The Guide: VHDL -> Avalon -> Device Driver

The idea here is a one-stop shop for example files, compilation commands, and more to go from a VHDL file to a Linux device driver.

VHDL (on your PC)

Write a VHDL file to implement the necessary functionality. Then, write a VHDL file that instantiates the VHDL component and sets it up for use over the avalon bridge. An example pair of files for a component and Avalon file is as follows:

Component:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
library IEEE;
library std;
use std.standard;

entity pwm_controller is
  generic (
    CLK_PERIOD : time := 20 ns
  );
  port (
    clk : in std_logic;
    rst : in std_logic; -- active high
    -- PWM repetition period in milliseconds;
    -- datatype (W.F) (32.24)
    period : in unsigned(31 downto 0);
    -- PWM duty cycle between [0 1]; out-of-range values are hard-limited
    -- datatype (W.F) (32.31)
    duty_cycle : in std_logic_vector(31 downto 0);
    output     : out std_logic
  );
end entity pwm_controller;

architecture arch of pwm_controller is

  -- assigned data types
  constant PERIOD_INT_BITS : natural := 8;
  constant PERIOD_FRAC_BITS : natural := 24;
  constant DUTY_INT_BITS : natural := 1;
  constant DUTY_FRAC_BITS : natural := 31;

  -- clock frequency as natural (keep in ms as period is provided in ms)
  constant FREQ_INTEGER : natural := integer(real(1 ms / CLK_PERIOD));
  -- bits needed for frequency
```

```
  constant FREQ_BITS : natural := natural(ceil(log2(real(FREQ_INTEGER))));
  -- clock frequency as unsigned
  constant FREQ : unsigned((FREQ_BITS - 1) downto 0) :=
to_unsigned(FREQ_INTEGER, FREQ_BITS);

  -- counter max
  signal counter_max_fullprec : unsigned((FREQ_BITS + PERIOD_INT_BITS +
PERIOD_FRAC_BITS - 1) downto 0);
  signal counter_max_int : natural;
  -- duty cycle max
  signal duty_cycle_max_fullprec : unsigned((FREQ_BITS + DUTY_INT_BITS +
DUTY_FRAC_BITS - 1) downto 0);
  signal duty_cycle_max_int : natural;
  -- count
  signal count  : integer := 0;

begin
    -- calculate counter max, accounting for the fact period is provided in
milliseconds
    counter_max_fullprec <= freq * period;
    counter_max_int <= to_integer(counter_max_fullprec((FREQ_BITS +
PERIOD_INT_BITS + PERIOD_FRAC_BITS - 1) downto PERIOD_FRAC_BITS));

    -- calculate duty cycle max
    duty_cycle_max_fullprec <= freq * unsigned(duty_cycle);
    duty_cycle_max_int <= to_integer(duty_cycle_max_fullprec((FREQ_BITS +
DUTY_INT_BITS + DUTY_FRAC_BITS - 1) downto DUTY_FRAC_BITS));


    OUTPUT_DRIVER : process(clk, rst)
    begin
        if(rst = '1') then
            count <= 0;
            output <= '1';
        elsif(rising_edge(clk)) then
            if(count < counter_max_int) then
              count <= count + 1;
              if(count < duty_cycle_max_int) then
                  output <= '1';
              else
                  output <= '0';
              end if;
            else
                count <= 0;
                output <= '1';
            end if;
        end if;
    end process OUTPUT_DRIVER;
end architecture arch;
```

Avalon File:

```vhdl
-- altera vhdl_input_version vhdl_2008

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library std;
use std.standard;

entity pwm_rgb_controller_avalon is
  port (
    clk : in std_ulogic;
    rst : in std_ulogic;
    -- avalon memory-mapped slave interface
    avs_read      : in std_logic;
    avs_write     : in std_logic;
    avs_address   : in std_logic_vector(1 downto 0);
    avs_readdata  : out std_logic_vector(31 downto 0);
    avs_writedata : in std_logic_vector(31 downto 0);
    -- external I/O; export to top-level
    red_out        : out std_logic;
    green_out : out std_logic;
    blue_out : out std_logic
  );
end entity pwm_rgb_controller_avalon;

architecture arch of pwm_rgb_controller_avalon is

  -- duty cycles are provided in the following format: (W.F) (32.31)
  -- set duty cycles to 50% initially
  signal reg_red_duty_cycle       : std_logic_vector(31 downto 0) := (30 =>
'1', others => '0');
  signal reg_green_duty_cycle     : std_logic_vector(31 downto 0) := (30 =>
'1', others => '0');
  signal reg_blue_duty_cycle      : std_logic_vector(31 downto 0) := (30 =>
'1', others => '0');

