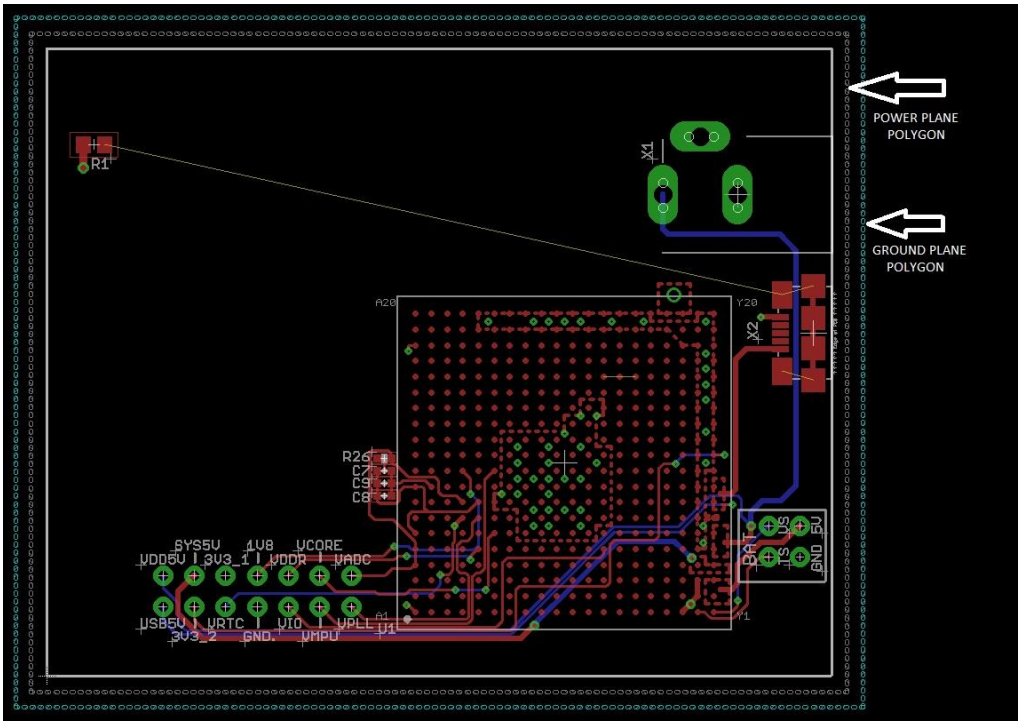


Figure 17 Power and Ground plane polygons

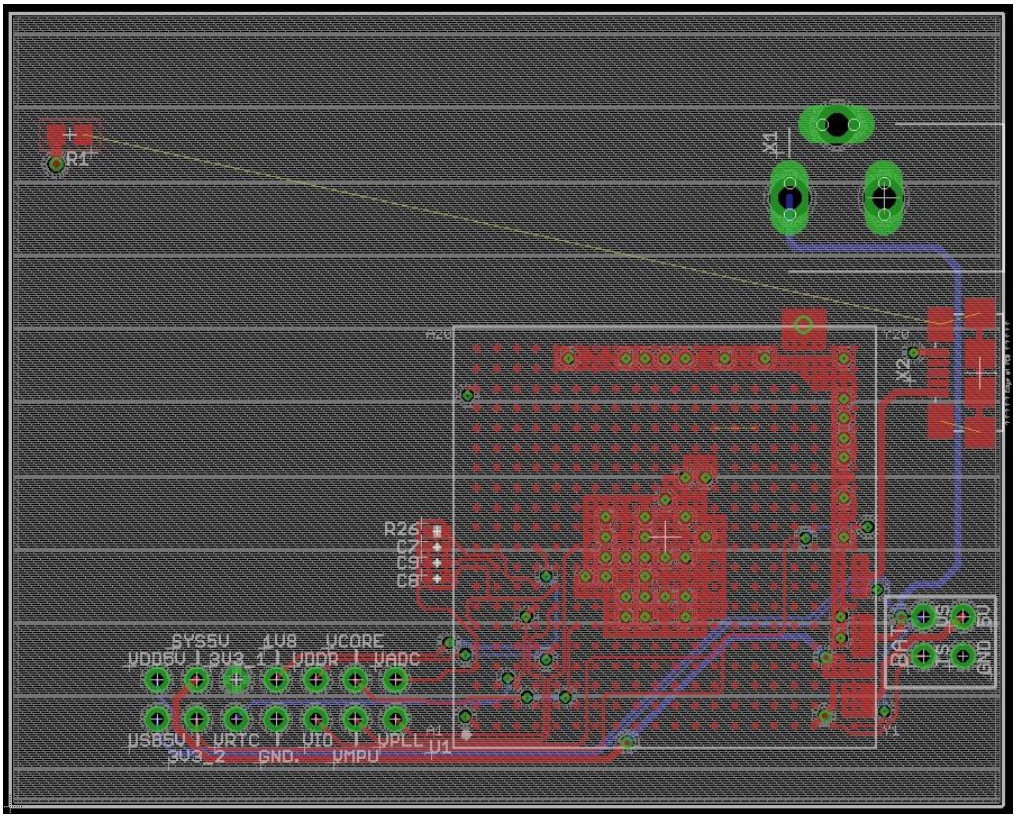


Figure 18 Power Plane

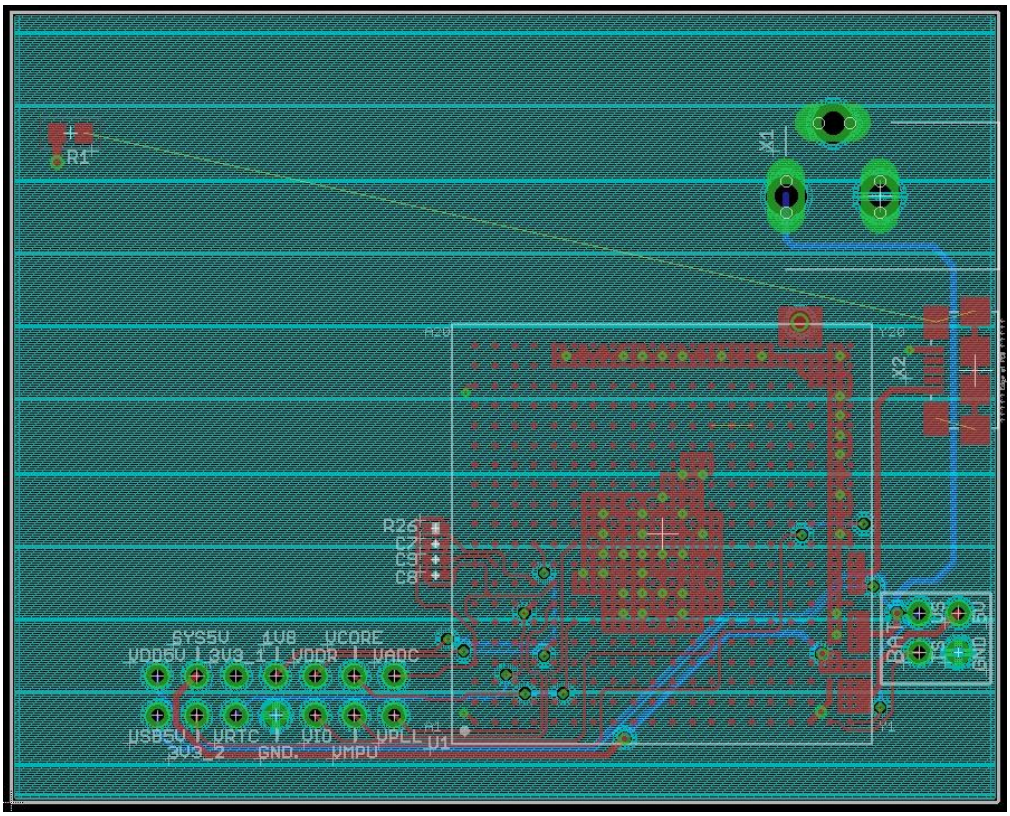


Figure 19 Ground Plane

****

5 OSD335x Power Management

5.1 Introduction

This article is the third part of the OSD335x Reference Design Lesson1 Power circuitry articles. It will mainly focus on topics related to control and management of power for the OSD335x. As we discuss the power circuitry, we will build the schematics and layout the corresponding traces.

5.2 I2C interface

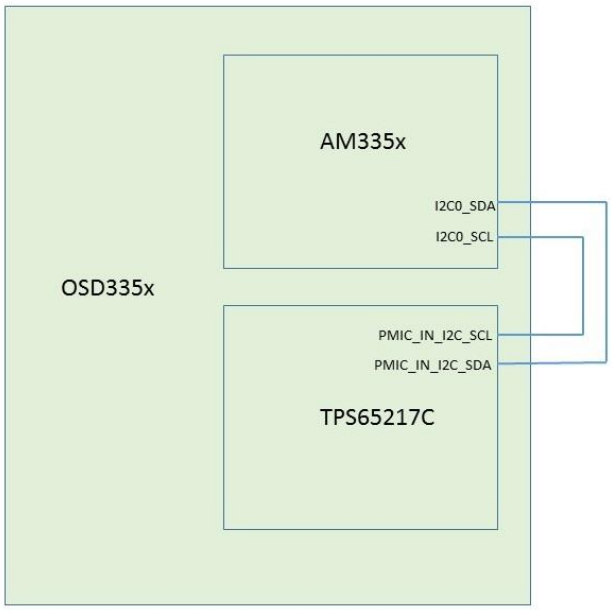


Figure 20 OSD335x I2C connections

For most designs, the AM335x processor communicates with the TPS65217C PMIC through the I2C0 interface. The I2C lines for both the processor and the PMIC are brought out to the BGA balls of the OSD335x to give flexibility in how things are connected. Almost always, the I2C0 pins of the AM335x and the I2C pins of the TPS65217C PMIC are connected externally to enable I2C communication between the processor and PMIC. The I2C pins that need to be connected are placed close to each other on the OSD335x schematic symbol to facilitate easy connection. The necessary connections are shown in Figure 20 block diagram.

The I2C0 pins of the AM335x processor have internal 4.7K ohm pullup resistors. This is sufficient if nothing else is connected to the I2C0 bus. In the case that other components are connected to the I2C0 bus, it is good practice to put additional external pull-up resistors. This allows the strength of the pull-ups on the I2C0 bus to be set based on the other bus components. For this design, given we are connecting

to some unknown components, we are going to use 1.5k ohm external pull-ups whose value can be adjusted later if necessary.

Most of the functionality of the TPS65217C PMIC can be controlled and various parameters can be set using the I2C interface. Some of the features that can be controlled are:

- Battery charger voltage.
- Charge safety timer control.
- Buck and Boost converter output voltage.
- LDO output voltage.
- Power up and power down sequences.
- Over current and over temperature thresholds.

Details about the I2C interface and its usage can be found in the ***TPS65217x datasheet***.

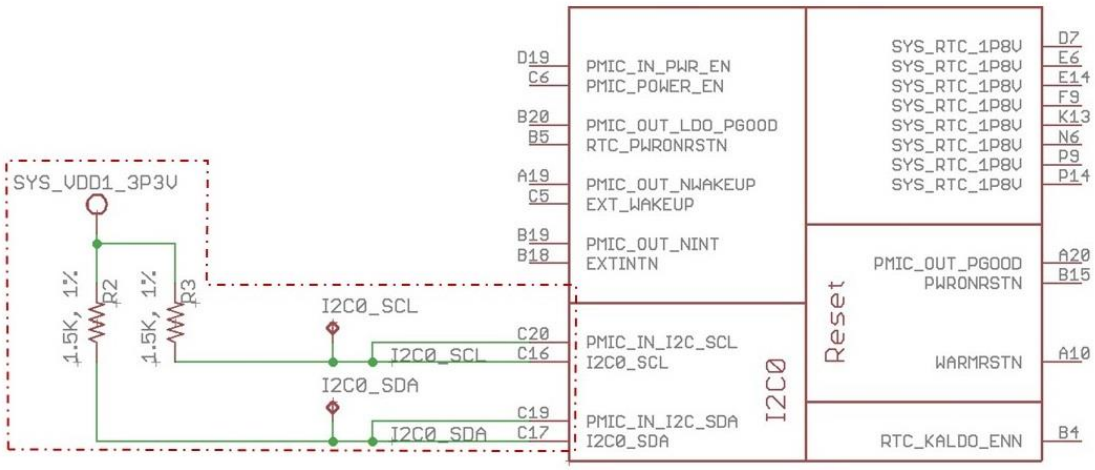


Figure 21 Schematic for OSD335x I2C connections

Let's make I2C connections on the schematic as shown in Figure 21 (Updates made to the schematics are shown using dotted lines).

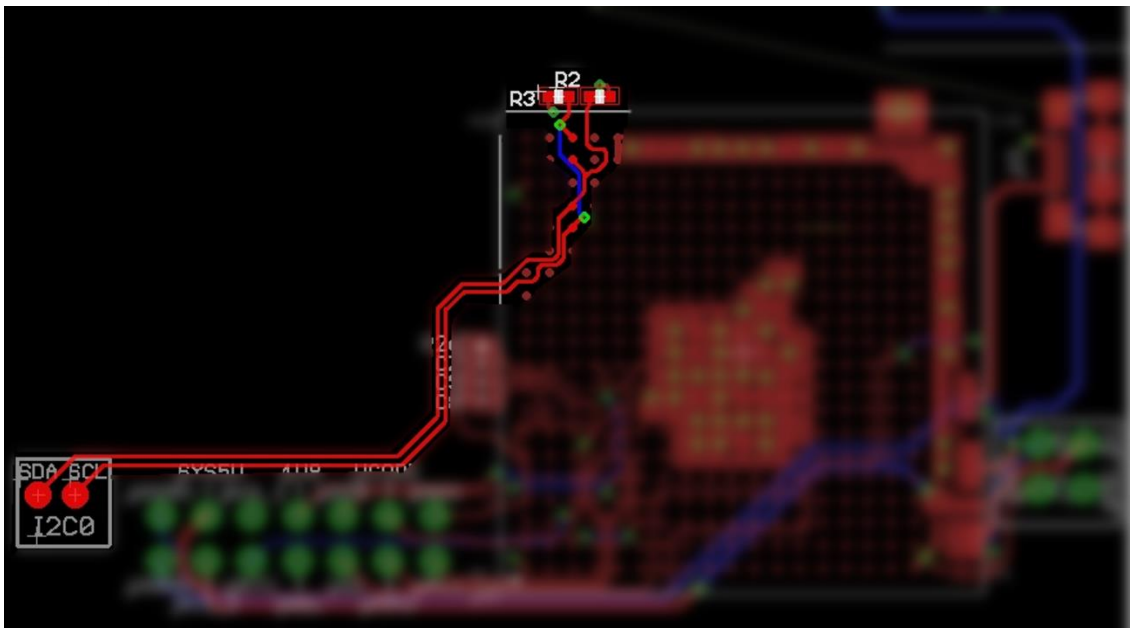


Figure 22 Layout for I2C connections

The highlighted routes in Figure 22 ***represent the I2C connections in the layout.***

5.3 PMIC control and status

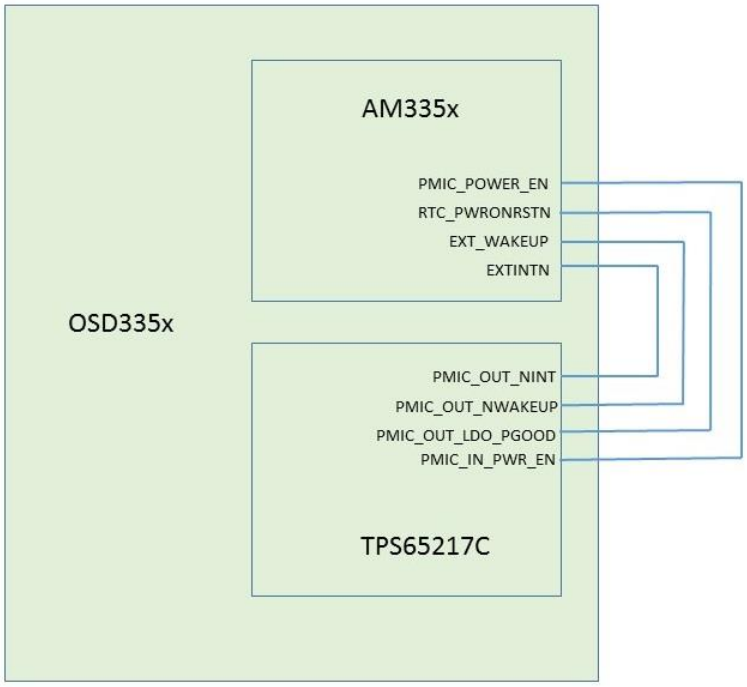


Figure 23 External connections required between AM335x processor and TPS65217C PMIC

Besides the I2C connections, there are a few more signals that must be connected between the processor and PMIC for the OSD335x to function correctly. Figure 23

shows the external signal connections that we must make to allow the AM335x and the TPS65217C PMIC to coordinate their operations. These signals were brought out of the OSD335x so that the user can have more control over power sequencing if necessary. The description of these signals are:

- **PMIC_POWER_EN:** This pin is used by the AM335x to control the power up sequence of the PMIC.
- **PMIC_IN_PWR_EN:** Enable input for buck converters and LDOs on the PMIC. Pulling this pin high will start the power up sequence.
- **RTC_PWRONRSTN:** Independent Power On Reset Pin of the AM335x RTC.
- **PMIC_OUT_LDO_PGOOD:** LDO power good output. This pin goes high when both LDO1 and LDO2 output voltages are within regulation and goes low when either one of them are out of regulation.
- **EXT_WAKEUP:** Dedicated input pin of the AM335x for external wake up events.
- **PMIC_OUT_NWAKEUP:** Signal to host to indicate a power on event (active low)
- **EXTINTN:** External interrupt input of the AM335x.
- **PMIC_OUT_NINT:** Interrupt output (active low) of the PMIC.

For this design, we do not need any additional control of the power sequencing, so we can directly connect the pins. However, let's add test points so that we can probe these signals during bring-up. Let's make the necessary connections on the schematic as shown in Figure 24 (Updates made to the schematics are shown using dotted lines).

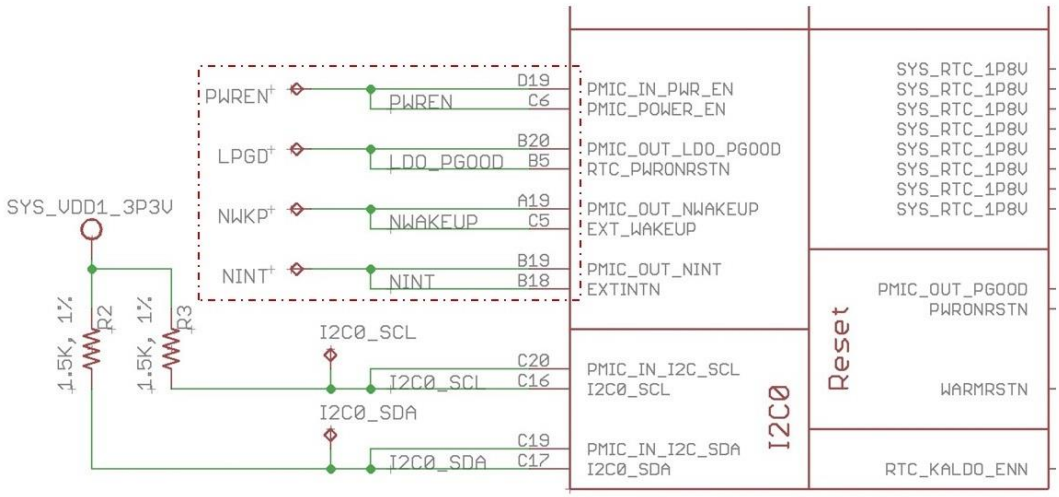


Figure 24 Schematic for OSD335x Control and Status connections

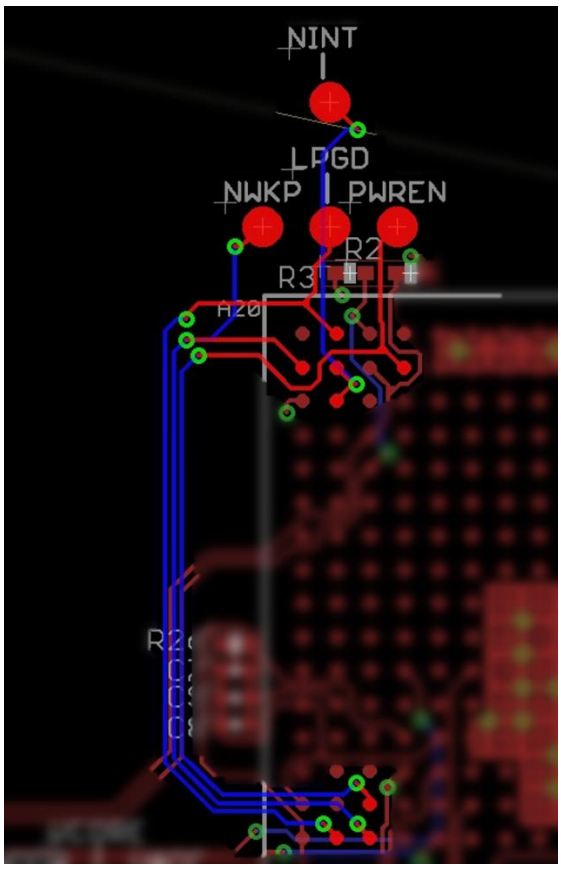


Figure 25 PMIC control and status connections

The highlighted routes in Figure 25 ***represent the PMIC control and status connections in the layout.***

5.4 Power button

The TPS65217C PMIC inside the OSD335x has an active low reset input which is brought out through the PMIC_IN_PB_IN pin of the OSD335x and can be connected to a push button. This input has a 50ms deglitch time and an internal pull-up resistor to an always-on supply. The power button has the following functions:

- The PMIC is powered up from OFF or SLEEP mode upon detecting a falling edge on PMIC_IN_PB_IN.
- PMIC is power cycled/reset when PMIC_IN_PB_IN is held low for more than 8 s. All rails will be shut down by the sequencer and all register values are reset to their default values. Rails not controlled by the sequencer are shut down immediately. The device remains in this state for as long as this pin is

held low. However, the device will remain in RESET state for a minimum of 1 second before it returns to ACTIVE state.

- If the PMIC_IN_PB_IN pin is kept low for an extended amount of time, the device will continue to cycle between ACTIVE and RESET state, entering RESET every 8 seconds.

Perk:

You can find more information about power button and PMIC modes at:

http://www.ti.com/product/TPS65217/datasheet/detailed-description#SLVSB641234

http://www.ti.com/product/TPS65217/datasheet/detailed-

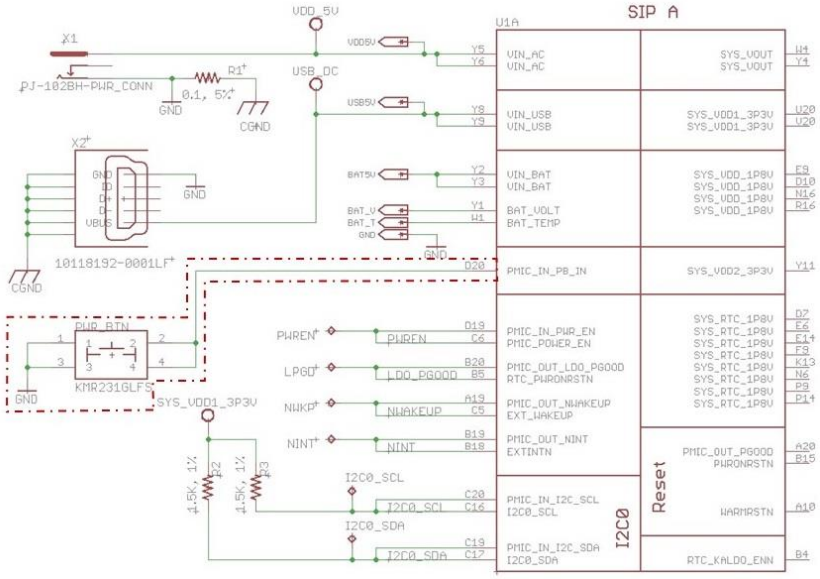


Figure 26 OSD335x Power Button

The schematic is developed as shown in Figure 26 (Updates made to the schematics are shown using dotted lines). The power button we're using can be found under the device name ***KMR231GLFS*** in the given library.

Caveat:

Long-pressing the power button on some boards will power off the system instead of power cycling it. This state occurs when the TPS65217C PMIC shuts off SYS_5V at the start of the power down sequence. If the SYS_5V rail powers down before the regulated supplies, which it powers, it can result in a PMIC fault. This causes the system to go to the OFF-state instead of going through the normal power-cycle sequence.

Figure 27 Power button layout

The highlighted routes in Figure 27 ***represent the power button connections in the layout.***

5.5 Power indicator LED

For this design, we want to have an LED to indicate that power is on. Since SYS_VDD2_3P3V can only supply 150mA externally, we used this power output to indicate power is on as shown in Figure 28 (Updates made to the schematics are shown using dotted lines).

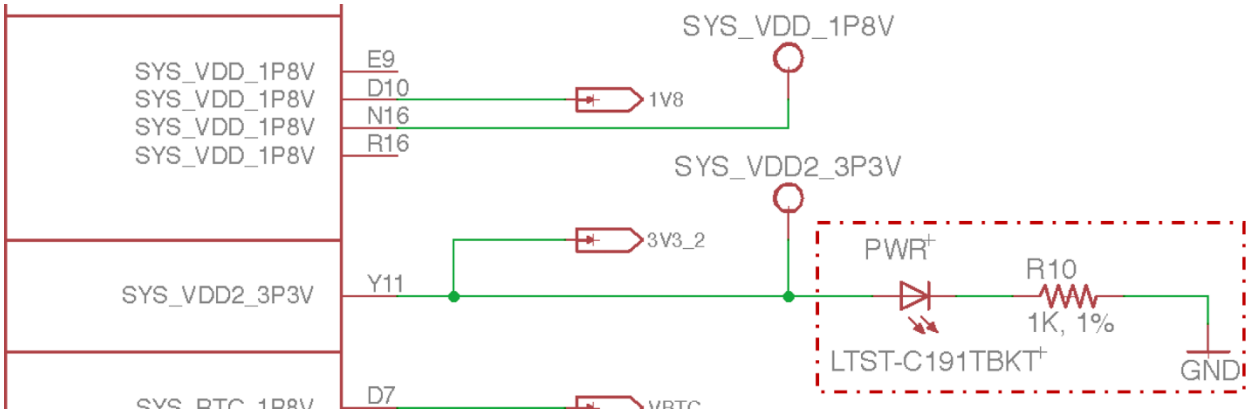


Figure 28 Schematic for OSD335x Power LED

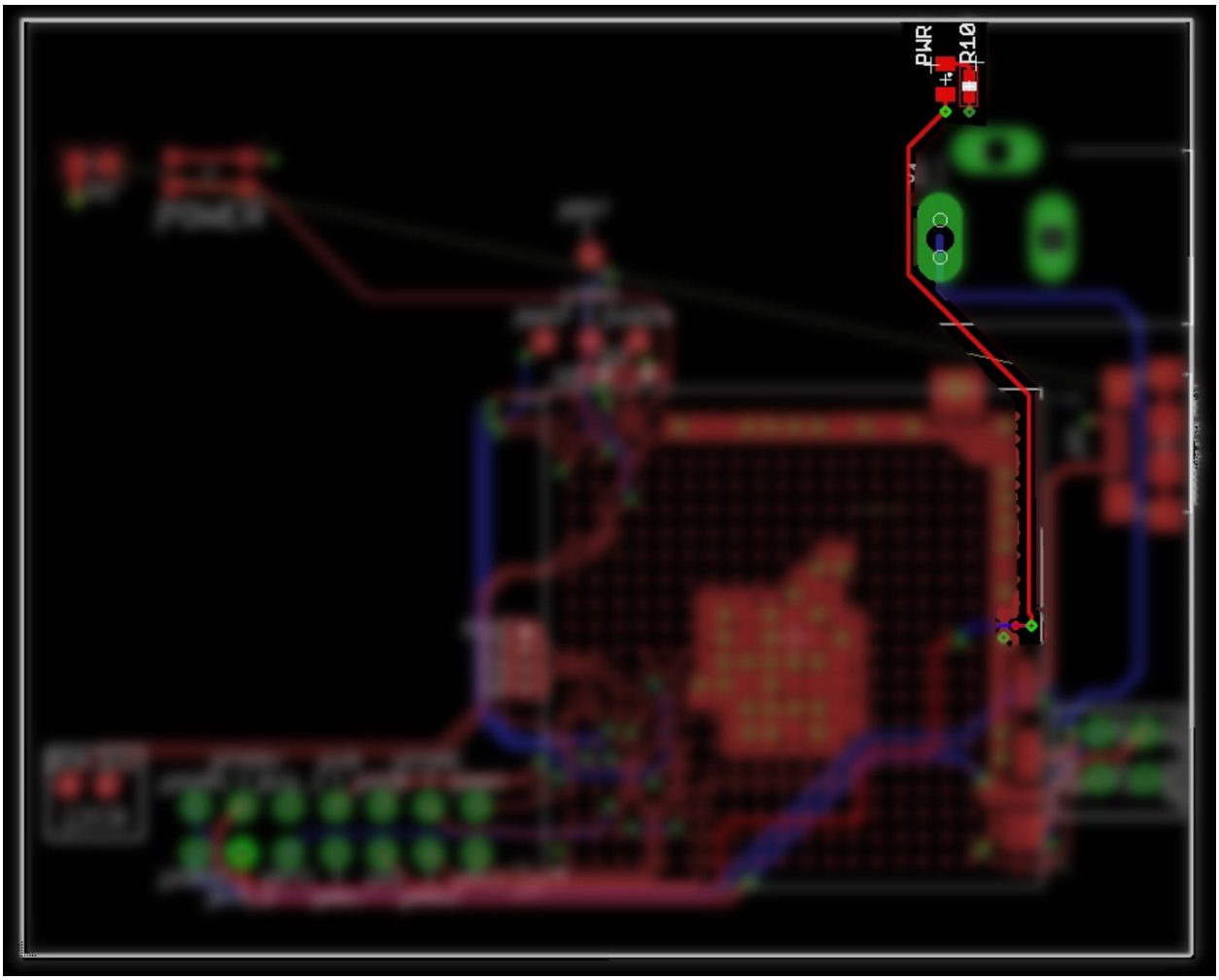


Figure 29 Power LED layout

The highlighted routes in Figure 29 ***represent the power LED connections in the layout.***

*****

6 OSD335x Clamping Circuit

6.1 Introduction

This article is the fourth part of the OSD335x Reference Design Lesson1 Power circuitry discussion. It will focus on a ***clamping circuit*** which may be needed by your application. A clamping circuit is a type of circuit that maintains the voltage level of an input with respect to another input. As we discuss the clamping circuit, we will build the schematic and layout the corresponding traces.

6.2 AM335x Power-Down Requirements

The AM335x datasheet requires that the voltage difference between the power rails VDDS (1.8V) and VDDSHVx [1-6] (3.3V) of the AM335x processor be less than 2V during the entire power-down sequence (More information about this can be found under ***Use of a Clamping Circuit for Simultaneous Ramp Down*** section of ***this user guide***). The VDDS power input of the AM335x processor is connected within the OSD335x to the SYS_RTC_1P8V power rail and the VDDSHVx [1-6] power inputs are connected within the OSD335x to the VDDSHV_3P3V power rail. The voltage difference between these two power rails could exceed 2V if VDDSHV_3P3V rail remains high, possibly because of large output capacitance or no load being present on the output, while the SYS_RTC_1P8V rail ramps down quickly, as if it were fully loaded.

If you cannot guarantee that the voltage difference between the two voltage rails (SYS_RTC_1P8V and VDDSHV_3P3V) of the OSD335x will be less than 2V during the entire power down sequence in your design, then you will need a clamping circuit.

6.3 Clamping circuit

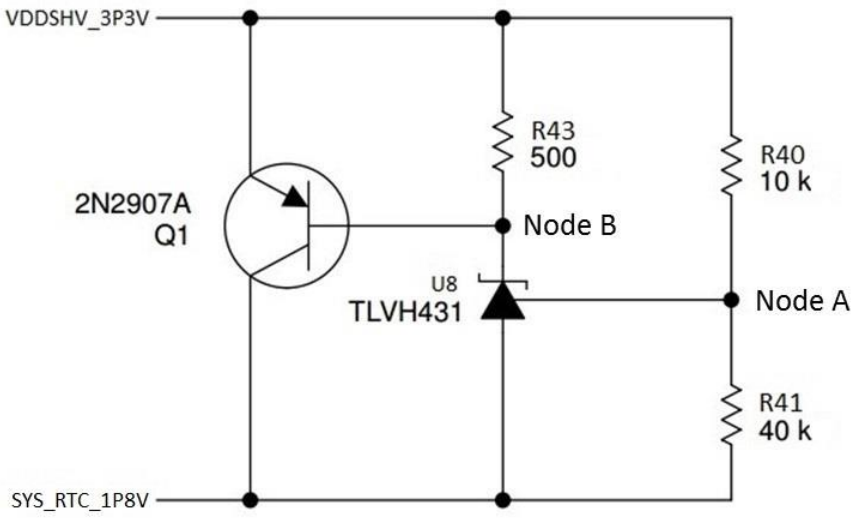


Figure 30 Clamping circuit

The clamping circuit (shown in Figure 30) will make sure that the voltage difference between the given two voltage rails (VDDSHV_3P3V and SYS_RTC_1P8V) is less than 2V. Its operation will be discussed in three phases.

You need to keep the following assumptions in mind before analyzing the operation of clamping circuit:

- It is assumed that, once the power down sequence of the AM335x begins, the voltage rail VDDSHV_3P3V remains high due to a large output capacitance or no load and the voltage rail SYS_RTC_1P8V ramps down relatively quickly due to a full load.
- It is also assumed that, the voltage rail, VDDSHV_3P3V, will be able to source enough current to pull up the voltage of SYS_RTC_1P8V rail whenever necessary.

We recommend you to read the datasheets of the shunt regulator ***TLVH431*** and ***2N2907A*** transistor before trying to understand the operation of clamping circuit.

Few quick tips about TLVH431 and 2N2907A essential to understand the operation of clamping circuit:

- The shunt regulator, TLVH431, will be OFF (acts as open switch) whenever the voltage at its reference input connected to node A (in Figure 30) is less than 1.24V. It will turn ON (act as closed switch) whenever voltage at its reference input (node A) is greater than 1.24V.
- The PNP transistor, 2N2907A, will allow current to flow from its emitter terminal to the collector terminal when the voltage difference between its

base terminal and emitter terminal is greater than or equal to 0.4V. In other words, it acts like a closed switch when the voltage difference between node B and the VDDSHV_3P3V rail is greater than 0.4V and acts like an open switch when the voltage difference between node B
and the VDDSHV_3P3V rail is less than 0.4V.

For the circuit analysis below, we will assume that TLVH431 will be in the OFF state whenever the reference input of the TLVH431 is at 1.24V.

6.3.1 Phase 1 – AM335x in normal operation / just before power down (Clamping circuit in standby):

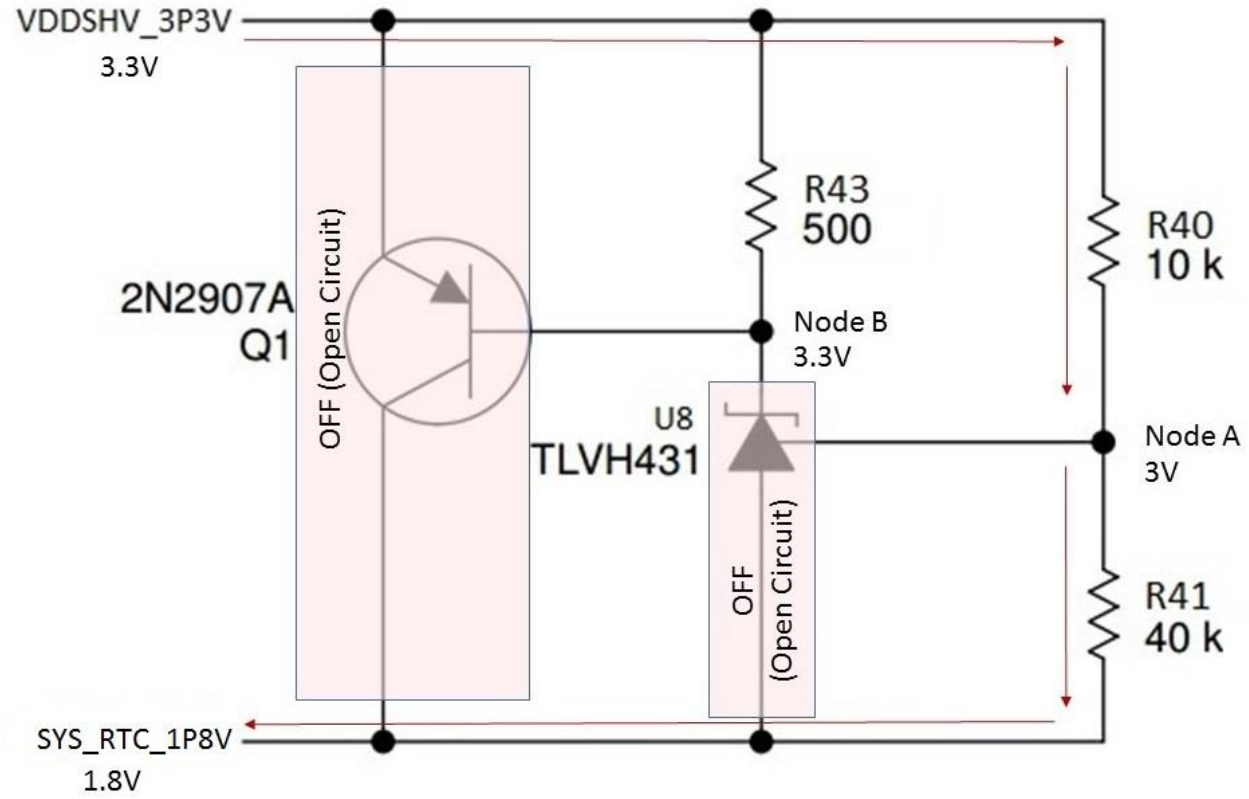


Figure 31 Clamping circuit (Phase 1)

The clamping circuit will operate in this phase when both the voltage rails are at nominal voltages (i.e., VDDSHV_3P3V is at 3.3V and SYS_RTC_1P8V is at 1.8V) as shown in Figure 31.

The voltage divider circuit consisting of resistors R40 and R41 provides the reference voltage for U8 (TLVH431) at node A. In this phase of operation, node A will be at 3V relative to ground. But, node A will be at 1.2V (3V – 1.8V) with respect to SYS_RTC_1P8V voltage rail. As a result, U8 will turn OFF (will not sink current from node B) and voltage at node B will remain at 3.3V. Hence, Q1 (2N2907A) will turn OFF.

6.3.2 Phase 2 – AM335x power down sequence begins (Clamping circuit actively maintaining the voltage difference between the two power rails):

The clamping circuit will operate in this phase when the voltage difference between its two power rails, VDDSHV_3P3V and SYS_RTC_1P8V, is greater than or equal to 1.55V. Let's discuss the two possible cases:

6.3.2.1 Case 1: Voltage difference between VDDSHV_3P3V and SYS_RTC_1P8V is equal to 1.55V

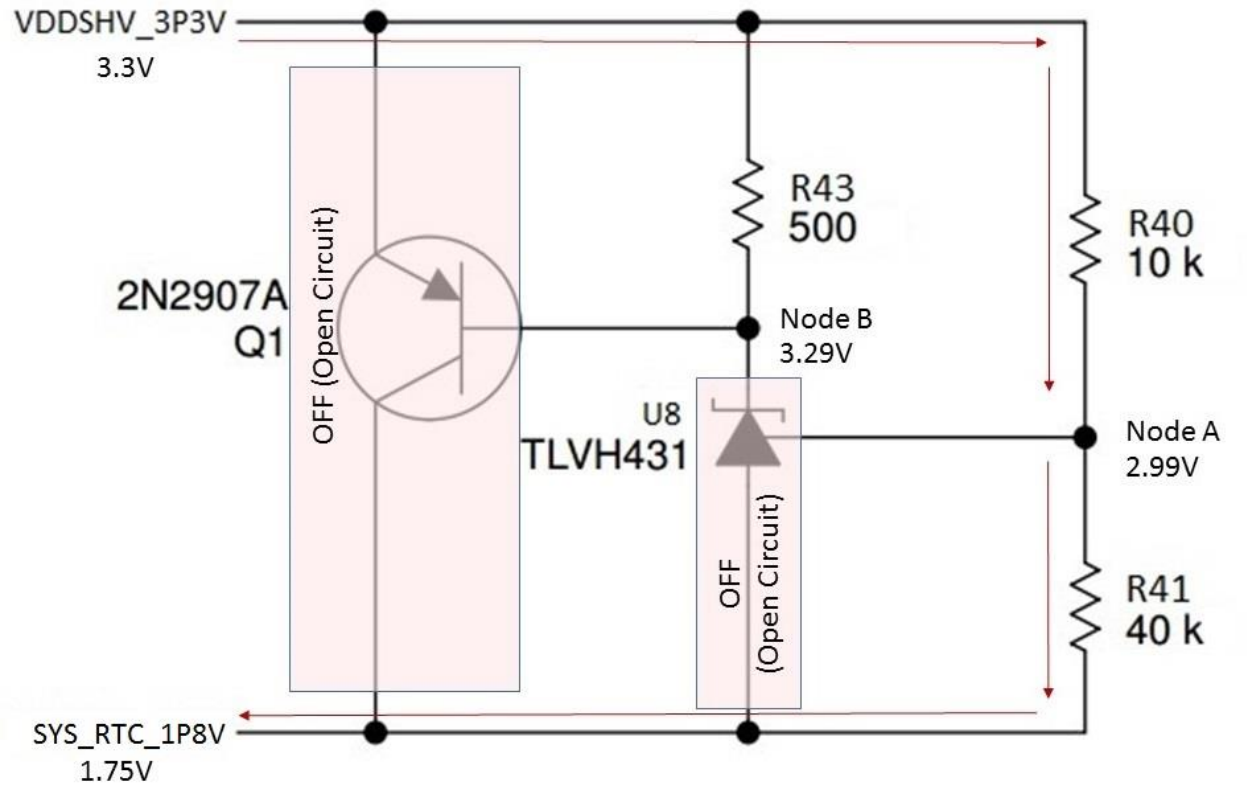


Figure 32 Clamping circuit (Phase 2, Case 1)

Let's understand case 1 through an example. For this example, let's assume VDDSHV_3P3V is at 3.3V and SYS_RTC_1P8V is at 1.75V as shown in Figure 32 (Fixing the voltage will help us understand the circuit better). Now, the voltage difference between the two power rails is 1.55V (3.3V – 1.75V = 1.55V).

When SYS_RTC_1P8V is at 1.75V, the voltage at node A will be 2.99V (i.e., 2.99V-1.75V = 1.24V with respect to SYS_RTC_1P8V rail). As a result, U8 (TLVH431) turns OFF and will not sink current from node B. Therefore, node B will be at 3.29V and the voltage difference between the base and emitter of transistor Q1 will be less than 0.4V. Hence, Q1 will turn OFF. This is similar to Phase 1 where both U8 and Q1 are off.

