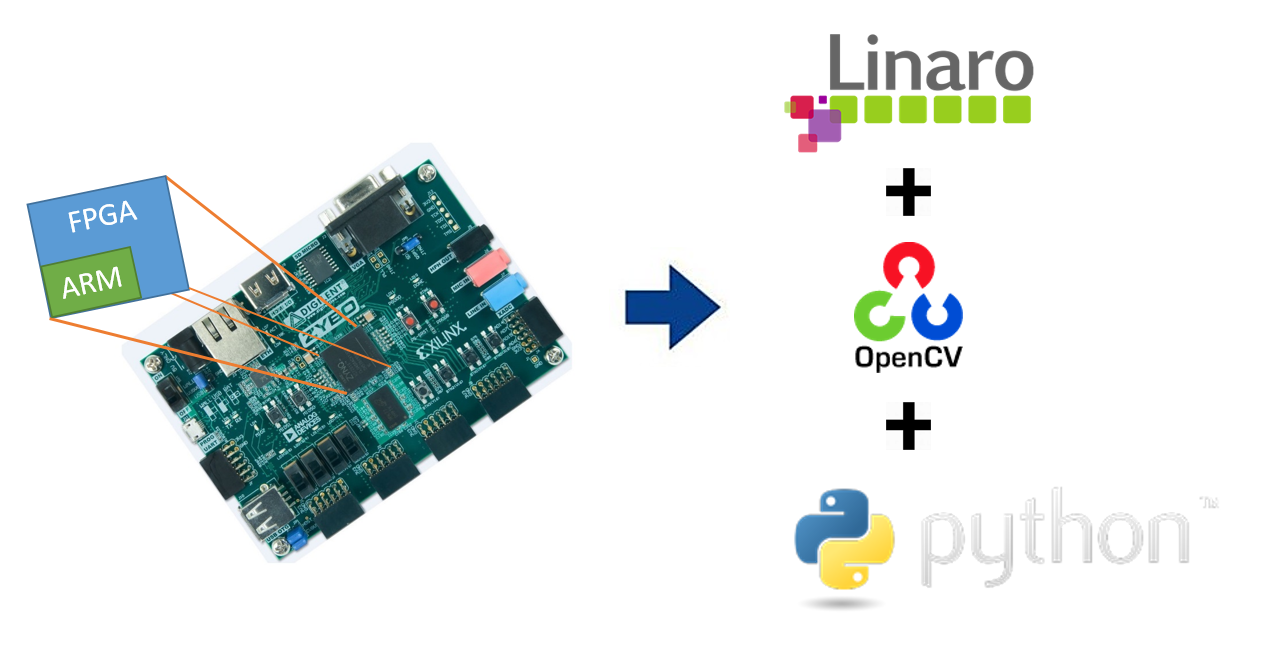
Getting Started

with ZYBO



目录

[1 Zynq操作系统安装指南 4](#_Toc438421446)

[1.1 Linaro Ubuntu系统 4](#_Toc438421447)

[1.2 在ZYBO上使用Linaro Ubuntu 4](#_Toc438421448)

[2 ZYBO硬件功能和接口介绍 7](#_Toc438421449)

[2.1 GPIO 8](#_Toc438421450)

[2.2 PWM 9](#_Toc438421451)

[2.3 UART 9](#_Toc438421452)

[2.4 IIC 9](#_Toc438421453)

[2.5 ADC 10](#_Toc438421454)

[2.6 正交编码模块 10](#_Toc438421455)

[2.7 SPI 10](#_Toc438421456)

[2.8 CAN模块硬件接口 11](#_Toc438421457)

[2.9 超声波传感器接口 11](#_Toc438421458)

[2.10 外设综述 11](#_Toc438421459)

[3 软件功能 13](#_Toc438421460)

[3.1 C&C++编译环境 13](#_Toc438421461)

[3.2 OpenCV 13](#_Toc438421462)

[3.3 Python 14](#_Toc438421463)

[4 在Linux Image上进行实验 15](#_Toc438421464)

[4.1 硬件实验 15](#_Toc438421465)

[4.1.1 GPIO 15](#_Toc438421466)

[4.1.2 PWM 15](#_Toc438421467)