  -- period in milliseconds is provided in the following format: (W.F) (32.24)
  -- set period initially to 1 ms
  signal reg_period : std_logic_vector(31 downto 0) := (24 => '1', others =>
'0');

  component pwm_controller is
      port (
         clk : in std_logic;
         rst : in std_logic; -- active high
         -- PWM repetition period in milliseconds;
         -- datatype (W.F) (32.24)
         period : in unsigned(31 downto 0);
         -- PWM duty cycle between [0 1]; out-of-range values are hard-limited
         -- datatype (W.F) (32.31)
         duty_cycle : in std_logic_vector(31 downto 0);
```

```
        output     : out std_logic
      );
  end component pwm_controller;

begin

  RED_PWM_CTL : component pwm_controller
    port map
     (
        clk => clk,
        rst => rst,
        period => unsigned(reg_period),
        duty_cycle => reg_red_duty_cycle,
        output => red_out
      );

    GREEN_PWM_CTL : component pwm_controller
      port map
     (
      clk => clk,
      rst => rst,
      period => unsigned(reg_period),
      duty_cycle => reg_green_duty_cycle,
      output => green_out
      );

      BLUE_PWM_CTL : component pwm_controller
        port map
       (
        clk => clk,
        rst => rst,
        period => unsigned(reg_period),
        duty_cycle => reg_blue_duty_cycle,
        output => blue_out
        );

  avalon_register_read : process (clk)
  begin
    if rising_edge(clk) and avs_read = '1' then
      case avs_address is
        when "00" => avs_readdata   <= reg_red_duty_cycle;
        when "01" => avs_readdata   <= reg_green_duty_cycle;
        when "10" => avs_readdata   <= reg_blue_duty_cycle;
        when "11" => avs_readdata <= reg_period;
        when others => avs_readdata <= (others => '0');
      end case;
    end if;
  end process;

  avalon_register_write : process (clk, rst)
  begin
    if rst = '1' then
      reg_red_duty_cycle <= (others => '0');
```

```vhdl
      reg_green_duty_cycle <= (others => '0');
      reg_blue_duty_cycle <= (others => '0');
      reg_period <= (21 => '1', others => '0');
    elsif rising_edge(clk) and avs_write = '1' then
      case avs_address is
        when "00"   => reg_red_duty_cycle <= avs_writedata(31 downto 0);
        when "01"   => reg_green_duty_cycle         <= avs_writedata(31 downto
0);
        when "10"   => reg_blue_duty_cycle     <= avs_writedata(31 downto 0);
        when "11" => reg_period <= avs_writedata(31 downto 0);
        when others => null; -- ignore writes to unused registers
      end case;
    end if;
  end process;