6.3.2.2 Case 2: Voltage difference between VDDSHV_3P3V and SYS_RTC_1P8V is greater than 1.55V

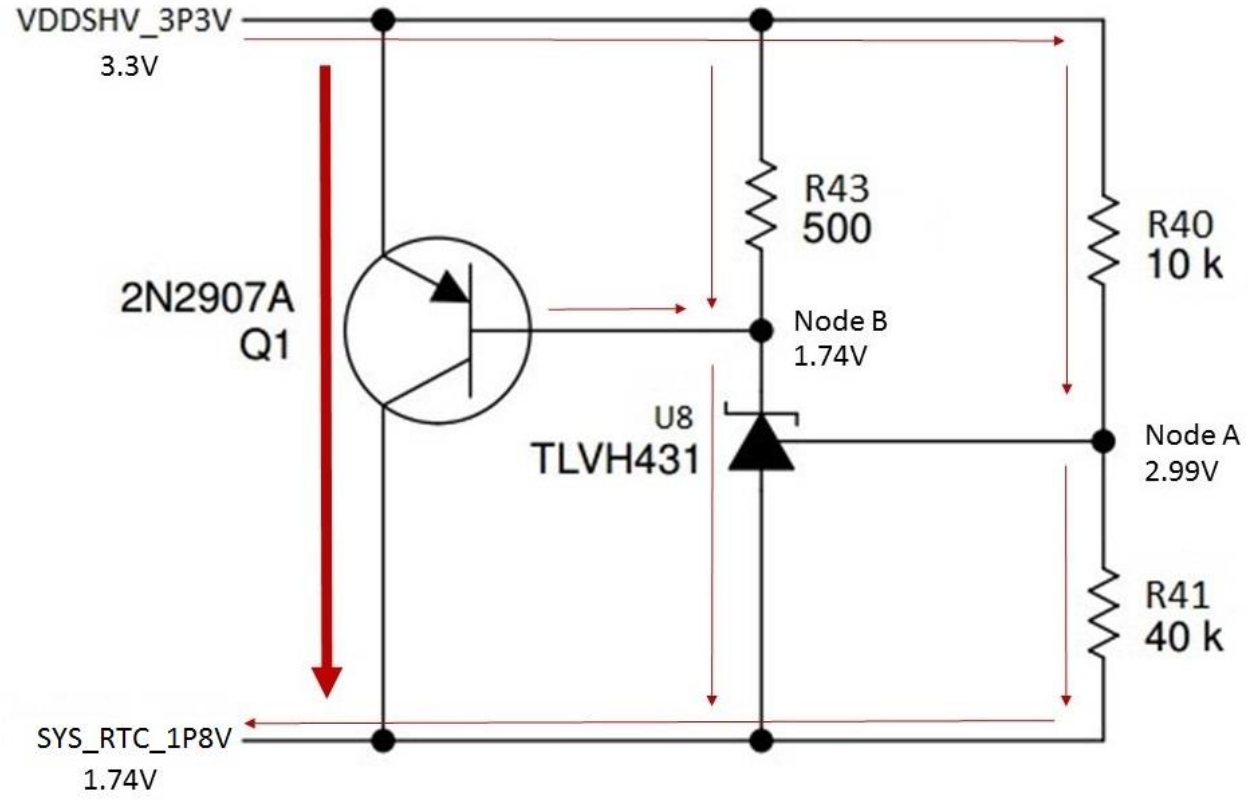


Figure 33 Clamping circuit (Phase 2, Case 2)

Now, let's understand the behavior of clamping circuit when the voltage difference between its two power rails is greater than 1.55V. Let's assume VDDSHV_3P3V is at 3.3V and SYS_RTC_1P8V is at 1.74V as shown in Figure 33 (Fixing the voltage will help us understand the circuit better). Hence, the voltage difference between the two power rails becomes 1.56V (3.3V – 1.74V = 1.56V).

When SYS_RTC_1P8V is at 1.74V, the voltage at node A will be 2.99V (i.e., 2.99V-1.74V = 1.25V with respect to SYS_RTC_1P8V rail). As a result, U8 (TLVH431) turns ON and sinks current from node B. This will lower the voltage at node B to 1.74V and the voltage difference between the base and emitter of Q1 will exceed 0.4V. Hence, Q1 will turn ON and current begins to flow from VDDSHV_3P3V rail to SYS_RTC_1P8V rail. This will increase the voltage of SYS_RTC_1P8V rail to 1.75V. The voltage at node A with respect to ground will now become 1.24V (2.99V – 1.75V = 1.24V) and U8 will turn OFF. So, Q1 will turn OFF and current flow from VDDSHV_3P3V to SYS_RTC_1P8V will cease.

This cycle continues until one or both of the voltage rails drop down to 0V at the end of the power down sequence. Now, this is an idealized analysis so in the real

operation there will be some overshoot and undershoot but based on the voltage ripple should be less than ±5%.

6.3.3 Phase 3 - End of AM335x power down sequence (Clamping circuit back to standby):

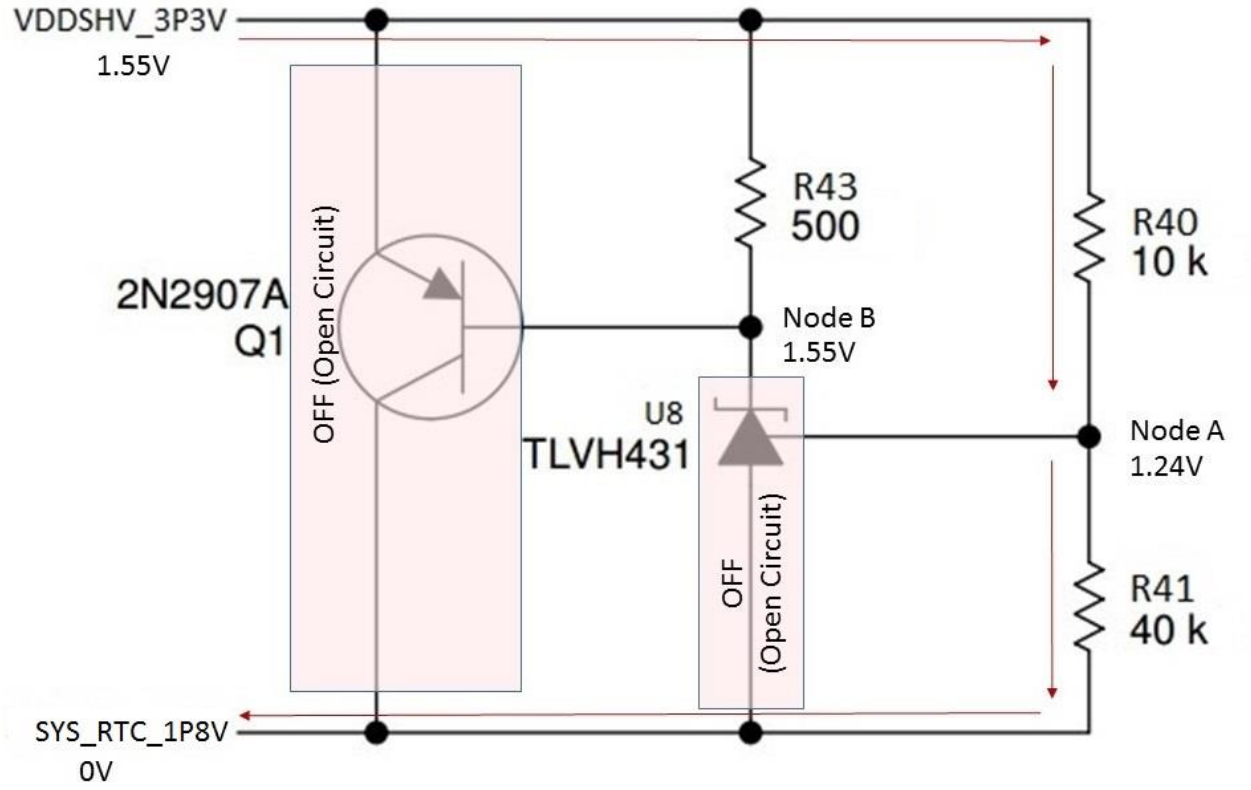


Figure 34 Clamping circuit (Phase 3)

The clamping circuit will operate in this phase when the AM335x is almost at the end of its power down sequence and after SYS_RTC_1P8V drops down to 0V while VDDSHV_3P3V is still at 1.55V as shown in Figure 34.

When SYS_RTC_1P8V is at 0V, the voltage at node A will be 1.24V. As a result, U8 (TLVH431) turns OFF and will not sink current from node B. This will lower the voltage at node B and the voltage difference between the base and emitter of Q1 will be less than 0.4V. Hence, Q1 will turn OFF. This is similar to Phase 1 where both U8 and Q1 are off.

The operation of clamping circuit and its three phases can be better understood with the help of a voltage vs time graph as shown in Figure 35.

Phase 1: The clamping circuit will be in standby and Q1 is OFF.

Phase 2: Initially, the clamping circuit will turn ON when the voltage difference between the given two power rails increases to 1.56V (shown at the beginning of

Phase 2 in Figure 35). After this, the clamping circuit will actively maintain the voltage difference between VDDSHV_3P3V and SYS_RTC_1P8V at approximately 1.55V.

Phase 3: The clamping circuit will go back to standby mode since SYS_RTC_1P8V will drop down to 0V while VDDSHV_3P3V is at 1.55V and the difference between the two voltage rails is 1.55V.

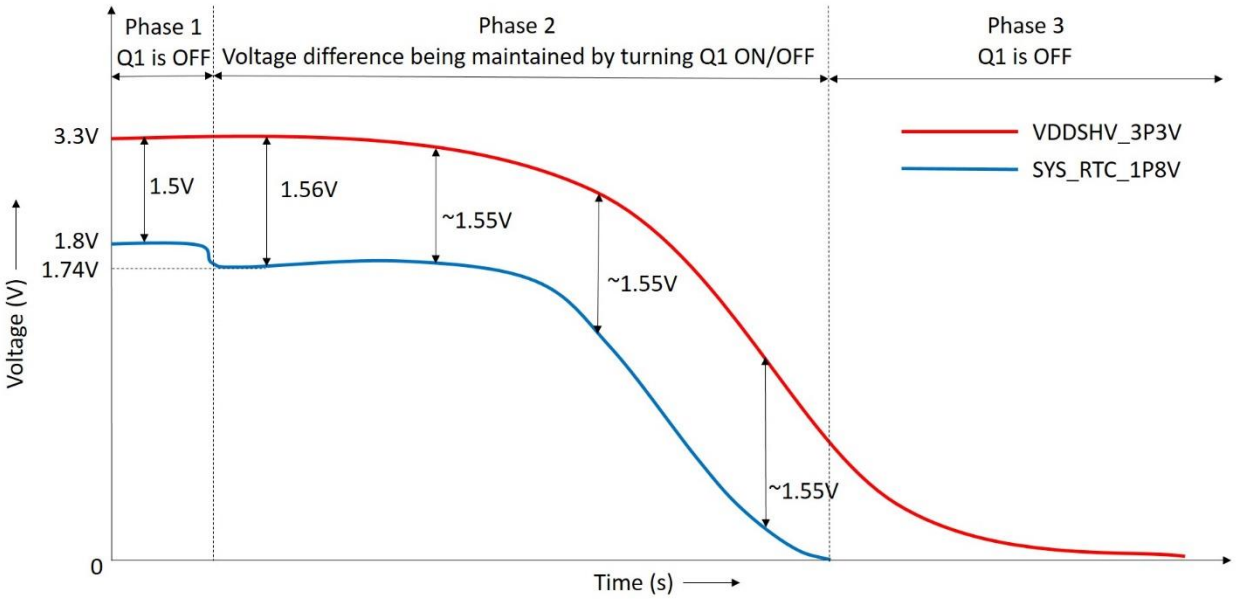


Figure 35 Clamping circuit voltage curves

Let's build the schematic of the clamping circuit as shown in Figure 36.

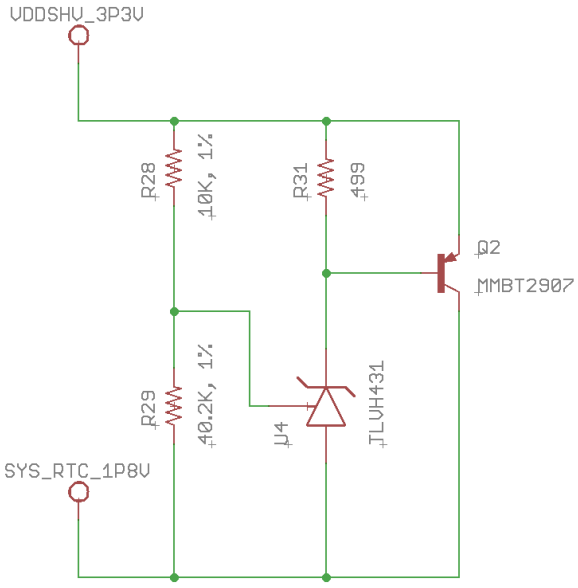


Figure 36 Schematic for OSD335x Clamping Circuit

Connections in layout are made as shown in Figure 37.

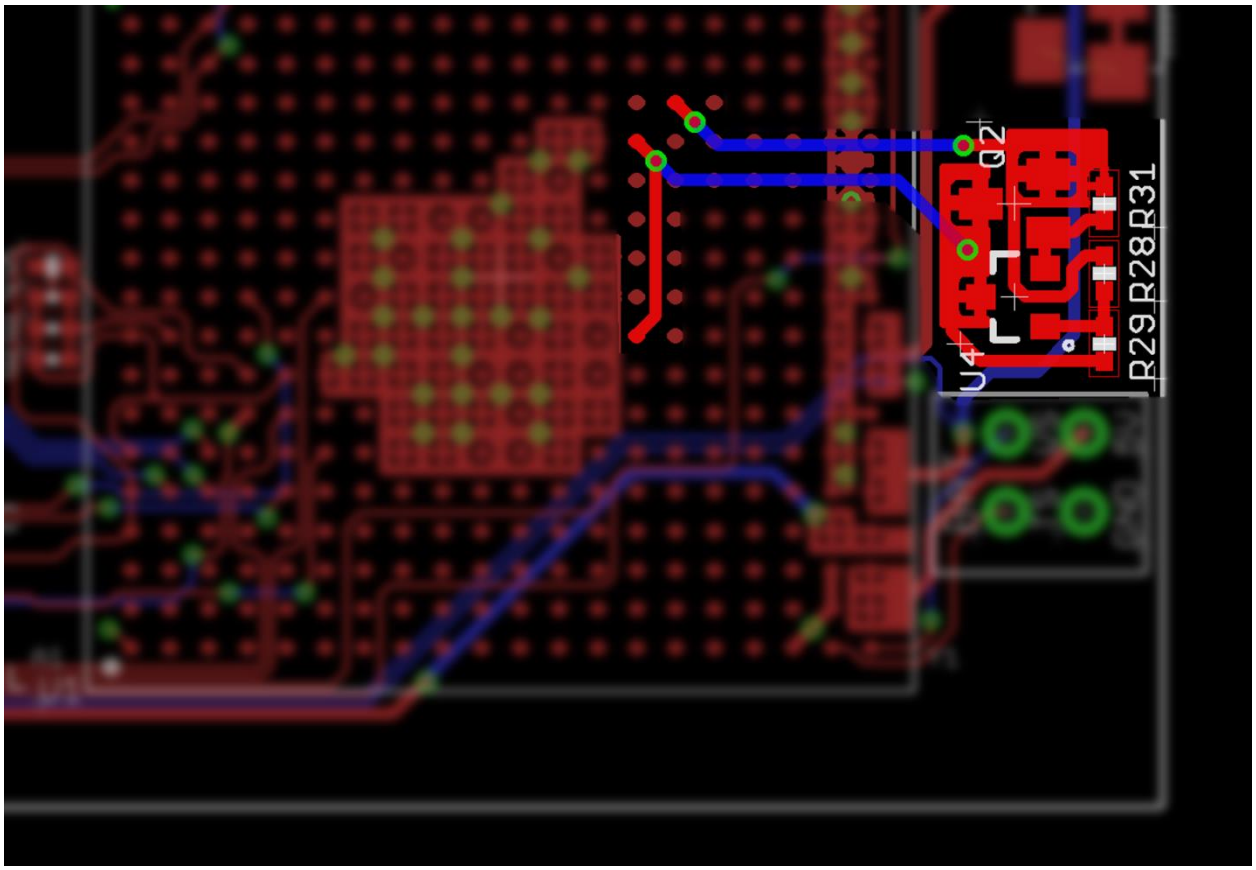


Figure 37 OSD335x Clamping Circuit layout with pour

For this design, we used copper pours as well as 15 mil traces so that there is good electrical as well as thermal connections.

Since we only wanted to put components on a single side of the board for this design, the power outputs were routed to the clamping circuit as shown in Figure 37. However, if placing components on the back side of the board, clamping circuit can go directly under the power connections.

*****

7 OSD335x ESD Protection

7.1 Introduction

Electrostatic discharge (ESD) is the flow of a static electric charge from one object to another when two charged objects come into contact. Familiar examples of ESD include the shock we receive when we walk across a carpet and touch a metal door knob and the static electricity we feel after drying clothes in a clothes dryer. This flow of static charge involves build-up of a very high voltage (around 10KV) for a very short duration of time. Generally, a Printed Circuit Board (PCB) receives an ESD strike whenever a person (whose skin is statically charged) touches it. Most electronic components (without internal ESD protection) get damaged since they cannot withstand such high voltage. Hence, ESD protection for a PCB is essential to protect all the components on it.

This article is the fifth part of the OSD335x Reference Design Lesson1 Power Circuitry Discussion. It will focus on providing ESD protection to a PCB with the OSD335x and other components on it. As we discuss the ESD protection circuitry, we will build the schematics and layout the corresponding traces.

7.2 ESD protection

In order to protect the board from ESD strikes which can result from a person touching the PCB, we have created a ground ring, CGND, that runs along the edge of the PCB on both the top and bottom layers. The ground ring connects all of the mounting holes (annotated in Figure 41) and shield pins (annotated in Figure 41) of metal connectors, such as USB connectors. These components are most susceptible to ESD strikes. An ESD strike has a very high voltage (around 10KV), but relatively little charge (low energy). Therefore, we can use a couple of different methods to help dissipate the energy from an ESD strike.

First, we can make use of the mounting holes as shown in Figure 38. Four mounting holes are placed at the corners of the PCB for mechanical attachments to the board. Generally, PCBs are bolted through the mounting holes to metal cases that are externally grounded. As a result, when an ESD strike occurs, the charge safely flows through the mounting holes to the bolts and finally to external ground.

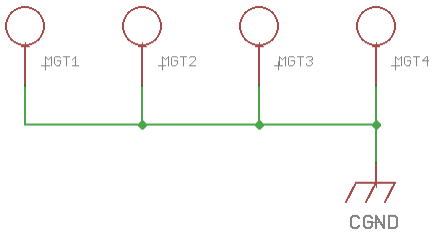


Figure 38 Mounting holes connected to CGND

Second, since a PCB may not be mounted and not have a path to external ground, we can add circuitry to dissipate the energy through the ground (GND) plane. Therefore, CGND is connected to the GND plane of the PCB through a 0.1 ohm resistor (R1) as shown in Figure 39 (This picture is from OSD335x Power Inputs and Outputs article).

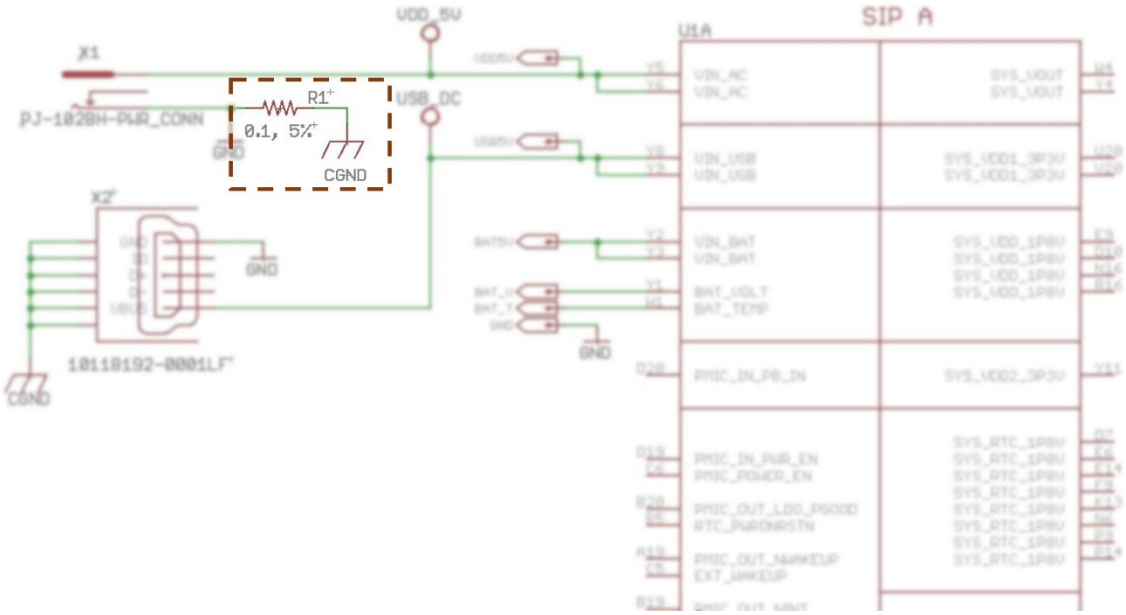


Figure 39 ESD energy dissipation

This will make sure any ESD strike can be safely dissipated into the GND plane. The 0.1 ohm resistor should have a higher wattage (0.125W or larger) to make sure there are no thermal issues (damage to the resistor due to power flow that exceeds its rating). Alternately, a 0.1uF capacitor and 100K Ohm resistor in parallel can be used to connect CGND to the GND plane (In this design, we have opted for the single resistor approach to dissipate energy since it is more economical).

The CGND polygon pour for ESD protection should form a complete ring along the edge of the board and must be well connected to the shield pins of the connectors. The CGND polygon pour must be present on both sides of the board. For this design, we will use a pour of at least 50 mils as shown in Figure 40 and Figure 41.

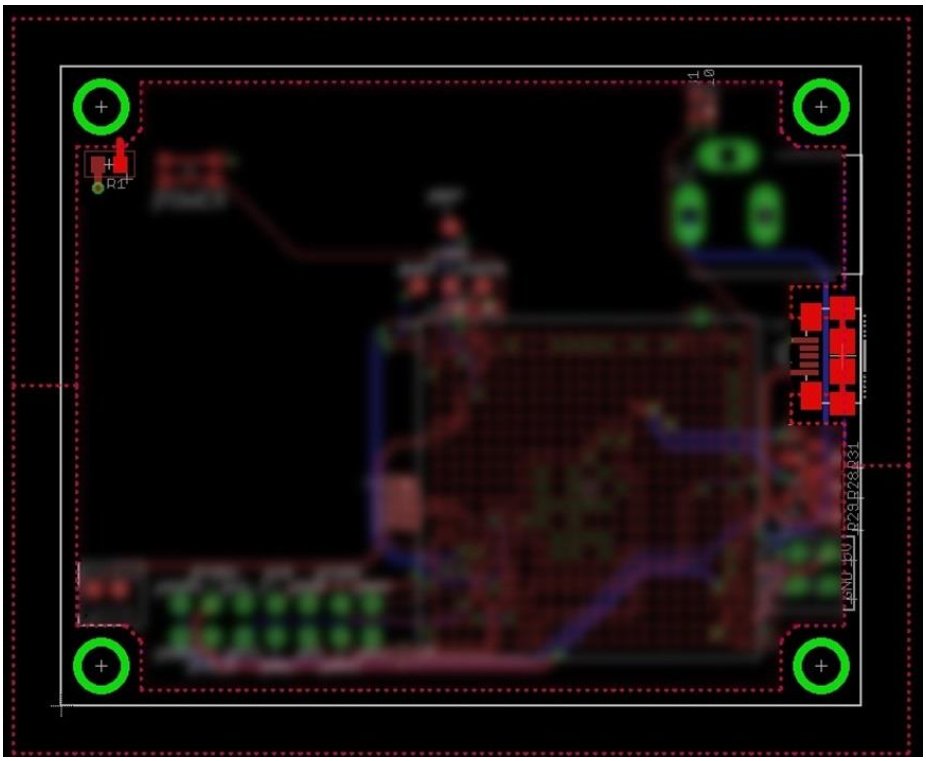


Figure 40 OSD335x mounting holes with CGND pour outline for ESD protection

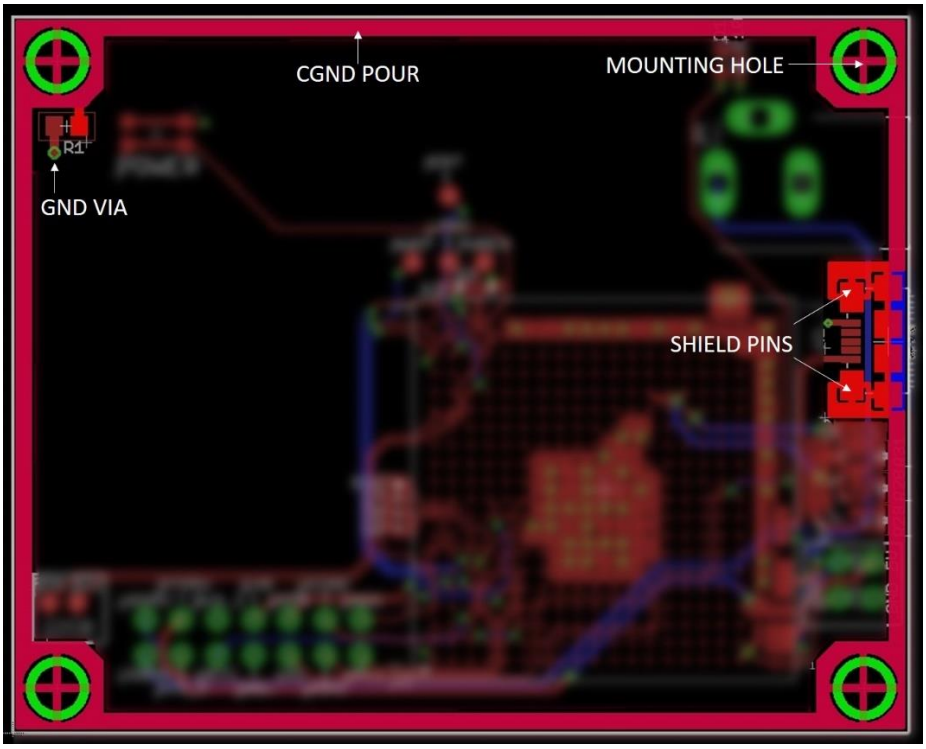


Figure 41 OSD335x mounting holes with CGND pour for ESD protection

*****

8 OSD335x Reset Circuitry

8.1 Introduction

This article will help you understand the OSD335x Reset Circuitry design methodology.

A Reset circuit/pin is typically used to help a microprocessor reinitialize itself and resume its normal operation whenever it runs into condition which is undesirable for the current activity and when all other recovery mechanisms fail. It is also used during power-up to make sure the microprocessor and all its modules start their operation from a known state.

8.2 Reset Types

Before we jump into the OSD335x specifics, let's look at the type of reset inputs typically provided by Texas Instruments (TI). TI generally provides two types of resets: Cold Reset and Warm Reset. Let's look at the properties of both these reset types:

8.2.1 Cold reset

- It affects all the logic within the given entity (sub-system, module or macro-cell).
- It is non blockable (This signal cannot be interrupted or blocked using software or any other internal module. Once this signal is triggered, the device immediately performs all the necessary operations irrespective of its state).
- A cold reset takes place during device power-up and power domain power-up.
- Cold reset is synonymous with Power-On-Reset.

8.2.2 Warm reset

- It is a partial reset which doesn't affect all the logic within the given entity.
- It is used to reduce Reset recovery time (Time required to resume normal operation after application of reset signal).

In general, a reset signal is asserted during device startup to make sure the device begins operation from a known initial state each time it is powered up. This signal is applied until the power supplies are stable and the device can begin normal operation. A reset signal is also applied during device operation when the microprocessor runs into an error condition which is undesirable for the current activity and all other error recovery mechanisms fail.

The OSD335x provides three reset inputs PWRONRSTN, WARMRSTN and RTC_PWRONRSTN which are directly connected to the AM335x processor reset inputs of the same name. Let's look at each of them in more detail:

8.2.2.1 PWRONRSTN

- It is a cold reset.
- It needs to be driven low during device power-up until all the input power lines have ramped up and are stable.
- It is non blockable (PWRONRSTN signal cannot be interrupted or blocked using software or any other internal module. Once this signal is triggered, the device immediately performs all the necessary operations irrespective of its state).
- Entire system is affected except RTC (Real Time Clock) module.
- SYSBOOT (boot configuration) pins are latched when reset is de-asserted.

8.2.2.2 WARMRSTN

- It is a warm reset.
- It can be blocked by EMAC (Ethernet Media Access Controller) switch.
- PLLs are not affected.
- Most debug logic subsystems are not affected. This allows us to maintain debug session even after warm reset event.
- SYSBOOT pins are not latched with warm reset.
- Some PRCM (Power, Reset and Clock Management) and control module registers are warm reset insensitive.
- Warm reset assumes power supply and clock is stable from assertion through de-assertion.

8.2.2.3 RTC_PWRONRSTN

- Dedicated Power-On-Reset input for the RTC module.
- RTC module is not affected by device Power-On-Reset (PWRONRSTN). Similarly, RTC_PWRONRSTN will not have any effect on the rest of the device.

8.3 Reset external connections

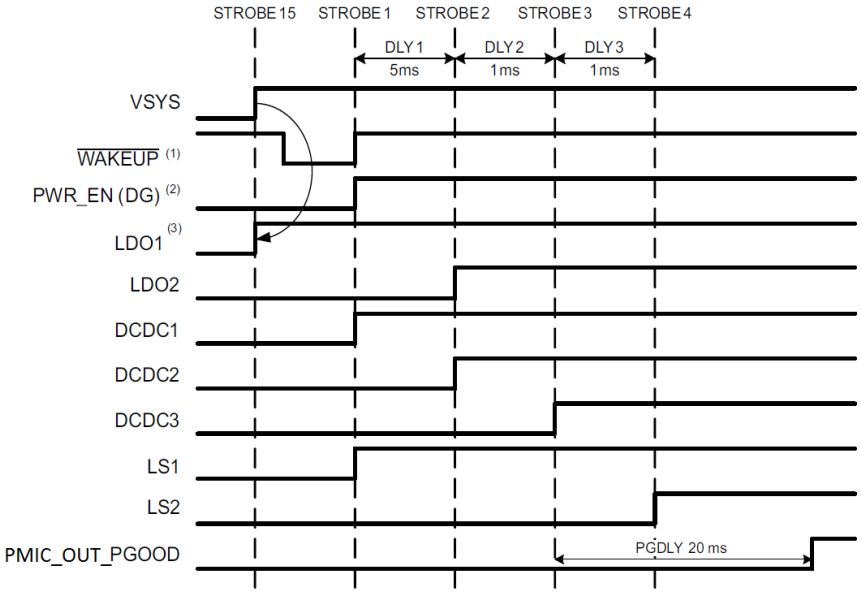


Figure 42 TPS65217 PMIC Power-Up Sequence (PMIC_OUT_PGOOD/PGOOD behavior during power-up)

During power-up, voltages on the input power rails of the AM335x processor will be ramping up as shown in Figure 42 (More information about Power-Up sequence of various power rails can be found in ***Power-Up Sequencing*** section of the ***TPS65217x datasheet***). The PWRONRSTN pin should be driven low until all the power rails have ramped up and are stable. The PMIC_OUT_PGOOD pin will be maintained low when power rails are ramping up or when the power on any of the power rails is below the required value. It will go high only when the power on all power rails are stable as shown in Figure 42. Hence, the PWRONRSTN pin needs to be driven by the PMIC_OUT_PGOOD pin. Both the PWRONRSTN and the PMIC_OUT_PGOOD signals are brought out of the OSD335x and they need to be connected externally as shown in Figure 43.

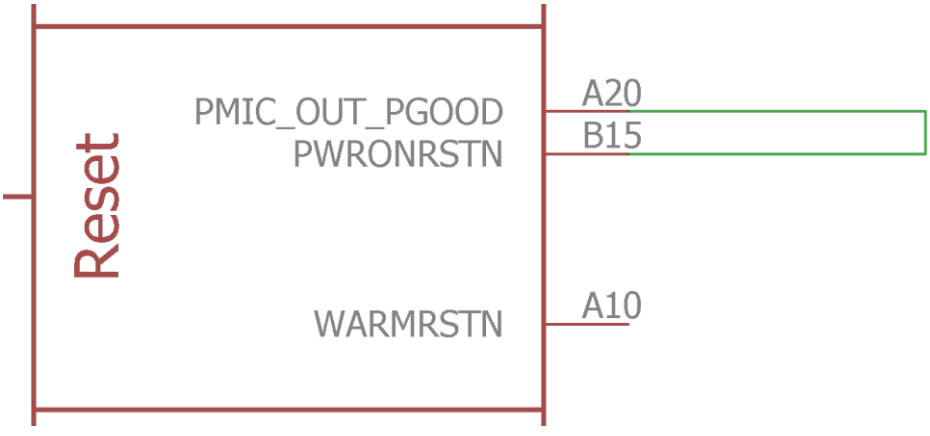


Figure 43 PWRONRSTN and PMIC_OUT_PGOOD connection

As described in the ***PORz Sequence*** section of the ***AM335x TRM***, for the WARMRSTN pin (nRESETIN_OUT) to maintain a valid low state until the supplies are ramped, it should also be driven by PMIC_OUT_PGOOD. However, due to I/O voltage differences (PMIC_OUT_PGOOD and PWRONRSTN pins operate at 1.8V while the WARMRSTN pin operates at 3.3V) and the need for independent reset control, WARMRSTN should be driven by PMIC_OUT_PGOOD thru an open-drain buffer (SN74LVC1G07). For more information on the requirement of open-drain buffer, see section 8.1.7.3.2 PORz Sequence (page 1236) of the ***AM335x TRM (Rev. P)***.

To manually reset the OSD335x in the case of a software or hardware error condition, we will also add a push button as a reset source for WARMRSTN. Push buttons are susceptible to ground bounce which may lead to multiple resets or partial resets. To overcome this problem, a reset supervisor circuit can be used. The APX811 is an efficient and cost effective solution which allows us to consolidate the reset sources for WARMRSTN and drive the signal cleanly. The entire reset supervisor circuit can be seen in Figure 44.

Perk:

A supervisor circuit (with manual reset input) has two specific functions. It asserts a reset signal for a fixed period of time whenever the:

- supply voltage falls below a preset voltage.
- the manual reset input is asserted. (it may care of input button de-bouncing also)

You can find more information about APX811's operation by going through its datasheet ***here***.

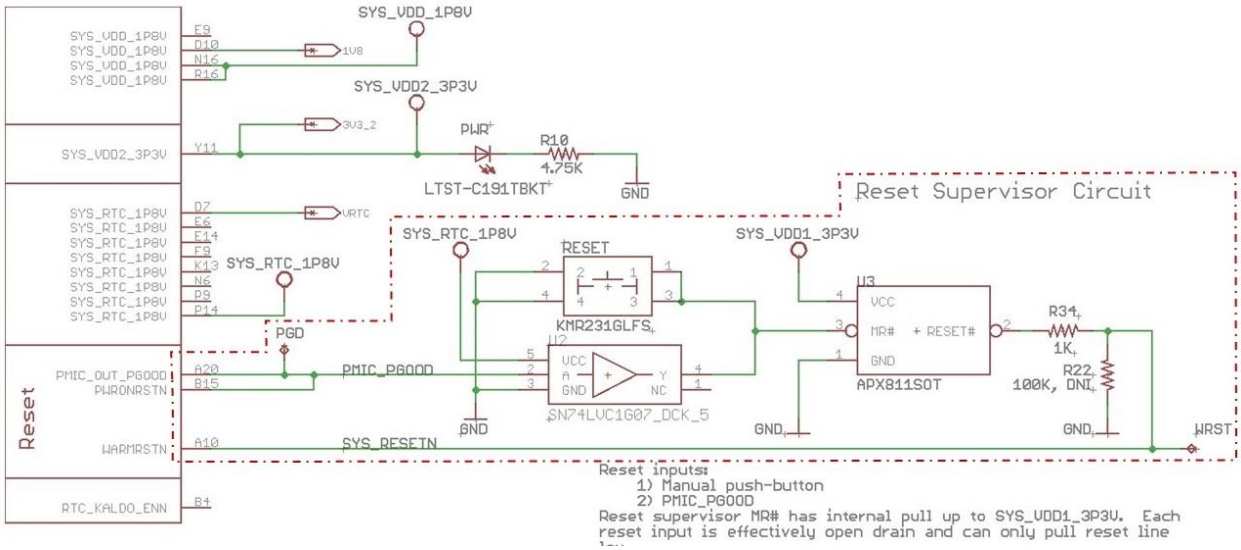


Figure 44 Reset button and buffer with APX811 supervisor circuit

The connections are made as shown in Figure 44 (Schematic updates are shown using dotted lines).

Let's go layout corresponding traces to complete the reset circuit design process as shown in Figure 45.

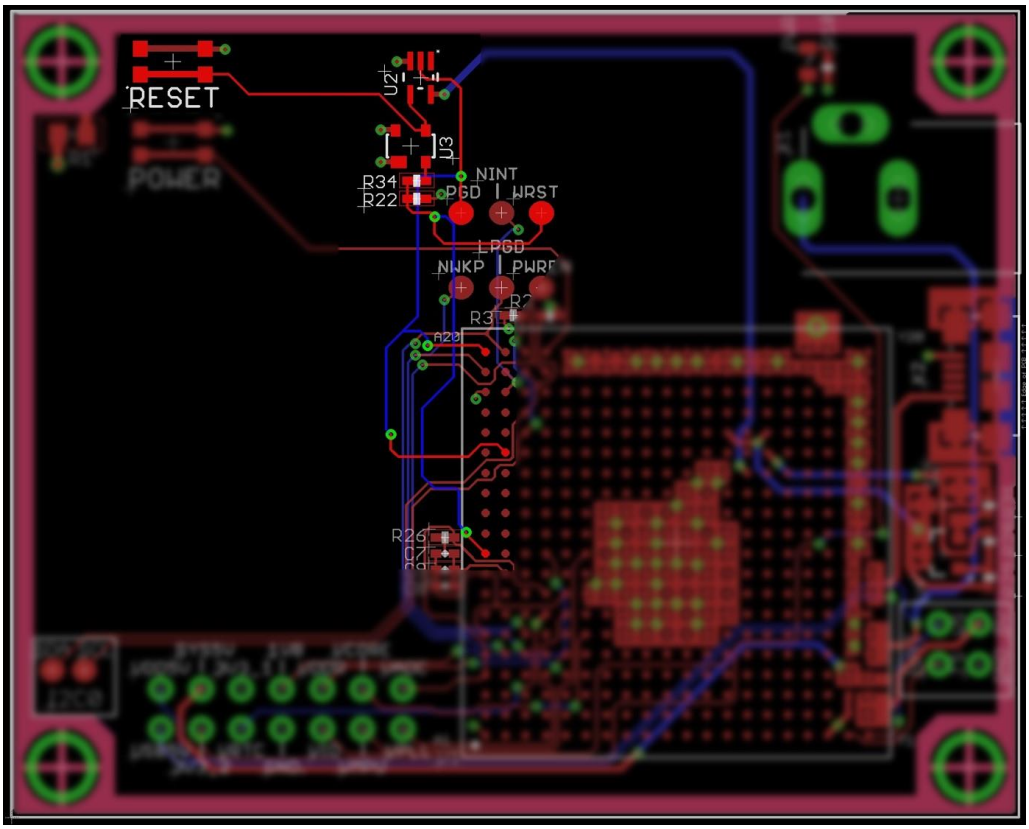


Figure 45 Reset circuitry layout

The components are placed in a specific manner in to accommodate future components and facilitate easy routing.

*****

9 OSD335x Clock Circuitry

9.1 Introduction

A clock is essential for the operation of any microprocessor. Therefore, proper design of the clock circuit is pivotal to achieve reliable operation. This article will help you understand the OSD335x Clock Circuitry design methodology.

9.2 The OSD335x OSC0 and OSC1

The OSD335x has two clock inputs. They are:

OSC0: This is the High Frequency Oscillator Input. This clock source is also called the Master Oscillator. It operates at either 19.2MHz, 24MHz ,25 MHz or 26MHz. This clock source provides reference for all non-RTC functions. The OSC0_IN, OSC0_OUT and OSC0_GND pins are used for this clock input.

OSC1: This is the Low Frequency Oscillator Input. This clock source provides a reference clock for the Real Time Clock (RTC) and operates at 32.768kHz. The OSC1_IN, OSC1_OUT and OSC1_GND pins are used for this clock input.

OSC1 is disabled by default when power is applied. This clock input is optional and is not required if the RTC (Real Time Clock) module is configured to receive clock from internal 32kHz RC oscillator or if the RTC modules is not needed at all.

For more information on clock sub-systems, you can refer the ***Clock Management*** section of the ***AM335x Technical Reference Manual***.

The crystal oscillator circuit for OSC0 is shown in Figure 46.

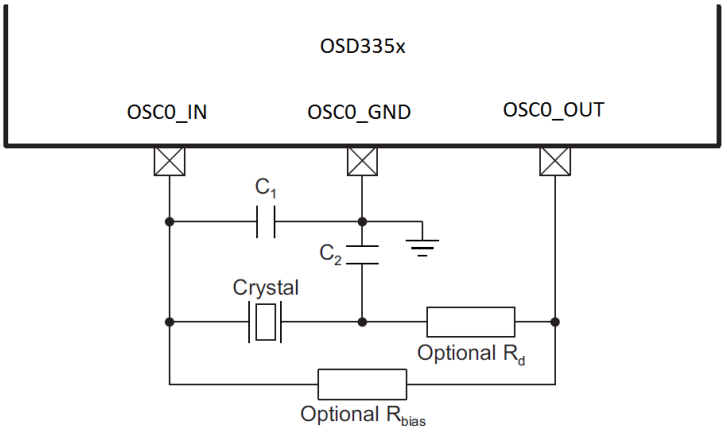


Figure 46 Crystal Oscillator circuit for OSC0

- According to the ***AM335x datasheet***, resistors R_{bias} and R_d are optional. For the reference design, we added the footprint for R_{bias} to give us the flexibility to add a resistor to the circuit if the default configuration (i.e., just C_1 and C_2 are connected)

did not provide the desired crystal performance. Additionally, R_{bias}'s footprint can be left unpopulated if it is not needed since it is in parallel with OSC0_IN and OSC0_OUT pins.

However, R_d is in series with the crystal and the OSC0_OUT pin, so it must always be populated even if it is not needed, which adds cost to the design. We decided not to add the footprint for R_d since we didn't think we needed it and it makes the layout cleaner.

For the reference design, we were happy with the performance of the crystal when we just used a 1 MOhm resistor for R_{bias}. You will have to decide how to handle R_{bias} and R_d based on your design and your crystal.

- C_1 and C_2 represent the total capacitance of the respective PCB trace, load capacitor, and other components (excluding the crystal) connected to each crystal terminal. The value of capacitors C_1 and C_2 should be selected to provide the total load capacitance, C_L, specified by the crystal manufacturer. The total load capacitance is $C_L = [(C_1 \times C_2) / (C_1 + C_2)] + C_{shunt}$, where C_{shunt} is the crystal shunt capacitance (C_0) specified by the crystal manufacturer. As long as the layout guidelines are followed, we can assume that the capacitance of the PCB trace and other components is small and can be ignored in the calculation of C_1 and C_2.
- For recommended crystal circuit component values for OSC0, check ***OSC0 Crystal Circuit Requirements*** table in the ***AM335x datasheet***.
- The AM335x supports either 19.2MHz, 24MHz,25MHz or 26MHz clock input for OSC0. However, the software we will be using in future lessons assumes a 24MHz input clock. Hence, we will be using 24MHz crystal oscillator ***7A-24.000MAAJ-T*** from TXC for OSC0. Based on the above mentioned guidelines for capacitor selection, we will be using 18pF capacitors for both C_1 and C_2 for our OSC0 design.

> Caveat:
>
> The choice of OSC0 clock frequency will affect the boot-configuration pull down/pull-up resistor setup. See the ***SYSBOOT Configuration Pins*** section of the ***AM335x Technical Reference Manual*** for more information about boot configuration resistor setup.