[4.1.3 ADC 16](#_Toc438421468)

[4.1.4 IIC 18](#_Toc438421469)

[4.1.5 SPI和OLED 18](#_Toc438421470)

[4.1.6 UART 19](#_Toc438421471)

[4.1.7 温度传感器 20](#_Toc438421472)

[4.2 使用Linux驱动硬件的原理 21](#_Toc438421473)

[4.2.1 使用mmap 21](#_Toc438421474)

[4.2.2 调用系统驱动 22](#_Toc438421475)

[4.3 OpenCV 22](#_Toc438421476)

[4.3.1 边缘检测 22](#_Toc438421477)

[4.4 Python实验 24](#_Toc438421478)

[4.4.1 ds18b20 Python实验 24](#_Toc438421479)

[4.4.2 OpenCV Python实验 24](#_Toc438421480)

[5 ZYBO标准硬件搭建 26](#_Toc438421481)

[5.1 通过Tcl建立硬件 26](#_Toc438421482)

[6 构建Zynq Linaro系统 27](#_Toc438421483)

[6.1 BOOT.bin构建 27](#_Toc438421484)

[6.2 devicetree构建 31](#_Toc438421485)

[6.3 uImage构建 32](#_Toc438421486)

[6.4 系统启动流程 33](#_Toc438421487)

[7 修改硬件和软件的方法 35](#_Toc438421488)

[7.1 设计一个IP 35](#_Toc438421489)

[7.2 添加用户IP到工程文件 36](#_Toc438421490)

[7.3 设计一个SDK软件程序 36](#_Toc438421491)

[8 软硬件交互, 使用硬件加速计算 39](#_Toc438421492)

[9 参考书籍 40](#_Toc438421493)

[10 附录 41](#_Toc438421494)

[10.1 附录A Linux API文档 41](#_Toc438421495)

[10.1.1 GPIO API 41](#_Toc438421496)

[10.1.2 PWM API 42](#_Toc438421497)

[10.1.3 UART API 43](#_Toc438421498)

[10.1.4 SPI API 44](#_Toc438421499)

[10.1.5 IIC API 45](#_Toc438421500)

[10.1.6 XADC API 45](#_Toc438421501)

# Zynq操作系统安装指南

## Linaro Ubuntu系统

在本次发布的Zynq开源硬件使用的是一款开源的轻量级Linux操作系统，名为Linaro Ubuntu。

## 在ZYBO上使用Linaro Ubuntu

本节主要讲解的是如何制作SD卡镜像，并通过SD卡启动Linaro Ubuntu系统。

（1）制作Linaro Ubuntu镜像需要准备一张MicroSD卡（推荐8G，至少4G），并在连接了Internet的Linux操作系统的环境下进行。

（2）SD卡分区并格式化：

将SD卡插入Linux电脑（虚拟机系统需要将其挂载进虚拟机）。

在/dev目录下找到SD卡设备，比如/dev/sdX（这可能是sdb,或sdc,sdd等，在部分内建读卡器的电脑中，该设备可能为/dev/mmcblkX,通过df命令可以查看所有挂载的系统，一般而言，外部存储设备会挂在在/media目录下或者/mnt目录下。）

使用如下命令卸载分区(注意在df里面挂在了几个分区就卸载几个分区，下面的示例中只卸载了第一分区)

* **bash$** sudo umount /dev/sdX1

输入以下命令进入分区工具(一般而言，请不要对sda进行分区，这回破坏你的系统！)：

* **bash$** sudo fdisk /dev/sdX

用**m**命令可以查看该工具的命令，之后请按顺序使用如下命令：

* 清除该SD卡的原有分区
* **n**，建立第一分区，之后程序会询问：
* 是否建立主分区？（回车，默认为是）
* 分区号？（回车，默认为最小分区号）
* 分区起始扇区？（回车，默认最小起始扇区）
* 分区末尾扇区？（输入“+512M”，回车，创建一个512M的分区）
* **n**，建立第二分区，之后程序会询问：
* 是否建立主分区？（回车，默认为是）
* 分区号？（回车，默认为最小分区号）
* 分区起始扇区？（回车，默认最小起始扇区）
* 分区末尾扇区？（回车，默认将剩余的空间全部给第二分区）
* **w**，将之前对SD卡的操作写入SD卡

