

# 1 Linux Device Tree

## 1.1 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.

## 1.2 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 1.

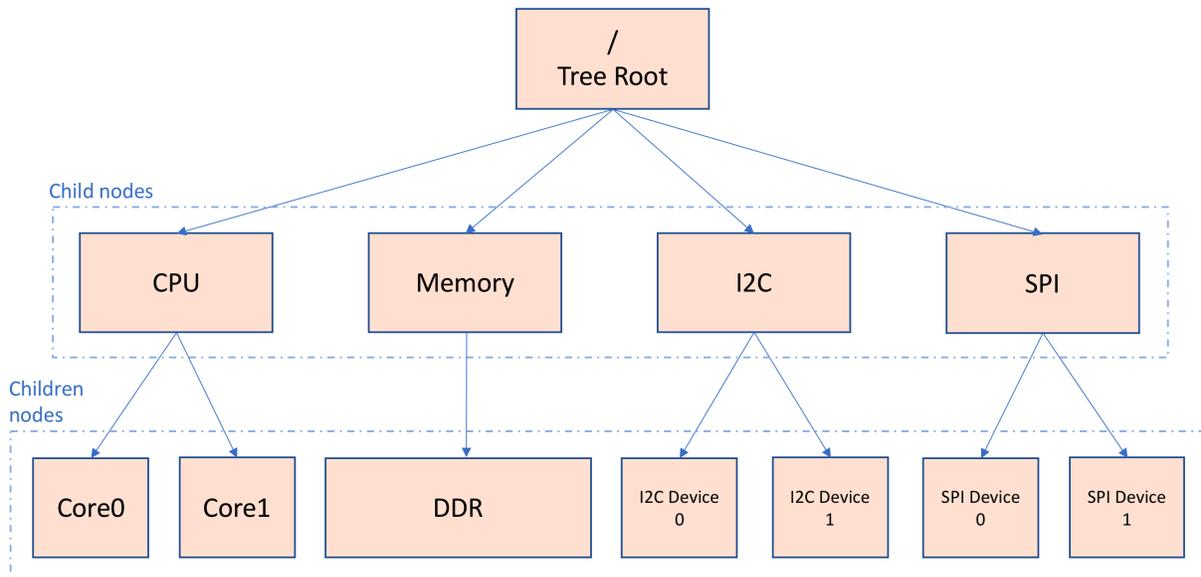


Figure 1 Block diagram of a simple device tree structure

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 2) that shows the generic syntax of the root node, child nodes, children nodes and property-value pairs.

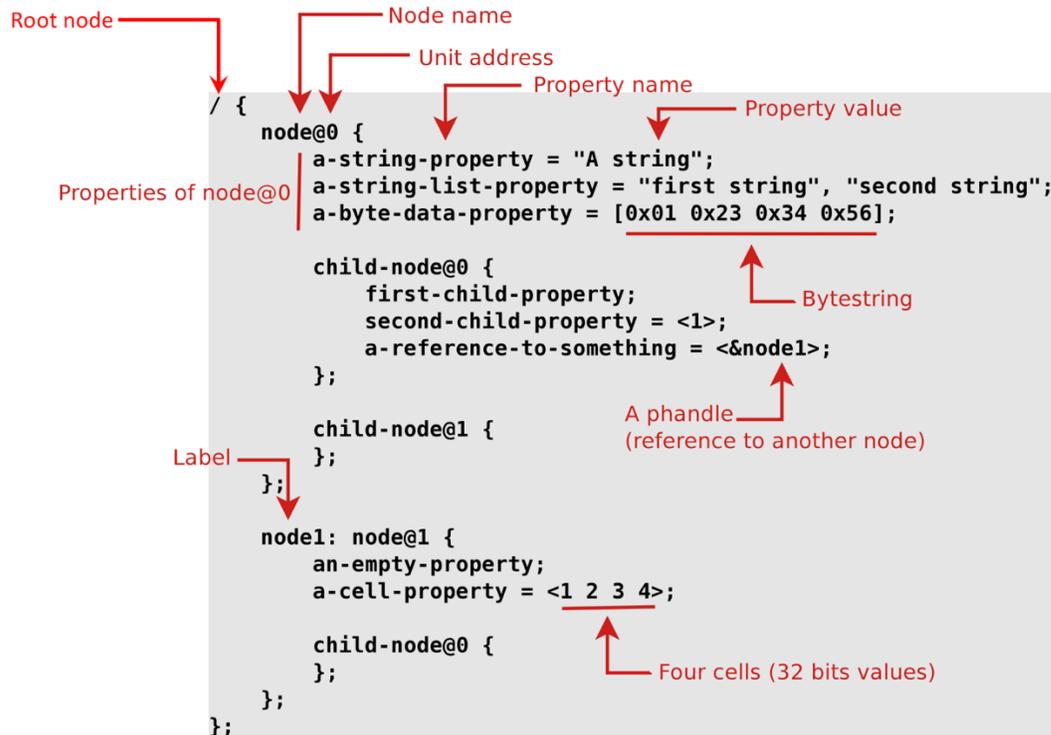


Figure 2 Simple Device Tree Template (© Thomas Petazzoni, Device Tree for Dummies)

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.