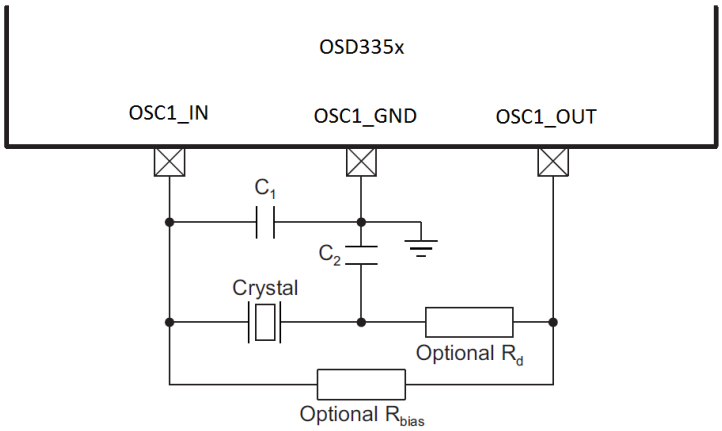


Figure 47 Crystal Oscillator circuit for OSC1

- The crystal oscillator circuit for OSC1 is shown in Figure 47 and is the same as the oscillator circuit for OSC0. For the reference design, we do not need either R_{bias} or R_d since OSC1 has an internal resistor. However, as discussed above, we have placed a footprint for R_{bias} but not for R_d to give us the flexibility to adjust the crystal performance without increasing the design cost. For recommended crystal circuit component values for OSC1, check ***OSC1 Crystal Circuit Requirements*** table in the ***AM335x datasheet***.

  OSC1 circuit operates in the same way as OSC0 circuit.

- The AM335x only supports a 32.768kHz clock input for OSC1 input. Therefore, we will be using the 32.768kHz crystal oscillator ***ABS07-32.768KHZ-T*** from Abracon LLC for OSC1 design. Based on the guidelines above for capacitor selection, we will be using 18pF capacitors for both C_1 and C_2 for our OSC1 design.

Now let's add OSC0 and OSC1 circuits into our schematics as shown in Figure 48.

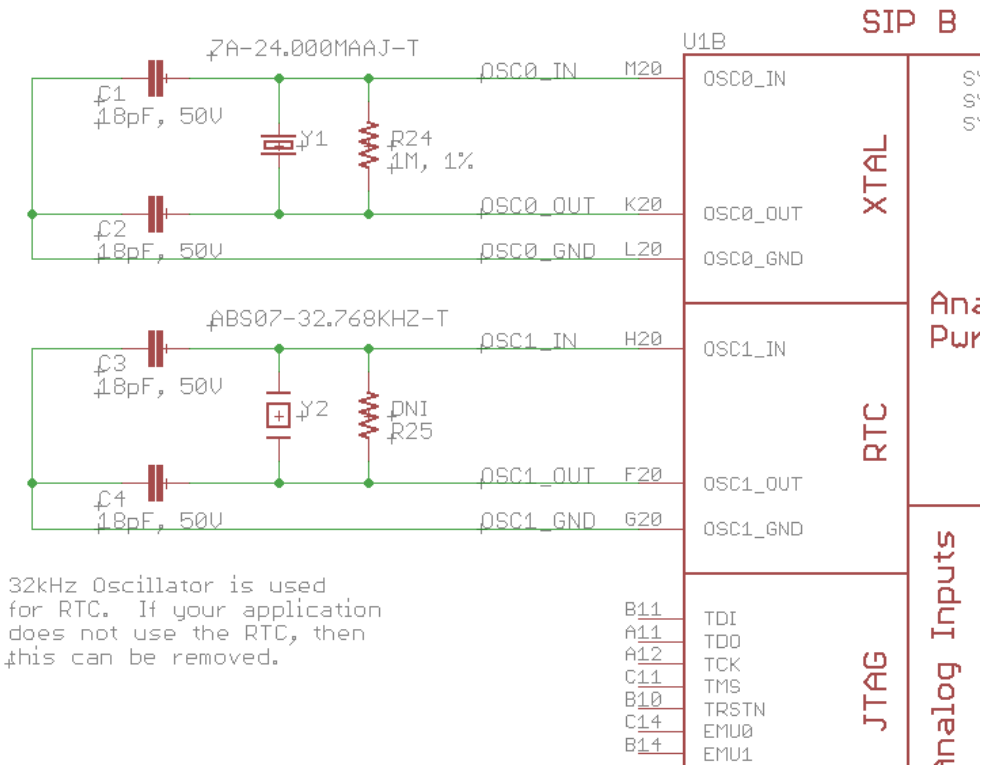


Figure 48 OSD335x Clock circuitry schematic

Perk:

OSC0 and OSC1 clock inputs can also be sourced from digital oscillator chips like ***SIT8008BCE7-18E*** (for OSC0) and ***ASDK2-32.768KHZ-LRT*** (for OSC1). Use of digital oscillators will significantly reduce the complexity of the circuit but does add cost. A discussion on digital oscillator chip is beyond the scope of this article.

9.3 Layout guidelines

Here are the guidelines we used for crystal oscillator layout in this design:

- The crystal oscillator is sensitive to noise from other signals. Other digital activities on the board may also distort the small amplitude sine wave from the crystal oscillator. Therefore, care should be taken when placing components or routing signals near the oscillator circuit to avoid capacitive coupling.
- The crystal oscillator circuit components should be placed close to the OSD335x.
- For the OSD335x, each oscillator has an oscillator ground. This should be used as the ground reference for the oscillator.
- Try not to route any other signal under the oscillator circuit wherever possible.
- If routing signals under the oscillators, then try to make sure that the signal traces cross at right angles vs running parallel to oscillator traces to minimize coupling.
- Avoid right angle traces.
- Avoid vias for clock signals if possible. If not possible, then make sure that there is enough keep out in the inner planes to not cause excessive noise on inner plane layers.
- The length of clock signal traces should be matched as much as possible.

Let's layout the traces for OSC0 and OSC1 as shown in Figure 49.

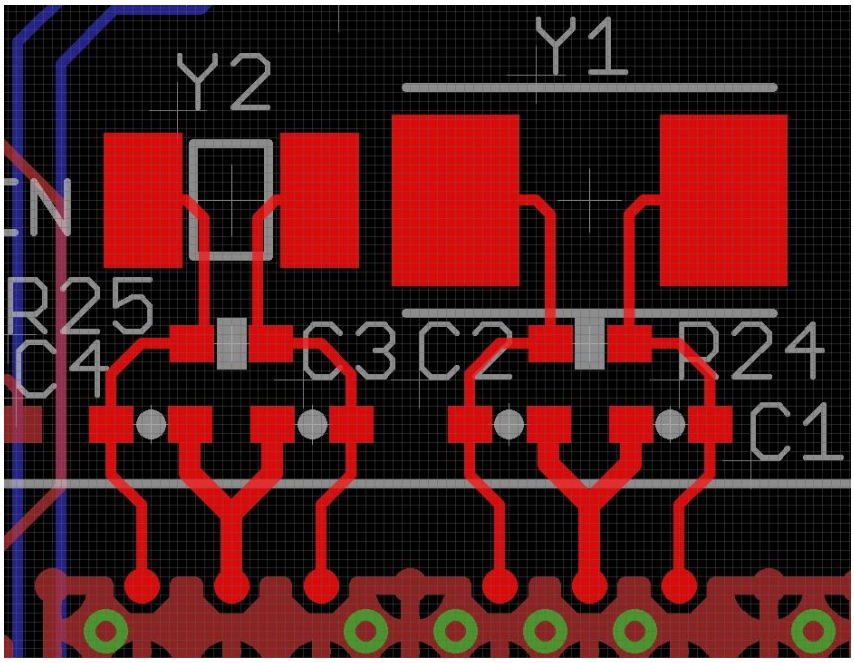


Figure 49 OSD335x OSC0 and OSC1 layout

9.4 RTC_KALDO_ENN

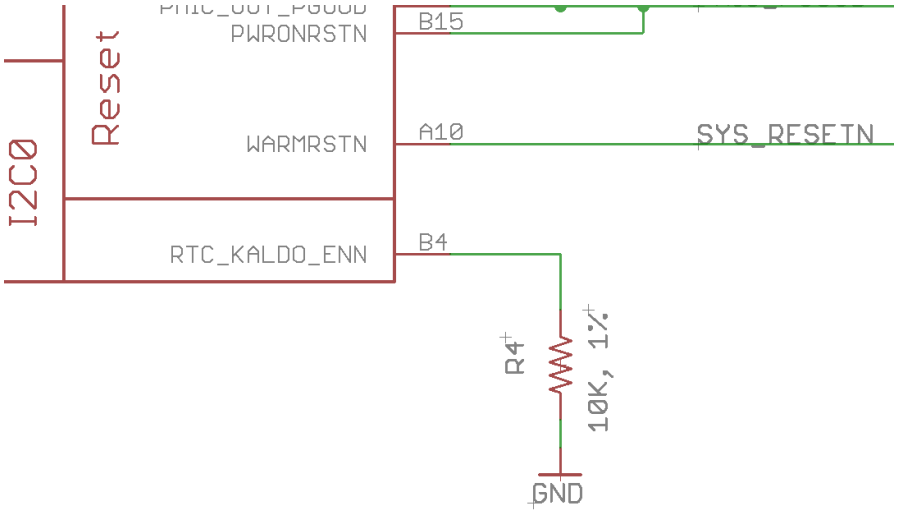


Figure 50 Grounding RTC_KALDO_ENN pin

The AM335x processor inside the OSD335x consists of an RTC (Real Time Clock) with the potential to support an RTC-Only mode. However, the OSD335x **DOES NOT** support RTC-Only mode because we have used the ***C version of TPS65217*** which ***does not support RTC-Only power mode***. Although RTC-Only mode is not available, we would like to enable and use RTC digital core along with other modules of the processor to keep track of time.

The RTC digital core has an internal RTC LDO which can supply power to it. CAP_VDD_RTC (the supply pin for the RTC core) gets power from the internal RTC LDO if the RTC_KALDO_ENn (active low) pin is pulled low as shown in Figure 50. If RTC_KALDO_ENn is pulled high, the internal RTC LDO will be disabled and CAP_VDD_RTC will have to be connected to VDD_CORE to power the RTC core externally.

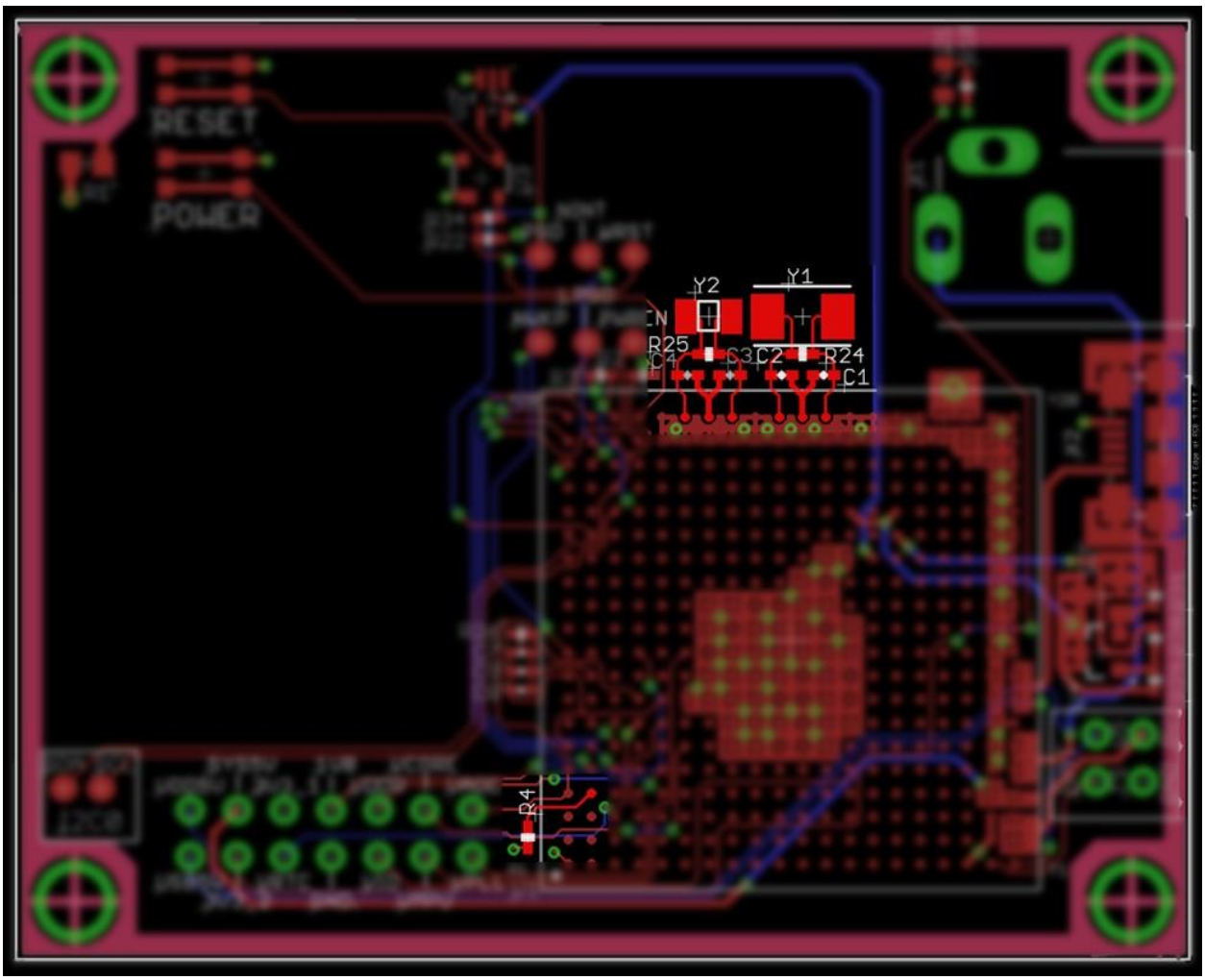


Figure 51 OSD335x clock circuitry layout

On completion of clock circuitry layout, the board should look like Figure 51 (assuming the board is also populated with power and reset circuitry from previous articles).

*****

10 OSD335x Peripheral Circuitry

10.1 Introduction

Now that the power, reset and clocks are all connected, we need to add the ability to program the OSD335x to make our design useful. To program the OSD335x, we are going to use the JTAG connection to the processor (later lessons will look at other boot devices). To enable us to do some fun things with our design, we will add a couple of general purpose LEDs, switches, and a multipurpose peripheral header. The header will allow us to connect daughter boards to extend functionality.

10.2 JTAG

For this design, we do not include any non-volatile storage to allow us to boot and run an OS by default. Therefore, all programs will need to be loaded via the JTAG interface.

> Perk:
>
> More information about JTAG interface can be found at ***here***.

The JTAG circuit can be built on the schematic as shown in Figure 52. This uses a standard 20-pin connector and supports most JTAG debuggers. The JTAG header can be found in the ***given library*** under the device name ***CTI-JTAG***.

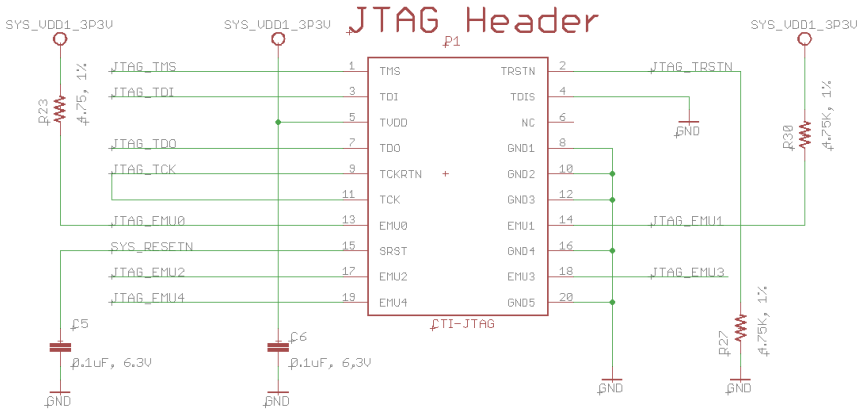


Figure 52 JTAG schematics

The JTAG connections can be made in the layout as shown in Figure 53.

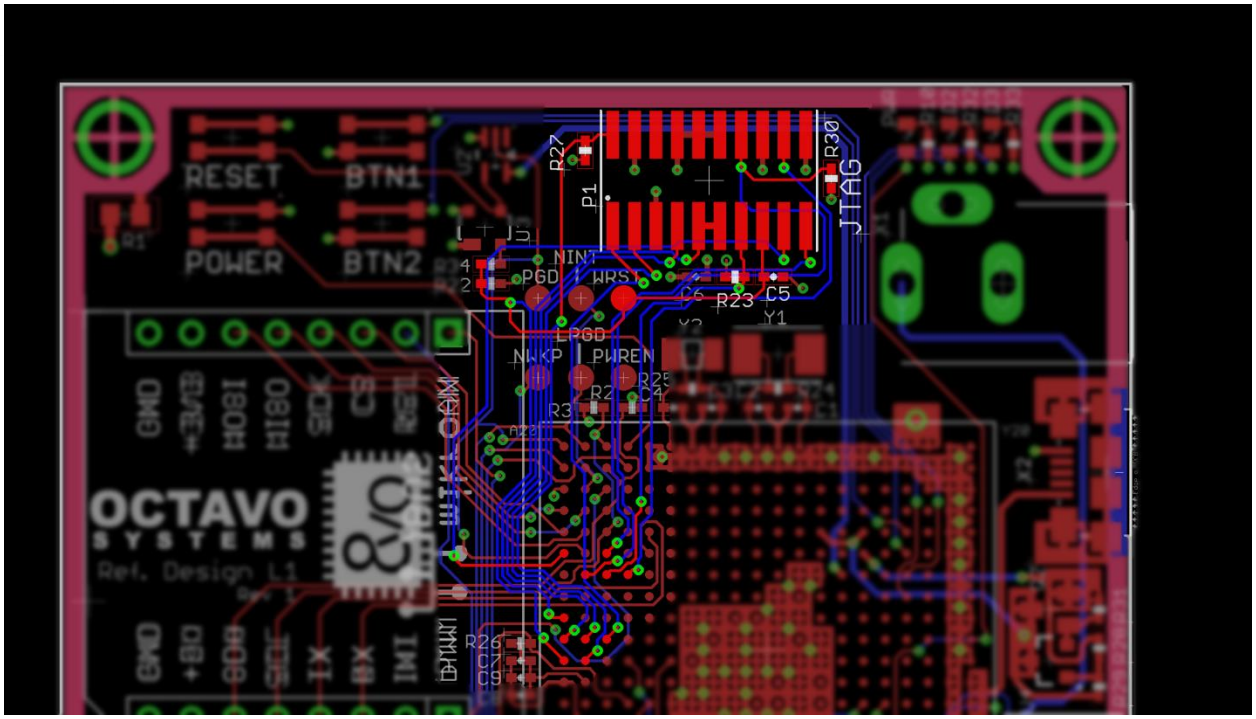


Figure 53 JTAG layout

10.3 Boot configuration

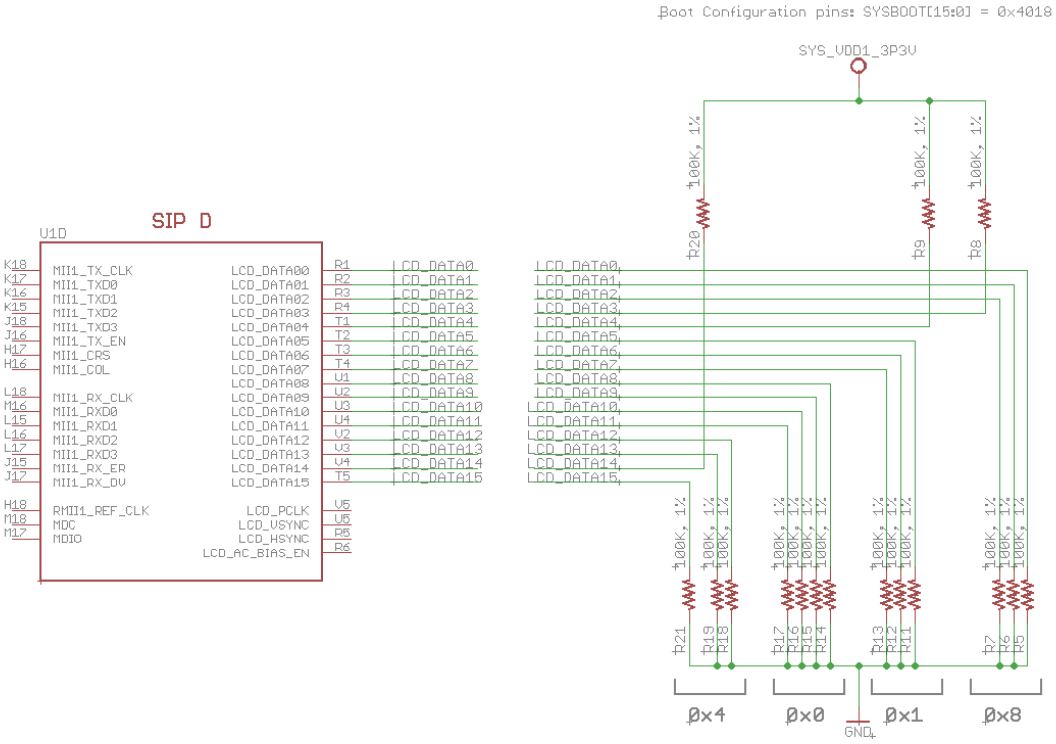


Figure 54 Boot configuration schematic

LCD_DATA0 to LCD_DATA15 pins are multiplexed with the SYSBOOT boot configuration pins of the OSD335x. More information about the function of each of these pins can be found in ***SYSBOOT Configuration Pins*** section of the ***AM335x Technical Reference Manual (TRM)***. SYSBOOT[0] corresponds to LCD_DATA0 while SYSBOOT[15] corresponds to LCD_DATA15.

Using the boot configuration pins for this design, we will:

- Set clock frequency to 24MHz.
- Disable CLKOUT1 output through XDMA_EVENT_INTR0 since this pin will only be used for JTAG emulation.
- Set the boot sequence to SPI0 -> MMC0 -> USB0 -> UART0

By default, if no valid boot images are found during the boot sequence, then JTAG can take control of the processor to allow software to be loaded.

To achieve this, we need to set SYSBOOT[15:0] = 0x4018. We can build the schematic for the boot configuration as shown in Figure 54.

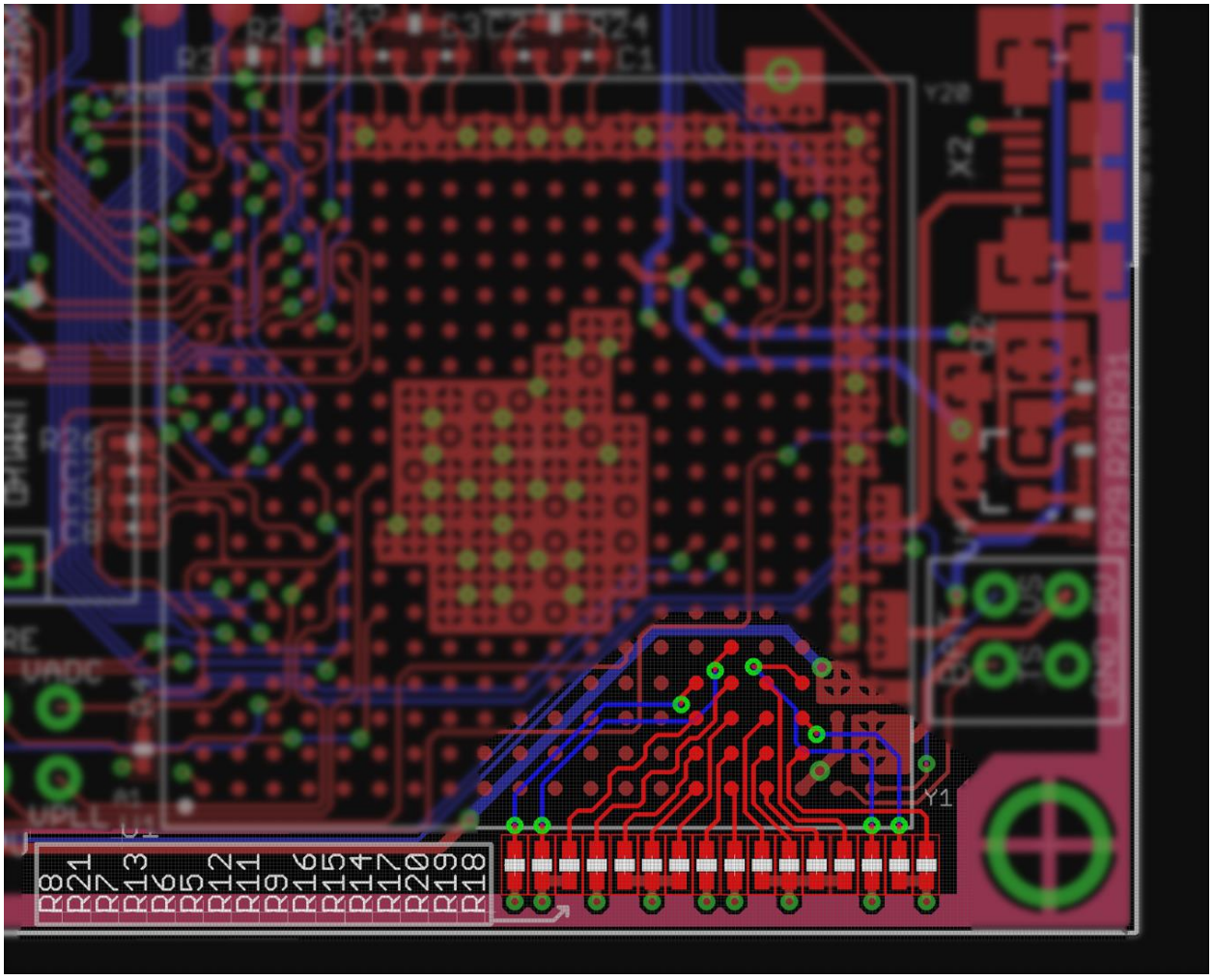


Figure 55 Boot configuration layout

The boot configuration connections in the layout are highlighted in Figure 55.

10.4 Buttons and LEDs

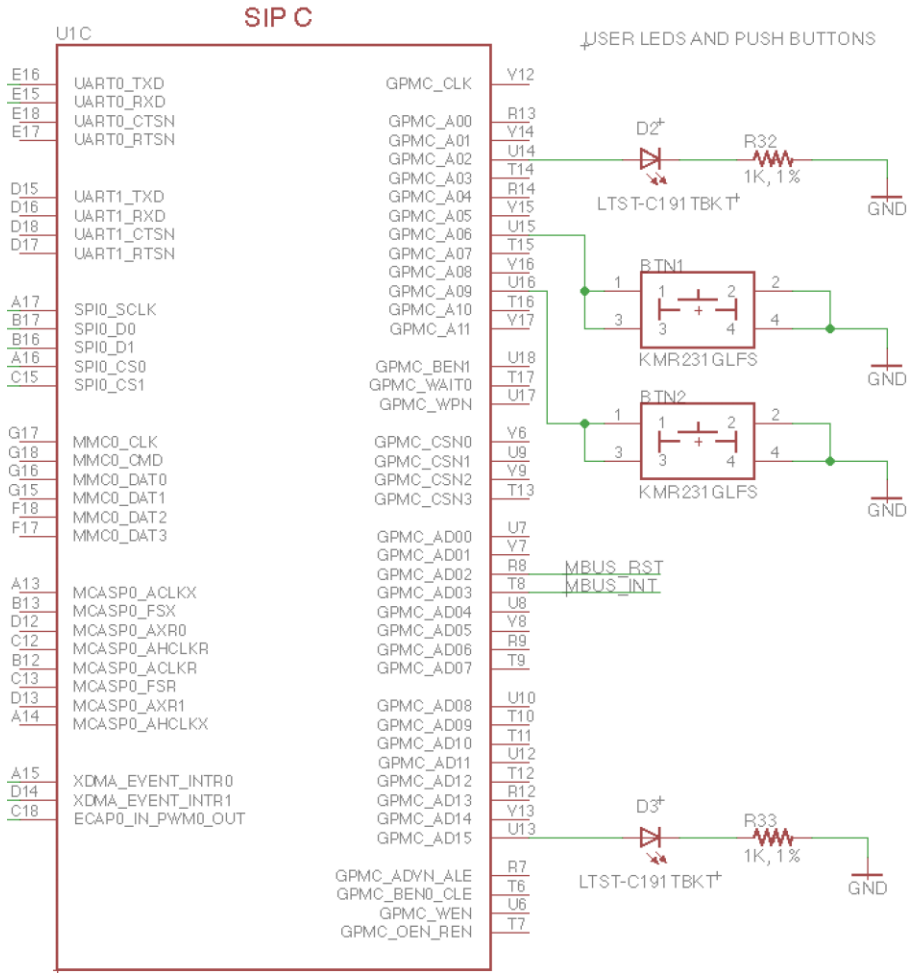


Figure 56 User LEDs and user buttons

For this design, we will add two push buttons and two LEDs. The buttons can be connected to GPIOs GPMC_A06 and GPMC_A09. These two pins were chosen to make routing easier. There was no de-bounce circuitry added to the buttons so any de-bouncing must be done in software.

Also, let's add two LEDs, one to GPIO GPMC_A02 and the other to a PWM capable GPIO GPMC_AD15 so that we can exploit the PWM capability of the OSD335x. The schematic for these connections is shown in Figure 56.

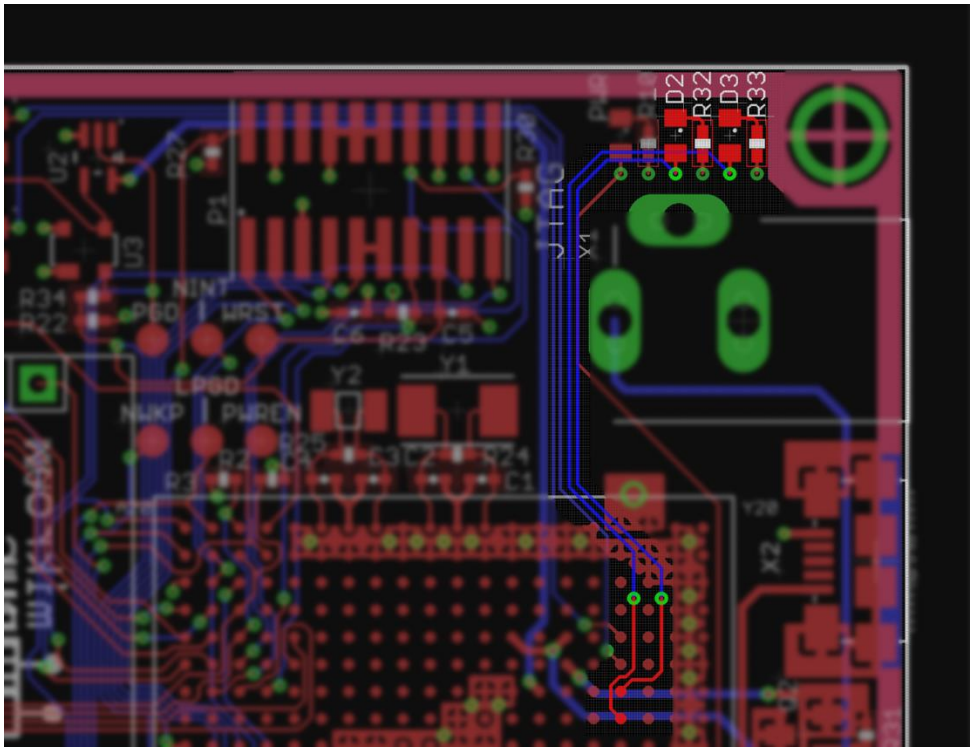


Figure 57 User LEDs layout

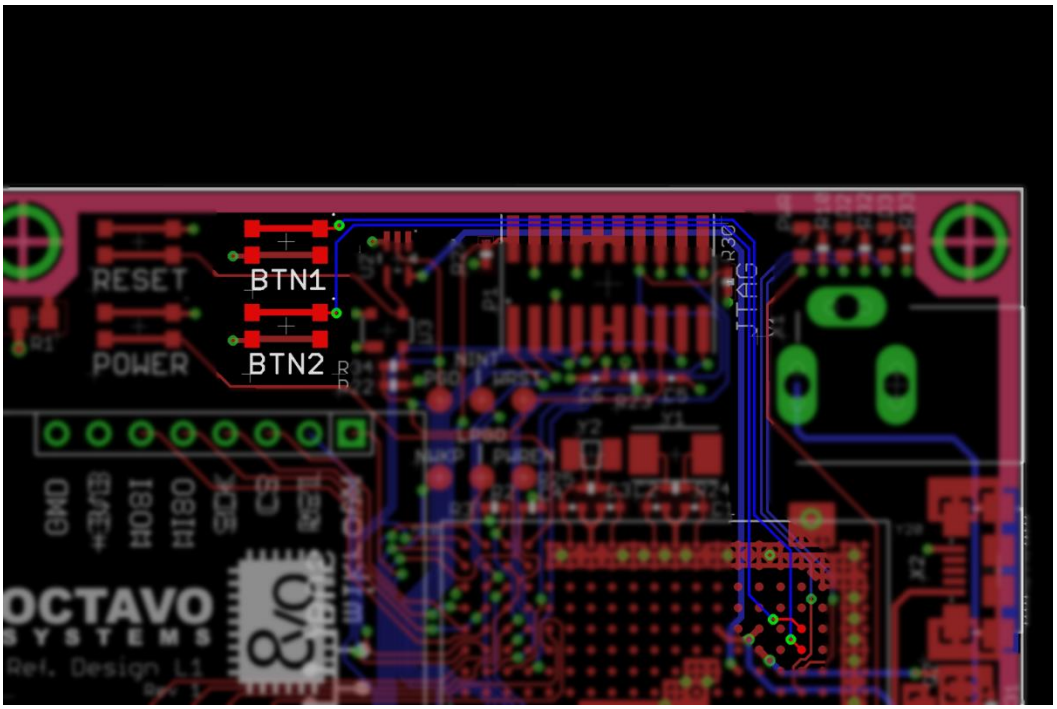


Figure 58 User buttons layout

The layout connections for user LEDs and user buttons can be made as shown in Figure 57 and Figure 58.

10.5 Peripheral header

The OSD335x supports several serial communication protocols like UART, SPI and I2C. These protocols can be used to communicate with and control external devices.

We can map these signals to a standard peripheral header to provide expandability and additional functionality.

The peripheral header we're using consists of a pair of 1x8 female headers with pin configuration as shown in Figure 59. In the ***given library***, the peripheral header can be found under the device name ***MIKROE_CLICKMINI (MIKROE_CLICK)***. This configuration is compatible with mikroBus® Click Board™. Click Boards are daughter boards with a single IC, module or circuit that brings specific functionality to a target main board. Hundreds of click boards are available with various types of sensors, transceivers and functionality.

More description about mikroBus socket and Click boards can be found at:

https://www.mikroe.com/mikrobus/

You can browse through available click boards at:

https://shop.mikroe.com/click.

The peripheral header schematic is given below:

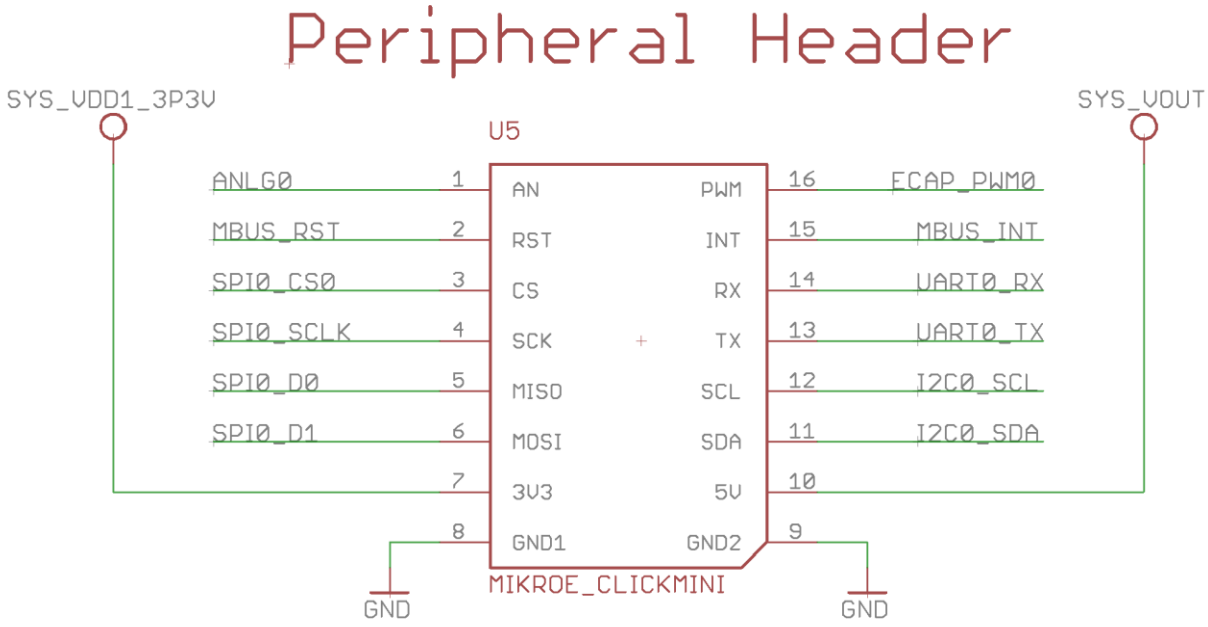


Figure 59 Peripheral header schematic

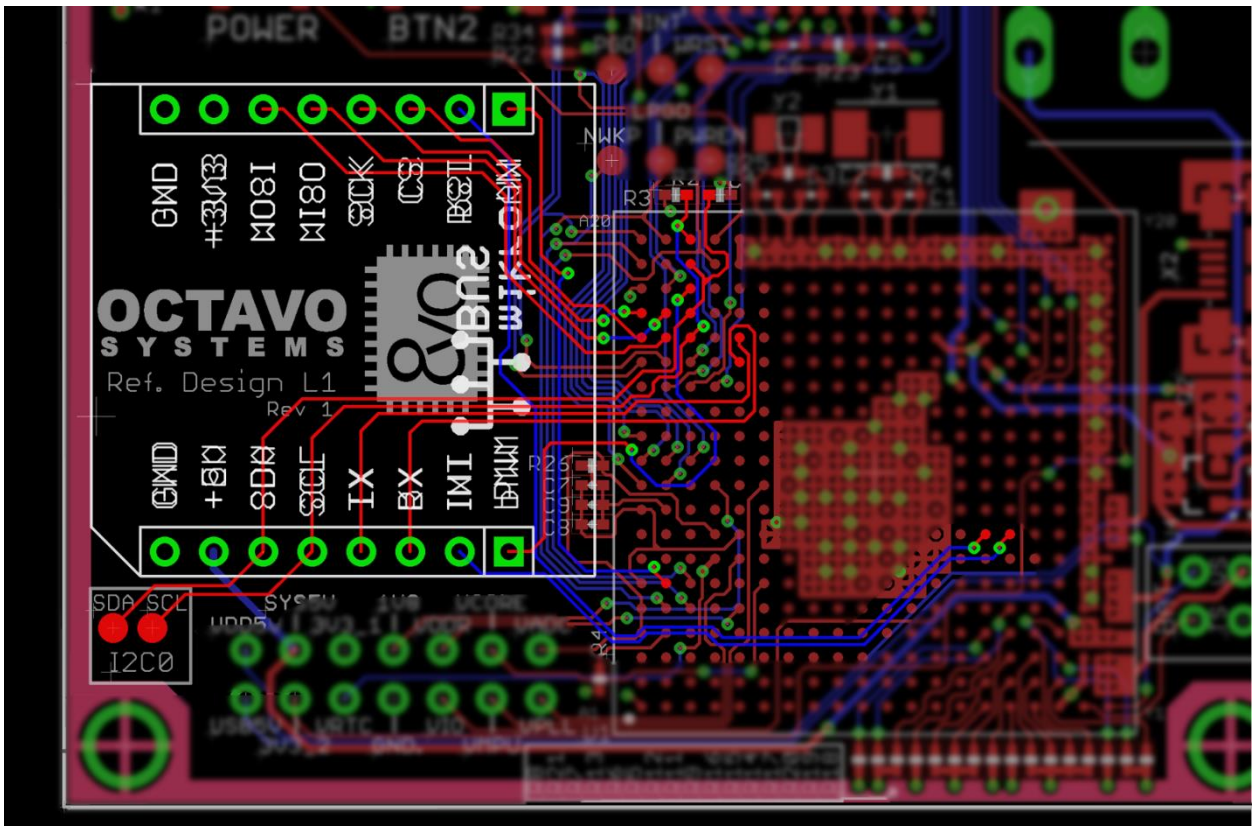


Figure 60 Peripheral header layout

The footprint for peripheral header is shown in Figure 60 and the connections are highlighted.

10.6 Finalizing the silkscreen

Silkscreen on a PCB is used to identify components, test points, warning symbols, certification symbols, company logos, PCB name and revision, and any other text that is necessary to document the functionality of the board. During the placement and routing phases of the board design, you should not really worry about the silkscreen. However, now that we have completed placement and routing of all the components on the PCB, we need to make sure that the silkscreen is readable. We need to make sure:

- All the component designators are placed appropriately next to their corresponding components.
- Pin 1 is marked for necessary components. This includes all ICs, connectors and polarized capacitors.
- There is no silkscreen over any component pads. This can lead to bad solder joints.
- The font size of the component designators is as large as possible given the space constraints of the board so that the text is readable.

- Try to not place silkscreen over vias. Silkscreen has a hard time adhering to the annular ring of a via which can make the text difficult to read.

When designing a board, the actual size of the text can be deceiving since the view in the design tool is generally many times larger than the actual board. The minimum font size you should use is 24mils with 8% ratio. Depending on your eyes, a font size of 32mils with an 8% ratio is readable without magnification. This is a good size for passive designators, though in general bigger is better. You can also increase the ratio of the text to make the line width thicker. This can help make the text more readable but only up to a point. However, once you get larger fonts (over 50mils), you should use larger ratios (12% or greater) to make the text more readable. It is recommended that once you are done with updating the silkscreen, print out the silkscreen layers to check their readability.

You can also import a picture of your logo into Eagle and add it into your layout. You can learn more about the procedure to import images ***here***. For our design, we have added ***Octavo Systems*** logo along with the lesson name and revision number as shown in Figure 62, Figure 63 and Figure 64.

10.7 Expected outcome

Now that we have completed building the schematic and layout, the complete schematic should look similar to Figure 61 and complete layout should look similar to Figure 62, Figure 63 and Figure 64.