查看/dev目录下是否有/dev/sdX1和sdX2设备，如果没有就将SD卡退出重插就有了。之后输入命令:

* ***bash$*** *sudo mkfs.msdos /dev/sdX1 #将sdX1格式化为FAT分区*
* ***bash$*** *sudo mkfs.ext4 /dev/sdX2 #将sdX2格式化为EXT4分区*

下载dosfstools，重定义卷名，输入如下命令:

* **bash$** sudo apt-get install dosfstools
  + - ***bash$*** *sudo dosfslabel /dev/sdX1 BOOT*
    - ***bash$*** *sudo e2label /dev/sdX2 FS*

重新插拔SD卡，在/media目录下会自动挂在BOOT和FS两个分区

（3）在Xilinx XUP的Github上下载FAT分区文件，FAT分区需要三个文件：boot.bin（引导文件）,devicetree.dtb（设备树文件）,uImage（内核镜像），并将他们复制到SD卡的FAT分区文件。

（4）在Xilinx XUP的Github上下载经过修改的Linaro Ubuntu文件系统到，并将系统文件包复制到用户目录：

* + - ***bash$*** *cp $MATERIAL/linaro.tar ~*

将压缩包解压并进入文件系统：

* + - ***bash$*** *tar -zvf linaro.tar*
    - ***bash$*** *cd linaro*

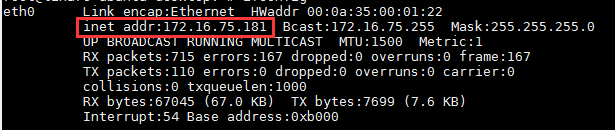
将文件系统同步到SD卡的EXT4分区：

* + - ***bash$*** *rsync -av . /media/FS*

等待完成后，将SD卡插入ZYBO板卡，连接电源、串口并确认启动方式为SD卡启动后，将ZYBO上电启动，此时，板卡右边LD11和LD10相继点亮，证明系统已经开始启动。

（5）通过使用终端软件通过串口连接ZYBO板卡就可以看到命令行界面。

（6）ZYBO网络连接。使用网线将ZYBO与交换机（或者路由器）相连后，在串口终端中输入ifconfig命令查看当前IP地址，如下图：（用户使用的系统中IP地址可能不是该值）



通过vim编辑工具可修改IP地址到用户自己的IP：

* **bash$** cd /etc/network
* **bash$** vim interfases

将里面的内容修改为：

auto lo

iface lo inet loopback

auto eth0

iface eth0 inet static

address \*\*\*

netmask \*\*\*

gateway \*\*\*

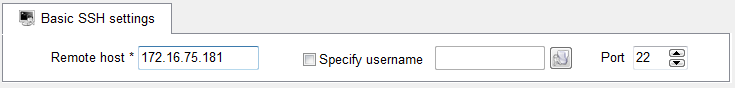
其中\*\*\*为用户自己的IP address、netmask、gateway。