end architecture arch;
```

Platform Designer (on your PC)

1. Open up platform designer and create new component
2. In `Component 1`, set `name` and `display name` both to the name of the component
3. In `Files` add the avalon file and all of the files it depends on. Make sure the avalon file is set as the top file by double-clicking in the attributes column and checking the top-level file box. `Analyze Synthesis Files` and debug any VHDL errors.
4. In `Signals & Int`, delete any outputs under `avalon_slave_0`.
5. Click `<<add interface>>` and add a `Clock Input`. Rename to `clk`. Add a signal, and rename to `clk`.
6. Click `<<add interface>>` and add a `Reset Input`. Rename to `rst`. Add a signal, and rename to `rst`.
7. Click `<<add interface>>` and add a `Conduit`. Rename to the name of your component. Add signals for each hardware i/o for your VHDL file. Set name and signal type to the same thing.
8. Select `avalon_slave_0`, and set clock and reset to `clk` and `rst` with the dropdown menus. Click `finish`.
9. Back on the main page of Platform Designer, double-click on the component you just created to add it.
10. Connect `clk` and `rst` on the component to `clk` and `clk_rst` on `fpga_clk`.
11. Connect `avalon_slave_0` on the component to `master` on `jtag_master` and `h2f_lw_axi_master` on `hps`.
12. Double click in the conduit row under the `Export` column in the component and rename to the name of the component.
13. Set `Base` to a unique address to be used as the base address for the component.
14. Click `Generate HDL`. NOTE: you will have to repeate this generation any time you change your VHDL code.
15. In the top menu under `Generate`, click `Instantiation Template` and copy the new signals at the end of the component declaration. Add those lines to the `soc_system` component declaration and instantiation in the project's top level file.

Quartus Project (on your PC)

1. Back in the Quartus project, go File>Open and navigate to `./soc_system/synthesis/` and open `soc_system.qip`. Then go Project>Add Current File to Project.
2. Compile the project and fix any bugs.
3. Go File>Convert Programming Files
4. Change `Programming file type` to `Raw Binary File (.rbf)`.
5. Change `Programming Mode` to `Passive Parallel x16`.
6. Change `File Name` to `soc_system.rbf`.
7. Select `SOF Data`, click `Add File`, navigate to `./outputfiles`, and select `de10nano_top.sof`.
8. Click `Generate`.

Setting Up FPGA Boot Process (in the VM)

1. Create a directory for your ***project*** if it is not already created in /srv/tftp/de10nano/ and switch to that directory
2. Copy the rbf file (FPGA bitstream) for your component to that directory
3. In the github repo, navigate to /linux/dts and add the following to the project (not component) `.dts` file:

```
<component_name>: <component_name>@<hardware_base_address> {
compatible = "<last_name>,<component_name>";
reg = <<hardware_base_address> 16>;
};
```

Repalce indicated spots with correct values. The hardware base address should be `0xff200000` plus whatever you set the base address as for the component in Platform Designer. The `.dts` file could look something like this:

```
#include "socfpga_cyclone5_de10nano.dtsi"

/{
pwm_rgb: pwm_rgb@ff210000 {
compatible = "jensen,pwm_rgb";
reg = <0xff210000 16>;
};
};
```

4. USING ABSOLUTE PATHS, create a symoblic link from `linux-socfpga/arch/arm/boot/dts/intel/socfpga/*.dts` to the file in your repository with the following commands:

```
ln -s <your-repo>/linux/dts/*.dts <linux-
socfpga>/arch/arm/boot/dts/intel/socfpga/
```

5. Ensure that the .dtb file is added to the Makefile at `linux-socfpga/arch/arm/boot/dts/intel/socfpga/Makefile`.
6. Navigate to `linux-socfpga/` and build the dtb by running

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
```

7. Copy the `.dtb` file from `arch/arm/boot/dts/intel/socfpga` to `/srv/tftp/de10nano/<project_name>` and rename the file to `<project_name.dtb>`.
8. Navigate to `/srv/tftp/de10nano/bootscripts`, and create a ***project*** .script file with the following contents:

```
# file directories
setenv tftpkerneldir ${tftpdir}/kernel
setenv tftpprojectdir ${tftpdir}/<project_name>
setenv nfsrootdir /srv/nfs/de10nano/ubuntu-rootfs

# kernel bootargs
setenv bootargs console=ttyS0,115200 ip=${ipaddr} root=/dev/nfs rw
nfsroot=${serverip}:${nfsrootdir},v4,tcp nfsrootdebug earlyprintk=serial

# file names
setenv fpgaimage <project_name>.rbf
setenv dtbimage <project_name>.dtb
setenv bootimage zImage

# memory addresses where files get loaded into
setenv fpgadata 0x2000000
setenv fpgadatasize 0x700000
setenv dtbaddr 0x00000100
setenv kerneladdr 0x8000

# commands to get files, configure the fpga, and load the kernel
setenv getfpgadata 'tftp ${fpgadata} ${tftpprojectdir}/${fpgaimage}'
setenv getdtb 'tftp ${dtbaddr} ${tftpprojectdir}/${dtbimage}'
setenv getkernel 'tftp ${kerneladdr} ${tftpkerneldir}/${bootimage}'
setenv loadfpga 'fpga load 0 ${fpgadata} ${fpgadatasize}'

# get all of the files and boot the device
run getfpgadata;
run loadfpga;
run getdtb;
run getkernel;
run bridge_enable_handoff;
bootz ${kerneladdr} - ${dtbaddr}
```

Make sure to update the file to have the correct project name in the three indicated spots.