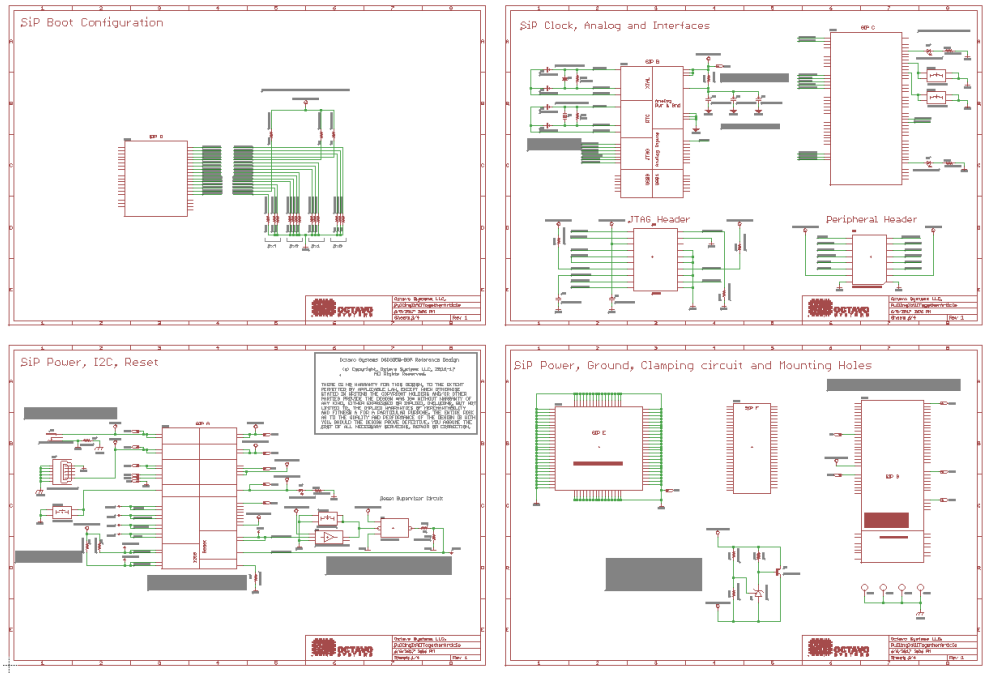


Figure 61 Lesson 1 complete schematic

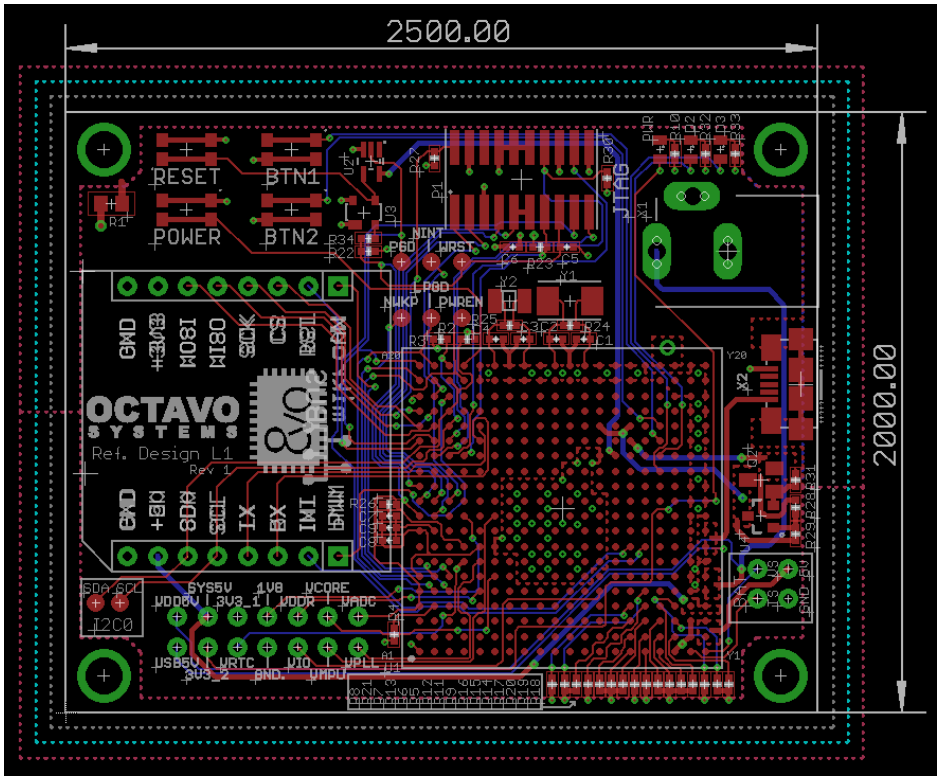


Figure 62 Lesson 1 complete layout with pour outlines

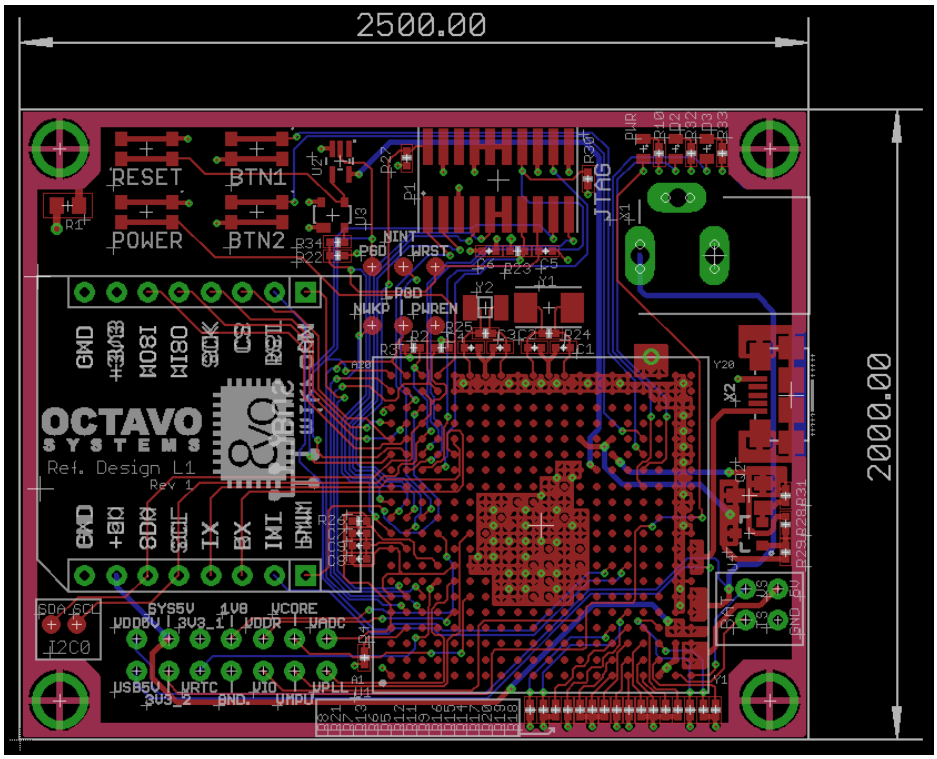


Figure 63 Lesson 1 complete layout with pour

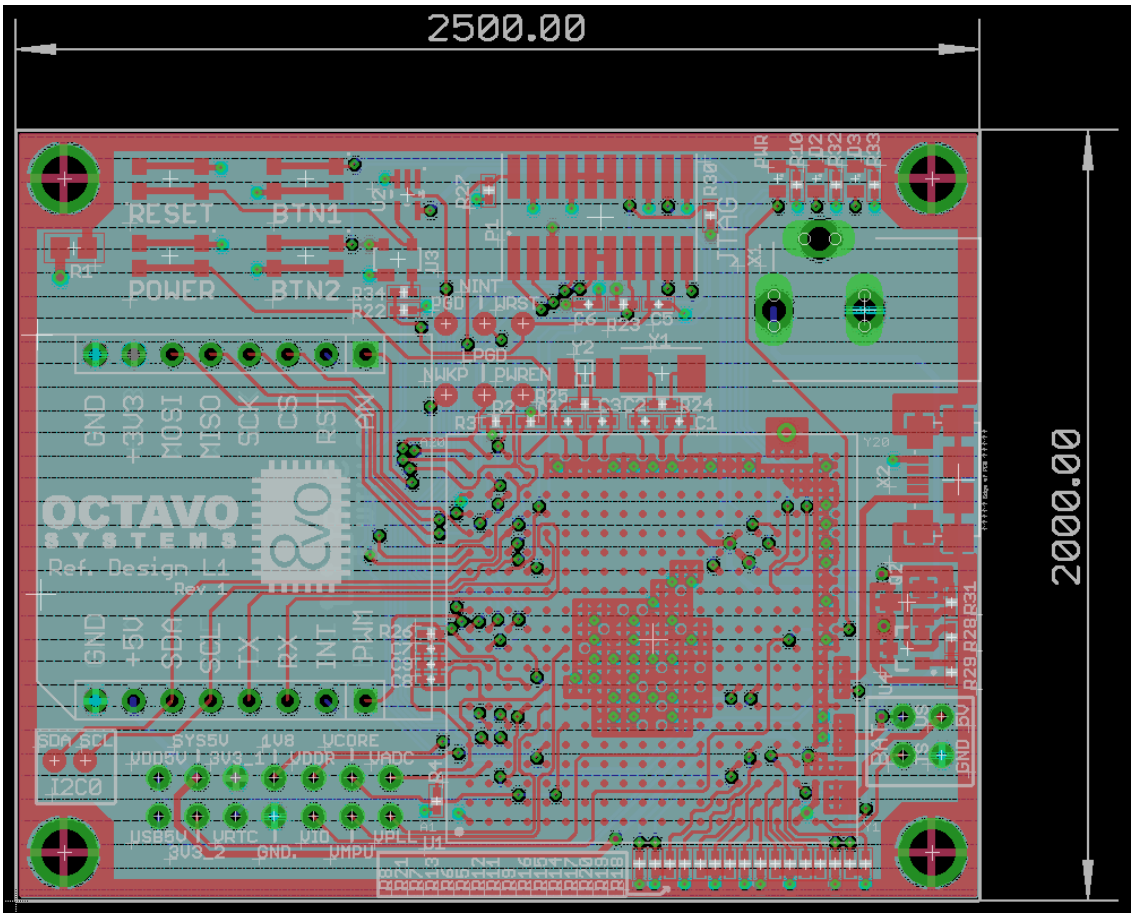


Figure 64 Lesson 1 complete layout with all layers turned on

10.8 PCB order process

The PCB order process begins with Bill Of Material (BOM) generation. EAGLE is capable of maintaining manufacturer and part number for each component so the schematic can be the controlled document for manufacturing information. The BOM.ulp can be used to output this data in an easily readable format. Once BOM is generated, the components can be purchased.

The next step is gerber file generation for PCB manufacturing. More information on gerber file generation, gerber verification and order placement can be found ***here***.

*****

11 OSD335x Bare Minimum Board Boot Process

11.1 Introduction

This document will take you through the bring-up process of the Printed Circuit Board (PCB) developed as part of the OSD335x Reference Design Lesson 1.

We begin with bringing-up the newly manufactured board (PCB) to make sure it is functional followed by the necessary software environment setup. We conclude with the demo applications to verify the overall functionality of the board.

11.2 The Board (PCB)

The board built as part of previous articles of this lesson should look similar to Figure 65 (Assuming you chose Red color for solder mask and white color for silk screen):

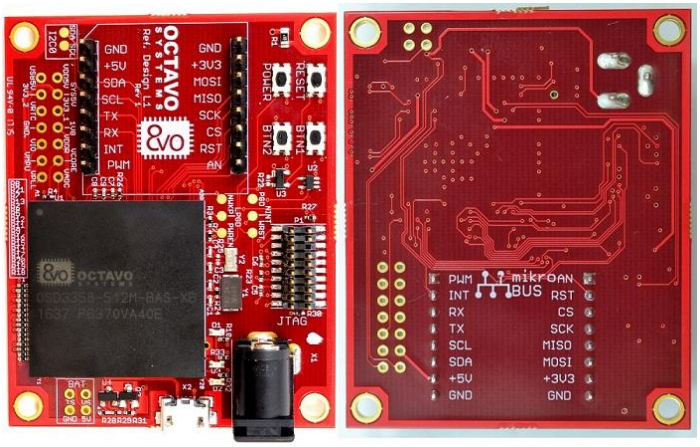


Figure 65 OSD335x Bare Minimum Board

11.3 Basic board bring-up

This section describes the various tests that need to be done during the board bring-up process.

11.3.1 Tests before board power-up

The newly manufactured PCB should be examined before powering it up for the first time to prevent damage to the board.

- Use a Digital Multi Meter (DMM) to make sure there are no shorts (resistance<=100ohms) between:
  - Power input pins and ground.
  - Power output pins and ground.
  - Between power inputs and power outputs - This is where all the test points can be very helpful.

- If there are any shorts, try to locate the source of the short. This can be done by:
    - Examining schematics and layout.
    - In many cases, the only way to isolate shorts is to remove components connected to the short one by one until the short is resolved.
    - Once you have discovered the source of the problem, you can try to modify the board to fix the issue. This can involve cutting traces or re-wiring components. Unfortunately, sometimes boards cannot be fixed and the design must be re-spun (i.e., schematics and layout updated and boards re-manufactured). If you have to re-spin a board, please make sure to update the revision code on the board.

- Use a DMM to check the resistance between:
    - All power inputs, power outputs and ground.
    - Between power planes and ground.
    - In any case, if the resistance is less than 100ohms, you need to be very careful during power-up to make sure no components have issues.

- Check if the correct components are used by reading the marking on the devices.
- Check if the orientations of the components are correct.

After performing all the above tests, we can power up the board using the DC barrel jack. If everything goes well, you should see the power LED ***PWR*** light up as shown in Figure 66.

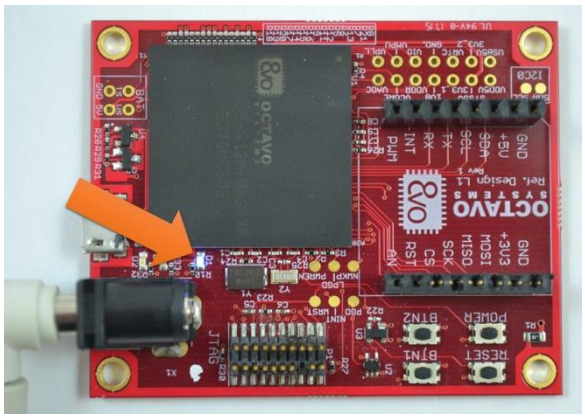


Figure 66 PWR LED after successful power up

Perk:

More information about using Digital Multi Meters can be found ***here***.

Caveat:

Boards should only be powered up for the first time in a good work environment with proper safety equipment like fire extinguisher, non-flammable ESD workbench, safety glasses, fume extractor (to remove soldering smoke) etc. Please make sure your work environment is proper before doing any board bring-up work.

11.3.2 Possible problems after board power-up

Once the board is powered up, if you observe sparking, overheating or smoke from any of the components, TURN OFF POWER TO THE BOARD IMMEDIATELY to avoid further damage to the board and prevent a possible fire hazard. For any component that failed, please make sure:

- The symbol pinout of the component in schematic matches with its datasheet pinout. Please make sure that you have the latest version of the datasheet since documentation can be updated / corrected between the time of symbol creation and board manufacture.
- The footprint of the component on PCB matches with the footprint specified on its datasheet.
- The symbol to footprint pin mapping is correct.
- All the components are supplying power within their specified output current limits. For example, if you notice that the voltage of one of the power outputs of the SiP, let's say, SYS_VDD1_3P3V, is low (less than 3.3V), there is a possibility that too much current is being drawn from it. Make sure the current output of SYS_VDD1_3P3V is within its maximum value to restore its output voltage to 3.3V.
- If possible, try to solve the problem by manually cutting suitable traces on the PCB/making connections using a thin wire. If the problem cannot be solved using these methods, you will have to re-spin the board.

11.3.3 Tests after power-up

If the power LED is lit and you don't observe any smoke, sparking or heating issues then we can consider it a successful power-up. But, this does not mean the board is fully functional. We can test for functionality by running demo applications. Before running the demo applications:

- Check the following voltage levels using the available test points/pads:

- Verify if VIN_AC is at 5V (assuming you're using DC barrel connector for power input).
- Verify if SYS_VOUT is at 5V (assuming you're using a 5V power input).
- Verify if SYS_VDD1_3P3V is at 3.3V.
- Verify if SYS_RTC_1P8V is at 1.8.
- Verify if the test pad PGD (PMIC_OUT_PGOOD) is at 1.8V and WRST (WARMRSTN) is at 3.3V to make sure the OSD335x is not being held in reset.

If any of the above voltages are not at the desired level, please use the test points/test pads with an oscilloscope and DMM to find out which component(s) is responsible for the erroneous voltage. One thing to be aware of is that the voltage on a power rail or signal pin can drop if the load is trying to draw more current than the source can provide.

If you've made it this far through the article and if everything looks good on your board, pat yourself on the back. Good job! Now you're ready to run some code!

11.4 Setting up software environment (for Windows 7,8 and 10 OS)

11.4.1 Installing Code Composer Studio

For our design, we will use the Code Composer Studio (CCS) Integrated Development Environment (IDE) to compile, debug and load programs to the OSD335x. The OSD335x uses the AM335x processor from Texas Instruments (TI). Therefore, for the AM335x, we will use the IDE developed and supported by TI. You can use third party IDEs and compilers if necessary.

Steps to install Code Composer Studio:

- You can download the ***CCS 7.2.0.00013*** (which was the latest version at the time of writing this article) installer from ***this page***. The below mentioned steps are for the offline installer. All the demo applications have been tested on ***CCS 7.2.0.00013*** and ***CCS 6.2.0.00050***. You may use ***CCS 6.2.0.00050*** if you face compatibility issues with ***CCS 7.2.0.00013***.
- Midway through the installation process, the installer will ask you to ***Select Product Families to be installed***. Choose ***Sitara*** as shown in Figure 67 and click next (if you plan to use CCS for other TI processors, you should select to install those now; else you will have to completely re-install CCS if you need to add support for other processors).

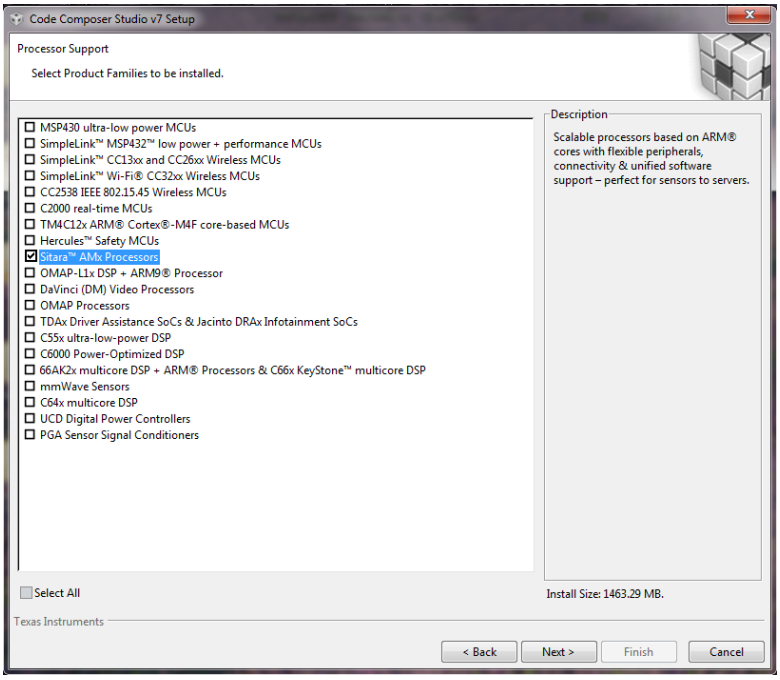


Figure 67 CCS Product family selection

- We are using XDS110 debug probe. Therefore, we will choose ***XDS*** under ***debug probes***, as shown in Figure 68 and hit next. If you are using a different

debugger, please select and install those instead. Follow the onscreen instructions to complete the installation.

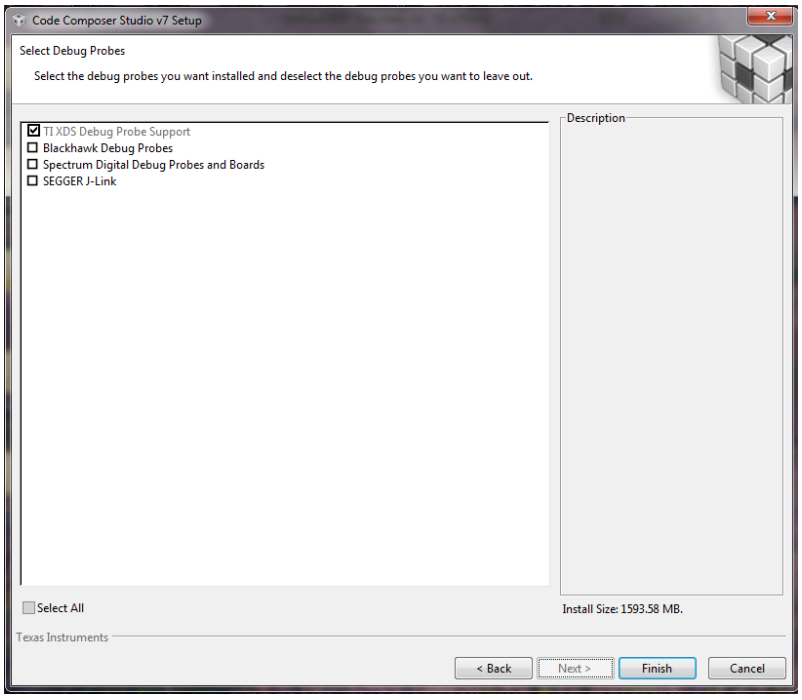


Figure 68 CCS debug probe selection

11.4.2 Installing StarterWare

StarterWare is a free software development package that provides ***bare metal*** (non-OS) platform support for ARM and DSP TI processors. StarterWare includes Device Abstraction Layer (DAL) libraries and example applications that demonstrate the capabilities of the peripherals on the TI processors. StarterWare also provides pre-built binaries for quick evaluations on the target. To keep things simple and to avoid using an OS like Linux, we will be using StarterWare platform for this lesson. Linux will be introduced in the next lesson.

To install StarterWare:

- Go to ***http://processors.wiki.ti.com/index.php/StarterWare.***
- Download StarterWare 02.00.01.01 for the AM335x (TI Account Login required. You can download the software only if TI approves it).
- Install StarterWare 02.00.01.01 using on screen instructions.

11.4.3 Debugger

To load the program to the OSD335x, we need a debugger. We have tested the demo applications using XDS110 from Spectrum Digital (shown in Figure 69). You are free to use other debuggers if you have them. We have only validated operation on the XDS110 which you can get ***here***.

Figure 69 XDS110 Debugger

11.5 Demo Applications

11.5.1 Demo Application 1: LED Dimmer

The objective of this application is to demonstrate the use of the GPIO and EHRPWM peripherals within the OSD335x.

The Demo Application 1 generates a hardware PWM signal on the EHRPWM1A(GPIO1_18) pin which is connected to LED D2 on the board and a software PWM signal on the GPIO1_15 pin which is connected to LED D3 on the board. The brightness of each of these LEDs will be controlled by their respective PWM duty cycles and will be set to the maximum value (i.e., always on) in the beginning.

The duty cycle of the hardware PWM signal for LED D2 will be controlled by button BTN1. When the button is first pressed and held, the duty cycle will slowly decrease, which will cause the LED to dim. This will continue until the minimum duty cycle is reached, and the LED will be off. If you continue to press the button, the duty cycle will then increase, which will cause the LED to brighten back up to the maximum duty cycle. This process repeats itself as long as BTN1 is pressed. The duty cycle of the software PWM signal for LED D3 will be controlled by button BTN2. The behavior of the PWM duty cycle controlled by BTN 2 is same as that of BTN1.

All the CCS design files required for Demo Application 1 and Demo Application 2 can be found ***here***.

11.5.2 Running Demo Application 1

- Open CCS.
- Create a new CCS Project (In Project Explorer, right click > New >CCS Project)
- Name the project as ***RefDesL1Demo1***. Configure the project as shown in Figure 70 and hit ***Finish***.

  Target: AM3358.

Connection: Texas Instruments XDS110 USB Debug probe.

Compiler version: TI v16.9.3.LTS or higher.

Project template: Empty Project (with main.c).

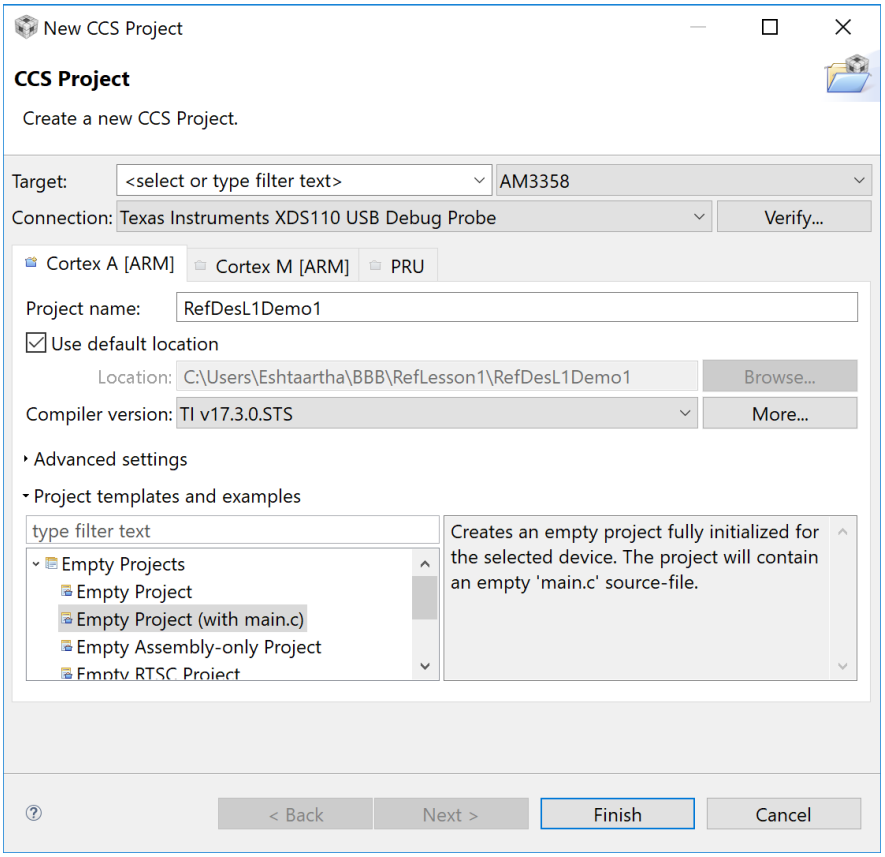


Figure 70 Demo Application 1 project settings

The compiler will need access to many different StarterWare folders and libraries in order to compile our code. Add the following paths to the ***ARM Compiler's Include Options*** as shown in Figure 71 (To find Compiler Include Options, right click on Project Name in Project Explorer > Properties > CCS Build > ARM Compiler > Include Options. The below paths are given assuming you have installed StarterWare at C:\ti. If not, alter the path suitably):

- *C:\ti\AM335X_StarterWare_02_00_01_01\include*
- *C:\ti\AM335X_StarterWare_02_00_01_01\include\armv7a*
- *C:\ti\AM335X_StarterWare_02_00_01_01\include\armv7a\am335x*
- *C:\ti\AM335X_StarterWare_02_00_01_01\include\hw*

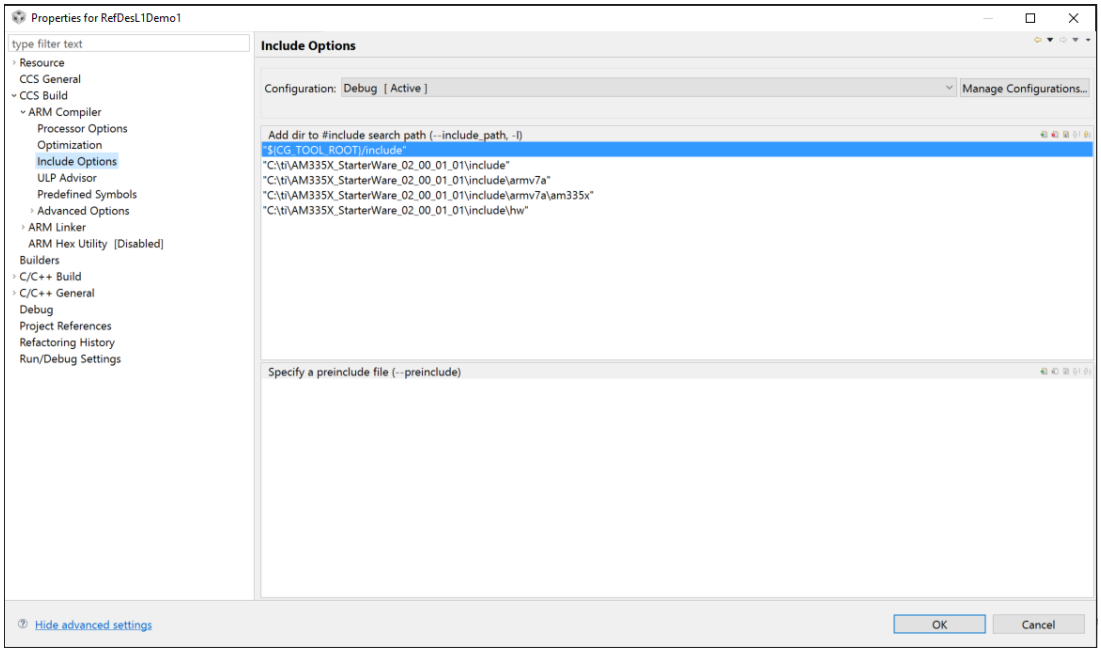


Figure 71 Compiler Include Paths

The linker will need access to many different StarterWare folders and libraries in order to build our code. Add the following paths to the ***ARM Linker's File Search Path*** as shown in Figure 72 (To find Linker File Search Path, right click on Project Name in Project Explorer > Properties > CCS Build > ARM Linker > File Search Path. The below paths are given assuming you have installed StarterWare at C:\ti.If not, alter the path suitably):

- *C:\ti\AM335X_StarterWare_02_00_01_01\binary\armv7a\cgt_ccs\utils/${ConfigName}\utils.lib*
- *C:\ti\AM335X_StarterWare_02_00_01_01\binary\armv7a\cgt_ccs\am335x\drivers/${ConfigName}\drivers.lib*
- *C:\ti\AM335X_StarterWare_02_00_01_01\binary\armv7a\cgt_ccs\am335x\system_config/${ConfigName}\system.lib*
- *C:\ti\AM335X_StarterWare_02_00_01_01\binary\armv7a\cgt_ccs\am335x\evmskAM335x\platform/${ConfigName}\platform.lib*

Caveat:

Make sure you use forward slash (/) with the project macro ${ConfigName} as ***/${ConfigName}*** in the paths above. Otherwise, CCS will not resolve ${ConfigName} into its value.

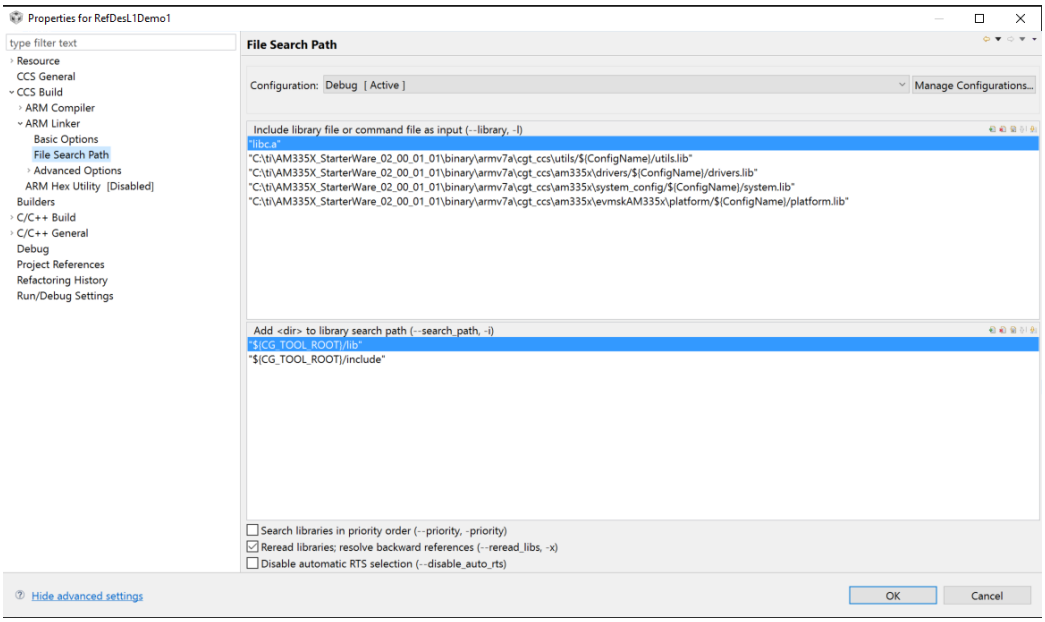


Figure 72 Linker Search Paths

- Delete ***main.c*** from your project. Download ***RefDesL1Demo1.c*** from and add it to the project.
- The default linker command file, ***AM335x.cmd*** and the given startup code file, ***init.asm*** that CCS provides are not suitable for our Demo Applications because of the following reasons:
  - The default linker command file uses the default startup code which sets the processor to ***User Mode*** before calling ***main()*** function. But, the processor needs to be in ***Privilege Mode/System Mode*** to be able to write to GPIO control module registers for the GPIO or EHRPWM registers as part of our demonstration projects.
  - The given startup code file, ***init.asm***, is configured to operate using external DDR3 with 4KB of stack assigned for interrupt requests. While the DDR3 memory exists in the OSD335x, we do not have the DDR initialization code in the demonstration program and therefore need to run the program using only the AM335x internal memory. Given that we are using internal memory, we will have to reduce interrupt request stack size to 256B.

    To solve the above problems, Octavo Systems provides two files: ***AM335x.cmd*** and init.asm, which make CCS use custom startup code before calling ***main()*** function. The custom startup code sets the processor to ***System Mode*** before calling ***main()*** function. It also reduces the interrupt request stack size to 256B.

    Delete the default ***AM335x.cmd*** file under your project files and replace it with the modified ***AM335x.cmd*** file (provided as part of CCS design files. Link given above) as shown in Figure 73.

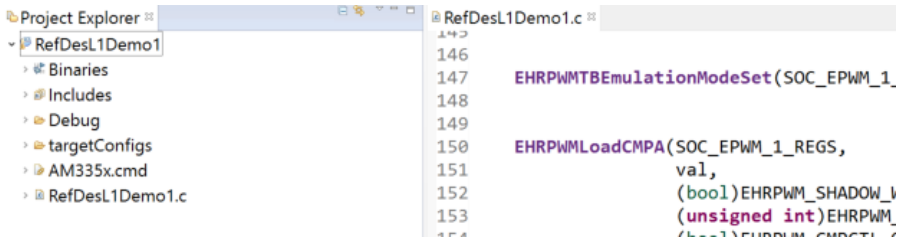


Figure 73 CCS Project Explorer

Import ***system_config*** project from (folder path is given assuming you have installed StarterWare at C:\ti, otherwise adjust the path accordingly):

C:\ti\AM335X_StarterWare_02_00_01_01\build\armv7a\cgt_ccs\am335x

Delete the default ***init.asm*** file under this project and replace it with the modified ***init.asm*** file (can be downloaded from the link above) as shown in Figure 74. Build the project using the ***build button*** as shown in Figure 75.

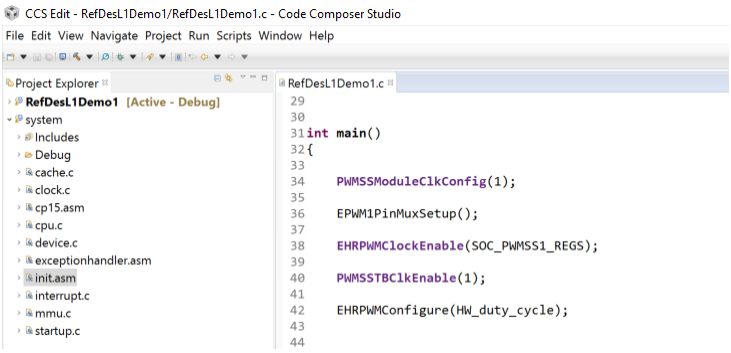


Figure 74 Adding custom init.asm file

- Before we can run the demo application, we need to build 3 libraries: ***utils.lib***, ***drivers.lib***, and ***platform.lib***. We will be using many APIs from these libraries in our Demo Applications. To build the libraries, you need to (folder paths are given assuming you have installed StarterWare at ***C:\ti***, otherwise adjust the path accordingly):

  - Import ***utils*** project from *C:\ti\AM335X_StarterWare_02_00_01_01\build\armv7a\cgt_ccs*
  - Import ***platform*** project from *C:\ti\AM335X_StarterWare_02_00_01_01\build\armv7a\cgt_ccs\am335x\evmskAM335x*
  - Import ***drivers*** project from *C:\ti\AM335X_StarterWare_02_00_01_01\build\armv7a\cgt_ccs\am335x*
  - Build each of the above projects by selecting the project and hitting the ***build*** button as shown in Figure 75.

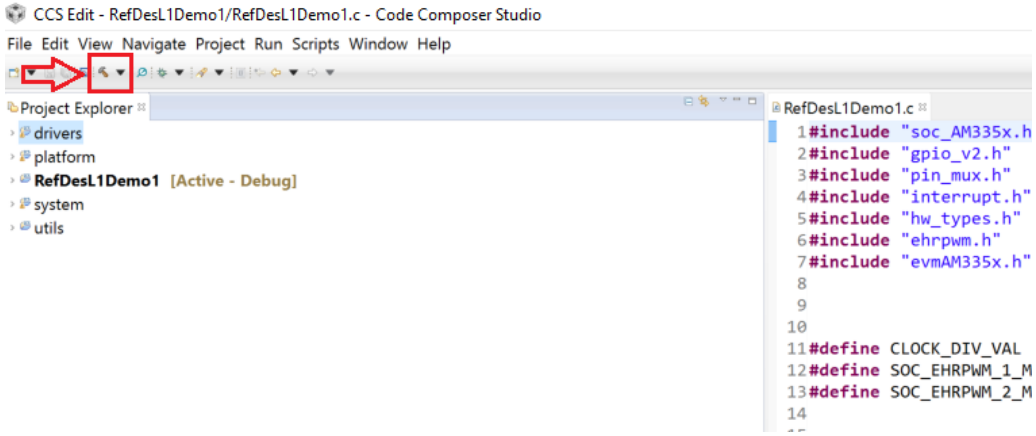


Figure 75 CCS Build button

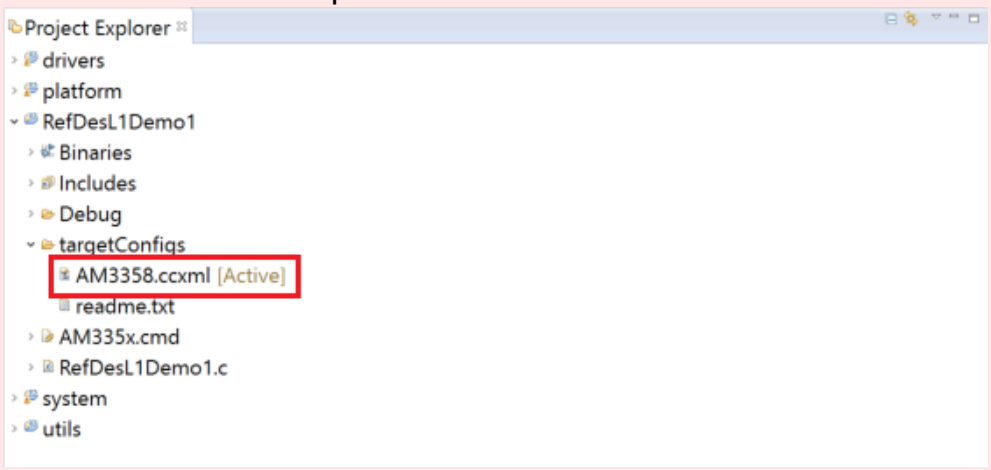
- Now we are ready to build ***Demo Application 1***. Select ***RefDesL1Demo1*** project and build it. If you have followed all the instructions given above, it should build without errors.
- Connect the debugger to your computer.
- Connect the 20 pin JTAG connector of the debugger to the JTAG header on the board.
- On power-up, the AM335x processor will look for valid boot images during the boot sequence (SPI0 -> MMC0 -> USB0 -> UART0 in case of OSD335x Bare Minimum Board). Since OSD335x Bare Minimum Board has no boot image on any of its interfaces, the AM335x processor will continue to look for an image. At this point, JTAG will be able to take control of the processor and allow the software to be loaded through the JTAG debugger and CCS.

  (If you are trying this step on OSD3358-SM-RED board instead of OSD335x Bare Minimum Board, then, make sure the SD card slot is empty and hold the SD boot button (S3) before applying power to the RED board to make sure AM335x cannot find a valid boot image during boot sequence. This is essential to allow JTAG debugger to take control of the AM335x processor and be able to load programs through CCS).

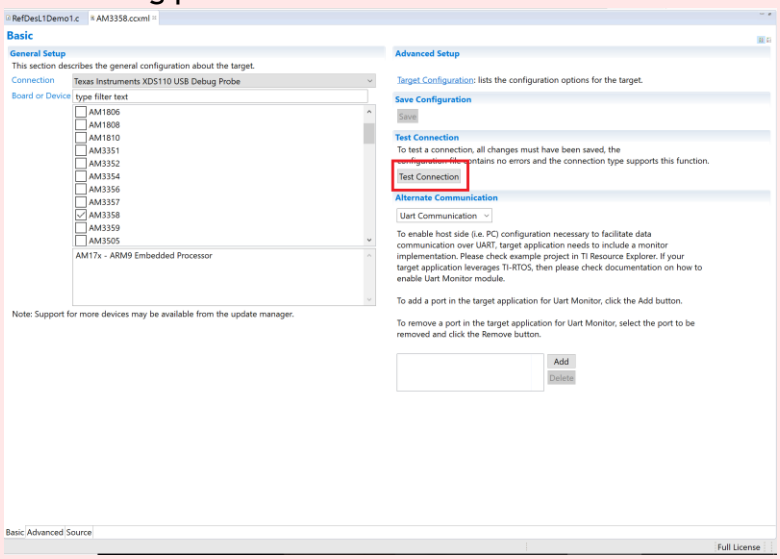
Caveat:

Before you try to load the Demo Applications through the JTAG debugger, you need to make sure the JTAG connection is setup properly and there are no connection issues. To check the JTAG connection, follow the steps below:

- In the drop down menu of your project folder, open ***targetConfigs*** drop down menu and then open ***AM3358.ccxml*** file as shown below.



- Click on ***Test Connection*** button to test the JTAG connectivity as shown below. CCS will perform a series of tests and you should receive the ***JTAG DR Integrity scan-test has succeeded*** message or something similar at the end of the testing process.



You can use the TI's Debugging JTAG Connectivity Problems wiki page to resolve any errors, if any. Probe the clock signals and use an oscilloscope to make sure OSC0 is operating at 24MHz and OSC1 is operating at 32.768KHz if JTAG connectivity test fails repeatedly.

You can continue with the Demo Application debug process once the JTAG connection is successfully verified.

- Press the ***Debug*** button as shown in Figure 76.

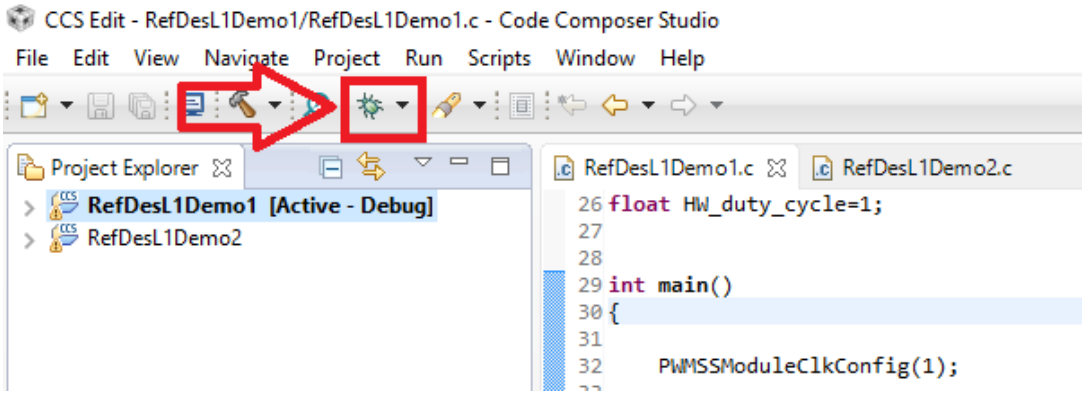


Figure 76 CCS Debug button

- Once the program gets loaded, you should see the below screen (Figure 77). Hit the ***Resume*** button to run the program on the board.

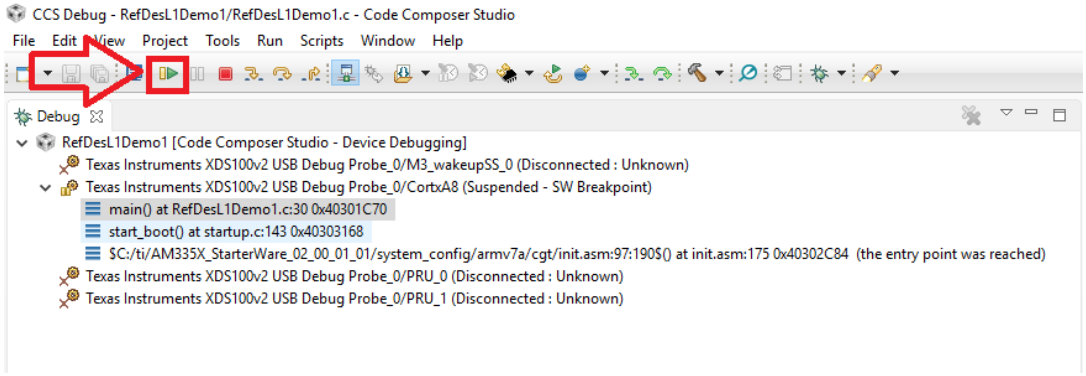
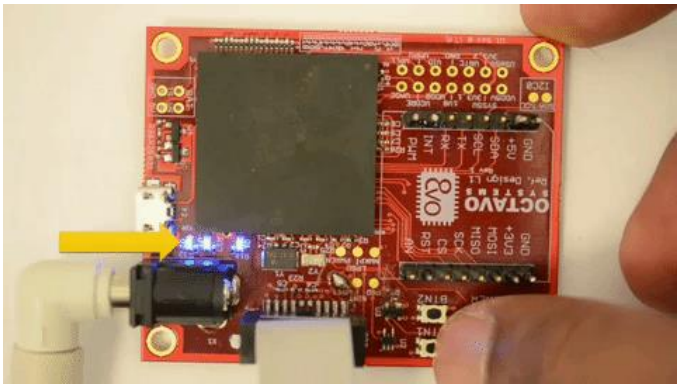


Figure 77 CCS Run button

- Verify the functionality of Demo Application 1 by pressing the user buttons BTN1 and BTN2 and observing the brightness of user LEDs D2 and D3 as shown in Figure 78.