之后重启ZYBO网络。

* **bash$** /etc/init.d/networking stop
* **bash$** /etc/init.d/networking start

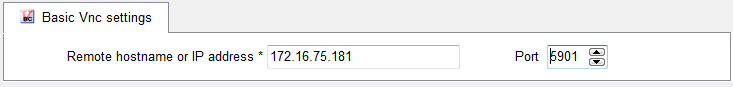
（7）网络设置后，同样也可以通过网络连接ZYBO，方法有两种：SSH和VNC。

SSH连接：输入上步所设置的IP地址，端口号为默认的22。

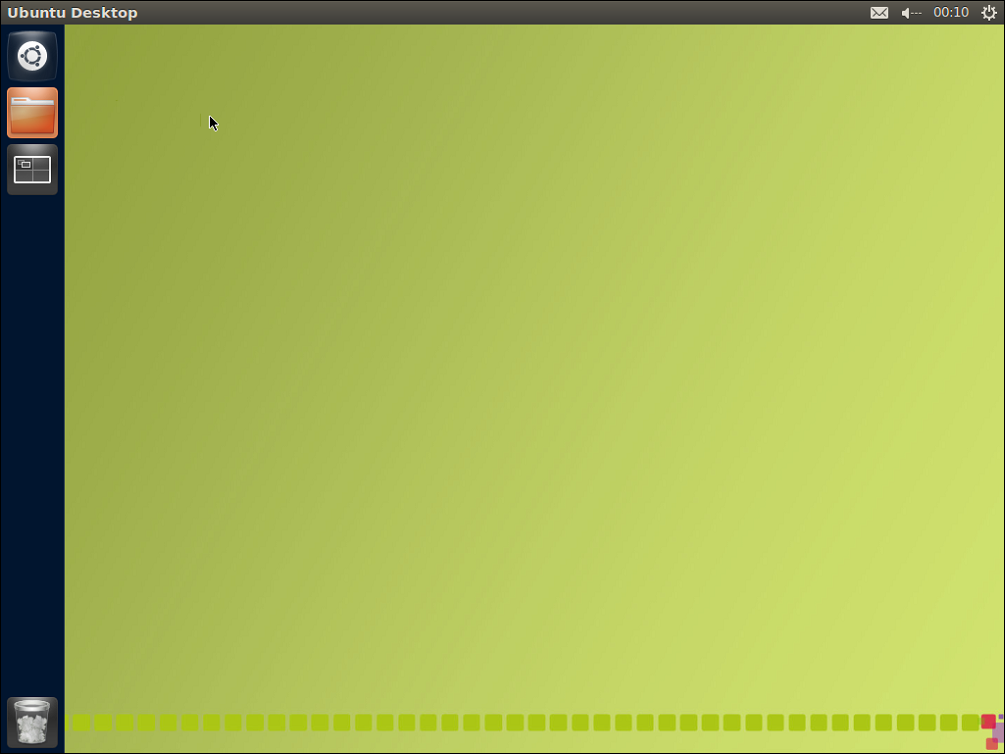


输入SSH用户名为root，默认密码为111，即可登录。

VNC连接：VNC是远程图形桌面软件，在预置系统中VNC默认开启，通过VNC连接可以看到远程桌面。



默认情况下端口（Port）为5901，密码为1q2w3e4r。

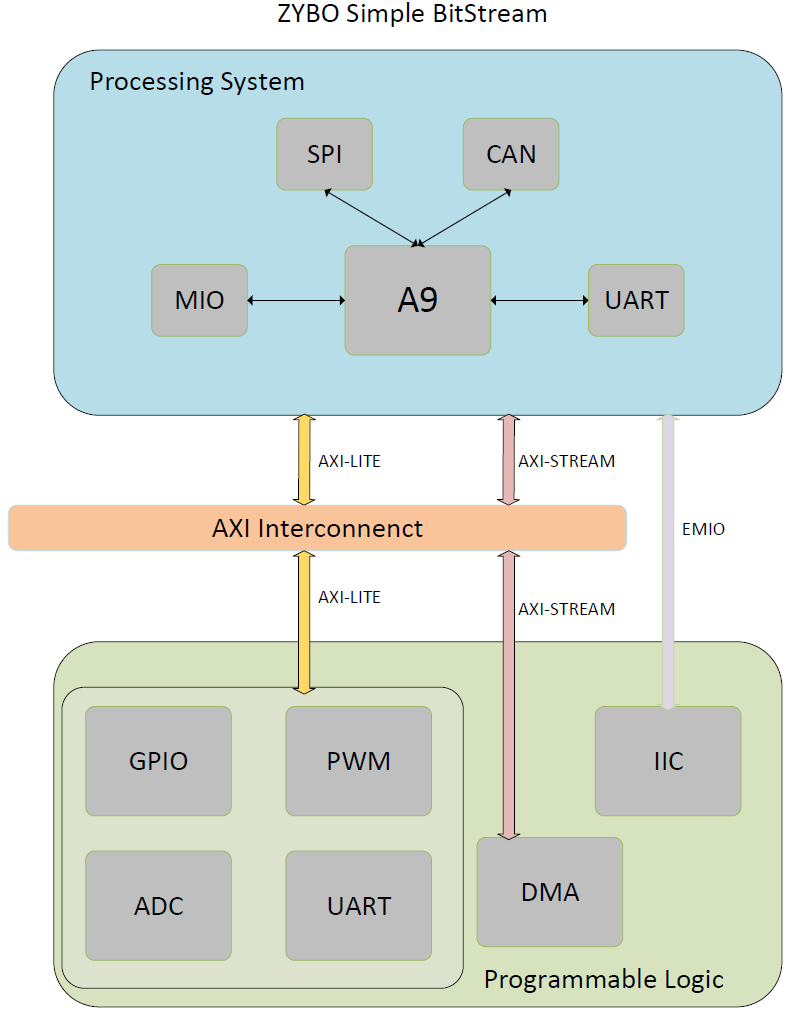


上图即为通过VNC连接后显示的图形界面。

# ZYBO硬件功能和接口介绍

本次使用的ZYBO板卡集成了ARM Cortex-A9双核（Processing System, PS）和FPGA逻辑资源（Programmable Logic, PL）。在本次开发的硬件工程中，大部分基础的SoC接口功能都被引出。

下面是本次硬件工程的框图：



## GPIO

在这次发布的硬件工程中，总共有26个GPIO从PL端通过AXI\_GPIO的IP核引出。其具体连接如下表所示。

表2-1 PL端GPIO接口对应表

|  |  |  |
| --- | --- | --- |
| GPIO接口 | Zynq-7000引脚 | PMod管脚或器件名 |
| GPIO0\_CH1\_0 | M14 | LD0 |
| GPIO0\_CH1\_1 | M15 | LD1 |
| GPIO0\_CH1\_2 | G14 | LD2 |
| GPIO0\_CH1\_3 | D18 | LD3 |
| GPIO0\_CH2\_0 | G15 | SW0 |
| GPIO0\_CH2\_1 | P15 | SW1 |
| GPIO0\_CH2\_2 | W13 | SW2 |
| GPIO0\_CH2\_3 | T16 | SW3 |
| GPIO0\_CH2\_4 | R18 | BTN0 |
| GPIO0\_CH2\_5 | P16 | BTN1 |