### 1.3 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 enable 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 3 lists all the nodes that are present in this file. The nodes that are marked with **X** should be removed.



Figure 3 osd3358-bsm-refdesign.dts before modifications

## OSD335x Lesson 2: Linux Device Tree

Rev.1 5/30/2018



After the modifications, the *osd3358-bsm-refdesign.dts* will look like Figure 4. Since it now reflects the Lesson 2 hardware, the file can be renamed to *osd335x-lesson2.dts*.

### osd335x-lesson2.dts

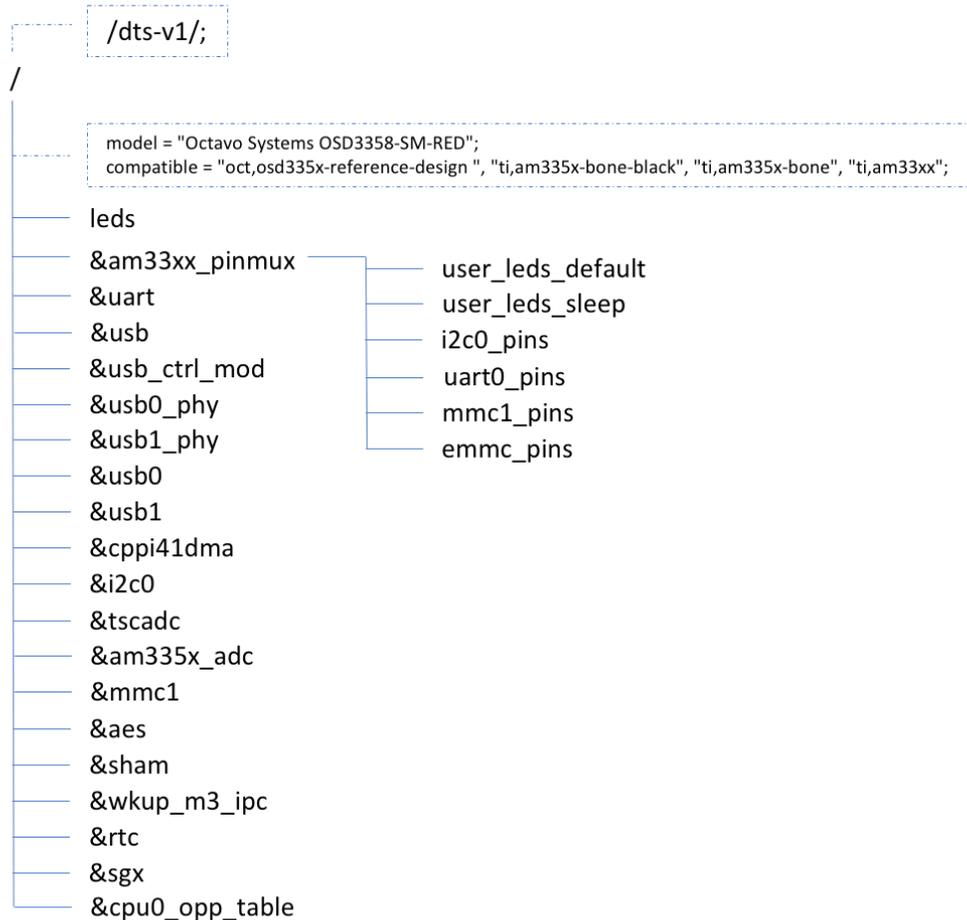


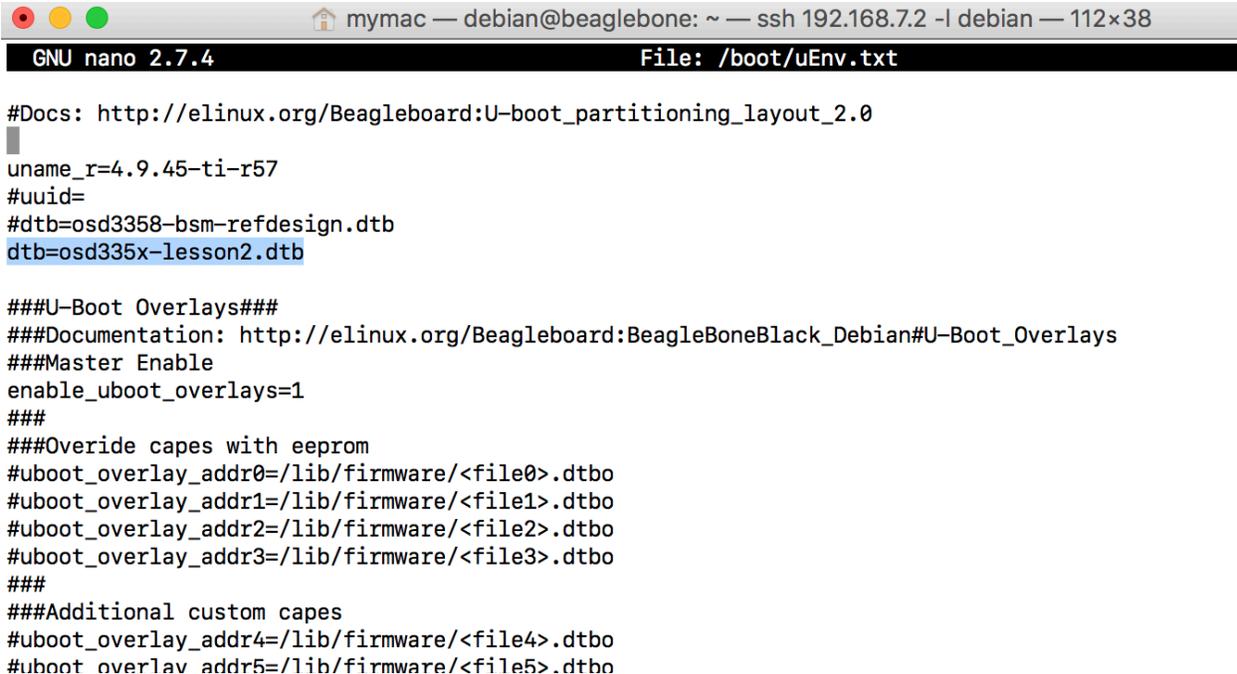
Figure 4 *osd3358-bsm-refdesign.dts* after modifications to reflect Lesson 2 hardware

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](#).

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

1. Download the latest **OSD3359-SM RED Debian** image and flash a microSD card with it. Boot the Lesson 2 board from the microSD card.
2. Download and copy the **dtb-rebuilder** to your microSD card.
3. Copy the **osd335x-lesson2.dts** and **osd335x-sm.dtsi** to **dtb-rebuilder/src/arm** directory.
4. Use the **make** command to build the files.
5. The **osd335x-lesson2.dtb** file will be in the **dtb-rebuilder/src/arm** directory once the build process is complete.
6. Move the **osd335x-lesson2.dtb** file to **/boot/dtbs/<kernel version>/** directory to make it available to the Linux Kernel during boot.
7. Configure the boot configuration file (uEnv.txt) to use the new device tree:
  - a. Open the **/boot/uEnv.txt** configuration file using a text editor. E.g., **nano /boot/uEnv.txt**
  - b. Comment out the existing **dtb** variable and add the line **dtb=osd335x-lesson2.dtb** as shown in Figure 5. Save the file.
  - c. Reboot the board using the command: **sudo reboot -h now**.

The board should now boot using the new device tree.



```
GNU nano 2.7.4 File: /boot/uEnv.txt
#Docs: http://elinux.org/Beagleboard:U-boot_partitioning_layout_2.0
uname_r=4.9.45-ti-r57
#uuid=
#dtb=osd3358-bsm-refdesign.dtb
dtb=osd335x-lesson2.dtb

###U-Boot Overlays###
###Documentation: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian#U-Boot_Overlays
###Master Enable
enable_uboot_overlays=1
###
###Override capes with eeprom
#uboot_overlay_addr0=/lib/firmware/<file0>.dtbo
#uboot_overlay_addr1=/lib/firmware/<file1>.dtbo
#uboot_overlay_addr2=/lib/firmware/<file2>.dtbo
#uboot_overlay_addr3=/lib/firmware/<file3>.dtbo
###
###Additional custom capes
#uboot_overlay_addr4=/lib/firmware/<file4>.dtbo
#uboot_overlay_addr5=/lib/firmware/<file5>.dtbo
```

Figure 5 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 the ***touch \**** command in the *dtb-rebuilder* directory and the *dtb-rebuilder/src*, *dtb-rebuilder/src/arm* sub-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.

## 1.4 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 multiplexing 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.

For any questions or concerns, you can reach us at:

<https://octavosystems.com/forums/>