Perk:

Generally, most errors that arise are due to compiler or linker path problems. Please pay extra attention while setting these paths.

Code Composer Studio does a good job in pointing out errors. Whenever there is an error, please go through the error message carefully. This will help you isolate the problem quickly so that it can be fixed.

Figure 78 Demo App 1 in action

11.5.3 Demo Application 2: Motion Detector

The objective of this app is to demonstrate the use of peripheral header with the OSD335x.

This project uses the MOTION click board (***https://shop.mikroe.com/click/sensors/motion***). It can be directly plugged into the peripheral header. The MOTION click board detects motion of living bodies. It picks up the IR radiation emitted by living bodies using its PIR sensor. Whenever a motion is detected, it sends an interrupt to the OSD335x using the ***INT*** pin of peripheral header. The OSD335x detects this interrupt and alternatively blinks LED D2 and LED D3 to indicate motion detection.

11.5.4 Running Demo Application 2

The procedure to run Demo Application 2 is exactly same as that of Demo Application 1 except that you have to use ***RefDesL1Demo2.c*** from instead of ***RefDesL1Demo1.c***. Figure 79 shows Demo App 2 in action.

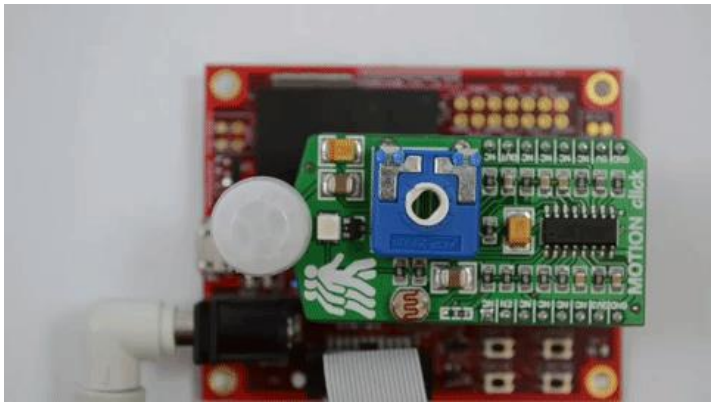


Figure 79 Demo Application 2 in action

****

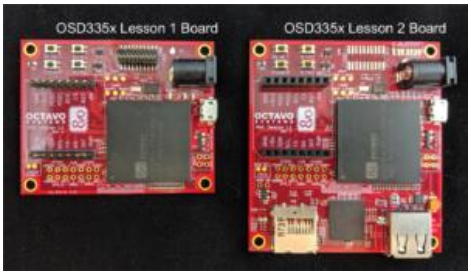
OSD335x Lesson 2
Introduction to Bare Minimum Circuitry for Linux Boot

The objective of this lesson is to help you become familiar with the bare minimum setup required to boot ***Linux*** on the OSD335x by building upon concepts and PCB design that was presented as part of Lesson 1. Lesson 2 is covered in Chapters 12 through 18.

12 Introduction to Bare Minimum Circuitry for Linux Boot

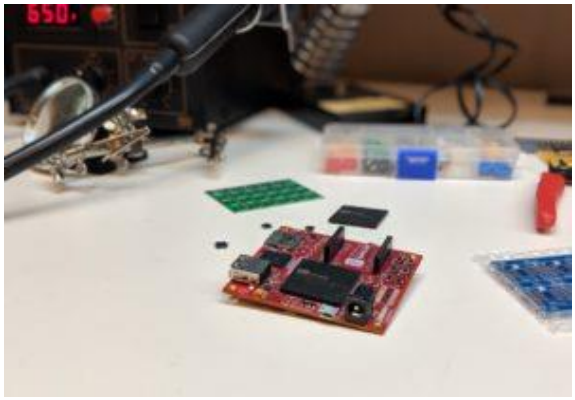
12.1 Introduction

The objective of this lesson is to help you become familiar with the bare minimum setup required to boot ***Linux*** on the OSD335x by building upon concepts and PCB design that was presented as part of Lesson 1. Similar to Lesson 1, this lesson will consist of a series of articles which will walk you through every step of the design process. We start from specifications and guide you through every step till debugging the manufactured Printed Circuit Board (PCB). The lesson will conclude with a PCB that verifies the design by putting together everything that was taught.



Recall that in Lesson 1, we built a PCB that could boot a bare metal application (using TI StarterWare) on OSD335x without the need of an OS. In Lesson 2, we will be adding USB connectivity and non-volatile memory to the Lesson 1 PCB design so that it can boot Linux. Lesson 1 PCB and expected Lesson 2 PCB designs that you can build yourself by following the steps.

Similar to Lesson 1, this lesson will consist of a series of articles, which will walk you through every step of the design process. We start from specifications and guide you through every step till debugging the manufactured Printed Circuit Board (PCB). The lesson will conclude with a PCB that verifies the design by putting together everything that was taught.



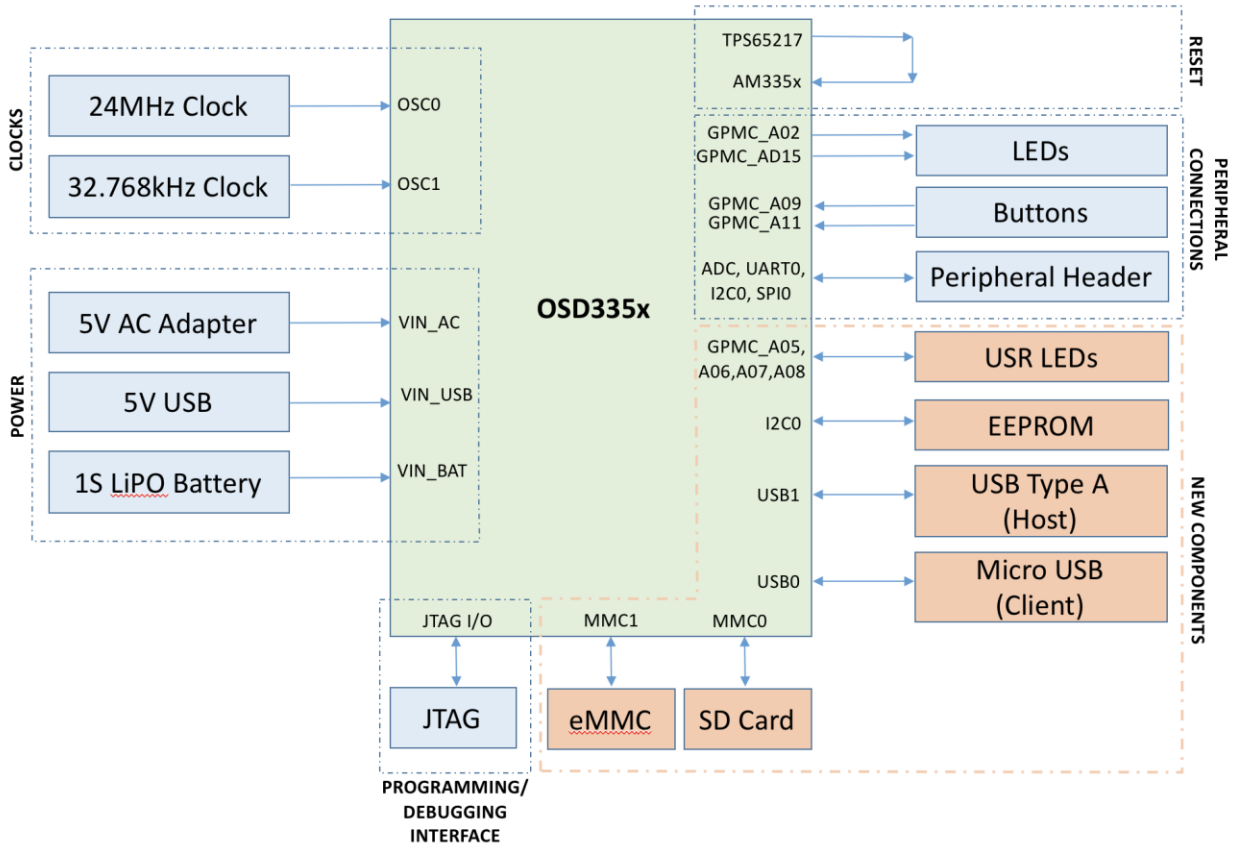


Figure 80 OSD335x Lesson 2 Block Diagram

For Lesson 2, we will be starting from the Lesson 1 block diagram and adding connectivity and non-volatile storage in order to demonstrate some of the capabilities of Linux as shown in Figure 1. We will walk through articles on:

- USB circuitry which will provide connectivity
- MMC circuitry which will provide non-volatile storage
- EEPROM circuitry which will provide non-volatile board configuration information
- Useful information on configuring the Linux Device Tree for a new board
- Tips on board bring-up and debug as well as demo examples

12.2 CAD Environment Setup

The Lesson 1 PCB design will serve as a foundation for Lesson 2. Hence, we need to first download and open Lesson 1 Eagle PCB Design files. You can download Lesson 1 design files here.

To accommodate the additional components, we will need to increase the length of Lesson 1 PCB. Except the length all other parameters can be maintained the same:

- New Board Size: 2500mil x 2750mil (2.5inch x 2.75inch)
- Number of layers: 4 layers
- Trace width: 6mil (approx. 0.15mm). Since power traces generally carry more current, we will be using larger traces (at least 15mil or 0.4mm) for them
- Trace spacing: 6mil (approx. 0.15mm)
- Minimum drill and via size: 12mil (approx. 0.30mm) drill and 24mil (approx. 0.60mm) finished via diameter (i.e., 6 mil annular ring)

Using these standard PCB design rules will help us reduce manufacturing cost. For your design, you are free to select the appropriate rules for your manufacturer and components that suit your design.

12.2.1 Library Setup

Octavo Systems provides an updated Eagle library for Lesson 2, ***OSD3358_BAS_RefDesignParts_L2.lbr***, that contains the schematic symbol and footprint for all the additional components required for Lesson 2. The library can be downloaded here.

12.2.2 Schematic Setup

- Rename the Lesson 1 schematic file RefDesL1.sch as ***RefDesL2.sch*** and layout file RefDesL1.brd as ***RefDesL2.brd***.
- Make sure Eagle is setup properly to use the updated Lesson 2 library (***OSD3358_BAS_RefDesignParts_L2.lbr***).
- Open ***RefDesL2.sch***. The schematic should look similar to Figure 5. This will serve as a good starting point for Lesson 2 design.
- It is up to you to change the label, date and revision number on each schematic page as necessary.

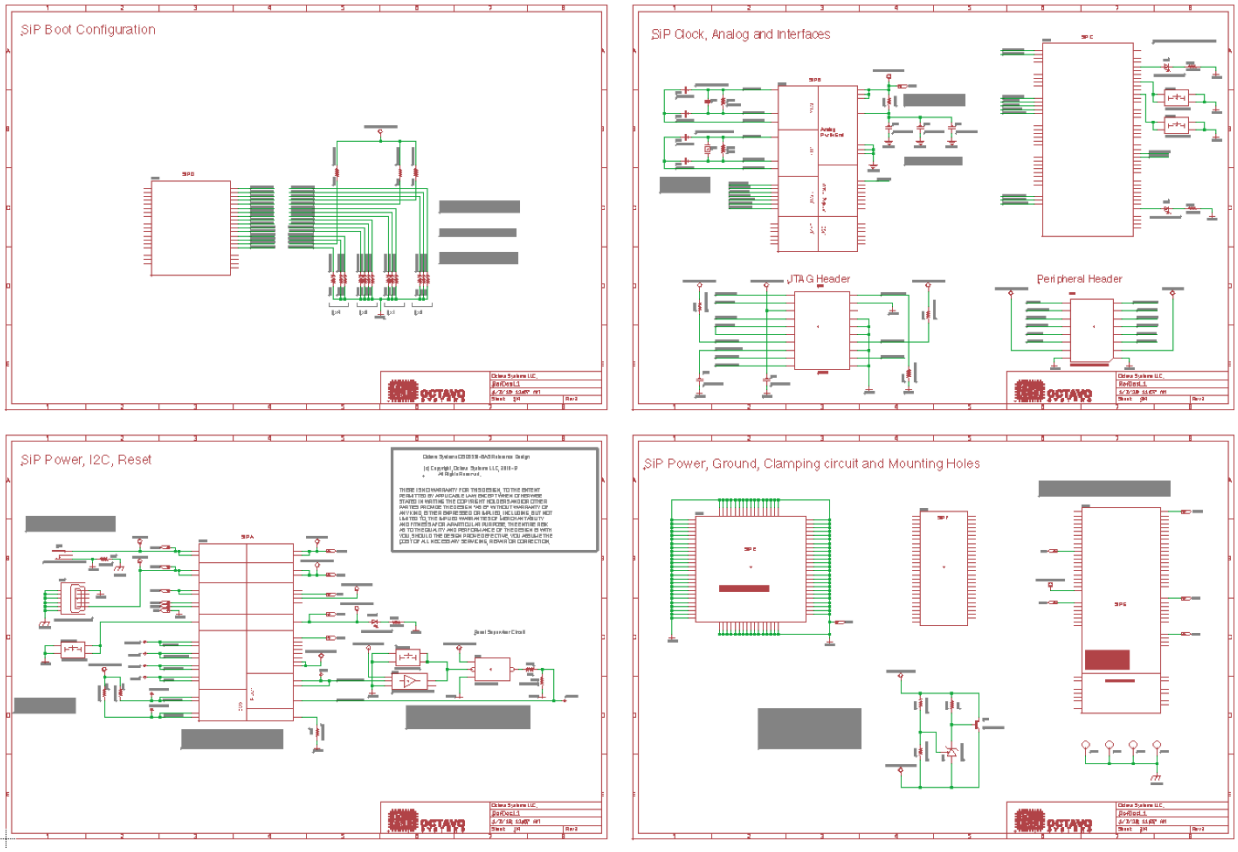


Figure 81 Lesson 1 schematic serving as a good starting point for Lesson 2

12.2.3 Layout (Board) Setup

- Open RefDesL2.brd file.
- The layer stack up is shown in Figure 6. We'll be using ***Top*** and ***Bottom*** layers for signal routing. ***Route2*** layer acts as power plane. ***Route15*** layer acts as ground plane.

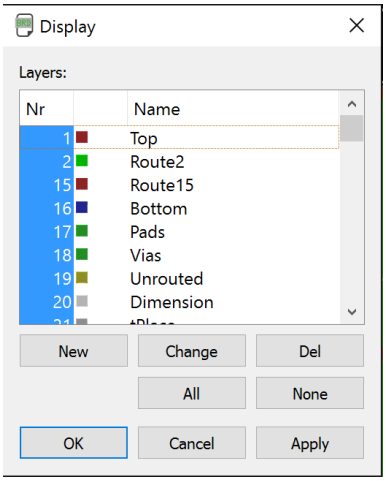


Figure 82 Layout Layer stack up

- The layout should look similar to Figure 7.

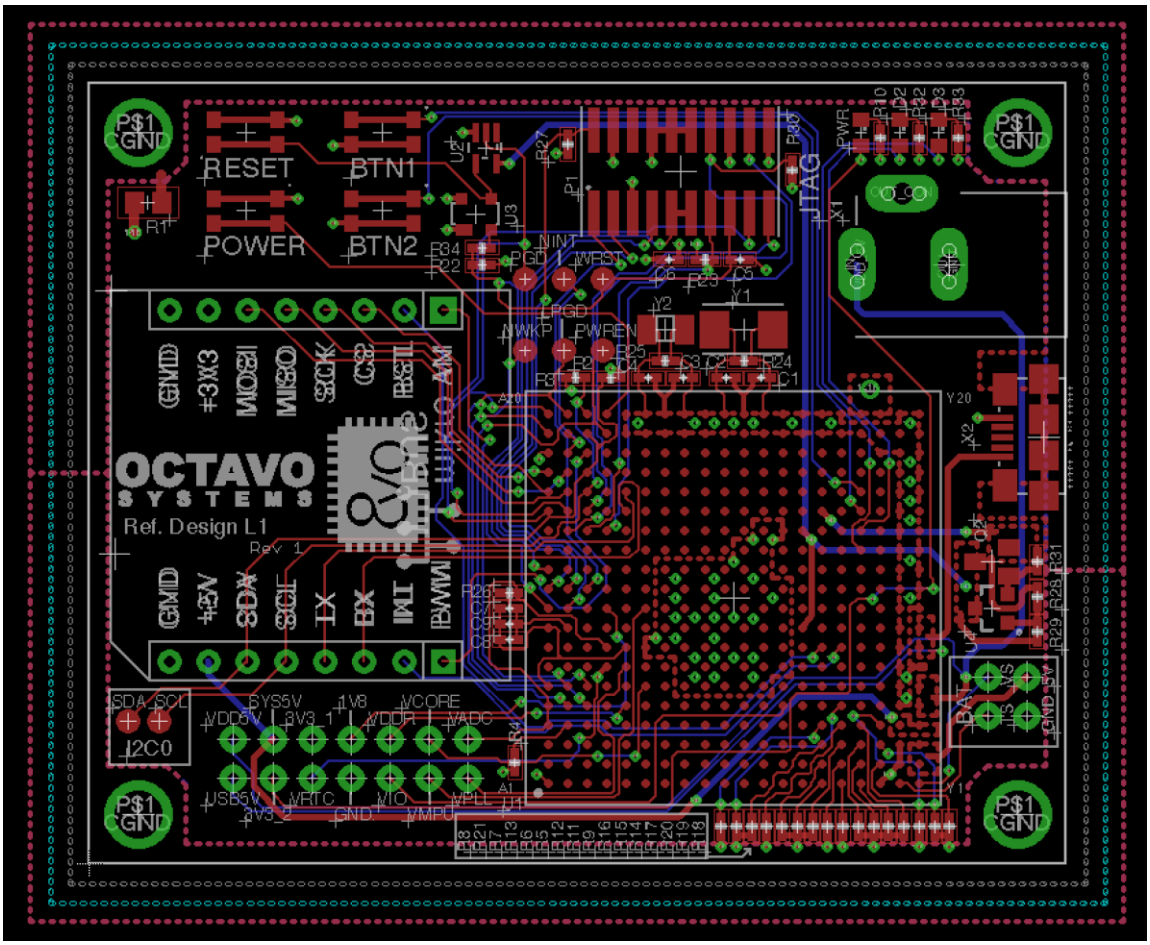


Figure 83 Lesson 1 layout serving as a good starting point for Lesson 2

- Extend the lower end of the PCB boundary and move the ESD rings by 750mils to make space for new components that will be added in this lesson. After the extension, the layout should look similar to Figure 84.

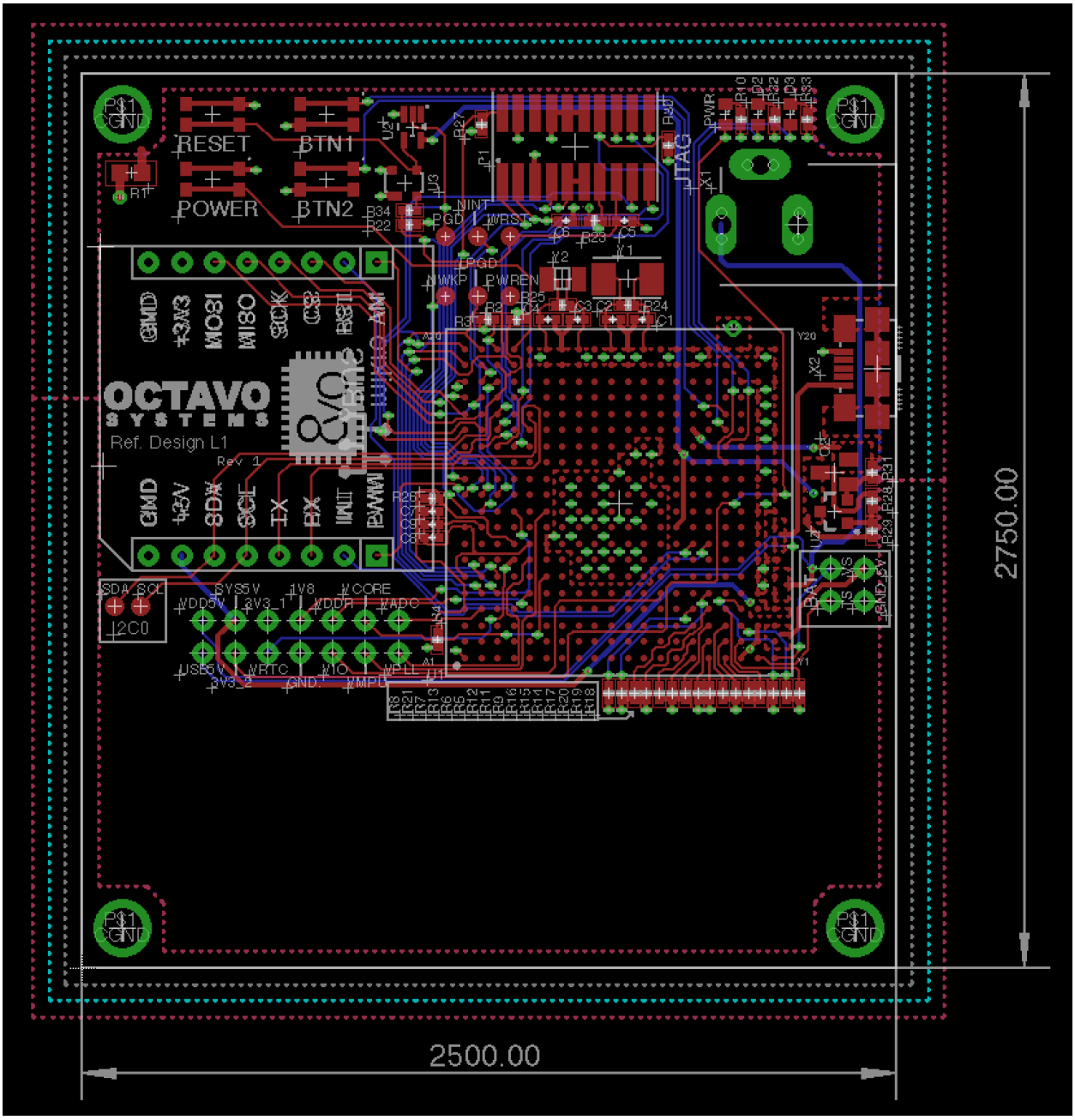


Figure 84 Lower end of PCB extended by 750mils

12.3 OSD3358-512M-BAS Pin Distribution

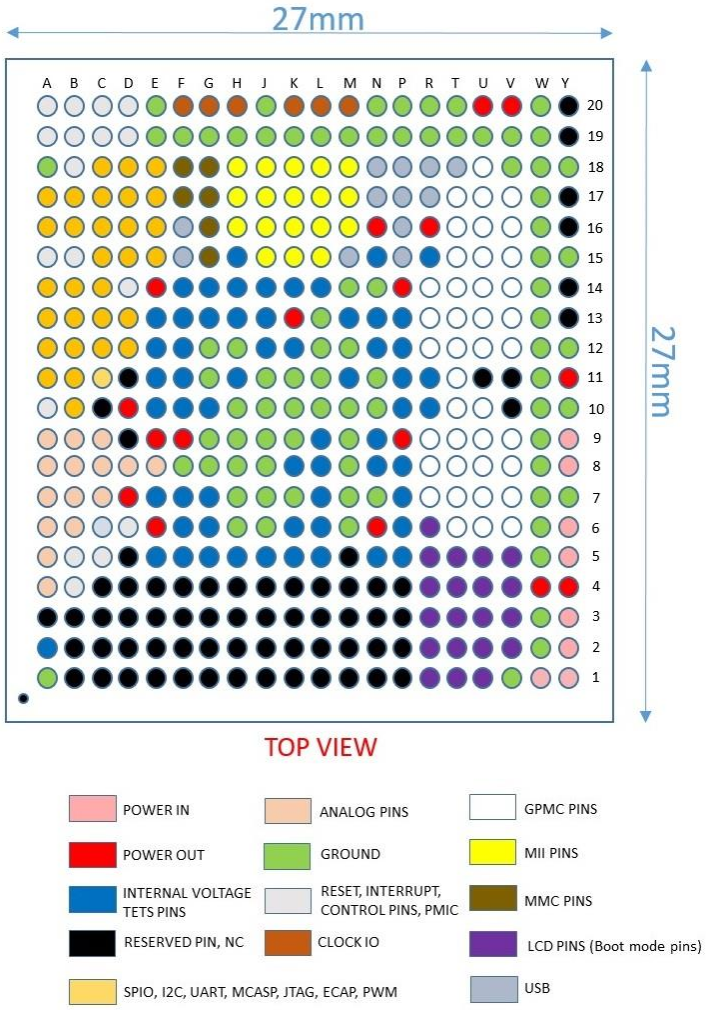


Figure 85 OSD335x BGA pin arrangement

Figure 8 gives visual representation of the OSD335x BGA pin arrangement. This will help us plan the placement of additional components on Lesson 2 board.

****

13 USB Circuitry

13.1 Introduction

The Universal Serial Bus (USB) is an industry standard that was introduced in 1996 to standardize the connection between computer peripherals. USB interfaces are used to exchange data and power between two or more devices. USB has been successful in replacing several older and slower interfaces like serial ports and parallel ports.

The OSD335x has two independent identical USB 2.0 peripherals (USB0 and USB1). Each peripheral supports USB Host, Peripheral (Client), and On-The-Go (OTG) modes with a line/bus speed of up to 480Mbps.

This article will give you necessary information about the different USB pins, how to put the USB peripherals of the OSD335x into the different USB modes, schematic/layout methodology and USB Testing for OSD335x.

13.2 USB Pins

USB 2.0 connectors have either 4 or 5 pins, depending on the form factor / USB mode. These pins are:

Table 1 USB Connector Pins

Pin	Type	Description
VBUS	Power	5V power rail
DM (or D-)	I/O	USB Data Differential Pair
DP (or D+)	I/O	
ID	I	USB Mode Control Pin (not present on 4 pin connectors)
GND	Power	Ground

A USB 2.0 Host connector has 4 pins (ID is assumed to be grounded) and can provide up to 500 mA of current at 5V. A USB 2.0 Peripheral (Client) port has 4 pins (ID is assumed to be high) and can source up to 500mA of current at 5V. A USB 2.0 OTG port requires 5 pins. This allows the port to be dynamically configured to be either a Host or a Peripheral (Client) by setting the ID pin voltage level.

When configured as a Host port, the USB controller will act as a master and send data and commands to the attached slaves. When configured as a Peripheral (Client) port, the USB controller will act as a slave and respond to data and commands sent to it by the master (Host).

The USB peripherals of the OSD335x contain the physical interface circuitry (PHY) to allow them to talk directly to other USB devices and each can be configured

independently as either a Host or a Client. To support this configurability, each USB peripheral has seven pins which are listed in Table 2.

Table 2 USBx Pins

Pin	Type	Description
USBx_DP	I/O	USBx Data Differential Pair
USBx_DM	I/O	
USBx_DRVVBUS	O	USBx external power logic control
USBx_VBUS	I	USBx external power logic voltage sense
USB_ID	I	USBx Mode Control Pin
USB_CE	O	USBx Charge Enable

- **USBx_DP/DM:** These are differential I/O pins that carry data.
- **USBx_DRVVBUS**: The USB peripherals themselves cannot supply the necessary current to support Host mode. Therefore, the USBx_DRVVBUS output pin was provided to control external 5V power logic used to supply power to a Host port.
- **USBx_VBUS:** This input pin is used to sense the voltage of the VBUS pin . For the USB peripheral to be enabled, a valid voltage (>=4.4V) should be placed on this pin.
- **USB_ID:** The mode of USB peripheral depends on the state of this pin.
- **USB_CE:** Each USB PHY contains circuitry that can automatically detect the presence of charger on the USB port. If a charger is detected, this pin goes high and remains high until the charger is disabled through software. For more information, please refer ***USB Charger Detect*** section of AM335x TRM (Section 9.2.4.4.2 of SPRUH73P). In general, this pin is not used.

13.3 USB Modes and their Configuration on the OSD335x

The three different USB modes supported by the USB peripherals of the OSD335x are described in this section.

13.3.1 USB Host Mode

In Host mode, the USB port will be able to supply power, control and communicate with all the connected Client devices. For example, the USB ports on most desktop computers and laptops operate in Host mode.

The USB peripherals of the OSD335x can be configured in Host mode in 2 ways:

Hardware Method: Ground the USBx_ID pin and use an appropriate USB A type connector on the cable end.

Software Method: Program the firmware to set the respective USBxMODE register IDDIG bit to 0.

Once the USB controller determines its role is a Host, it will drive USBx_DRVVBUS pin high to enable external 5V power logic and wait for USBx_VBUS input pin to go high. If USBx_VBUS does not go high (>=4.4V) within the next 100ms, it will generate a VBUS error interrupt that can be handled by software. However, if a valid voltage is found on USBx_VBUS, the controller will wait for a Client device to connect.

13.3.2 USB Peripheral (Client) Mode

In Client mode, the USB device receives power from the Host and the device can only communicate with the Host. The USB peripherals can be configured in Peripheral (Client) mode in 2 ways:

Hardware Method: Leave the USBx_ID pin floating and use an appropriate USB B type connector on the cable end.

Software Method: Program the firmware to set the respective USBxMODE register IDDIG bit to 1.

The USB controller will begin operating in Peripheral (Client) mode only if it detects a valid voltage (>=4.4V) on USBx_VBUS input.

13.3.3 USB OTG Mode

On-The-Go (OTG) mode allows the USB port to dynamically switch between Host and Peripheral (client) mode based on requirement in space constrained applications like tablets or smartphones.

The USB peripherals in OTG mode assume the role of Host or Peripheral (Client) based on the status of ID pin which is directly controlled by the USB cable.

The configuration of the USB peripherals for different modes in summarized in Table 3.

Table 3 USBx Mode Configuration

	Configuration		
	Through Hardware		**Through Software***
USB Host	Ground USBx_ID	OR	Set IDDIG bit of USBxMODE register to 0
USB Client	Leave USBx_ID floating		Set IDDIG bit of USBxMODE register to 1
USB OTG	USBx_ID is directly controlled by USB Cable Connector		Set IDDIG bit to either 0 or 1 based on mode requirement

* If the USBx mode is being controlled purely through software, you may have to suitable manage other registers also along with IDDIG pin. For more information, refer ***USB Controller Host and Peripheral Modes Operation*** section of AM335x TRM.

13.4 USB Schematics

In this section, we will build schematics for to configure the USB1 peripheral in Host mode and the USB0 peripheral in OTG mode.

Let's begin with the USB1 as a Host port. As described in the previous section, a USB Host port is required to supply power to its clients. Hence, we will need an external 5V power switch to supply the appropriate amount of power. The OSD335x will use the USB1_DRVVBUS output to enable/disable the power switch and will use the USB1_VBUS input to sense the presence of required output voltage. Given that there is generally a lot of interaction with the USB port, it is important to add electro-static discharge (ESD) protection to the signals going to the processor. Therefore, a 4-channel ESD Protection chip has also been added to the circuit.

For this lesson, we will be using be using a TPS2041 external power switch, a TPD4S012 4-channel ESD Solution, and an AMP 787616-1 USB A Female Connector. You may use different components in your design based on your requirements.

In order to isolate the power supplied by the board from any external noise on the USB VBUS line, a Ferrite bead is placed between the output of the power switch and the connector. Similarly, in order to isolate the USB1_VBUS input from any power supply noise on the board, a Ferrite bead will be placed between the output of the power switch and the USB1_VBUS input.

The symbols for the TPS2041, TPD4S012, AMP 787616-1 and Ferrite beads are available in the provided library. The USB Host circuit can be built as shown in Figure 86.

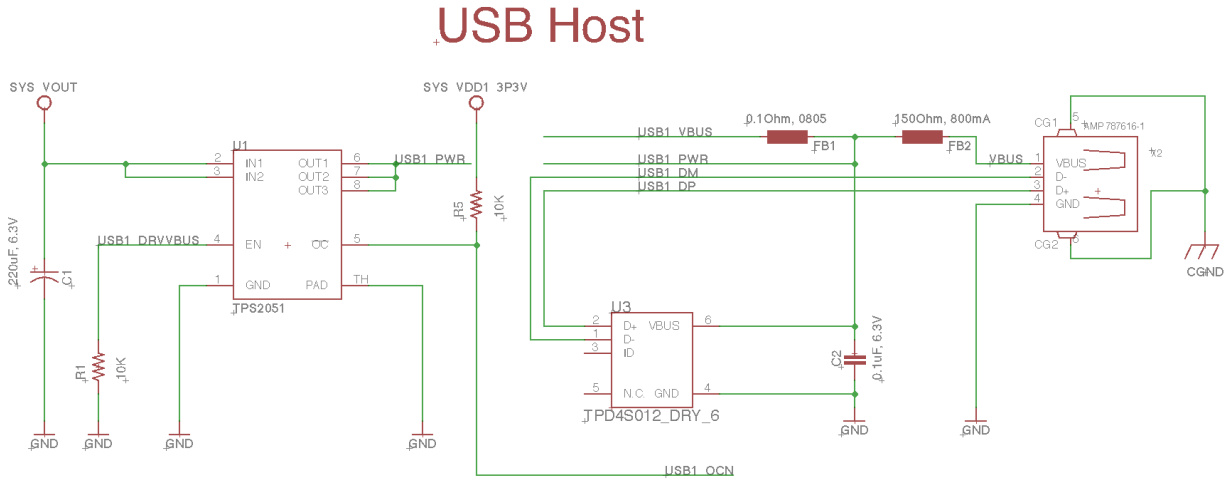


Figure 86 USB Host Schematic

The USB1 signal connections to the OSD335x are shown in Figure 87 (The changes made are highlighted in the dotted lines).

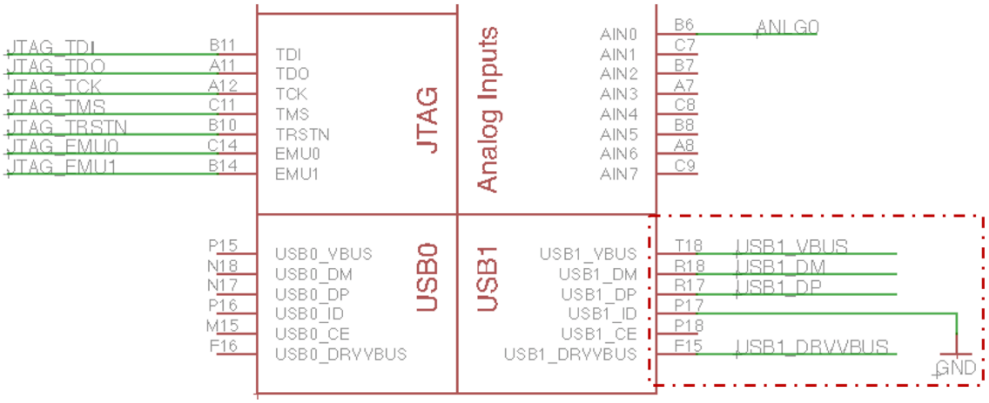


Figure 87 USB1 Signal Connections

The active low Over Current output of the Power Switch (pin 5 of the TPS2501 attached to the USB1_OCN signal in Figure 86) can be connected to any of the GPIOs of the OSD335x as shown in Figure 88 (GPMC_A10 pin is chosen in this case) to detect issues with USB Client devices trying to draw more power than allowed.

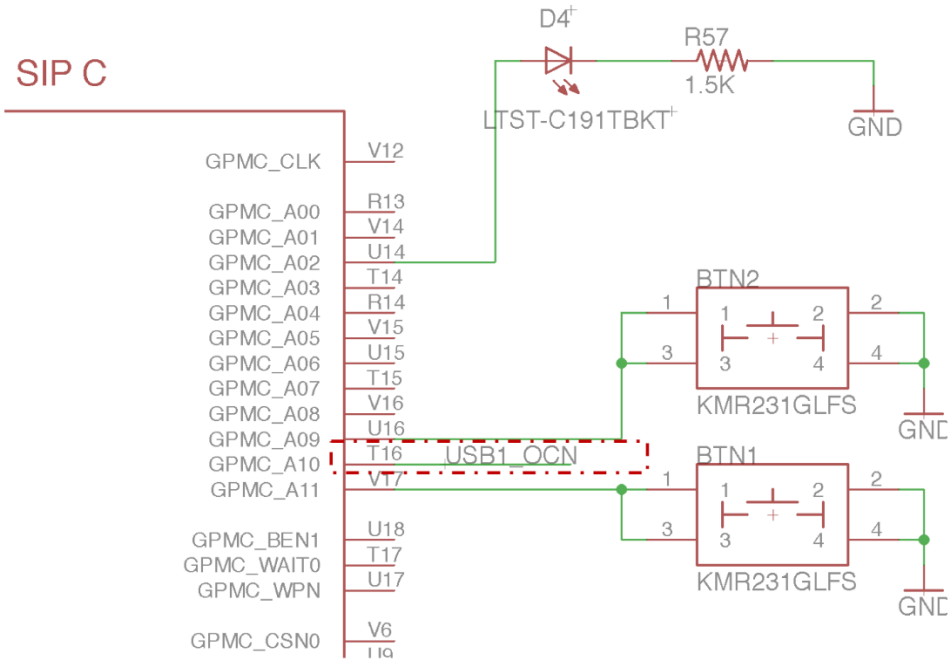


Figure 88 USB1_OCN connection

Now, let's configure USB0 as a USB Client (technically we will configure this as an OTG port since we will connect the ID pin, but primarily this port will be used as a client. To use this as a Host port, power will need to be applied externally to the USB5V test point.). A USB Client can receive power from its host using the VBUS line of the USB client connector. The OSD335x can source power from multiple input sources, including 5 volts from a USB connector. Therefore, to provide flexibility when powering this board, we can connect the VBUS line of the USB client connector to the USB Power Input pins of the OSD335x, i.e., VIN_USB. Similar to the USB Host connections, we also have to connect the VBUS input to the USB voltage sense input of the OSD335x, USB0_VBUS, and add a TPS2041 ESD protection chip.

The symbol for USB Client Connector (10118192-0001LF) is available in the provided library. The USB Client circuit is shown in Figure 89. This USB Client connector is the same connector used in Lesson 1 for the USB Input Power. Now, we have to connect the USB signals and ESD protection chip.

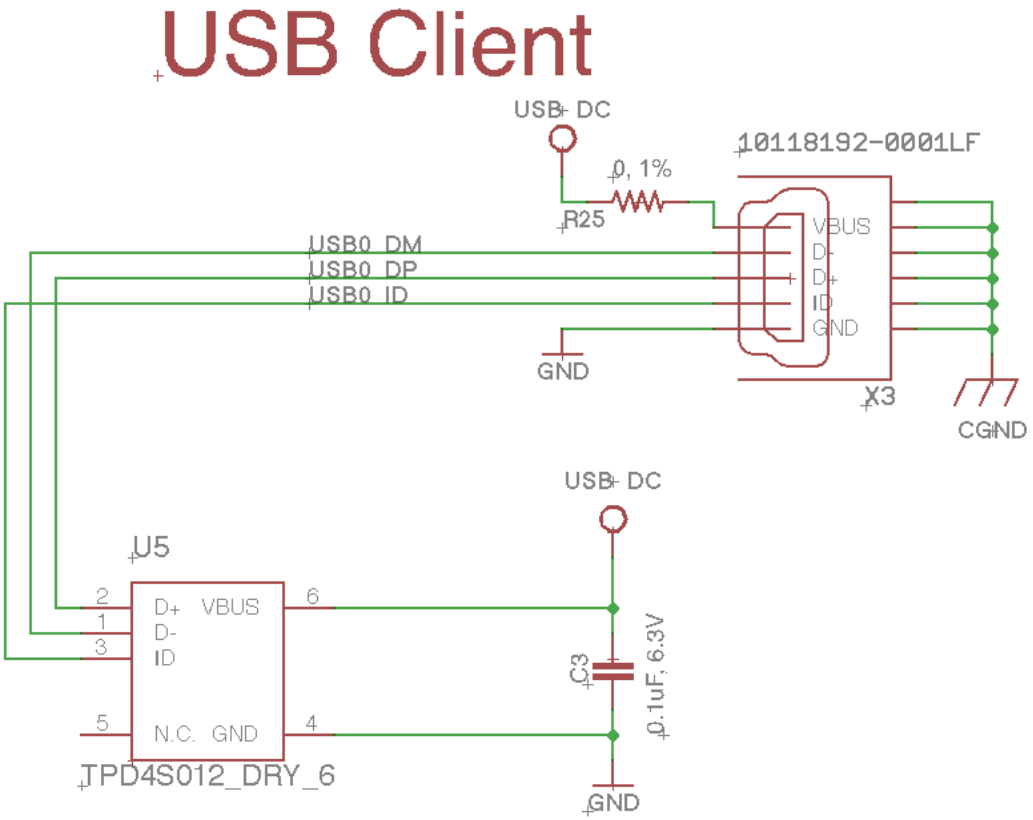


Figure 89 USB0 Client Circuit

The USB0 signal connections to the OSD335x are shown in Figure 90 (The changes made are highlighted in the dotted lines).

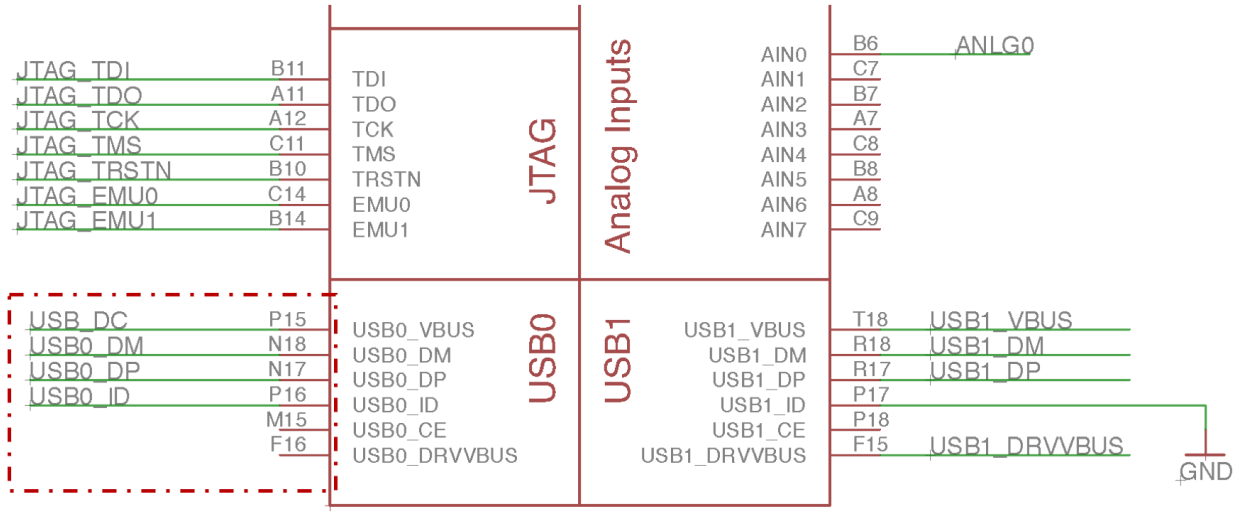


Figure 90 USB0 Signal Connections

13.5 USB Layout

USB lines operate at very high speeds (up to 480Mbps) and use differential signaling. Therefore, we need to follow certain guidelines to ensure good layout and proper operation.