| GPIO0\_CH2\_6 | V16 | BTN2 |
| GPIO0\_CH2\_7 | Y16 | BTN3 |
| GPIO1\_CH1\_0 | W20 | JB4 |
| GPIO1\_CH1\_1 | Y19 | JB8 |
| GPIO1\_CH1\_2 | W18 | JB9 |
| GPIO1\_CH1\_3 | W19 | JB10 |
| GPIO2\_CH1\_0 | V15 | JC1 |
| GPIO2\_CH1\_1 | W15 | JC2 |
| GPIO2\_CH1\_2 | W14 | JC7 |
| GPIO2\_CH1\_3 | Y14 | JC8 |
| GPIO2\_CH2\_0 | T14 | JD1 |
| GPIO2\_CH2\_1 | T15 | JD2 |
| GPIO2\_CH2\_2 | P14 | JD3 |
| GPIO2\_CH2\_3 | U14 | JD7 |
| GPIO2\_CH2\_4 | U15 | JD8 |
| GPIO2\_CH2\_5 | V17 | JD9 |

另外，从PS端还有3路GPIO接口引出，分别为ZYBO板上的LD4（MIO7）、BTN4（MIO50）和BTN5（MIO51）。

## PWM

共有6路PWM接口从PL端引出，PWM周期、占空比均可通过软件设置。

表2-2 PWM接口对应表

|  |  |  |
| --- | --- | --- |
| PWM接口 | Zynq-7000引脚 | PMod管脚 |
| PWM0 | V12 | JE1 |
| PWM1 | W16 | JE2 |
| PWM2 | J15 | JE3 |
| PWM3 | H15 | JE4 |
| PWM4 | T17 | JE9 |
| PWM5 | Y17 | JE10 |

## UART

共有两路串口分别通过PS和PL端在ZYBO硬件上被引出。从PS端引出的为UART1，是调试串口，用于用户调试，不建议在上面进行编程。PL端UART0为编程串口，接口对应如下表。

表2-3 UART0接口对应表

|  |  |  |
| --- | --- | --- |
| UART0接口 | Zynq-7000引脚 | PMod管脚 |
| TXD | V18 | JD10 |
| RXD | R14 | JD4 |