9. Convert the `.scrpit` file to a U-Boot image with `mkimage`:

```
mkimage -A arm -O linux -T script -C none -a 0 -e 0 -n "<project_name>
bootscript" -d <project_name>.script <project_name>.scr
```

10. To tell the FPGA to look at this boot image, we need to link it to `u-boot.scr` with `ln -s <project_name>.scr u-boot.scr`. The FPGA should now boot with the new boot script.

Creating the Device Driver (in the VM)

1. Navigate to `/linux/` in the Github repo and create a new directory for the component, and in it make a file `<component_name.c>`.
2. In this file, paste the following code in order:

Imports:

```
#include <linux/module.h>           // basic kernel module definitions
#include <linux/platform_device.h>  // platform driver/device definitions
#include <linux/mod_devicetable.h>  // of_device_id, MODULE_DEVICE_TABLE
#include <linux/io.h>               // iowrite32/ioread32 functions
#include <linux/mutex.h>            // mutex defintions
#include <linux/miscdevice.h>       // miscdevice defintions
#include <linux/types.h>            // data types like u32, u16, etc.
#include <linux/fs.h>               // copy_to_user, etc.
#include <linux/kstrtox.h>          // kstrtou8, etc.
```

Register Offsets:
For each register in your Avalon file, define the offsets (0x4 is 32 bits).

```
#define <register_name>_OFFSET 0x0

#define SPAN 16
```

Dev struct:
This struct is what is used to hold instance-specific info about the device. It includes a `void __iomem * <reg_name>;` for every Avalon register.

```
/**
* struct <component_name>_dev - Private <component> device struct.
* @base_addr: Pointer to the component's base address
* @<reg_name>: <reg_purpose>
* @miscdev: miscdevice used to create a character device
* @lock: mutex used to prevent concurrent writes to memory
*
```

```
* An <reg_name>_dev struct gets created for each led patterns component.
*/
struct <reg_name>_dev {
void __iomem *base_addr;
void __iomem *<reg_name>;
struct miscdevice miscdev;
struct mutex lock;
};
```

Show and Store functions:

These are the functions used to r/w to the registers via sysfs. You will have a set for every register in the Avalon file.

```
/**
* <reg_name>_show() - Return the <reg_name> value
* to user-space via sysfs.
* @dev: Device structure for the <component_name> component. This
* device struct is embedded in the <component_name>' device struct.
* @attr: Unused.
* @buf: Buffer that gets returned to user-space.
*
* Return: The number of bytes read.
*/
static ssize_t <reg_name>_show(struct device *dev,
struct device_attribute *attr, char *buf)
{
u32 <reg_name>;
struct <reg_name>_dev *priv = dev_get_drvdata(dev);

<reg_name> = ioread32(priv-><reg_name>);

return scnprintf(buf, PAGE_SIZE, "%u\n", <reg_name>);
}

/**
* <reg_name>_store() - Store the <reg_name> value.
* @dev: Device structure for the <component_name> component. This
* device struct is embedded in the <component_name>
* platform device struct.
* @attr: Unused.
* @buf: Buffer that contains the stop_button value being written.
* @size: The number of bytes being written.
*
* Return: The number of bytes stored.
*/
static ssize_t <reg_name>_store(struct device *dev,
struct device_attribute *attr, const char *buf, size_t size)
{
u32 <reg_name>;
int ret;
```

```
struct <component_name>_dev *priv = dev_get_drvdata(dev);

// Parse the string we received as an unsigned 32-bit int
// See https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c#L289
ret = kstrtouint(buf, 0, &<reg_name>);
if (ret < 0) {
// kstrtobool returned an error
return ret;
}

iowrite32(<reg_name>, priv-><reg_name>);

// Write was successful, so we return the number of bytes we wrote.
return size;
}
```

Define sysfs attributes:

You will need one of these for each register.

```
// Define sysfs attributes
static DEVICE_ATTR_RW(<reg_name>);
```