- Length matching the differential pair (D+ and D-) is critical in USB layout design. The thumb rule is to maintain the length difference < 1%.
- If the trace lengths are mismatched, try to match them at the mismatched end and not on the matched end using serpentine traces. When using serpentine traces, the please be careful that the width between the bends is greater than 3x the trace width and that all angles are at least 135 degrees. An example for this is shown in Figure 91.

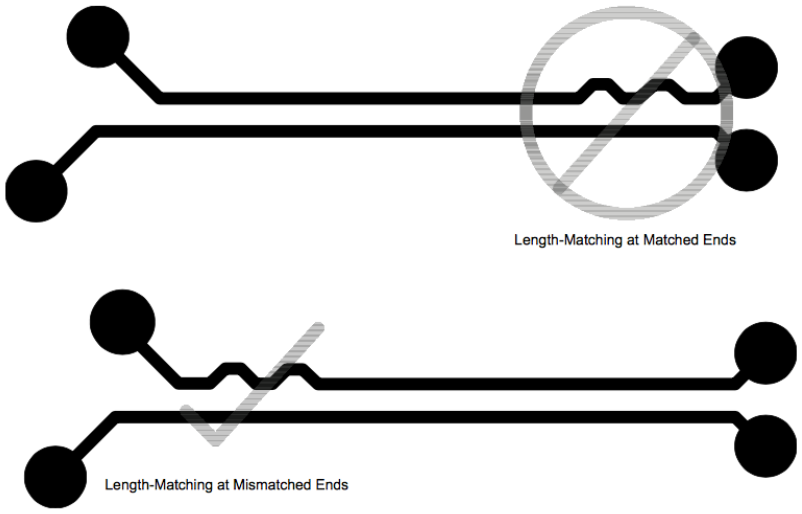


Figure 91 USB Length Matching (©Texas Instruments)

- Route the differential pair symmetrically and parallel to each other.

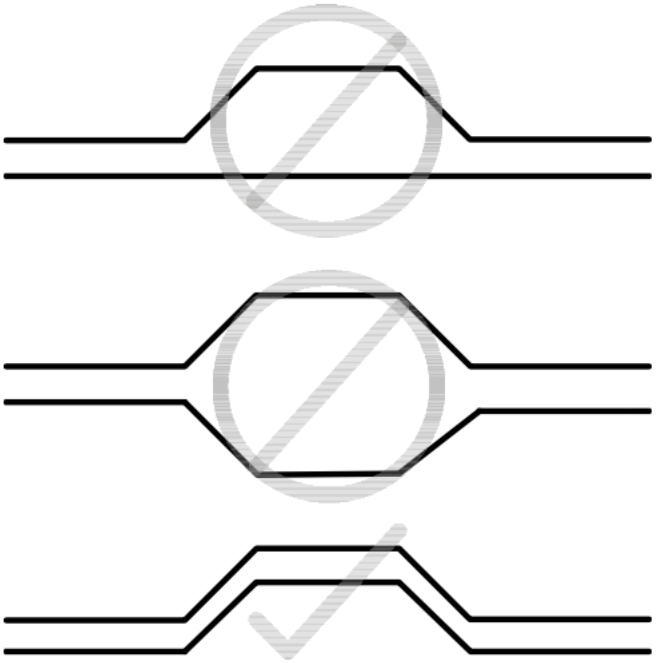


Figure 92 Differential Pair Symmetry (©Texas Instruments)

- It is best to route the differential pair over a solid Ground plane. Avoiding routing over split planes or plane voids.
- When through-hole USB connectors are used, it is better to route the differential pair on the bottom layer instead of top layer so that the connector pins can be directly reached on the bottom layer. This prevents the connector pins from acting as stubs.
- Try to avoid routing differential pair near clock signals or any other signals that may cause interference.
- The D+ to D- impedance must be 90 Ohms and the impedance of either line to ground must be 30 Ohms. This is straight forward to do on a 4 layer PCB. However, you can route USB on a two layer PCB. See ***http://www.focusembedded.com/blog/high-speed-usb-in-a-two-layer-pcb/*** for a discussion on additional considerations to accomplish this.

Keeping the above guidelines in mind, the USB Host and Client circuitry can be laid out as shown in Figure 93 and Figure 94 respectively.

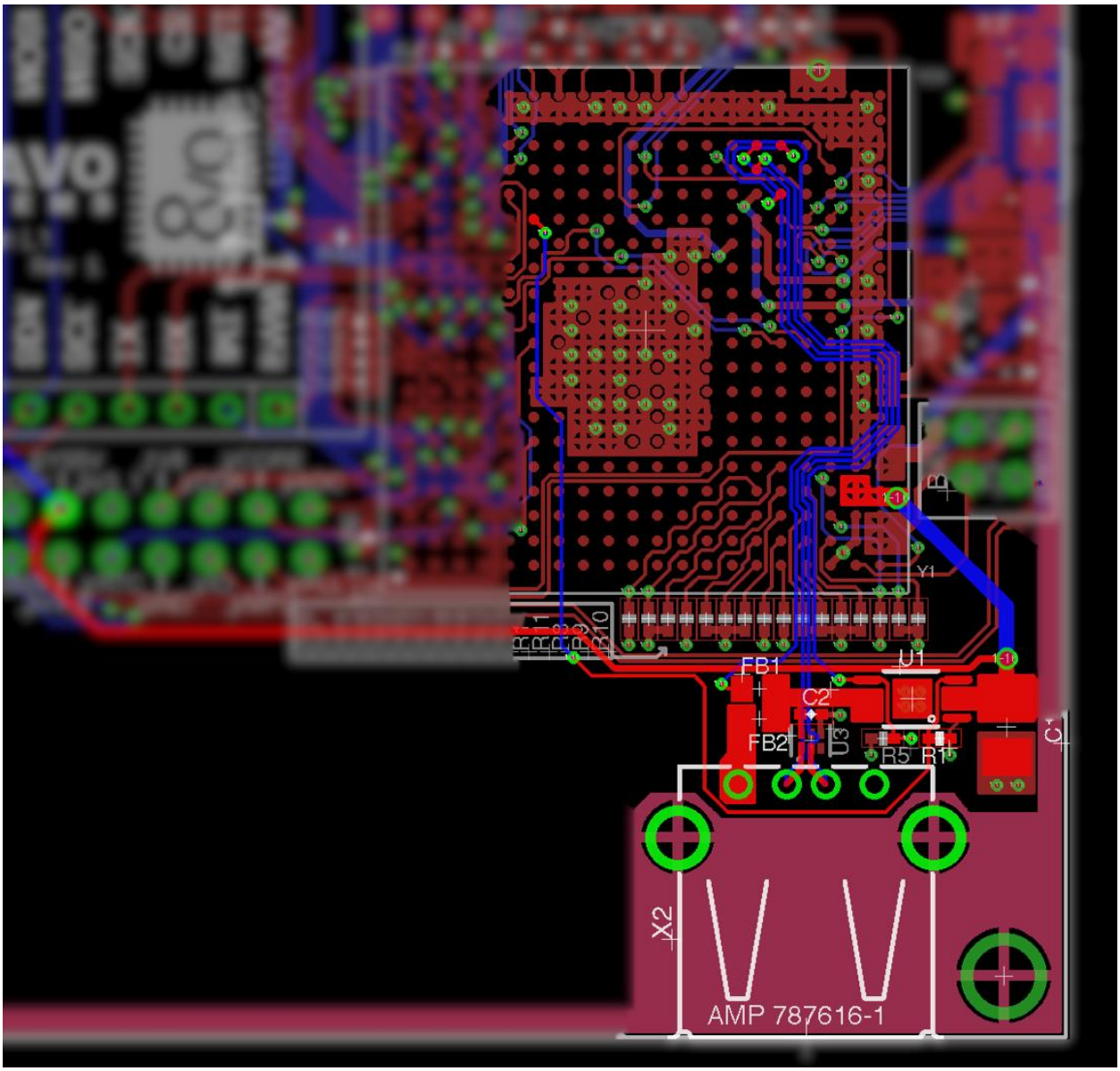


Figure 93 USB Host Circuit Layout

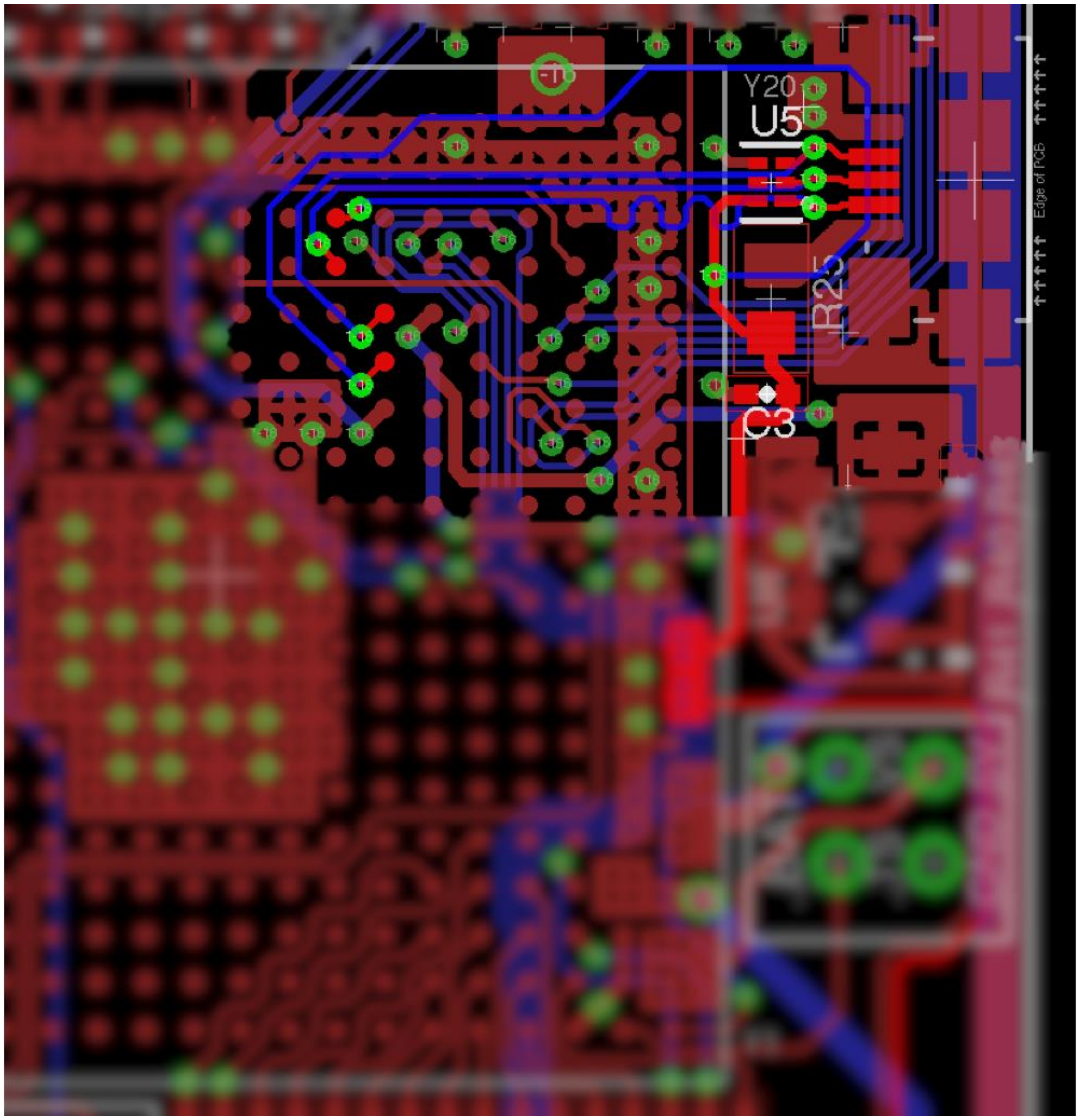


Figure 94 USB Peripheral (Client) Circuit

13.6 USB Testing

Testing the integrity of USB signal is very important since it operates at high speed. An eye diagram is commonly used for this purpose. The eye diagram test will measure the rise time, fall time, undershoot, overshoot and jitter of USB signal. You can put the official USB compliance logo on your device only after your device passes the appropriate tests. More information about USB compliance tests and procedure can be found here.

****

14 Adding Non-Volatile Storage

14.1 Introduction

In order to store programs, files, and data so that Linux can boot easily and retain information across power downs, non-volatile storage needs to be attached to the OSD335x. This can be done using floating gate non-volatile memories such as embedded Multi-Media Card (eMMC), Secure Digital (SD) Card, or electrically erasable programmable read-only memory (EEPROM). The floating gates are used to store information in the form of 0s and 1s and can retain data when power is removed from the device.

The main differences between an eMMC or an SD Card and EEPROM are speed, data density and longevity. EEPROM is typically byte addressable with limited read/write cycles and has a lower data density. It also operates relatively slowly but has a much lower cost when a small amount of non-volatile storage is required. Whereas, in an eMMC or an SD Card (also known as Flash memory devices), data is read and written in large blocks or pages which helps it operates faster. Since flash memory devices are not byte addressable, all the circuitry required for byte addressability is removed and hence they can pack a higher data density. Additionally, eMMC has circuitry that increases robustness against data corruption due to power down events. Hence, in embedded applications, eMMCs and SD Cards are generally used to store the Linux OS, filesystem, applications and related data whereas EEPROM is generally used to store board or device identification (ID) and hardware configuration information.

The OSD335x has three MMC/SD Controllers that can interface with flash memory devices like an eMMC or an SD Card. For our board, we will use MMC0 to interface with a microSD Card connector and MMC1 to interface with an eMMC device.

The OSD335x also has many I2C peripherals that can be used to interface with the EEPROM. For our board, we will use the I2C0 peripheral to interface with the EEPROM. The OSD335x has to use an external EEPROM for the board and device ID. However, the OSD335x-SM integrates an EEPROM within the package. Within the OSD335x-SM, the EEPROM is also connected on I2C0.

This article will help you connect an eMMC, SD card and EEPROM to the OSD335x by giving you necessary information about their corresponding circuits, schematic and layout. We will also be discussing the steps needed to finalize the Lesson 2 PCB design and sending it for manufacturing.

14.2 MMC/SD Circuitry

Each MMC/SD instance of the OSD335x has several pins. The name and function of the important pins are listed in Table 2.

Table 4 MMCx Pins

Pin	Type	Description
MMCx_CLK	I/O	MMC/SD serial clock output
MMCx_CMD	I/O	MMC/SD command signal
MMCx_DAT [7:0]	I/O	MMC/SD data signals
MMCx_SDCD	I	SD card detect (from connector)

- **MMCx_CLK:** This pin provides the clock to a MMC or SD device from the MMC/SD controller.
- **MMCx_CMD**: This pin is used for two-way communication between the connected MMC or SD device and the MMC/SD controller. The MMC/SD controller transmits commands to the device and the device drives responses to the commands on this pin.
- **MMCx_DAT [7:0]:** Depending on the device you are using, you may need to connect 1, 4, or 8 data lines. The number of DAT pins (the data bus width) is set by the Data Transfer Width (DTW) bit in the MMC control register (SD_HCTL) and DW8 bit in SD_CON register. For more information, see ***MULTIMEDIA_CARD Registers*** section of AM335x TRM .
- **MMCx_SDCD:** This input pin serves as the SD card detect. This signal is generated by a mechanical switch on an SD card connector.

14.2.1 SD Card

The Device Tree of the Linux image you will be using expects an SD card on MMC0 and an eMMC on MMC1 interface of the OSD335x. Therefore, we will use MMC0 for our SD card circuit. The SD card circuit can be built around MMC0 as shown in Figure 95 and Figure 96. According to JEDEC Standard No. 84-A43 which is the eMMC standard implemented by the AM335x processor inside the OSD335x, MMC DAT and CMD lines have to be pulled up to prevent them from floating when no card is inserted.

Perk:
The MMC peripherals in the AM335x support the SDIO 2.0 standard which implies a card size limit of 32GB. However, the SDIO 3.0 standard includes Standard (STD) and High Speed (HS) modes that are backward compatible with SDIO 2.0. This allows SD cards greater than 32 GB to potentially be supported, but without the higher data rates of the SDIO 3.0 standard. However, it is recommended that SD cards greater than 32GB be tested with an OSD335x development board before being used.

Caveat:

Since the OSD335x only supports 3.3V I/Os, 1.8V signaling for your SD card is not supported.

For this lesson, we used the ***SCHA5B0200*** SD card connector. Its symbol and footprint are available in the given library. This is a Push-Push Reverse Mount micro SD card connector. At this writing, this part has been end-of-life'd by the vendor and we would recommend the ***1040310811*** Push-Push top Mount micro SD card connector as a substitute.

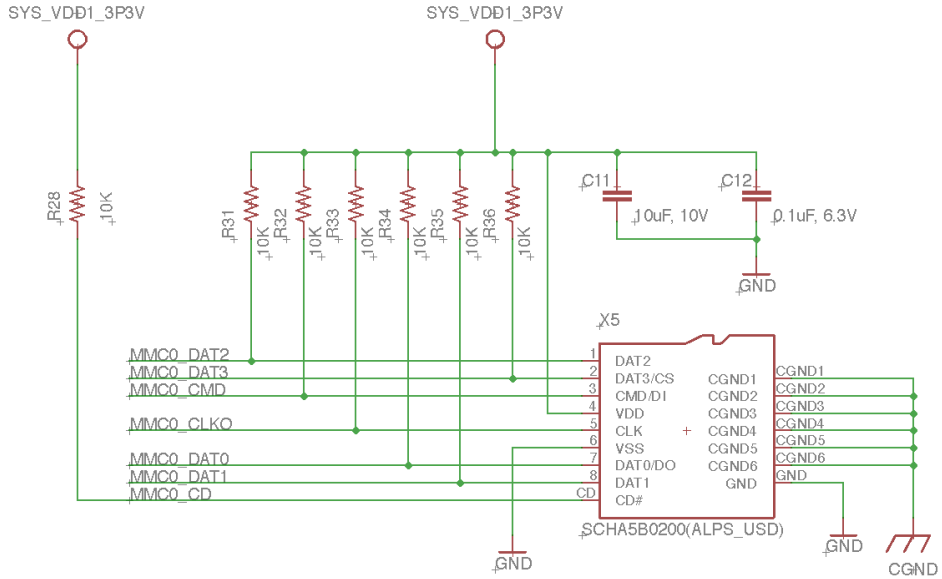


Figure 95 SD Card Circuitry

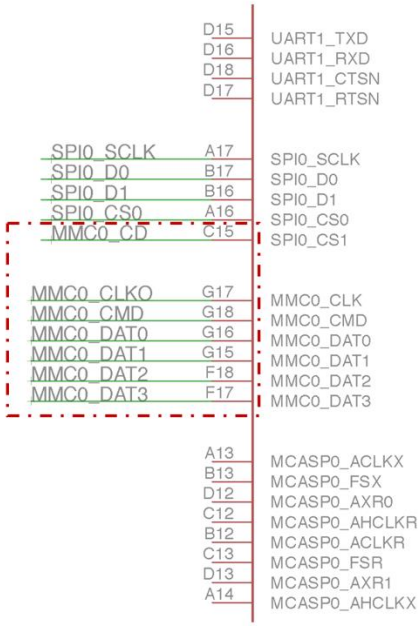


Figure 96 OSD335x SD Card connections

The SD card layout can be made as shown in Figure 97. The MMC bus operates at a moderately high speed. Therefore, you should try to keep all of the traces of the bus about the same length.

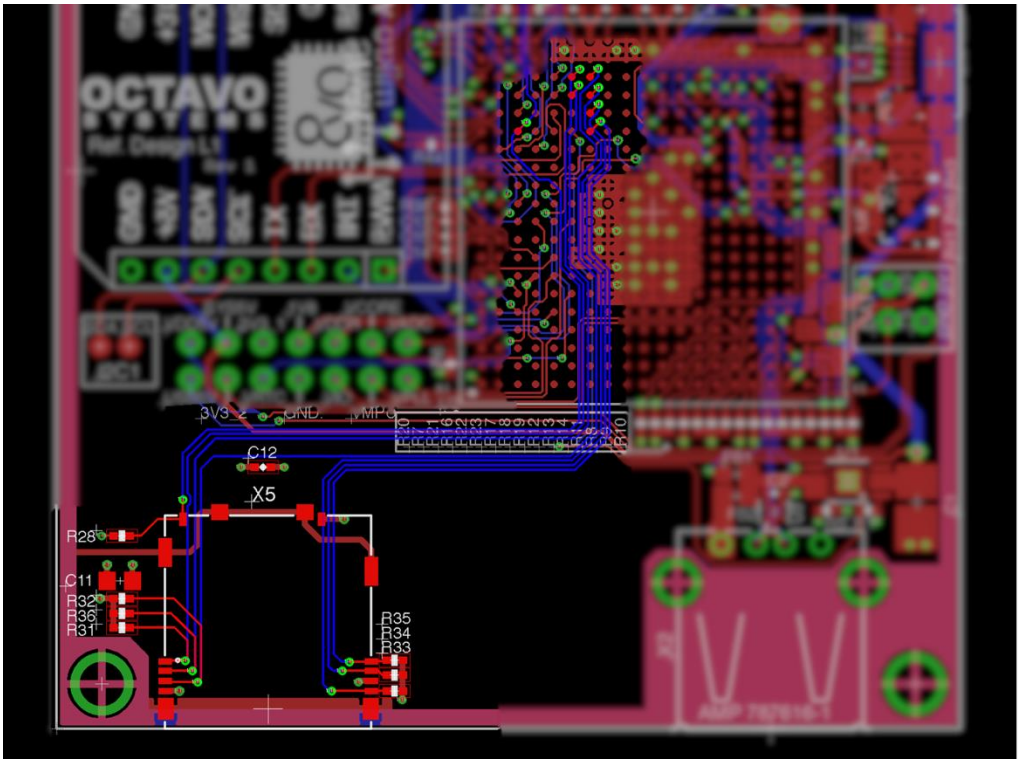


Figure 97 SD Card Layout

14.2.2 eMMC

Similar to the SD Card circuit, the eMMC circuit can be built around MMC1 as shown in Figure 98 and Figure 99. The RST# input of the eMMC can be controlled by any of the GPIOs of the OSD335x (we used the GPMC_A04 pin for this design).

For this lesson, we'll be using ***Kingston 153 (HS200) 16GB*** eMMC. Its symbol and footprint are available in the given library. However, it can be difficult to source eMMC memory. As long as the memory follows the 153 ball JEDEC eMMC standard, it can be used in this lesson.

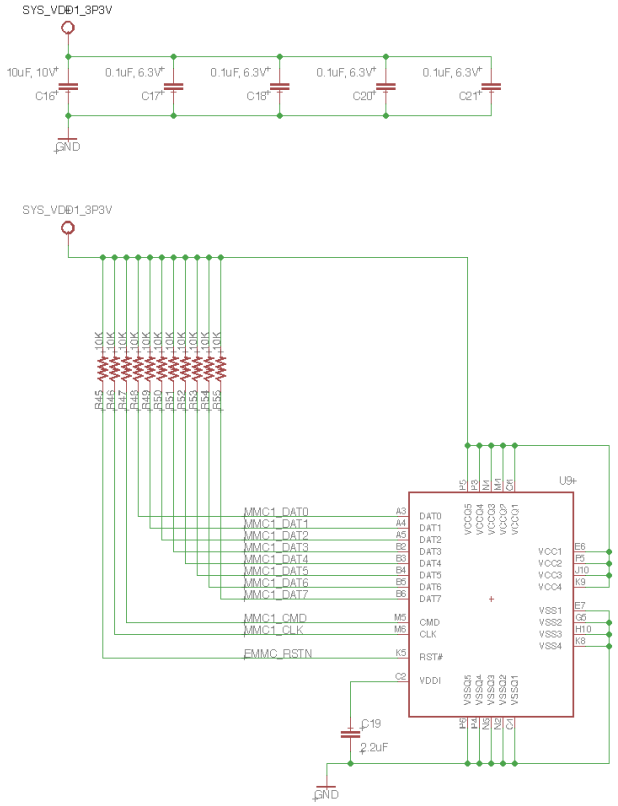


Figure 98 eMMC Circuitry

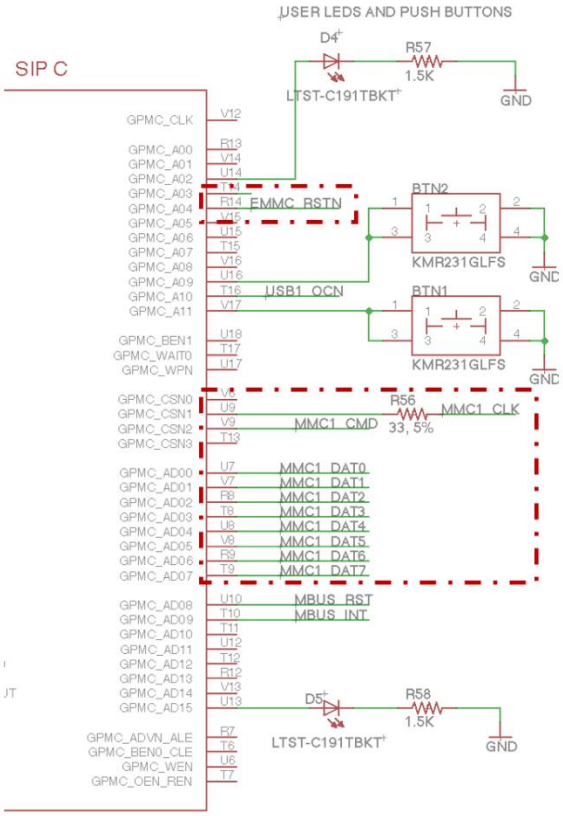


Figure 99 OSD335x eMMC connections

The eMMC traces can be laid out as shown in Figure 100. Similar to the SD card layout, you should try to keep all of the traces of the bus about the same length.

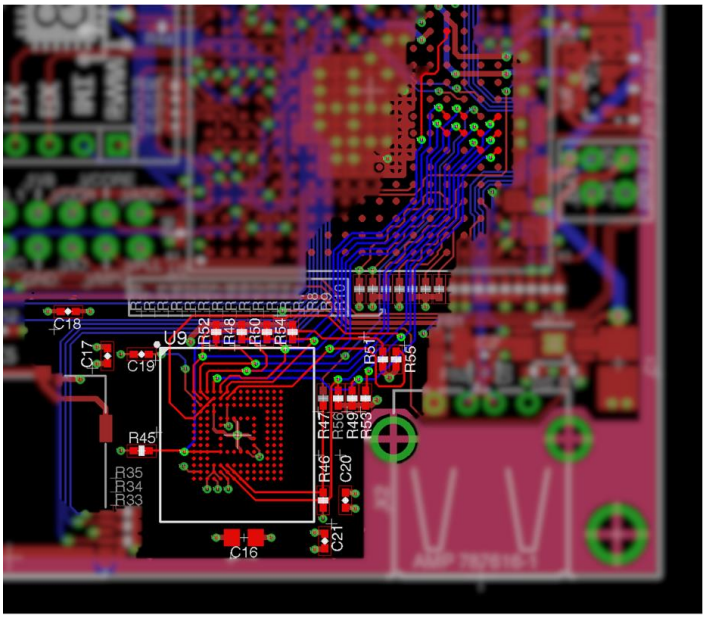


Figure 100 eMMC Layout

Perk:
The MMC peripheral within the AM335x supports the eMMC v4.3 JEDEC standard. However, this is an older standard and it is difficult to find parts that only support the eMMC v4.3 standard. Fortunately, the more recent eMMC v5.x standards are backward compatible. From a design and layout perspective, the only caveat is that there are a number of pins in the newer standard that must be understood. For example, there are a number of No Connect (NC) pins on the eMMC pinout. NC pins are purely structural and traces can route through those BGA balls to allow larger trace widths to be used to route eMMC devices. However, some of the NC pins in the eMMC v4.3 standard are actually used or reserved for future use (RFU) in the eMMC v5.x standards. Therefore, even though only the eMMC v4.3 signals are used, the device should be routed avoiding all of the other used or RFU pins. A more complete explanation is given in the ***Designing for Flexibility around eMMC*** application note. Also, the eMMC v5.x pinout is shown in Figure 101 for your reference.

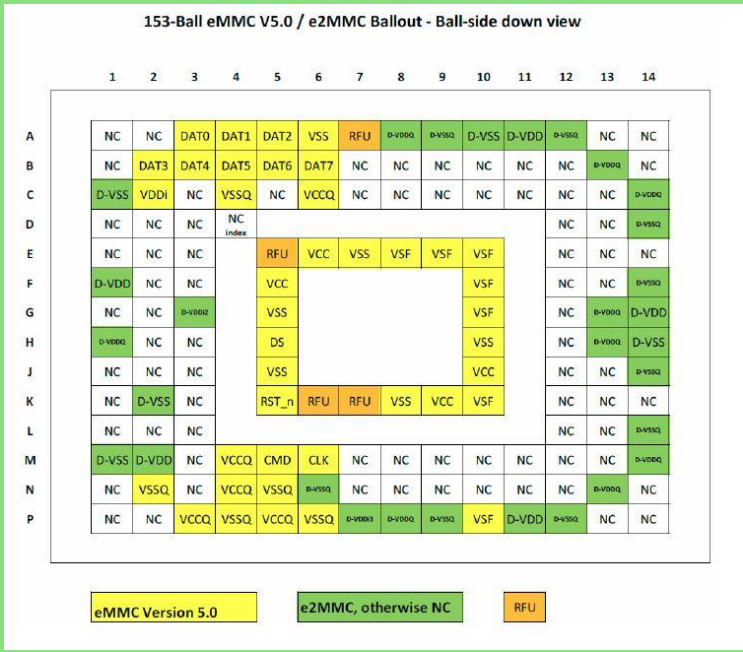


Figure 101 eMMC v5.x Pinout

14.3 EEPROM Circuitry

During boot, the U-boot bootloader of the Linux image you will be using read an on-board EEPROM for a board ID. This allows U-boot to configure the pins and set other appropriate boot environment variables. By default, the Linux kernel will not boot unless a valid board ID is found. However, this check can be bypassed in order to bring up a board or to perform the initial programming of the EEPROM. Given that we will be using a Linux image for this lesson, we need to have an EEPROM connected to I2C0 that is programmed with the appropriate information. More information about loading data to the EEPROM or bypassing EEPROM board ID check can be found on our forum posts #4733 and #4608.

The EEPROM circuit can be built as shown in Figure 102. By default, the Write Protect (WP) pin is pulled high through a 10K Ohm resistor. This means that the EEPROM is write protected and no data can be written to the EEPROM. In order to program the EEPROM, we need to provide test points such that it is easy to connect the Write Protect (WP) pin to a GND pin.

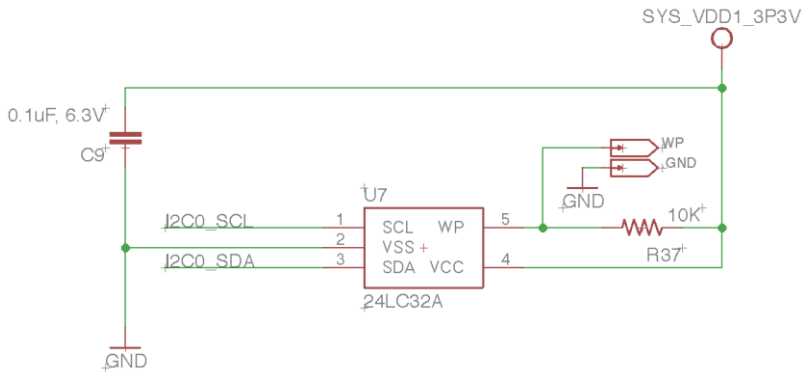


Figure 102 EEPROM Circuitry

EEPROM layout can be made as shown in Figure 103.

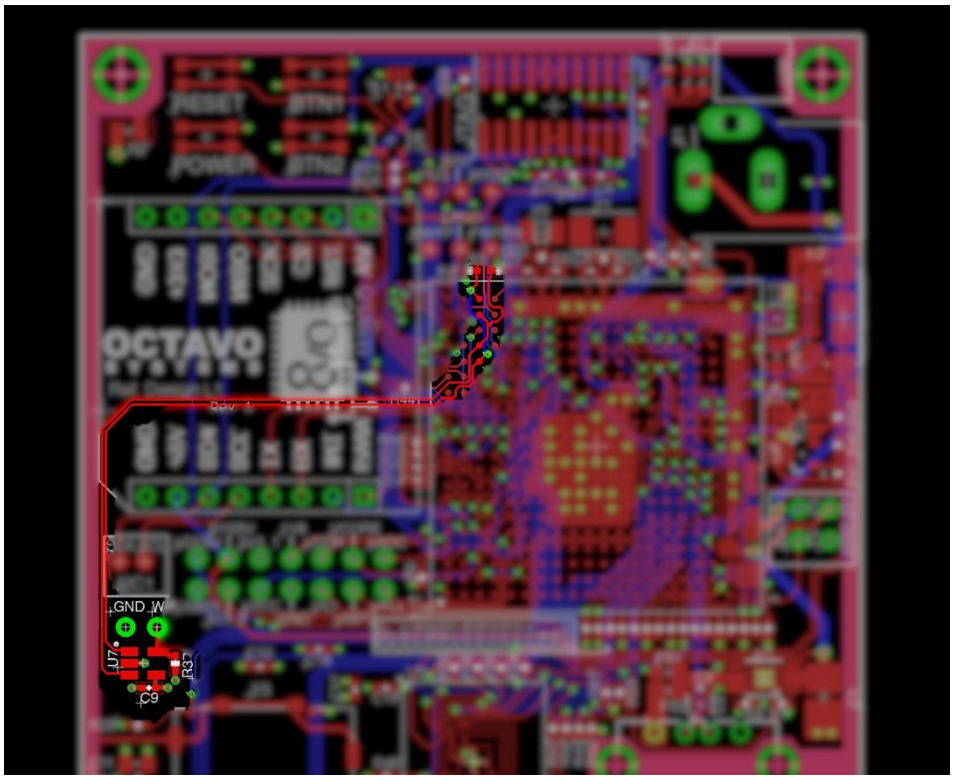


Figure 103 EEPROM Layout

****

15 Bringing Up a Custom Bare-Bones Linux PCB

15.1 Introduction

In this article, we will finalize the Lesson 2 board design and bring up the manufactured printed circuit board (PCB). This will allow us to explore new applications with our completed design and use the completed design as a starting point for future application specific designs.

The software operating system needed for the projects we will be running on the board is based on Linux. Because the OSD3358 is based on the Sitara® AM335x ARM® Cortex® A8 processor from TI, there are many Linux distributions available. The one we will be using is the ***Linux image from BeagleBoard.org*** ®. This Linux image is based on the Debian distribution of Linux and is robust and supported by a strong open source community. The ***BeagleBoard.org® Foundation*** is a US-based non-profit existing to provide education in and collaboration around the design and use of open-source software and hardware in embedded computing.

15.2 Finalizing the Lesson 2 Design

In this design, we have added USB ports and non-volatile storage. Besides the need to clean up the silkscreen on the board to fully document the design on the PCB, it is also useful to add some user defined LEDs that can be used to monitor the health of the PCB during operation.

15.2.1 Adding LEDs

During operation, it is important that a design provide feedback to a user so that they can understand if there are any issues with the operating state. One convenient way to provide feedback is with LEDs. Also, Linux provides an easy method, through the device tree, to connect LEDs to system events, such as CPU and memory activity.

By default, the Linux image from BeagleBoard.org ® or RED Linux image supports 4 user-defined (USR) LEDs. These LEDs must be connected to four GPIO pins as defined in the device tree in order to have the indicated functionality. The GPIOs corresponding to user LEDs and their default operation is summarized in Table 5.

Table 5 USR LED functionality

LED	AM335x GPIO Signal	Function
USR0	GPMC_A05	Blinks in heartbeat pattern
USR1	GPMC_A06	Blinks during SD Card activity
USR2	GPMC_A07	Blinks during CPU activity

USR3	GPMC_A08	Blinks during eMMC activity

You can see the mapping of these pins to their respective system function in the Linux device tree. The snippet of the device tree used trigger LEDs is shown in Figure 104.

```
17              device_type = "memory";
18              reg = <0x80000000 0x10000000>; /* 256 MB */
19      };
20
21      leds {
22              pinctrl-names = "default", "sleep";
23              pinctrl-0 = <&user_leds_default>;
24              pinctrl-1 = <&user_leds_sleep>;
25
26              compatible = "gpio-leds";
27
28              led@2 {
29                      label = "beaglebone:green:usr0";
30                      gpios = <&gpio1 21 GPIO_ACTIVE_HIGH>;
31                      linux,default-trigger = "heartbeat";
32                      default-state = "off";
33              };
34
35              led@3 {
36                      label = "beaglebone:green:usr1";
37                      gpios = <&gpio1 22 GPIO_ACTIVE_HIGH>;
38                      linux,default-trigger = "mmc0";
39                      default-state = "off";
40              };
41
42              led@4 {
43                      label = "beaglebone:green:usr2";
44                      gpios = <&gpio1 23 GPIO_ACTIVE_HIGH>;
45                      linux,default-trigger = "cpu0";
46                      default-state = "off";
47              };
48
49              led@5 {
50                      label = "beaglebone:green:usr3";
51                      gpios = <&gpio1 24 GPIO_ACTIVE_HIGH>;
52                      linux,default-trigger = "mmc1";
53                      default-state = "off";
54              };
55      };
56
57      vmmcsd_fixed: fixedregulator@0 {
58              compatible = "regulator-fixed";
```

Figure 104 BeagleBoard.org or RED Linux Device Tree snippet showing USR LED nodes

In the above code snippet, which can be found in ***dtb-rebuilder***,

- ***pinctrl*** parameters are used to setup pin multiplexing for LEDs. You can find more information at ***https://www.kernel.org/doc/Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt***.
- ***compatible*** property is used to bind the ***gpio-leds*** driver to the LEDs. See ***https://www.kernel.org/doc/Documentation/devicetree/bindings/leds/leds-gpio.txt***.
- ***label*** property is used to uniquely identify the LED.
- ***gpios*** property is used to assign a particular GPIO for the LED and also define its active state.

- ***linux, default-trigger*** is used to set a system event as trigger to the LED. See ***https://www.kernel.org/doc/Documentation/devicetree/bindings/leds/common.txt*** to know more about available triggers and their assignment.
- ***default-state*** determines the initial state of the LED.

To find more information about modifying the device tree for your custom board, please refer to our forthcoming articles on Linux.

The LED circuit can be built and added to Lesson 2 design as shown in Figure 105 and Figure 106.

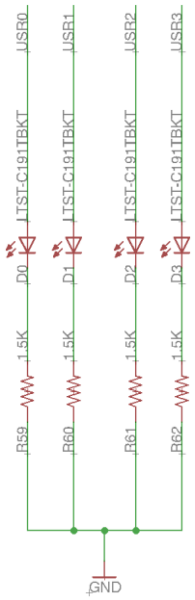


Figure 105 USR LED circuit

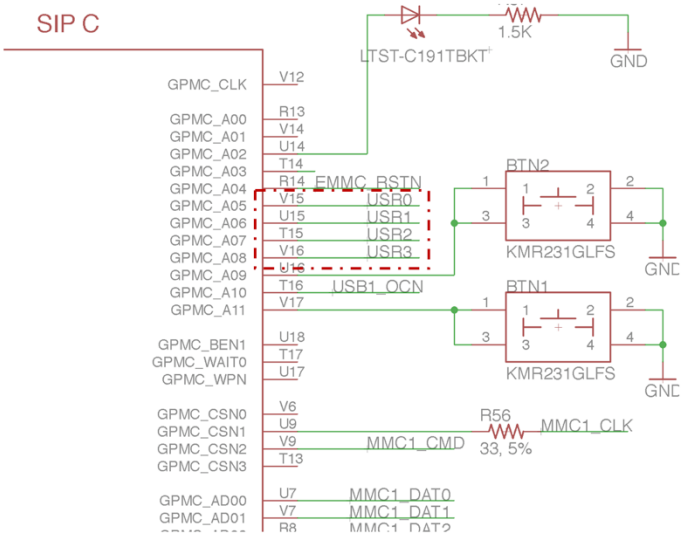


Figure 106 OSD335x USR LED connections

The LED circuit layout can be built as shown in Figure 107.

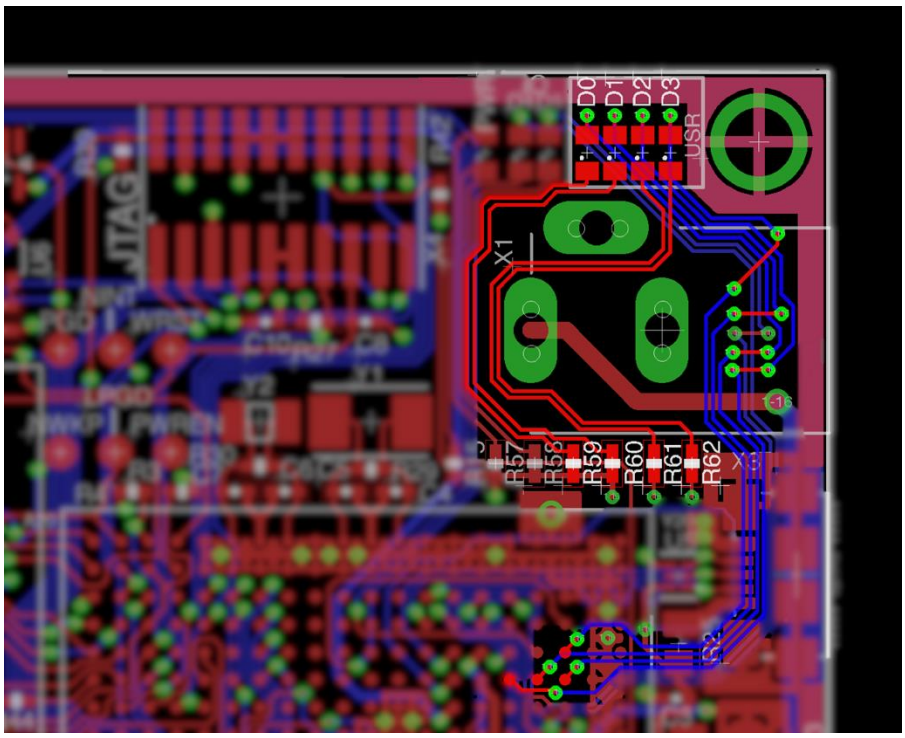


Figure 107 USR LED Layout

15.2.2 Finalizing the silkscreen

Once the design is complete, the silk screen should be finalized to fully document the PCB. The steps needed to finalize silkscreen are discussed as part of the ***Finalizing the silkscreen*** Section of OSD335x Peripheral Circuitry Article from Lesson 1.

15.2.3 Expected outcome

Now that we have completed the Lesson 2 design, the completed schematic should look similar to Figure 108 and completed layout should look similar to Figure 109, Figure 110 and Figure 111.