## IIC

从PL端引出两路IIC，IIC0给用户编程使用，IIC1板上默认连接了EEPROM芯片SSM2603，可供掉电时的数据存储用。IIC接口对应如下表：

表2-4 IIC接口对应表

|  |  |  |
| --- | --- | --- |
| IIC接口 | Zynq-7000引脚 | PMod管脚或器件名 |
| IIC0SCL | V13 | JE7 |
| IIC0SDA | U17 | JE8 |
| IIC1SCL | N18 | SSM2603 SCLK |
| IIC1SDA | N17 | SSM2603 SDIN |

## ADC

ADC用于模拟信号的采集。在ZYBO中，PMod接口JA中有4路12位精度模数信号采集模块，这个AD可采集差分的峰峰值为1V的模拟信号。ADC接口对应如下表：

|  |  |  |
| --- | --- | --- |
| ADC接口 | Zynq-7000引脚 | PMod管脚 |
| ADC14\_P | N15 | JA1 |
| ADC14\_N | N16 | JA7 |
| ADC7\_P | L14 | JA2 |
| ADC7\_N | L15 | JA8 |
| ADC15\_P | K16 | JA3 |
| ADC15\_N | J16 | JA9 |
| ADC6\_P | K14 | JA4 |
| ADC6\_N | J14 | JA10 |

## 正交编码模块

在这次所使用的硬件中，包含两路正交编码模块，可接两路正交编码器，可用于位置捕获、速度检测等。其硬件接口如下表所示：

表2-6 正交编码模块硬件接口

|  |  |  |
| --- | --- | --- |
| 正交编码模块接口 | Zynq-7000引脚 | PMod管脚 |
| M\_left\_phA | T10 | JC4 |
| M\_left\_phB | U12 | JC10 |
| M\_right\_phA | T11 | JC3 |
| M\_right\_phB | T12 | JC9 |

## SPI

本次硬件工程中包含1路SPI。SPI被接在PS端，从JF中被接出。其硬件接口如下表所示：

表2-7 SPI模块硬件接口

|  |  |  |
| --- | --- | --- |
| SPI接口 | Zynq-7000引脚 | PMod管脚 |
| MOSI | MIO10 | JF2 |
| MISO | MIO11 | JF3 |
| SCLK | MIO12 | JF4 |
| SS | MIO13 | JF1 |

## CAN模块硬件接口

本次硬件工程中包含1路CAN。SPI被接在PS端，从JF中被接出。其硬件接口如下表所示：

表2-8 CAN模块硬件接口表

|  |  |  |
| --- | --- | --- |
| CAN接口 | Zynq-7000引脚 | PMod管脚 |
| RX | MIO14 | JF9 |
| TX | MIO15 | JF10 |