Attribute Group:

You need one of these for the component. Make sure to add `&dev_attr_<reg_name>.attr,` lines as necessary for each register.

```
// Create an attribute group so the device core can
// export the attributes for us.
static struct attribute *<component_name>_attrs[] = {
&dev_attr_<reg_name>.attr,
NULL,
};
ATTRIBUTE_GROUPS(<component_name>);
```

Character device methods:

You just need one of these for the component as a whole.

```
/**
* <component_name>_read() - Read method for the <component_name> char device
* @file: Pointer to the char device file struct.
* @buf: User-space buffer to read the value into.
* @count: The number of bytes being requested.
* @offset: The byte offset in the file being read from.
*
* Return: On success, the number of bytes written is returned and the
* offset @offset is advanced by this number. On error, a negative error
```

```
* value is returned.
*/
static ssize_t <component_name>_read(struct file *file, char __user *buf,
size_t count, loff_t *offset)
{
size_t ret;
u32 val;

/*
* Get the device's private data from the file struct's private_data
* field. The private_data field is equal to the miscdev field in the
* <component_name>_dev struct. container_of returns the
* <component_name>_dev struct that contains the miscdev in private_data.
*/
struct <component_name>_dev *priv = container_of(file->private_data,
struct <component_name>_dev, miscdev);

// Check file offset to make sure we are reading from a valid location.
if (*offset < 0) {
// We can't read from a negative file position.
return -EINVAL;
}
if (*offset >= SPAN) {
// We can't read from a position past the end of our device.
return 0;
}
if ((*offset % 0x4) != 0) {
// Prevent unaligned access.
pr_warn("<component_name>_read: unaligned access\n");
return -EFAULT;
}

val = ioread32(priv->base_addr + *offset);

// Copy the value to userspace.
ret = copy_to_user(buf, &val, sizeof(val));
if (ret == sizeof(val)) {
pr_warn("<component_name>_read: nothing copied\n");
return -EFAULT;
}

// Increment the file offset by the number of bytes we read.
*offset = *offset + sizeof(val);

return sizeof(val);
}

/**
* <component_name>_write() - Write method for the <component_name> char device
* @file: Pointer to the char device file struct.
* @buf: User-space buffer to read the value from.
* @count: The number of bytes being written.
* @offset: The byte offset in the file being written to.
```

```
 *
 * Return: On success, the number of bytes written is returned and the
 * offset @offset is advanced by this number. On error, a negative error
 * value is returned.
 */
static ssize_t <component_name>_write(struct file *file, const char __user
*buf,
size_t count, loff_t *offset)
{
size_t ret;
u32 val;

struct <component_name>_dev *priv = container_of(file->private_data,
struct <component_name>_dev, miscdev);

if (*offset < 0) {
return -EINVAL;
}
if (*offset >= SPAN) {
return 0;
}
if ((*offset % 0x4) != 0) {
pr_warn("<component_name>_write: unaligned access\n");
return -EFAULT;
}

mutex_lock(&priv->lock);

// Get the value from userspace.
ret = copy_from_user(&val, buf, sizeof(val));
if (ret != sizeof(val)) {
iowrite32(val, priv->base_addr + *offset);

// Increment the file offset by the number of bytes we wrote.
*offset = *offset + sizeof(val);

// Return the number of bytes we wrote.
ret = sizeof(val);
}
else {
pr_warn("<component_name>_write: nothing copied from user space\n");
ret = -EFAULT;
}

mutex_unlock(&priv->lock);
return ret;
}
```

File Operations:

Defines operations supported by the device driver.

```
/**
* <component_name>_fops - File operations supported by the
* <component_name> driver
* @owner: The <component_name> driver owns the file operations; this
* ensures that the driver can't be removed while the
* character device is still in use.
* @read: The read function.
* @write: The write function.
* @llseek: We use the kernel's default_llseek() function; this allows
* users to change what position they are writing/reading to/from.
*/
static const struct file_operations <component_name>_fops = {
.owner = THIS_MODULE,
.read = <component_name>_read,
.write = <component_name>_write,
.llseek = default_llseek,
};
```

Probe and Remove:

These are called when the device driver is loaded into the kernel. Make sure to add in registers with their offsets where indicated!

```
static int <component_name>_probe(struct platform_device *pdev)
{

size_t ret;

struct <component_name>_dev *priv;
/*
* Allocate kernel memory for the <component_name> device and set it to 0.
* GFP_KERNEL specifies that we are allocating normal kernel RAM;
* see the kmalloc documentation for more info. The allocated memory
* is automatically freed when the device is removed.
*/
priv = devm_kzalloc(&pdev->dev, sizeof(struct <component_name>_dev),
GFP_KERNEL);
if (!priv) {
pr_err("Failed to allocate memory\n");
return -ENOMEM;
}
/*
* Request and remap the device's memory region. Requesting the region
* make sure nobody else can use that memory. The memory is remapped
* into the kernel's virtual address space because we don't have access
* to physical memory locations.
*/
priv->base_addr = devm_platform_ioremap_resource(pdev, 0);
if (IS_ERR(priv->base_addr)) {
pr_err("Failed to request/remap platform device resource\n");
```

```c
return PTR_ERR(priv->base_addr);
}

// Set the memory addresses for each register.
priv-><reg_name> = priv->base_addr + <reg_name>_OFFSET;

// Do any initializations when module is loaded
iowrite32(<value>, priv-><reg_name>);

// Initialize the misc device parameters
priv->miscdev.minor = MISC_DYNAMIC_MINOR;
priv->miscdev.name = "<component_name>";
priv->miscdev.fops = &<component_name>_fops;
priv->miscdev.parent = &pdev->dev;

// Register the misc device; this creates a char dev at /dev/<component_name>
ret = misc_register(&priv->miscdev);
if (ret) {
pr_err("Failed to register misc device");
return ret;
}

/* Attach the <component_name>'s private data to the platform device's struct.
* This is so we can access our state container in the other functions.
*/
platform_set_drvdata(pdev, priv);
pr_info("<component_name>_probe successful\n");
return 0;
}

/**
* <component_name>_remove() - Remove an <component_name> device.
* @pdev: Platform device structure associated with our <component_name> device.
*
* This function is called when an <component_name> devicee is removed or
* the driver is removed.
*/
static int <component_name>_remove(struct platform_device *pdev)
{
// Get the <component_name>'s private data from the platform device.
struct <component_name>_dev *priv = platform_get_drvdata(pdev);

// Any writes to do when the module is removed
iowrite32(0x0, priv-><reg_name>);

// Deregister the misc device and remove the /dev/led_patterns file.
misc_deregister(&priv->miscdev);
pr_info("<component_name>_remove successful\n");

return 0;
}
```

Last few bits:

Last couple of lines...

```
/*
* Define the compatible property used for matching devices to this driver,
* then add our device id structure to the kernel's device table. For a device
* to be matched with this driver, its device tree node must use the same
* compatible string as defined here.
*/
static const struct of_device_id <component_name>_of_match[] = {
    { .compatible = "<last_name>,<component_name>", },
    { }
};
MODULE_DEVICE_TABLE(of, <component_name>_of_match);

/*
* struct <component_name>_driver - Platform driver struct for the
<component_name> driver
* @probe: Function that's called when a device is found
* @remove: Function that's called when a device is removed
* @driver.owner: Which module owns this driver
* @driver.name: Name of the pwm_rgb driver
* @driver.of_match_table: Device tree match table
*/
static struct platform_driver <component_name>_driver = {
    .probe = <component_name>_probe,
    .remove = <component_name>_remove,
    .driver = {
        .owner = THIS_MODULE,
        .name = "<component_name>",
        .of_match_table = <component_name>_of_match,
        .dev_groups = <component_name>_groups,
    },
};

/*
* We don't need to do anything special in module init/exit.
* This macro automatically handles module init/exit.
*/
module_platform_driver(<component_name>_driver);

MODULE_LICENSE("Dual MIT/GPL");
MODULE_AUTHOR("<Full Name>");
MODULE_DESCRIPTION("<component_name> driver");
```

3. Run `make ARCH=arm` in the directory that `<component_name>.c` is in and fix any errors.
4. Copy the `.ko` file that gets generated to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc`.
5. Boot the FPGA - you should be able to load and remove the module from the home directory using `insmod <component_name>.ko` and `rmmod <component.ko>`. Check if it was loaded successfully

by running `dmesg | tail` and checking to see if the print statement from the probe function shows up.