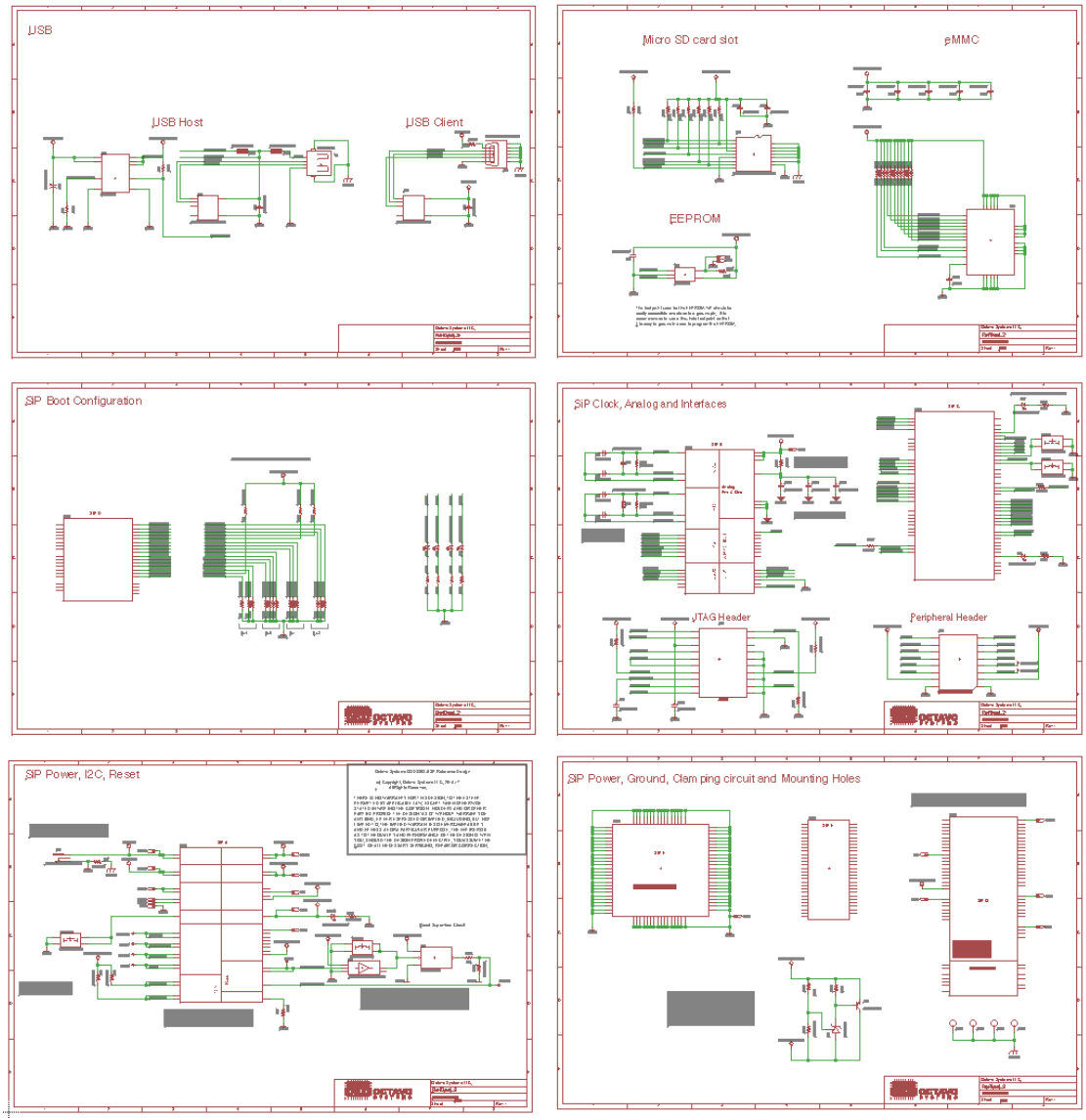


Figure 108 Lesson 2 complete schematic

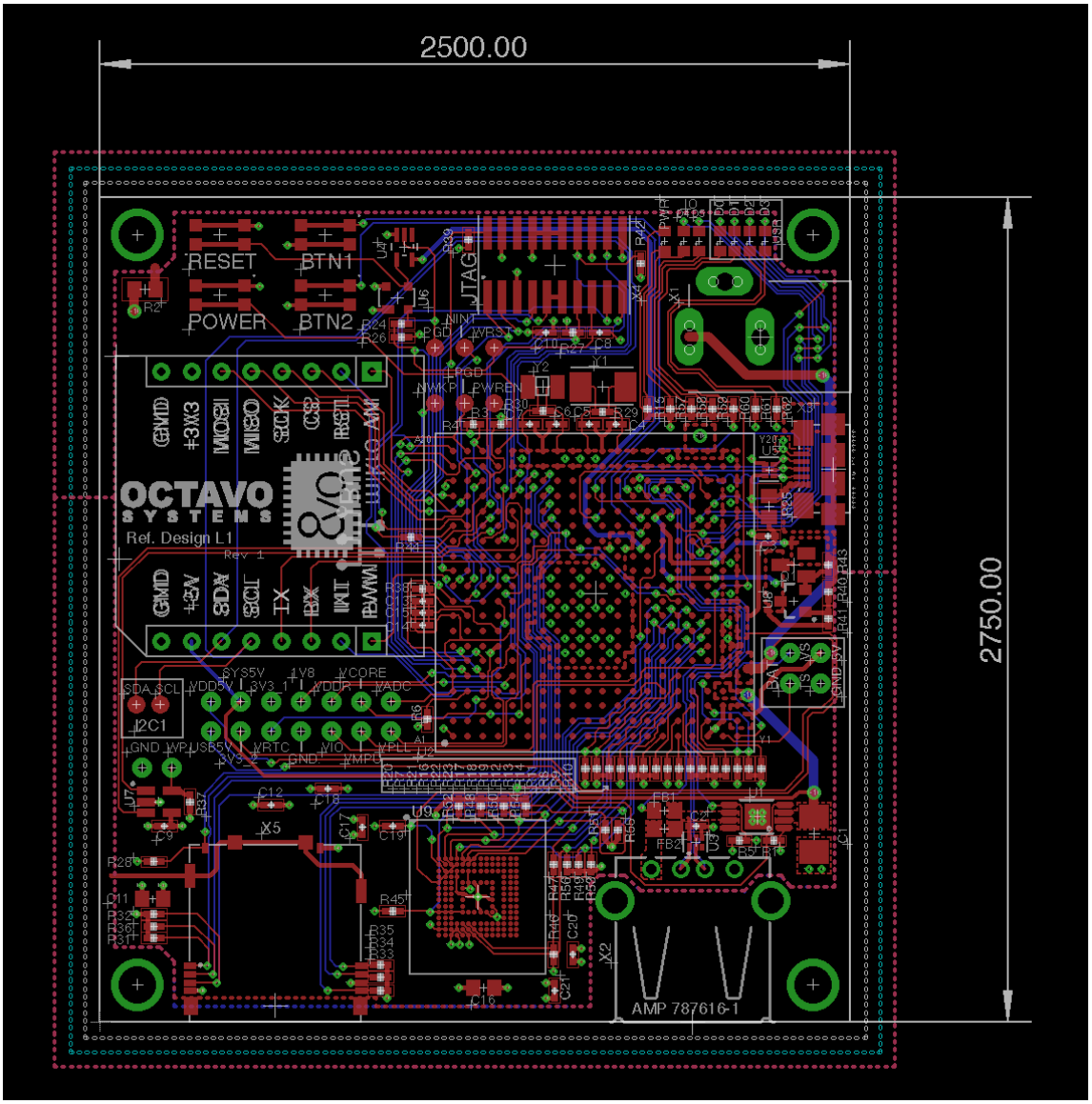


Figure 109 Lesson 2 complete layout with pour outlines

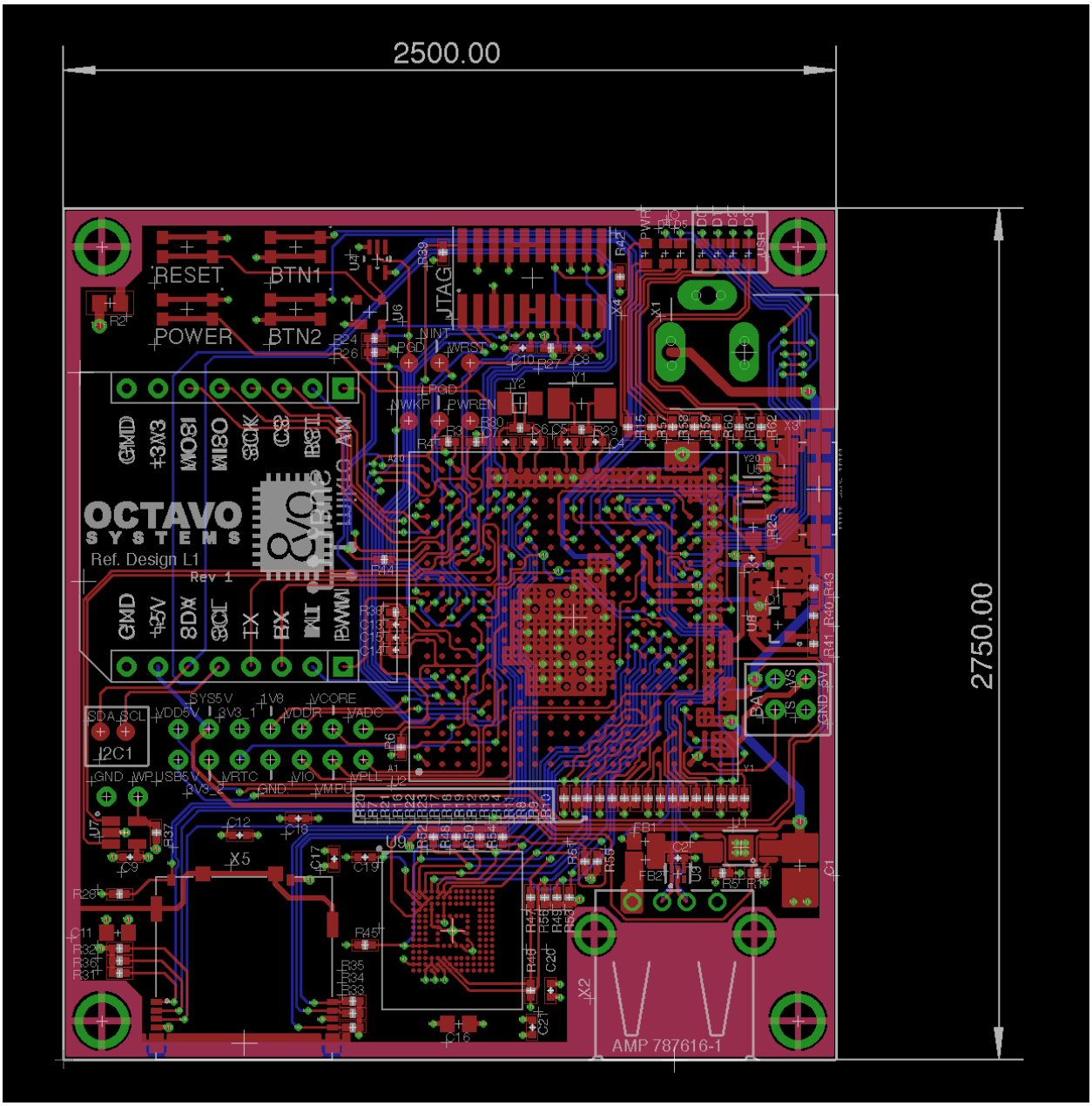


Figure 110 Lesson 2 complete layout with pour

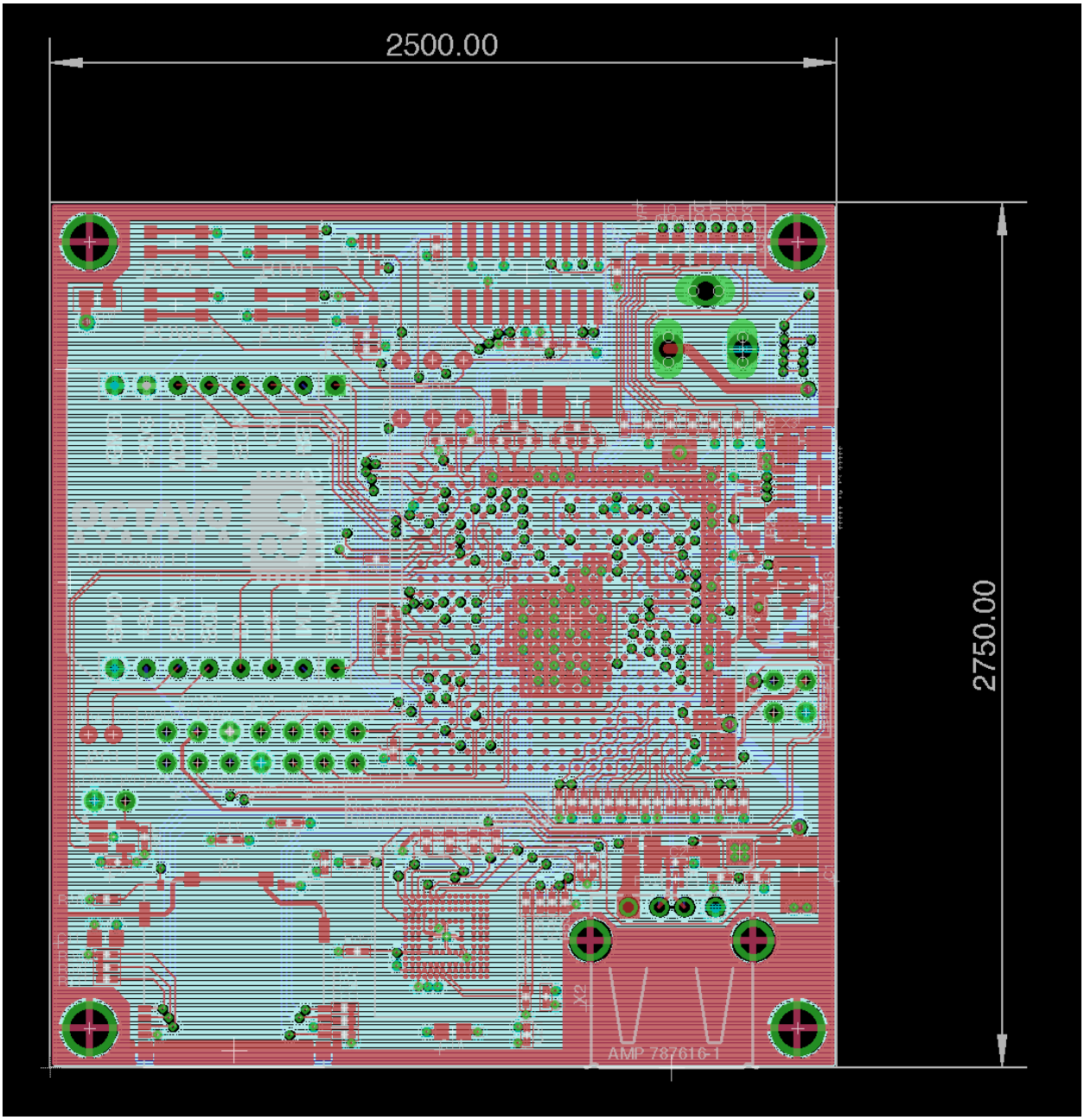


Figure 111 Lesson 2 complete layout with all layers turned on

15.3 PCB Manufacturing

To have the Lesson 2 design manufactured, please follow the procedure discussed in the ***PCB order process*** Section of OSD335x Peripheral Circuitry Article from Lesson 1.

15.4 Bringing up the PCB

Once you have the manufactured Lesson 2 design, please follow the procedure discussed in the ***Basic board bring-up*** Section of OSD335x Bare Minimum Board Boot Process Article from Lesson 1.

15.5 Booting Linux

The Lesson 2 design can boot any of the Linux images from BeagleBoard.org ® or the RED Linux image. However, the system functionality will be slightly different since the Lesson 2 design does not have the same peripheral set as other reference designs. The getting started page of BeagleBoard.org ® will help you flash the uSD card, eMMC and install necessary drivers. Although these instructions are intended for one of the BeagleBoard.org ® development boards, they can also be used to get started with the Linux image from BeagleBoard.org ® or the RED Linux image on the Lesson 2 design.

> Perk:
>
> Learn the basics of the booting Linux in Chapter 16 “Linux Boot Process”

One thing to be careful about when using the above images: the U-Boot bootloader within the images looks for a ***board id*** in an EEPROM attached to the I2C0 bus. The board id is used by U-Boot to identify the board so that the on-board hardware components can be initialized properly before booting the Linux kernel. If suitable information is not found in the EEPROM (e.g. the EEPROM is blank or has not yet been programmed) or if EEPROM is absent on the board, then U-Boot will not boot Linux kernel. Unfortunately, this check occurs before the serial console has been initialized so no boot messages will be displayed on the UART0 serial console making the board look like it is not working properly. We have developed an application note, ***EEPROM During Boot,*** specifically to guide you through any potential EEPROM configuration issues during boot.

This ***application note*** will discuss how the board ID is used in U-Boot, how to modify U-Boot to ignore the board ID, and how to program the board ID in an EEPROM either before boot or within U-Boot. Your preferred method to program the board ID, depends on the hardware in your system. For example, the tutorial board can boot from a microSD card, then modifying U-Boot to program the board ID will likely be your preferred method. If you have an external I2C programmer, that programming method is also covered in the application note. Finally, if you would like to program the EEPROM over USB from a host PC, software is provided in the associated zip file.

15.6 Demo Application

Once Linux is running, we can now explore different applications on the Lesson 2 design. For example, we can use a Motion Click to detect motion and alert a user by flashing some LEDs. The Motion Click, an add-on board from MikroElektronica, is built around a pyroelectric sensor that pulls the interrupt pin high whenever it senses motion of objects that emit infrared, such as living bodies. The demo app consists of a simple bash script which polls the status of the interrupt pin of Motion Click when it is plugged into the peripheral header and blinks LEDs D4 and D5 whenever motion is detected. The demo app can be ***downloaded here***.

To run the demo app on the Lesson 2 board:

a. Copy the demo app to the uSD card or eMMC of the Lesson 2 board.
b. Make the demo app file executable by using the command:

```
chmod +x DemoApp_L2.sh
```

c. Run the demo app with the command:

```
sh ./DemoApp_L2.sh
```

(1) ***dtb-rebuilder*** is developed and maintained by Robert C Nelson

****

16 Linux Boot Process

16.1 Introduction

Linux is a free and open-source operating system created by Linus Torvalds. Due to its openness, flexibility, and tremendous community support and development, Linux has become the operating system of choice for most embedded systems like industrial control systems, robotics applications and IoT devices. There are many Linux distributions, such as Debian and Ubuntu, that pull together utilities, libraries and application software around the Linux kernel to provide a development and execution environment for custom application software.

This article will focus on understanding the boot process of a OSD3358-SM-RED Debian Linux image running on OSD335x.

16.2 OSD335x Debian Linux Boot Process

Like many processors, the Texas Instruments AM335x processor inside the OSD335x Family of devices uses a multi-stage boot process to load and run the operating system. This is due to factors such as flexibility in the boot peripherals, boot speed, processor memory limitations, etc. The four boot stages for a standard Linux boot are shown in Figure 112.

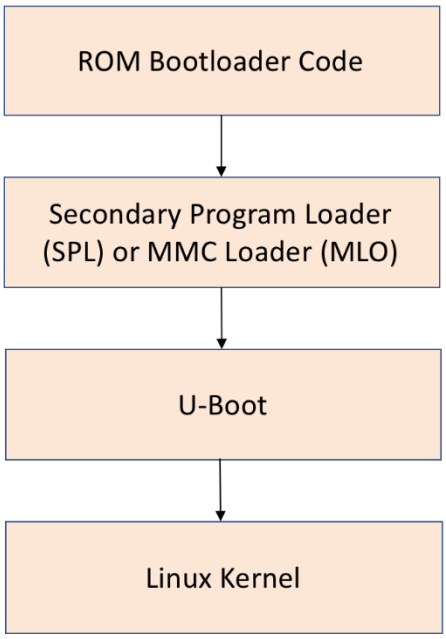


Figure 112 Linux Boot stages on OSD335x

16.2.1 Stage 1: ROM Bootloader

The AM335x contains a section of Read-Only Memory (ROM) that implements the first stage bootloader. The ROM Bootloader code is the first piece of code that is executed by the processor when it is released from reset. The ROM is a hard coded piece of software and cannot be changed for a given device but may change between revisions of the processor. The ROM bootloader has the following responsibilities:

- Initial configuration of the device and initialization of boot peripherals
  - Memory section setup (stack, heap, etc.)
  - Configuration of Watchdog Timer 1 (set to three minutes)
  - Configuration of PLL and System Clocks
- Load and begin execution of the next stage bootloader
  - Check boot peripherals for next stage bootloader (SPL/MLO)
  - Load bootloader from peripheral into the internal RAM of the AM335x and begin execution of the code

The AM335x defines sixteen (16) boot configuration pins (SYSBOOT[15:0]) that are set by the hardware design to inform the bootloaders about hardware system parameters. These parameters include input clock frequency, Ethernet or flash memory configuration (if applicable), output clock configuration, and boot sequence. The ROM bootloader uses the boot sequence parameter to search a given set of boot peripherals for the next stage bootloader. The value of the SYSBOOT pins can be read by the processor via the *control_status* register (address 0x44E1_0040).

For example, in the Lesson 2 design, by default we set SYSBOOT[15:0] = 0x4018 which sets a boot sequence value of 11000b. From the ***SYSBOOT Configuration Pins*** section (Table 26-7) in the AM335x Technical Reference Manual (TRM), this means that the ROM bootloader will search SPI0, MMC0, USB0 and UART0 for the next stage bootloader, in that order. If a properly formatted bootloader is not found, the ROM bootloader will continue to poll the last peripheral, i.e. UART0, until a bootloader is found, or the processor is reset. In the case of UART0, the polling of the bootloader can be recognized by the processor outputting a series of "***C***" characters on UART0.

It is important to set the boot sequence properly since only the peripherals defined by the boot sequence will be checked by the ROM bootloader for the next stage bootloader. Similarly, it is important to make sure the boot sequence order is correct if multiple peripherals will be used to boot, such as an SD Card and eMMC.

Caveat:

By default, the ROM code in the AM335x processor configures the UART0 port to operate at “115200 8N1” (i.e. 115200 baud rate, 8 data bits, no parity bits, and 1 stop bit). The Beagleboard.org Debian Linux image also configures the UART0 port with the same settings. You will need a UART serial communication client, like Putty, and a USB to Serial UART converter cable to communicate with the UART0 port of AM335x using a host computer. A good tutorial on how to use Putty and a USB to UART Cable with both PC, Mac ***can be found here***. If there is a mismatch between the clock settings (i.e. the SYSBOOT[15:14] pins) and the actual crystal frequency/oscillator frequency supplied to OSC0 input, then you may get strange characters on the UART0 interface because the UART port will not be operating in the correct frequency. If you see strange characters on the UART0 interface, then you should use an oscilloscope to measure the width or period of UART bits on the TX line to determine the baud rate / communication frequency.

16.2.2 Stage 2: Secondary Program Loader (SPL)

The second stage bootloader is known as the Secondary Program Loader (SPL). Previously, it was also known as the MMC Loader (MLO) but that term has been deprecated but may still appear in documentation. The SPL must operate entirely within the internal memory of the AM335x processor since only the boot peripherals have been initialized by the ROM bootloader. In the case of bare-metal applications that are small enough to fit within the internal memory, the boot process can end at this stage and the application can run. However, this is not the case for Linux.

To boot Linux, one of the most common methods is to use U-Boot (Universal Boot Loader) to perform all of the steps necessary to load and boot the Linux kernel. U-Boot provides a feature rich environment to accomplish its tasks. However, the feature rich U-Boot environment requires more memory than is available in the AM335x internal memory. Therefore, U-Boot is split into a first-stage and second-stage bootloader. The first stage of U-Boot is small and can be used as the SPL for the OSD335x Linux boot process. This split is done automatically during the build process for U-Boot, but the pieces are loaded into separate parts of the boot image.

The main function of the stripped-down SPL version of U-Boot is to perform hardware initialization of the DDR3 memory within the OSD335x, load the larger, fully featured version of U-Boot into DDR memory, and begin execution of that code.

16.2.3 Stage 3: U-Boot Bootloader

The third stage of the OSD335x Linux boot process is the second stage of the U-Boot bootloader. The full-featured version of U-Boot that is being run at this stage is extremely powerful and includes an interactive shell, environment variables, as well as command line utilities to initialize and interact with many different peripherals. These features make U-Boot a very popular bootloader for many embedded devices running Linux.

Once the U-Boot console has been activated, it is possible to stop the automatic boot process via a serial terminal and use the command line interface to interact with the hardware and test its functionality. For example, you are able to interact with I2C or MMC devices to make sure that they are powered and at the correct address. The U-Boot console typically uses the UART0 interface. However, there are interesting projects such as Netconsole that allows the serial terminal to be used over a network interface.

The main function of the U-Boot bootloader is to load and begin execution of the Linux kernel. To do this, it will typically look for a ***uImage*** file, which is contains both the Linux kernel and a header that describes the kernel. The *uImage* file can be found

in non-volatile memory attached to the processor, such as an eMMC or a microSD card, or over a network interface via a protocol like TFTP.

The U-Boot environment can be configured by setting environment variables. These environment variables can be 1) configured during the build of U-Boot; 2) set and saved during an interactive U-Boot session; or 3) set or overridden from a file called ***uEnv.txt*** which is stored in the */boot* directory of the filesystem.

16.2.4 Stage 4: Linux Kernel

In the final stage of boot, the Linux kernel is started which will boot and configure the Linux operating system. During this part of the boot process, all the necessary device drivers are loaded and configured so that the system can operate properly.

The Linux kernel is wrapped in a header that describes it and the two together create the ***uImage*** file that is loaded and executed. The header is 64kB and includes information like the target architecture, the operating system version, kernel size, checksum to verify the kernel image was loaded correctly, etc. When U-Boot loads the *uImage*, it displays the header information on the serial console as shown in Figure 113.

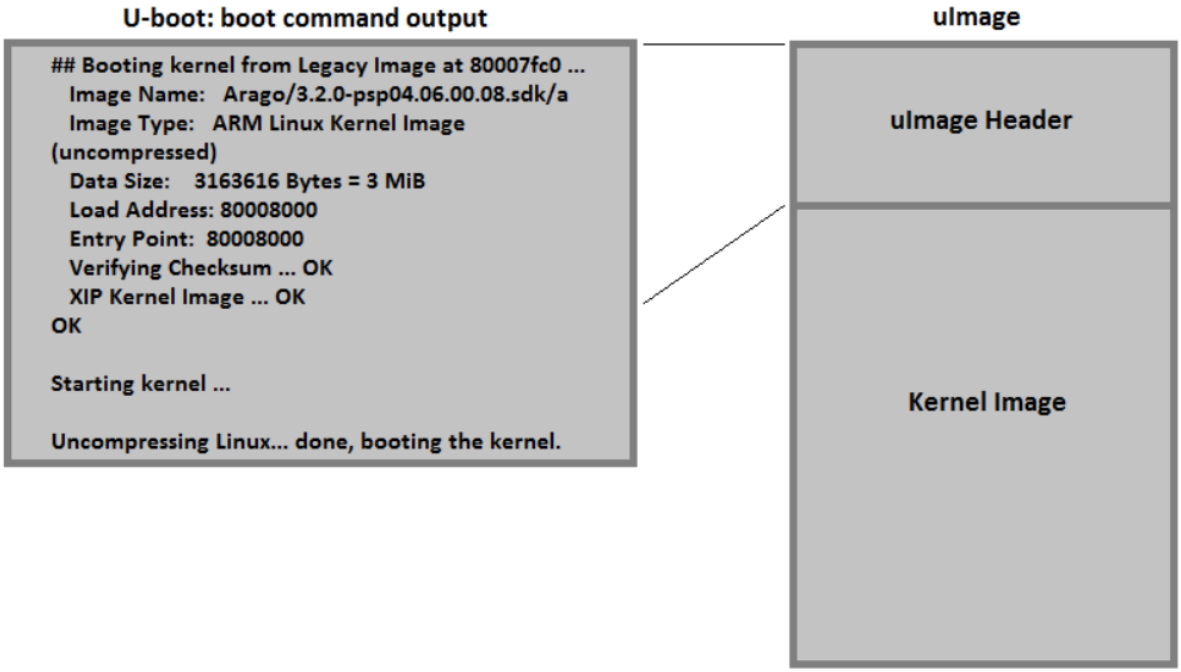


Figure 113 uImage header info displayed by U-Boot

Once the kernel boots, depending on the *loglevel* and *quiet* options that are part of the *cmdline* variable in *uEnv.txt*, various boot messages will appear on the serial console as drivers and services are loaded. These messages are important debug resources if there are any issues with hardware or software components of the system. Eventually, a login prompt will appear, see Figure X, on the serial console. You can use this to log into the system. This login is important if there are any issues during boot since it allows access to the system when other methods of interacting with the system, such as ssh over a network or a graphical user interface, might not be working properly.

```
[    3.023236] bone_capemgr bone_capemgr: slot #3: auto loading handled by U-Boot

Debian GNU/Linux 9 beaglebone ttyS0

BeagleBoard.org Debian Image 2017-08-31

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

beaglebone login:
```

Figure 114 UART0 serial login prompt

At this point, the Linux operating system has completely booted and you are now able to use it as you would any other Linux operating system to run programs and applications, communicate with system hardware, etc.

16.2.5 Boot Process Memory Usage

While it is important to understand the different stages of the boot process, it is also important to understand where each boot stage lives in memory and the different memory it uses during boot.

Before the device powers up, the boot code lives in one of two places. First, the ROM Bootloader resides permanently on the AM335x processor and cannot be modified. Second, all other pieces of the boot image (ie the second and third stage bootloaders, and the linux kernel and filesystem) are in non-volatile storage (e.g. a microSD card or eMMC). Once the device powers up, the ROM Bootloader will load the second stage bootloader, i.e. the SPL, into internal AM335x memory. The SPL will load the third stage bootloader, i.e. the full featured version of U-Boot, into the DDR memory that is within the OSD335x device. Finally, the last bootloader stage, i.e. the Linux kernel, will be pulled into the DDR and executed. At that point, the Linux operating system will run using all of the resources of the AM335x and the OSD335x as well as continuing to use the non-volatile storage as the file system. This is shown in Figure 115.

You can download pre-built Debian Linux images that consist of all necessary pieces to boot from BeagleBoard.org ® or the RED Linux Image Download.

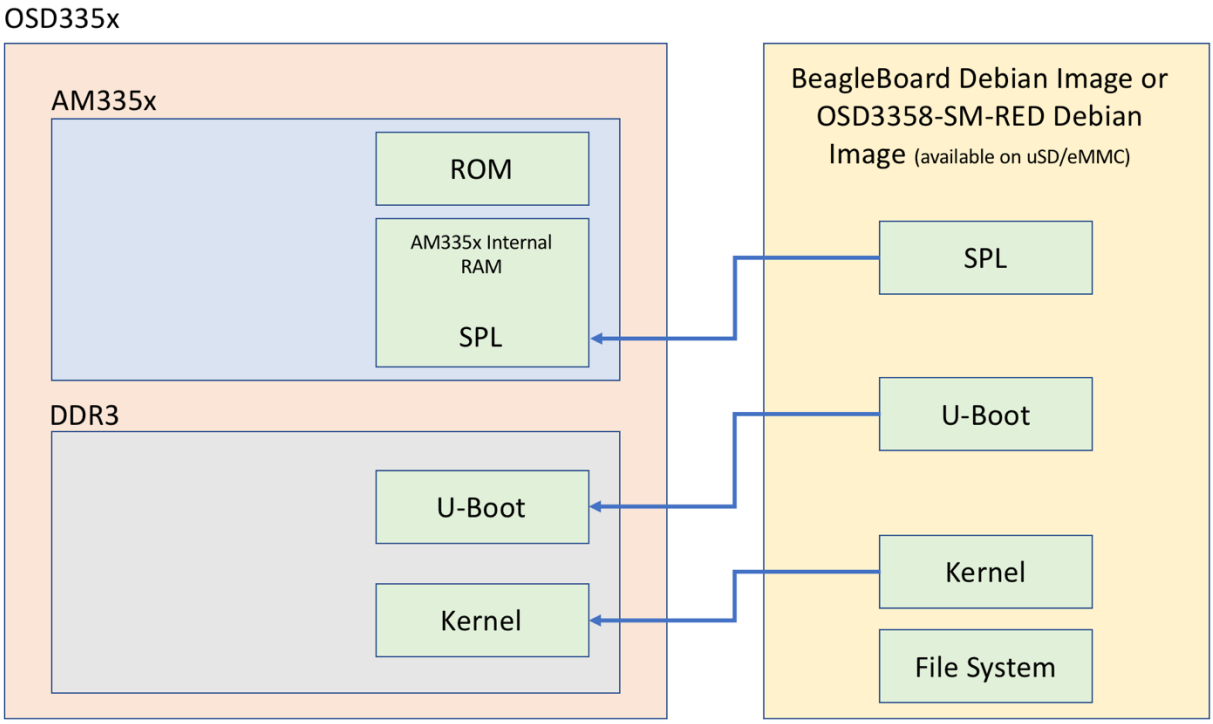


Figure 115 Boot Process Memory Usage

*****

17 Linux Device Tree

17.1 Introduction

Most modern general-purpose computers, like a desktop or laptop, will consist of several peripherals connected to a main processor through a bus such as PCI, USB, etc. An operating system, such as Windows or Linux, running on the computer can discover or learn about the connected peripherals through enumeration. Enumeration is a process through which the OS can enquire and receive information, such as the type of the device, the manufacturer, or the device configuration, about all the devices connected to a given bus. Once the OS gathers information about a device, it can load the appropriate driver for the device. However, this is not the case when it comes to most embedded systems.

In embedded systems, many peripherals are connected to the main processor with busses like I2C, SPI, and UART, which do not support enumeration. Therefore, when an embedded system uses an operating system, such as Linux, it is necessary to convey the hardware configuration of the system, i.e. all the information about the connected peripherals, to the OS in a standard form. The most common way to do this in Linux is to use a device tree. A device tree is a tree data structure that describes the hardware configuration of the system to the Linux operating system. During boot, the Linux kernel will use the information in the device tree to recognize, load appropriate drivers and manage the hardware devices in the system.

This article will help you get started with Linux device trees by introducing the structure of device trees, describing some properties of device trees and showing you how to modify an existing device tree for your custom hardware configuration.

> Perk:
>
> Devices which use buses that support enumeration do not need to be included in the device tree. For example, devices that are connected to a USB Host port do not need to be included in the device tree, since the USB bus protocol supports enumeration. Similarly, all devices that are connected to an I2C bus must be included in the device tree, since the I2C bus protocol does not support enumeration.

17.2 Device Tree Structure and Properties

A Linux device tree begins from a root node (i.e. the Tree Root) and will consist of a level of child nodes and one or more levels of children nodes. Each child node represents a hardware component of the micro-processor. For example, in the OSD335x, each child node represents a component of the AM335x processor, such as the CPU, an I2C peripheral, etc. Each children node represents a sub-component of a child node or a device attached to the child node. For example, in the OSD335x, the TPS65217C PMIC is attached to the I2C0 peripheral bus and appears as a children node under the I2C0 child node. Each node consists of property-value pairs that describe the hardware of the node. Each child node or children node can have only one parent and the root node has no parent. A block diagram of a simple device tree structure is shown in Figure 116.

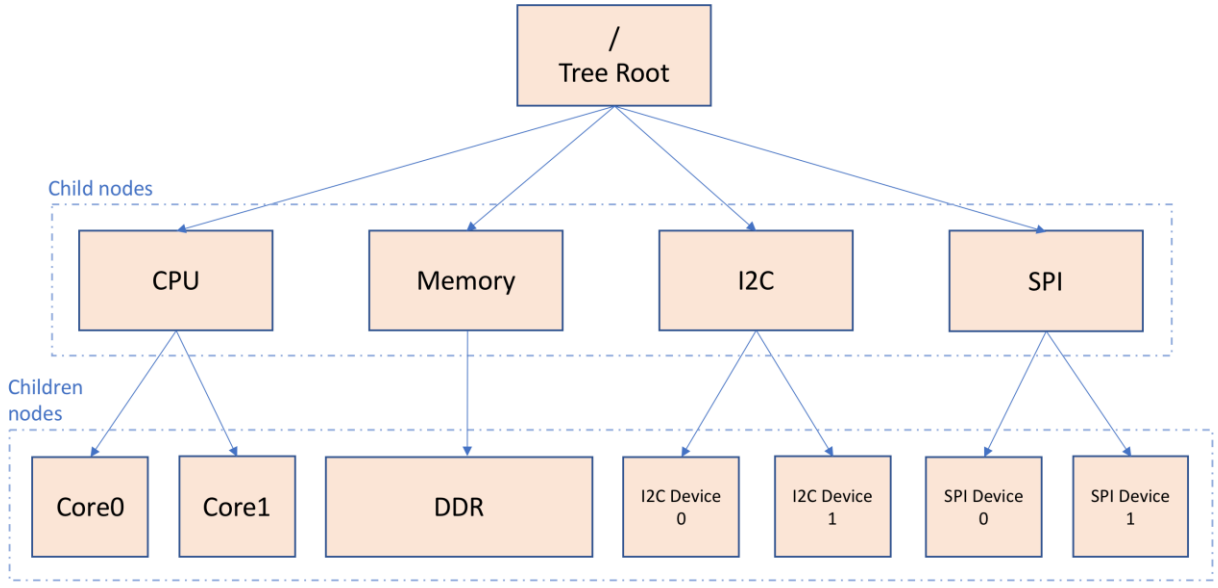


Figure 116 Block diagram of a simple device tree structure

In the above picture, you can see the parent-child relationship that exists between the child and children nodes. For example, all CPU cores are grouped as children nodes under CPU child node. Similarly, all I2C devices on a given I2C bus are grouped as children under that I2C node.

> Perk:
>
> As mentioned above, a parent-child relationship exists between child and children nodes. Although it is counter intuitive to call *parent node* a *child node* and *child node* a *children node*, this was done to match the terminology used in the official device tree documentation.

To express these relationships, the Linux device tree has a specific syntax and structure that should be followed to ensure a device tree works as intended. Let's look at a simple device tree template (Figure 117) that shows the generic syntax of the root node, child nodes, children nodes and property-value pairs.

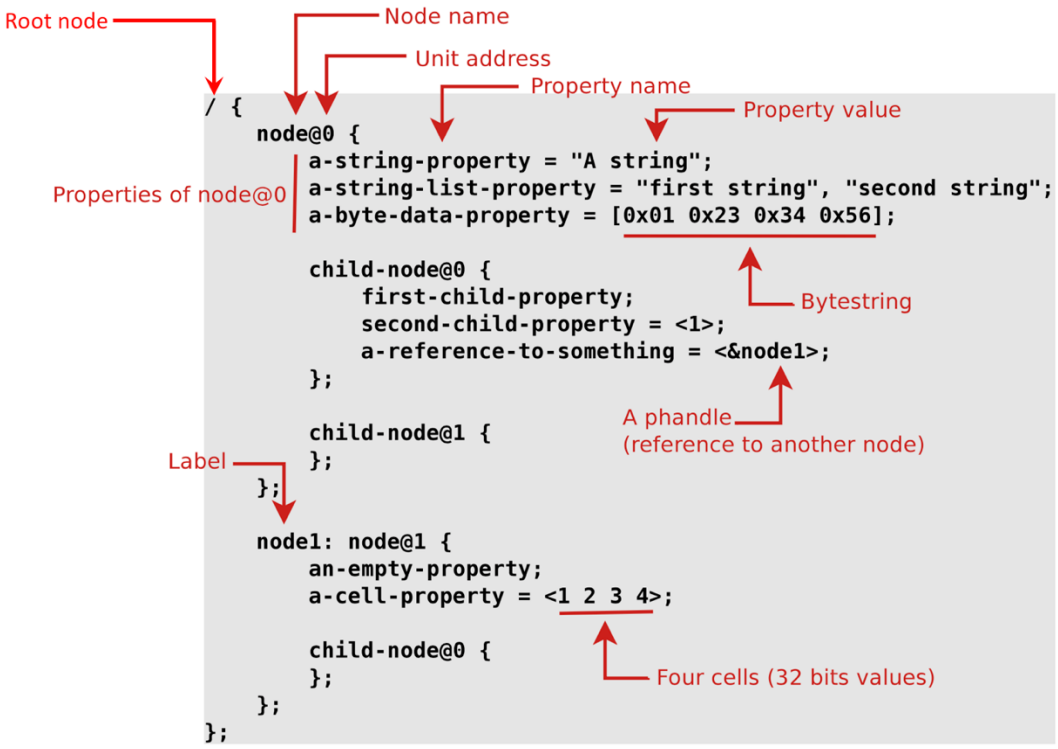


Figure 117 Simple Device Tree Template (© Thomas Petazzoni, Device Tree for Dummies)

In the above figure:

- ***node@0*** and ***node@1*** are child nodes.
- ***child-node@0*** and ***child-node@1*** are children nodes of their respective child nodes.
- ***node1*** is an easy to remember label of node@1.
- ***a-string-property***, ***a-string-list-property*** etc., are properties that are used to describe the child and children nodes.
- ***<&node1>*** is a phandle (i.e. a reference to node1 from node@0).

The official Device Tree Specification can be found here (this article will refer to information from version v0.2). A more detailed explanation of the device tree structure and basic concepts is available on the Device Tree Usage webpage. Please make sure you are familiar with these two documents before proceeding.

Now that the structure and the syntax of a device tree is clearer, the next question is: what property-value pairs should be used to describe a particular component and where can that information be found? The answer can be found in Device Tree Bindings. The properties that are necessary to describe a particular component in the device tree depends on the requirements of the Linux driver, or kernel module, for that component. All the required properties must be supplied to ensure that the Linux kernel recognizes the component properly and loads the appropriate drivers for that component. The Device Tree Bindings for a particular hardware device will give you this information. Therefore, it is essential to find the right Device Tree Bindings information for your component. More supporting documentation about device tree bindings can be found on the Device Tree Reference webpage.

As an example of Device Tree Bindings, the Lesson 2 board has a peripheral header that supports MikroElektronika Click Boards. To use a MPU 9DOF Click that has MPU-9150 9-axis motion tracking component, the appropriate information must be added to the device tree. This Click communicates with the host using I2C and has a single interrupt line to the processor. The device tree bindings for this component can be found here. Based on this, the following entry would be added to the device tree as a children node under the appropriate I2C child node:

```
mpu9150@69 {
      compatible = "invensense,mpu9150";
      reg = <0x69> ;
      interrupt-parent = <&gpio0> ;
      interrupts = <23 1> ;
};
```

The ***unit address*** (i.e. the *@69* after the *mpu9150*) and ***reg*** values are both ***69*** because the MPU 9DOF Click's address on the I2C bus is 0x69. The INT pin of the peripheral header, which is used for interrupts from the MPU 9DOF Click, is connected to the GPMC_AD9 pin of the OSD335x (which is the GPMC_AD9 pin of AM335x, see the Lesson 2 board schematics for more information). The GPMC_AD9 pin is also available to the system as bit 23 in GPIO bank 0 (see the ***Pin Attributes*** section of AM335x datasheet). Hence, the ***interrupt-parent*** for this device is a phandle for gpio0 node and the first value of ***interrupts*** property is 23. The second value of the ***interrupts*** property is used to describe the behavior of the interrupt, i.e. the trigger type and level, which can be found in the AM335x TRM. See here for more information about device tree interrupts.

The device tree bindings for all the hardware components must be consolidated to form a complete device tree so that the system can function properly. While this may seem like a daunting task, device trees are seldom built from scratch. The next section will discuss modification and reuse of existing device trees.

17.3 Modifying an Existing Device Tree

For an embedded Linux system, a device tree is generally a complex data structure requiring several hundred nodes to describe the entire system hardware architecture. Creating a device tree from scratch and validating it can be a daunting and time-consuming task. Therefore, you should look to reuse device tree include files and modify an existing device tree to meet your requirements.

A device tree for the Lesson 2 board can be created by modifying the published device tree from the OSD3358-SM RED board. You can download the device tree files here.

The OSD3358-SM RED board has several components that do not exist on the Lesson 2 board. Those components will need to be removed from the device tree to make it suitable for the Lesson 2 board. However, before doing this, you need to understand how the OSD3358-SM RED board device tree is structured.

The device tree consists of 2 files:

- ***osd335x-sm.dtsi*** – This is a ***device tree include*** file that describes all the hardware that is present in the OSD335x-SM System-in-Package (SiP) device.
- ***osd3358-bsm-refdesign.dts*** – This is the ***main*** device tree file that includes the previous file and adds nodes corresponding to the hardware that is specific to the OSD3358-SM-RED board.

Dividing the device tree into separate files helps with code reusability. Since the internal hardware of the OSD335x-SM remains the same irrespective of where it's used, the device tree corresponding to it can be put into an include file (***osd335x-sm.dtsi***). Then, this file can be used in many device trees that utilize the OSD335x-SM.

When using device tree include files, it is important to understand the status of each node (i.e. after loading the include file, is a given node in an ***okay*** or a ***disabled*** state). The main device tree file (***osd3358-bsm-refdesign.dts*** in this case) can then enabled or disable nodes depending on the system hardware. The nodes with the preceding ampersand (&) in the main device tree file are referencing nodes that are already declared in ***osd335x-sm.dtsi***.

At this point, the OSD3358-SM-RED board's device tree can be modified to match the hardware on Lesson 2 board. The nodes and node references that do not correspond to the Lesson 2 board can be removed in ***osd3358-bsm-refdesign.dts***. Figure 118 lists all the nodes that are present in this file. The nodes that are marked with ***X*** should be removed.