6. To check if the character device driver works, load the module, navigate to `/sys/devices/platform/` and then `cd` into the directory corrisponding to the component you created. You should be able to read and write to the registers using `cat <register_name>` and `echo <value> > <register_name>`
7. To control them via software, see this example file. Cross-compile it with `/usr/bin/arm-linux -gnueabihf -gcc -o <file_name> <file_name>.c`, and copy the executable to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc/`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

// TODO: update these offsets if your address are different
#define HPS_LED_CONTROL_OFFSET 0x0
#define BASE_PERIOD_OFFSET 0x8
#define LED_REG_OFFSET 0x4

static volatile int keep_running = 1;

void int_handler(int irrelevant)
/**
 * int_handler() - Switch FPGA to hardware control mode and exit program when
cntl-C is entered
 * @arg1: TODO
 *
 * TODO
 *
 * Return: void.
 */
{
    printf("\nLOOP KILLED!\n");
    keep_running = 0;
}

int main () {
    FILE *file;
    size_t ret;
    uint32_t val;

    file = fopen("/dev/led_patterns" , "rb+" );
    if (file == NULL) {
        printf("failed to open file\n");
        exit(1);
    }
```

```
    // Test reading the registers sequentially
    printf("\n************************************\n*");
    printf("* read initial register values\n");
    printf("************************************\n\n");

    ret = fread(&val, 4, 1, file);
    printf("HPS_LED_control = 0x%x\n", val);

    ret = fread(&val, 4, 1, file);
    printf("base period = 0x%x\n", val);

    ret = fread(&val, 4, 1, file);
    printf("LED_reg = 0x%x\n", val);

    // Reset file position to 0
    ret = fseek(file, 0, SEEK_SET);
    printf("fseek ret = %d\n", ret);
    printf("errno =%s\n", strerror(errno));


    printf("\n************************************\n*");
    printf("* write values\n");
    printf("************************************\n\n");
    // Turn on software-control mode
    val = 0x01;
    ret = fseek(file, HPS_LED_CONTROL_OFFSET, SEEK_SET);
    ret = fwrite(&val, 4, 1, file);
    // We need to "flush" so the OS finishes writing to the file before our
code continues.
    fflush(file);

    // Write cool pattern to LEDs as long until ctl-c
    signal(SIGINT, int_handler);
    int count = 0;
    char slow = 0x1;
    char fast = 0x80;
    while (keep_running)
    {
        fast = (fast >> 1) | (fast << 7);
        if (count > 16)
        {
            count = 0;
            slow = (slow << 1) | (slow >> 7);
        }
        val = fast | slow;
        ret = fseek(file, LED_REG_OFFSET, SEEK_SET);
        ret = fwrite(&val, 4, 1, file);
        fflush(file);
        usleep(20*1000);
        count = count + 1;
    }
```

```
    // Turn on hardware-control mode
    printf("back to hardware-control mode....\n");
    val = 0x00;
    ret = fseek(file, HPS_LED_CONTROL_OFFSET, SEEK_SET);
    ret = fwrite(&val, 4, 1, file);
    fflush(file);

    val = 0x12;
    ret = fseek(file, BASE_PERIOD_OFFSET, SEEK_SET);
    ret = fwrite(&val, 4, 1, file);
    fflush(file);

    sleep(5);

    // Speed up the base period!
    val = 0x02;
    ret = fseek(file, BASE_PERIOD_OFFSET, SEEK_SET);
    ret = fwrite(&val, 4, 1, file);
    fflush(file);


    printf("\n************************************\n*");
    printf("* read new register values\n");
    printf("************************************\n\n");

    // Reset file position to 0
    ret = fseek(file, 0, SEEK_SET);

    ret = fread(&val, 4, 1, file);
    printf("HPS_LED_control = 0x%x\n", val);

    ret = fread(&val, 4, 1, file);
    printf("base period = 0x%x\n", val);

    ret = fread(&val, 4, 1, file);
    printf("LED_reg = 0x%x\n", val);

    fclose(file);
    return 0;
}
```