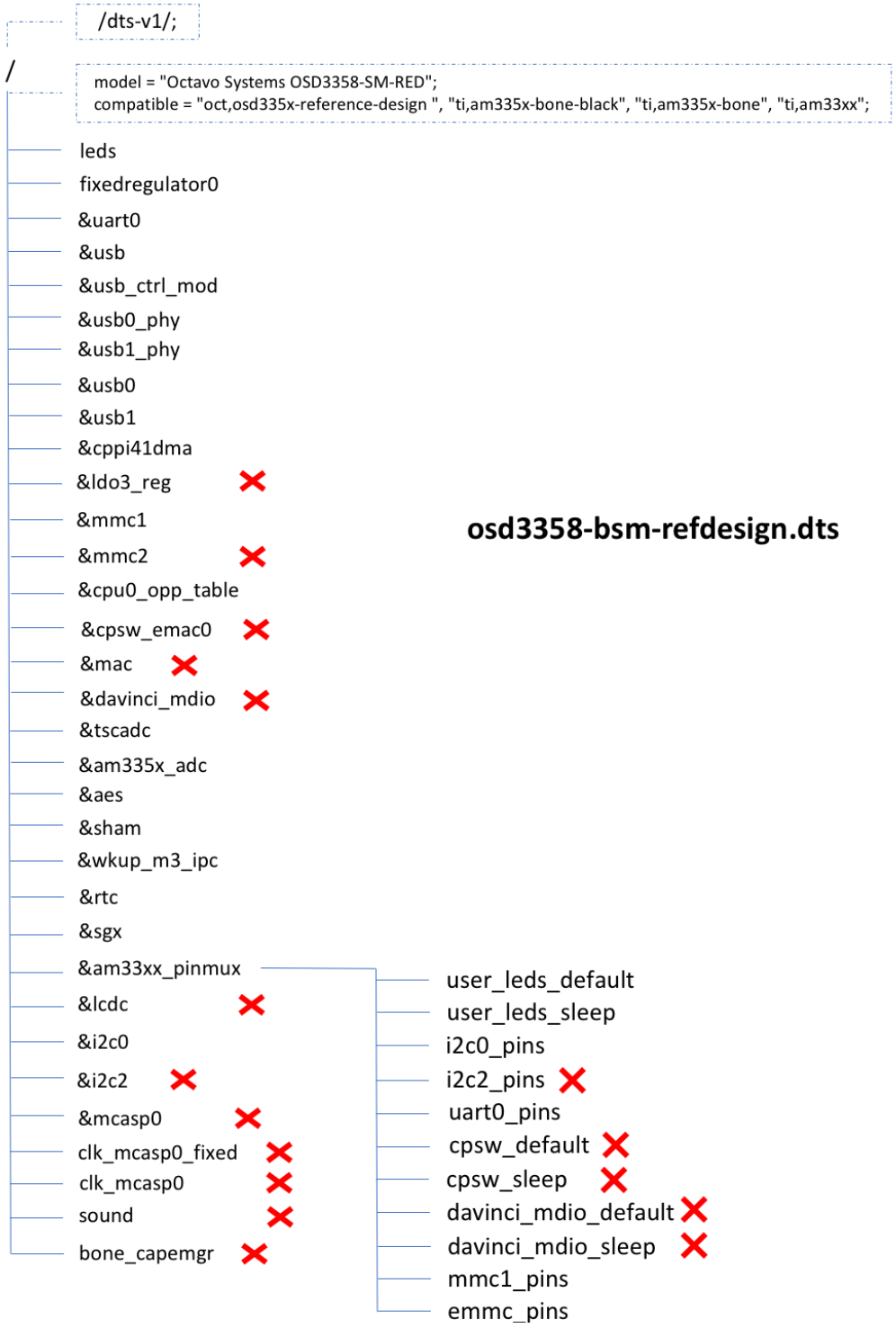


*Figure 118 **osd3358-bsm-refdesign.dts** before modifications*

After the modifications, the ***osd3358-bsm-refdesign.dts*** will look like Figure 119. Since it now reflects the Lesson 2 hardware, the file can be renamed to ***osd335x-lesson2.dts***.

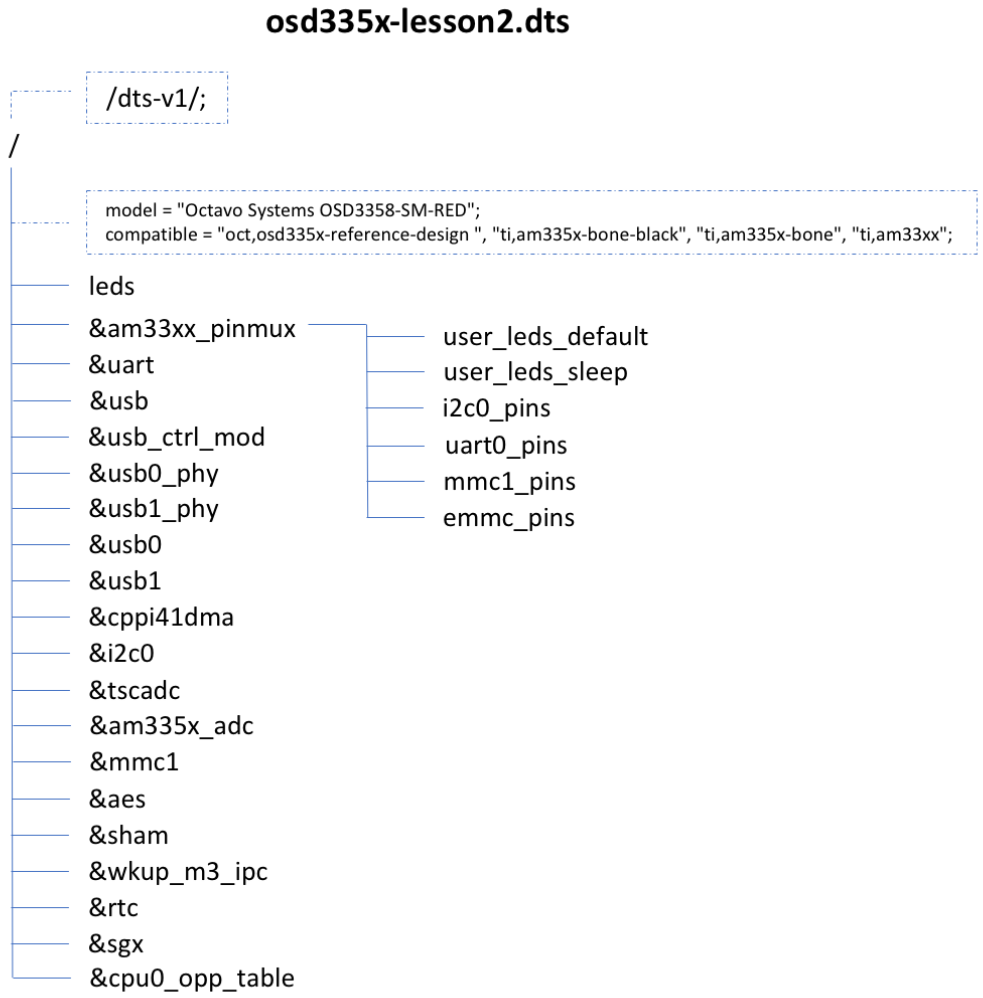


Figure 119 ***osd3358-bsm-refdesign.dts*** *after modifications to reflect Lesson 2 hardware*

You can directly download the ***osd335x-lesson2.dts*** device tree file for the Lesson 2 board here. The ***.dts*** device tree source files are human readable and can be viewed in your favorite text editor. Next, this source file will be converted to a ***Device Tree Blob*** or ***.dtb*** file, i.e. a binary file that is smaller and easier to parse by the Linux kernel. For the OSD335x Family of devices, Robert Nelson's ***Device Tree Rebuilder*** can be used to compile the device tree source. You can download it here (we have tested our ***.dts*** files using ***4.14-ti*** branch with the OSD3358-SM RED Debian image. https://octavosystems.com/files/osd3358-sm-red-linux-image/
For a different image, use a branch that matches the image's kernel version).

The steps to compile and use the new device tree on the Lesson 2 board are as follows:

1. Install a clean software image (optional)
    a. Download the latest OSD3358-SM RED Debian image (or a suitable Linux image from Beagleboard.org depending on your board) on your host PC
    b. Flash the image on a microSD card or the on-board eMMC
    c. Boot the board with the image
2. Install ***dtb-rebuilder*** by either
    a. Downloading ***dtb-rebuilder*** on your host PC and copying it your board. You can use Cloud9, SSH, or SFTP client app like WinSCP.
    b. Cloning the git repository if the board has internet connectivity
3. Copy source files
    a. Copy ***osd335x-lesson2.dts*** and ***osd335x-sm.dtsi*** files to the "*<install_path>/dtb-rebuilder/src/arm*" directory of the board.
4. Build the device tree
    a. Run the ***make*** command in the "*<install_path>/dtb-rebuilder*" directory of the board (see Caveat below for debug help).
    b. Result: The ***osd335x-lesson2.dtb*** file will be in the "*<install_path>/dtb-rebuilder/src/arm* directory.
5. Install the dtb file
    a. Copy the ***osd335x-lesson2.dtb*** file to "*/boot/dtbs/<kernel version>/*" directory to make it available to the Linux Kernel during boot.
6. Edit the boot configuration file (uEnv.txt) to use the new device tree:
    a. Open the /boot/uEnv.txt configuration file using a text editor. For example: "*nano /boot/uEnv.txt*"
    b. Comment out the existing ***dtb*** variable and add the line *dtb=osd335x-lesson2.dtb* as shown in Figure *120***.** Save the file.
    c. Reboot the board using the command: "*sudo reboot -h now*"

The board should now boot using the new device tree.

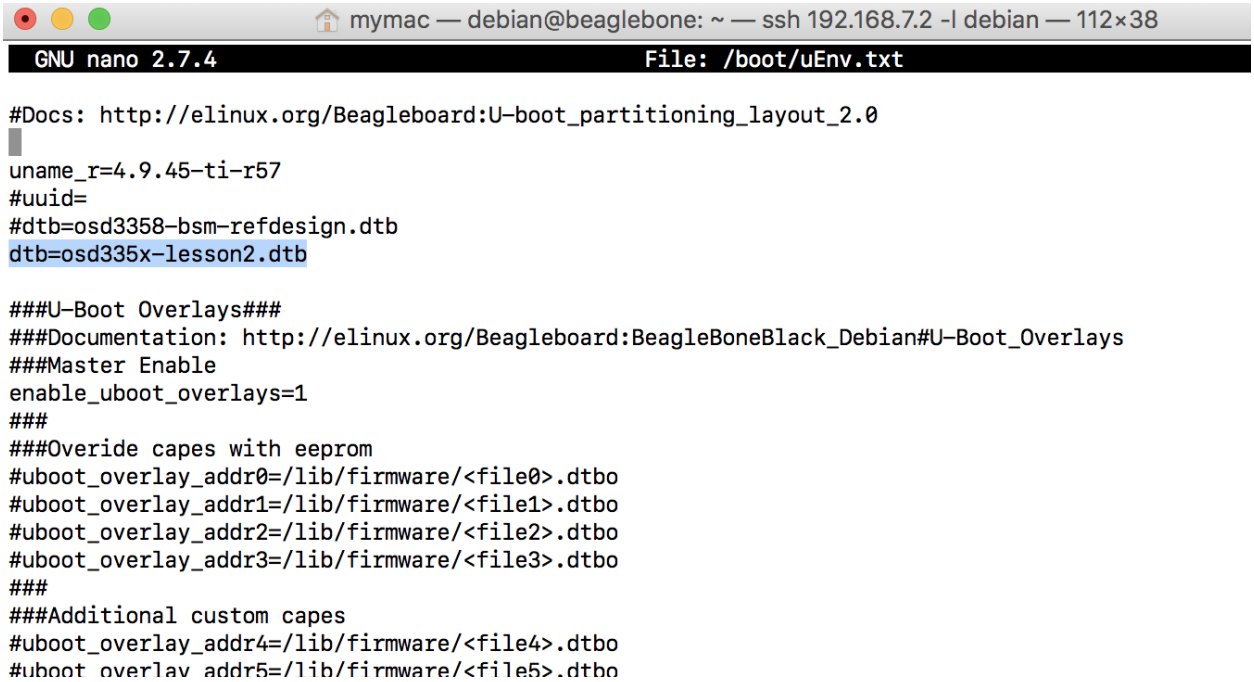


Figure 120 Setting Kernel Device Tree file in uEnv.txt

Perk:

To understand what was loaded during the boot process, you can look at the boot messages on the serial console (by default UART0) during boot or in the system logs after boot. This can help you debug any boot issues.

Caveat:

If you get ***time skew*** warnings while building the device tree, run "***touch ****" in:

- "*<install_path>/dtb-rebuilder*"
- "*<install_path>/dtb-rebuilder/src*"
- "*<install_path>/dtb-rebuilder/src/arm*"

directories to update the file modification time of all files to resolve the warning.

If there are any syntax errors in the device tree, the compiler will indicate the line number where the error was found. You can use this information to track and resolve the error. The most common error is phandle references to non-existing nodes. Since some nodes were deleted while modifying the device tree, any references to the deleted nodes might cause errors. Make sure to delete all phandle references to the deleted nodes to fix any errors.

17.4 Pin Multiplexing

The OSD335x family of devices provides access to all 123 signal pins of the AM335x processor. Each of these signal pins can have up to seven (7) different functions, or modes, which allows a given peripheral within the AM335x to be multiplexed to different pins. The pin modes for each pin can be found in the ***Pin Attributes*** table of AM335x datasheet (the OSD335x family uses ZCZ package of AM335x when referring to pin names / pin functionality). The ***RESET REL. MODE*** column of the table shows the default mode that will be assigned to the pin after the processor is released from reset. If this mode is not the mode required by system, you will need to add information to the device tree to set the pin mulitplexing so that the connected components interface properly with the AM335x processor.

For example, below shows a device tree snippet from the *am33xx_pinmux* node of the ***osd3358-bsm-refdesign.dts*** file. The ***pinctrl-single*** driver is used to set the appropriate pin configuration (more information about *pinctrl-single* driver can be found here). The AM33XX_IOPAD macro (which can be found in the *dt-bindings/pinctrl/omap.h* file, which is included by the *dt-bindings/pinctrl/am33xx.h* file, which is included by the *osd335x-sm.dtsi* file) helps configure each pin (more information about AM33XX_IOPAD macro can be found here).

```
&am33xx_pinmux {
       user_leds_default: user_leds_default {
              pinctrl-single,pins = <
                     AM33XX_IOPAD(0x854, PIN_OUTPUT_PULLDOWN | MUX_MODE7)
/*gpmc_a5.gpio1_21 */
                     AM33XX_IOPAD(0x858, PIN_OUTPUT_PULLUP | MUX_MODE7)
/*gpmc_a6.gpio1_22 */
                     AM33XX_IOPAD(0x85c, PIN_OUTPUT_PULLDOWN | MUX_MODE7)
/*gpmc_a5.gpio1_23 */
                     AM33XX_IOPAD(0x860, PIN_OUTPUT_PULLUP | MUX_MODE7)
/*gpmc_a5.gpio1_24 */
              >;
       };
};
```

You can find the absolute physical address of the pins, which is required for the macro, in the ***CONTROL_MODULE REGISTERS*** table of AM335x TRM. In the table, each signal pin name is prefixed with ***conf_***. For example, the absolute physical address of the GPMC_A5 pin, 0x854, is shown in Figure 6.

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
774h	vdd_mpu_opp_100		Section 9.3.1.42
778h	vdd_mpu_opp_120		Section 9.3.1.43
77Ch	vdd_mpu_opp_turbo		Section 9.3.1.44
7B8h	vdd_core_opp_050		Section 9.3.1.45
7BCh	vdd_core_opp_100		Section 9.3.1.46
7D0h	bb_scale		Section 9.3.1.47
7F4h	usb_vid_pid		Section 9.3.1.48
7FCh	efuse_sma		Section 9.3.1.49
800h	conf_gpmc_ad0	See the device datasheet for information on default pin mux configurations. Note that the device ROM may change the default pin mux for certain pins based on the SYSBOOT mode settings.	Section 9.3.1.50
804h	conf_gpmc_ad1		Section 9.3.1.50
808h	conf_gpmc_ad2		Section 9.3.1.50
80Ch	conf_gpmc_ad3		Section 9.3.1.50
810h	conf_gpmc_ad4		Section 9.3.1.50
814h	conf_gpmc_ad5		Section 9.3.1.50
818h	conf_gpmc_ad6		Section 9.3.1.50
81Ch	conf_gpmc_ad7		Section 9.3.1.50
820h	conf_gpmc_ad8		Section 9.3.1.50
824h	conf_gpmc_ad9		Section 9.3.1.50
828h	conf_gpmc_ad10		Section 9.3.1.50
82Ch	conf_gpmc_ad11		Section 9.3.1.50
830h	conf_gpmc_ad12		Section 9.3.1.50
834h	conf_gpmc_ad13		Section 9.3.1.50
838h	conf_gpmc_ad14		Section 9.3.1.50
83Ch	conf_gpmc_ad15		Section 9.3.1.50
840h	conf_gpmc_a0		Section 9.3.1.50
844h	conf_gpmc_a1		Section 9.3.1.50
848h	conf_gpmc_a2		Section 9.3.1.50
84Ch	conf_gpmc_a3		Section 9.3.1.50
850h	conf_gpmc_a4		Section 9.3.1.50
854h	conf_gpmc_a5		Section 9.3.1.50
858h	conf_gpmc_a6		Section 9.3.1.50
85Ch	conf_gpmc_a7		Section 9.3.1.50
860h	conf_gpmc_a8		Section 9.3.1.50

Figure 6 Physical Address of GPMC_A5

In addition, you can also use TI's PinMux Tool to ensure the pin muxing of your pins do not conflict with each other (You will need a TI user account to use the tool). You can watch the TI PinMux Tool getting started video here ***to learn more about the tool***.

*****

18 Linux Device Tree Overlay

18.1 Introduction

Since the adoption of the Device Tree standard to describe embedded Linux systems, there has been one major limitation: the static nature of device trees (please find more information on Linux device trees here<link to be added to previous article>). Device trees could not cope with changes in non-discoverable hardware, such as modifications to pin muxing, during run time. However, most modern embedded systems support adding and removing non-discoverable hardware during run time. This made it difficult to define the full hardware configuration statically at boot time in a device tree. Pantelis Antoniu, an active Linux kernel developer, implemented a solution to this issue using Device Tree Overlays (DTOs). The idea was to be able to dynamically insert a fragment of a device tree into a live device tree to update the hardware configuration of the system. For example, a fragment could change the status property of a device node from “disabled” to “okay” and then the device corresponding to that node would be created.

A device tree overlay is a file that consists of one or more device tree fragments that describe changes to the system hardware. This article will help you become familiar with device tree overlays by explaining the structure through an example, building a device tree overlay for the peripheral header of the Lesson 2 board, and then adapting the generic overlay for an example Click board.

18.2 Understanding Device Tree Overlays

The structure of a device tree overlay is a direct extension of a device tree. First, let’s understand the structure by looking at the ***PB-I2C1-MPU-9DOF-CLICK.dts*** device tree overlay file. This overlay was written for the MPU 9DOF Click board which can be attached to the headers of the BeagleBoard.org® PocketBeagle®. Since the PocketBeagle® uses the OSD335x-SM device (which is very similar to the OSD335x), it will be easy to leverage this overlay for the Lesson 2 board (The ***PB-I2C1-MPU-9DOF-CLICK.dts*** file is available for download here).

```
/*
 * Copyright (C) 2017 Robert Nelson <robertcnelson@gmail.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

#include <dt-bindings/board/am335x-bbw-bbb-base.h>
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>
```

```
/ {
	fragment@0 {
		target = <&am33xx_pinmux>;
		__overlay__ {

			mpu9150_pins: pinmux_mpu9150_pins {
				pinctrl-single,pins = <
					AM33XX_IOPAD(0x0824, PIN_INPUT | MUX_MODE7 )
/* (T10) gpmc_ad9.gpio0[23] INT */
				>;
			};
		};
	};

	fragment@1 {
		target = <&ocp>;
		__overlay__ {

			P2_03_pinmux {
				status = "disabled";
			};
		};
	};

	fragment@2 {
		target = <&i2c1>;
		__overlay__ {
			status = "okay";

			#address-cells = <1>;
			#size-cells = <0>;

			mpu9150@69 {
				compatible = "invensense,mpu9150";
				reg = <0x69>;
				interrupt-parent = <&gpio0>;
				interrupts = <23 1>;
				i2c-gate {
					#address-cells = <1>;
					#size-cells = <0>;
					ax8975@c {
						compatible = "ak,ak8975";
						reg = <0x0c>;
					};
				};
			};
		};
	};
};
```

The above overlay code consists of three major parts:

First, the overlay begins with tokens (shown below). The first token indicates the file version and the second one indicates that this file is a plugin (i.e. an overlay).

```
/dts-v1/;
/plugin/;
```

Next, the tokens are followed by include statements. This is where required header files are included in the overlay to add symbol definitions and macros.

```
#include <dt-bindings/board/am335x-bbw-bbb-base.h>
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>
```

Finally, the fragments describe the functional changes to the device tree for the overlay, starting from the root node (/):

The first fragment of the overlay (fragment@0) is used to set the ***P2_03*** pin of the PocketBeagle® (i.e. the GPMC_AD9 pin, bit 23 of the GPIO0 peripheral of the AM335x) to ***GPIO input mode***. When the MPU 9DOF Click board is connected to the PocketBeagle®, the ***INT*** pin of the Click connects to the ***P2_03*** pin. Therefore, the pin must be configured so that the PocketBeagle® can receive interrupt signals from the MPU 9DOF Click. To configure the pin, the ***target*** property is used to specify the device tree node that needs to be overlaid. In this case, the ***am33xx_pinmux*** node is specified to alter the pin muxing of the AM335x IO. The properties listed in the ***__overlay__*** node will be overlaid on the original properties of the target device tree node.

```
fragment@0 {
            target = <&am33xx_pinmux>;
            __overlay__ {

                  mpu9150_pins: pinmux_mpu9150_pins {
                        pinctrl-single,pins = <
                              AM33XX_IOPAD(0x0824, PIN_INPUT | MUX_MODE7 )
/* (T10) gpmc_ad9.gpio0[23] INT */
                        >;
                  };
            };
      };
```

The second fragment of the overlay (fragment@1) is then used to disable the current functionality associated with the ***P2_03*** pin. Currently in the PocketBeagle® device tree, the ***P2_03*** pin is configured to use the ***bone-pinmux-helper*** driver in the ***OCP*** (On Chip Peripheral) node (see am335x-pocketbeagle.dts). Therefore, the driver must be disabled to ensure that the ***P2_03*** pin is available to receive interrupts from the Click board.

```
fragment@1 {
            target = <&ocp>;
            __overlay__ {

                  P2_03_pinmux {
                        status = "disabled";
                  };
            };
      };
};
```

The third fragment (fragment@2) is used to enable the I2C communication with the Click board. The Click board communicates over the I2C1 bus. Therefore, the I2C1 interface needs to be enabled by setting the ***status*** property of the ***i2c1*** node to ***okay***. Also, all the required information of the IC on the MPU 9DOF Click board, such as the manufacturer name, IC name, device address on the I2C bus, etc. must be supplied in this fragment so the kernel can load the appropriate drivers. The device tree bindings for MPU-9150 IC on the MPU 9DOF Click can be found here.

```
fragment@2 {
            target = <&i2c1>;
            __overlay__ {
                  status = "okay";

                  #address-cells = <1>;
                  #size-cells = <0>;

                  mpu9150@69 {
                        compatible = "invensense,mpu9150";
                        reg = <0x69>;
                        interrupt-parent = <&gpio0>;
                        interrupts = <23 1>;
                        i2c-gate {
                              #address-cells = <1>;
                              #size-cells = <0>;
                              ax8975@c {
                                    compatible = "ak,ak8975";
                                    reg = <0x0c>;
                              };
                        };
                  };
            };
      };
```

18.3 Generic Device Tree Overlay for the Peripheral Header

The Lesson 2 board has a peripheral header that supports removable Click boards. Similar to the device tree overlay in the previous section, a generic device tree overlay (***osd335x_L2_generic.dts***) can be built for the Lesson 2 board's peripheral header as shown below (The overlay can also be downloaded here).

The overlay has three parts like the previous example, however, there are more fragments to enable the different peripherals used by the header. The first fragment is used to configure the pin muxing for all the interfaces on the peripheral header such as SPIO0, UART0, I2C1 and GPIO pins. The second, third and fourth fragments enable the I2C1, SPI0 and UART0 interfaces, respectively, and provide an outline to add device tree binding information corresponding to the type of click board that you will use.

> Perk:
>
> The device tree overlay we're building in this section is intended to be overlaid on the ***osd335x-lesson2.dts*** device tree which we built as part of the Linux Device Tree article. ***osd335x-lesson2.dts*** does not use ***bone-pinmux-helper*** driver. Hence, there is no need to disable the pinmux driver as in the example from the previous section.

```
/*
 * Copyright (C) 2017 2017 Octavo Systems - http://www.octavosystems.com/
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

#include <dt-bindings/board/am335x-bbw-bbb-base.h>
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>

/ {

fragment@0 {
      target = <&am33xx_pinmux>;
      __overlay__ {

            pwm_pins: pinmux_pwm {
                  pinctrl-single,pins = <
                        AM33XX_IOPAD(0x0964, PIN_OUTPUT | MUX_MODE0 ) /*
(C18) ECAP0_IN_PWM0_OUT */
```

```
                        >;
                };

                int_n_rst_pins: pinmux_int_rst {
                        pinctrl-single,pins = <
                                AM33XX_IOPAD(0x0824, PIN_INPUT | MUX_MODE7 ) /*
(T10) gpmc_ad9 MBUS_INT */
                                AM33XX_IOPAD(0x0820, PIN_OUTPUT | MUX_MODE7 ) /*
(U10) gpmc_ad8 MBUS_RST */
                        >;
                };

                i2c1_pins: pinmux_i2c1 {
                        pinctrl-single,pins = <
                                AM33XX_IOPAD(0x0968, PIN_INPUT_PULLUP | MUX_MODE3 )
/* (E18)  I2C1_SDA/UART0_CTSN */
                                AM33XX_IOPAD(0x096C, PIN_INPUT_PULLUP | MUX_MODE3 )
/* (E17)  I2C1_SCL/UART0_RTSN */
                        >;
                };

                spi0_pins: pinmux_spi0 {
                        pinctrl-single,pins = <
                                AM33XX_IOPAD(0x0950, PIN_INPUT | MUX_MODE0 ) /*
(A17) spi0_sclk.spi0_sclk */
                                AM33XX_IOPAD(0x0954, PIN_INPUT | MUX_MODE0 ) /*
(B17) spi0_d0.spi0_d0 */
                                AM33XX_IOPAD(0x0958, PIN_INPUT | MUX_MODE0 ) /*
(B16) spi0_d1.spi0_d1 */
                                AM33XX_IOPAD(0x095c, PIN_INPUT | MUX_MODE0 ) /*
(A16) spi0_cs0.spi0_cs0 */
                        >;
                };

        };
};


fragment@1 {

        target = <&i2c1>;
        __overlay__ {

                status = "okay";
                pinctrl-names = "default";
                pinctrl-0 = <&i2c1_pins>;

                /* Add device tree bindings
                 * for your I2C1 device here.
                 *
                 */

        };
```

```
};

fragment@2 {

      target = <&spi0>;
      __overlay__ {

            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&spi0_pins>;

            /* Add device tree bindings
             * for your SPI device here.
             *
             */

      };
};

fragment@3 {

      target = <&uart0>;
      __overlay__ {

            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&uart0_pins>;

            /* Add device tree bindings
             * for your UART device here.
             *
             */

      };
};
};
```

18.4 Adapting the Generic Device Tree Overlay for a Specific Click Board

When you decide to use a specific Click board with the Lesson 2 board, it is straight-forward to modify the generic device tree overlay created in the previous section, versus writing one from scratch, in order to support the Click board. Taking the MPU 9DOF Click example from Section 18.2, you would only need to add the MPU9150 device tree binding information under fragment@1 since MPU 9DOF Click uses the I2C bus for communication. The new device tree overlay (***osd335x_L2_generic_i2c1.dts***) would look like this (content that is different from generic DTO is highlighted):

```
/*
 * Copyright (C) 2017 Octavo Systems - http://www.octavosystems.com/
 *
```

```
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

#include <dt-bindings/board/am335x-bbw-bbb-base.h>
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>

/ {

fragment@0 {
      target = <&am33xx_pinmux>;
      __overlay__ {

            pwm_pins: pinmux_pwm {
                  pinctrl-single,pins = <
                        AM33XX_IOPAD(0x0964, PIN_OUTPUT | MUX_MODE0 ) /*
(C18) ECAP0_IN_PWM0_OUT */
                  >;
            };

            int_n_rst_pins: pinmux_int_rst {
                  pinctrl-single,pins = <
                        AM33XX_IOPAD(0x0824, PIN_INPUT | MUX_MODE7 ) /*
(T10) gpmc_ad9 MBUS_INT */
                        AM33XX_IOPAD(0x0820, PIN_OUTPUT | MUX_MODE7 ) /*
(U10) gpmc_ad8 MBUS_RST */
                  >;
            };

            i2c1_pins: pinmux_i2c1 {
                  pinctrl-single,pins = <
                        AM33XX_IOPAD(0x0968, PIN_INPUT_PULLUP | MUX_MODE3 )
/* (E18)  I2C1_SDA/UART0_CTSN */
                        AM33XX_IOPAD(0x096C, PIN_INPUT_PULLUP | MUX_MODE3 )
/* (E17)  I2C1_SCL/UART0_RTSN */
                  >;
            };

            spi0_pins: pinmux_spi0 {
                  pinctrl-single,pins = <
                        AM33XX_IOPAD(0x0950, PIN_INPUT | MUX_MODE0 ) /*
(A17) spi0_sclk.spi0_sclk */
                        AM33XX_IOPAD(0x0954, PIN_INPUT | MUX_MODE0 ) /*
(B17) spi0_d0.spi0_d0 */
                        AM33XX_IOPAD(0x0958, PIN_INPUT | MUX_MODE0 ) /*
(B16) spi0_d1.spi0_d1 */
                        AM33XX_IOPAD(0x095c, PIN_INPUT | MUX_MODE0 ) /*
(A16) spi0_cs0.spi0_cs0 */
                  >;
```

```
            };

      };
};


fragment@1 {

      target = <&i2c1>;
      __overlay__ {

            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&i2c1_pins>;

            #address-cells = <1>;
            #size-cells = <0>;

            mpu9150@69 {
                  compatible = "invensense,mpu9150";
                  reg = <0x69>;
                  interrupt-parent = <&gpio0>;
                  interrupts = <23 1>;
                  i2c-gate {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        ax8975@c {
                              compatible = "ak,ak8975";
                              reg = <0x0c>;
                        };
                  };
            };

      };
};


fragment@2 {

      target = <&spi0>;
      __overlay__ {

            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&spi0_pins>;

            /* Add device tree bindings
             * for your SPI device here.
             *
             */

      };
};
```

```
fragment@3 {

      target = <&uart0>;
      __overlay__ {

            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&uart0_pins>;

            /* Add device tree bindings
             * for your UART device here.
             *
             */

      };
};
};
```

You can also directly download the ***osd335x_L2_generic_i2c1.dts*** device tree overlay file here.

18.5 Building and using a Device Tree Overlay

The process to compile a device tree overlay is similar to the process to compile a device tree as discussed in Section 1.3 of Linux Device Tree article. For the OSD335x Family of devices, Robert Nelson's ***Device Tree Rebuilder*** can be used to build the DTO. You can download it here. The steps to build the DTO are as follows:

a. Copy the ***osd335x_L2_generic_i2c1.dts*** file to the *dtb-rebuilder/src/arm/* directory.
b. Build the file using ***make*** command.
c. Once the build process completes, the ***osd335x_L2_generic_i2c1.dtb*** file will be available in the *dtb-rebuilder/src/arm/* directory. Change the extension of the file to ***.dtbo*** (i.e., ***osd335x_L2_generic_i2c1.dtbo***) to make sure the linux kernel recognizes it as an overlay.
d. Copy the ***osd335x_L2_generic_i2c1.dtbo*** file to the /lib/firmware directory (You will have to run this command as root)
e. Open the ***uEnv.txt*** file in the */boot* directory using your favorite text editor and set *custom_cape* = *osd335x_L2_generic_i2c1.dtbo* as shown in Figure . Save and close the file.

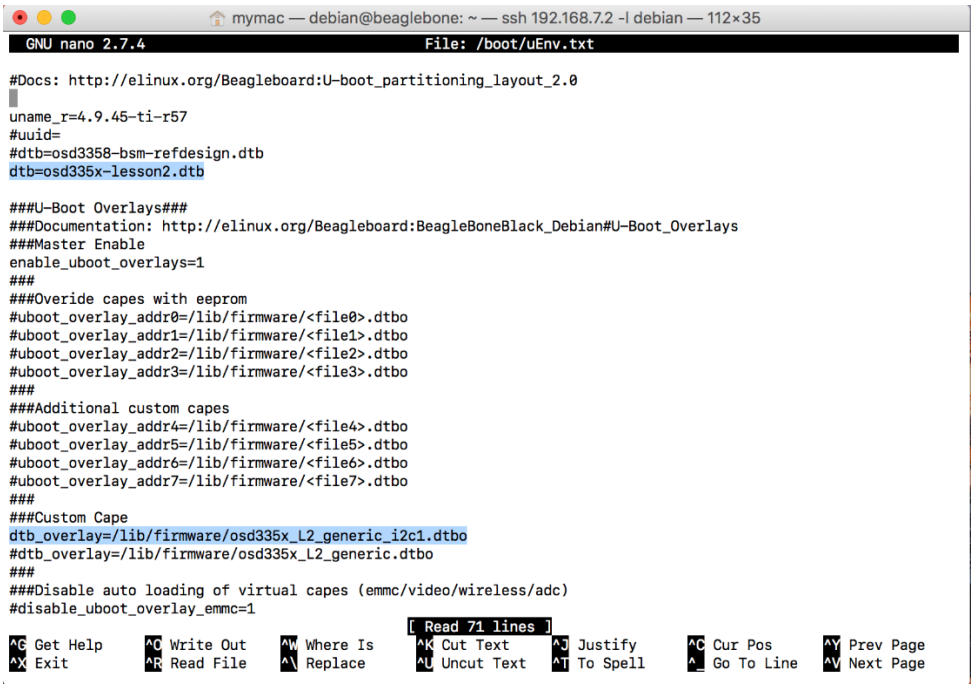


Figure 1 Setting Device Tree Overlay

f. Reboot the board. If you monitor the boot messages on the UART0 serial console, it will display the name of the device tree and device tree overlays that were loaded during boot-up as shown in Figure 121.

```
U-Boot SPL 2017.09-rc2-00002-g7c9353e752 (Aug 31 2017 - 09:25:14)
Trying to boot from MMC2


U-Boot 2017.09-rc2-00002-g7c9353e752 (Aug 31 2017 - 09:25:14 -0500), Build: jenkins-github_Bootloader-Builder-596

CPU  : AM335X-GP rev 2.1
I2C:   ready
DRAM:  512 MiB
No match for driver 'omap_hsmmc'
No match for driver 'omap_hsmmc'
Some drivers were not found
Reset Source: Global external warm reset has occurred.
Reset Source: Power-on reset has occurred.
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Using default environment

Board: BeagleBone Black
<ethaddr> not set. Validating first E-fuse MAC
BeagleBone Black:
BeagleBone: cape eeprom: i2c_probe: 0x54:
BeagleBone: cape eeprom: i2c_probe: 0x55:
BeagleBone: cape eeprom: i2c_probe: 0x56:
BeagleBone: cape eeprom: i2c_probe: 0x57:
Net:   eth0: MII MODE
Could not get PHY for cpsw: addr 0
cpsw
Press SPACE to abort autoboot in 2 seconds
board_name=[A335BNLT] ...
board_rev=[] ...
switch to partitions #0, OK
mmc0 is current device
SD/MMC found on device 0
** Bad device 0:2 0x82000000 **
** Bad device 0:2 0x82000000 **
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
gpio: pin 56 (gpio 56) value is 0
gpio: pin 55 (gpio 55) value is 0
gpio: pin 54 (gpio 54) value is 0
gpio: pin 53 (gpio 53) value is 1
switch to partitions #0, OK
mmc0 is current device
gpio: pin 54 (gpio 54) value is 1
Checking for: /uEnv.txt ...
Checking for: /boot.scr ...
Checking for: /boot/boot.scr ...
Checking for: /boot/uEnv.txt ...
gpio: pin 55 (gpio 55) value is 1
2158 bytes read in 43 ms (48.8 KiB/s)
Loaded environment from /boot/uEnv.txt
debug: [dtb=osd335x-lesson2.dtb] ...
Using: dtb=osd335x-lesson2.dtb ...
Checking if uname_r is set in /boot/uEnv.txt...
gpio: pin 56 (gpio 56) value is 1
Running uname_boot ...
loading /boot/vmlinuz-4.9.45-ti-r57 ...
9464664 bytes read in 631 ms (14.3 MiB/s)
uboot_overlays: dtb=osd335x-lesson2.dtb in /boot/uEnv.txt, unable to use [uboot_base_dtb=am335x-boneblack-uboot.dtb] ...
loading /boot/dtbs/4.9.45-ti-r57/osd335x-lesson2.dtb ...
49963 bytes read in 167 ms (292 KiB/s)
uboot_overlays: [fdt_buffer=0x60000] ...
uboot_overlays: loading /lib/firmware/AM335X-20-00A0.dtbo ...
378 bytes read in 81 ms (3.9 KiB/s)
uboot_overlays: loading /lib/firmware/BB-BONE-eMMC1-01-00A0.dtbo ...
1105 bytes read in 193 ms (4.9 KiB/s)
uboot_overlays: loading /lib/firmware/BB-HDMI-TDA998x-00A0.dtbo ...
4169 bytes read in 295 ms (13.7 KiB/s)
uboot_overlays: loading /lib/firmware/BB-ADC-00A0.dtbo ...
695 bytes read in 203 ms (2.9 KiB/s)
uboot_overlays: loading /lib/firmware/AM335X-PRU-UIO-00A0.dtbo ...
853 bytes read in 114 ms (6.8 KiB/s)
uboot_overlays: [dtb_overlay=/lib/firmware/osd335x_L2_generic_i2c1.dtbo] ...
```

Figure 121 Serial Console Boot Messages

We can now login to Debian (default username = debian, password = temppwd).

18.6 Checking if the Device Tree Overlay works as intended

If the new DTO works as intended, the Linux kernel should recognize the MPU 9DOF Click and load appropriate drivers. From the command line, we can check if appropriate driver has been loaded for the IC on the MPU 9DOF by listing the i2c bus drivers (as shown in Figure 122) using the command:

```
ls /sys/bus/i2c/drivers
```

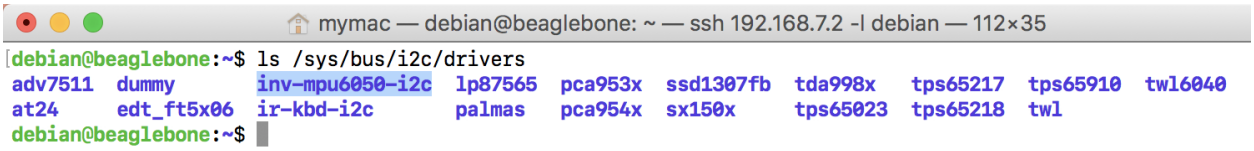


Figure 122 Checking if driver for 9DOF Click was automatically loaded

If the Invensense MPU6050 I2C driver appears, you know that the device was properly configured and the correct driver was loaded. Also, given a kernel driver was loaded for MPU 9DOF click, we should see ***UU*** (Unit Unavailable) on the I2C1 bus at address 0x69 when the ***i2cdetect*** command is used to scan the I2C1 bus for devices as shown in Figure 123. When a kernel driver manages a device, it will not allow ***i2cdetect*** to probe the device. Therefore, you will see ***UU*** when scanned versus ***69***, if the correct driver was not loaded.

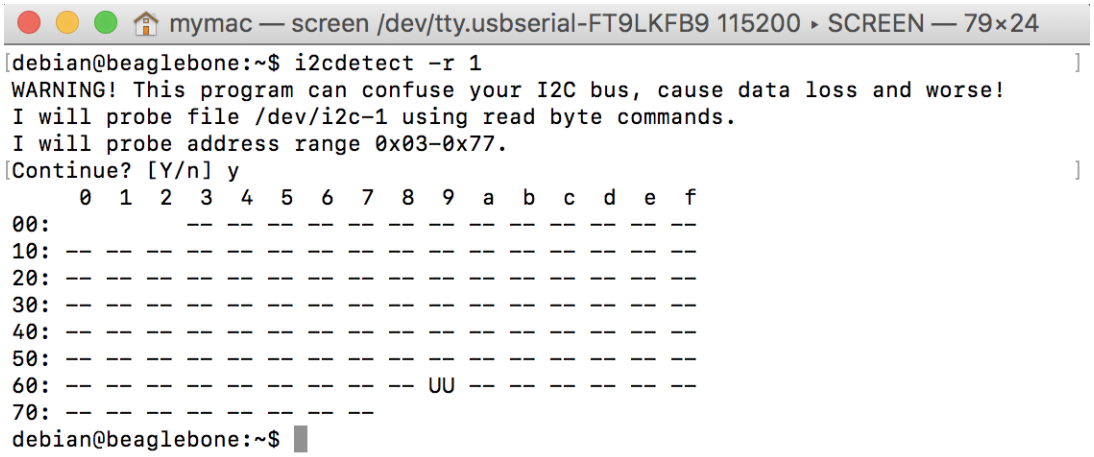


Figure 123 Detecting Click board on I2C bus

Finally, we can check if the MPU 9DOF Click appears as an IIO (Industrial IO) device by searching for IIO devices in IIO devices directory as shown in Figure 124.

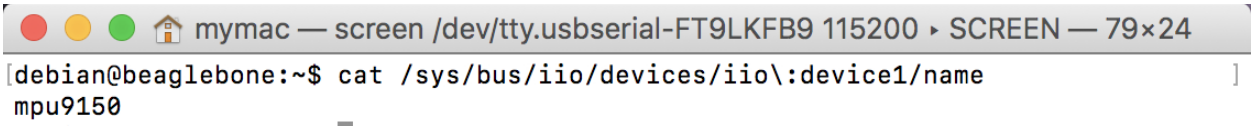


Figure 124 Checking if MPU 9DOF IC is recognized

If all the above checks succeed, the device is working properly and you can directly access the accelerometer, compass and other values from the MPU 9DOF Click using Sysfs (more on Sysfs here). Most of the readable registers of the MPU 9DOF Click are available as files under the IIO device directory for the Click (***iio:device1***, in the picture below. However, the device number maybe different in your system) as shown in Figure 125.

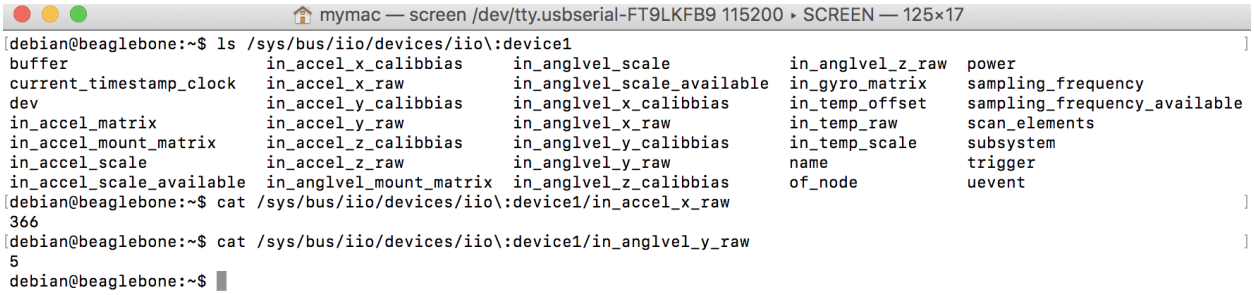


Figure 125 Reading MPU 9DOF files and output values using Sysfs

For any questions or concerns, you can reach us at:
https://octavosystems.com/forums/

*****