## 超声波传感器接口

在部分应用场合，超声波传感器经常被使用，故在本次的硬件工程中包含了3路硬件的超声波传感器采集模块。其接口如下表所示：

表2-9超声波传感器接口表

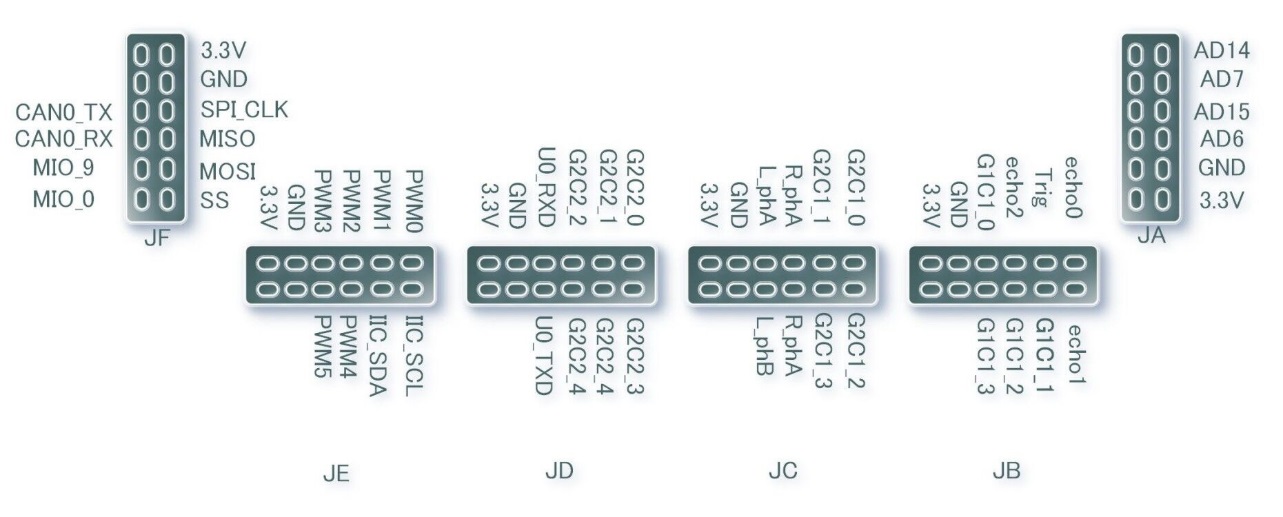
|  |  |  |
| --- | --- | --- |
| 超声波传感器接口 | Zynq-7000引脚 | PMod管脚 |
| Trig | U20 | JB2 |
| Echo0 | T20 | JB1 |
| Echo1 | Y18 | JB7 |
| Echo2 | V20 | JB3 |

## 外设综述

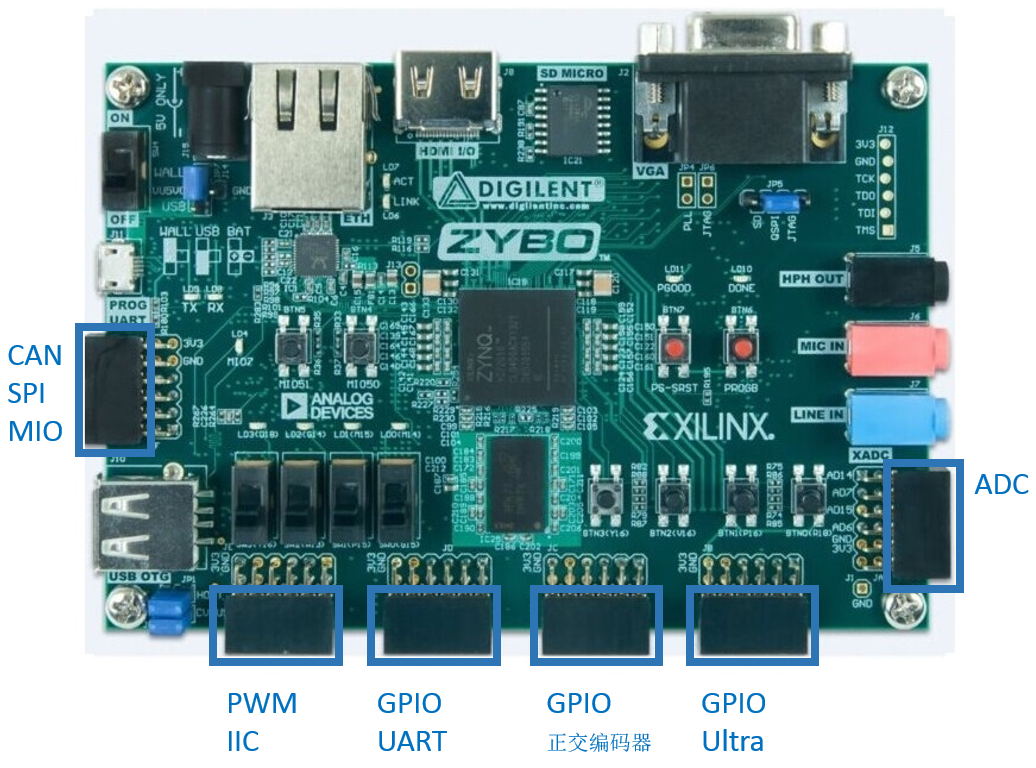
在第二章中主要介绍了本次硬件工程设计的所有外设接口，这些外设接口主要包含：

* 29个GPIO接口，其中3个连接Zybo的PS端，26个连接PL端；
* 6个周期、占空比均可设置的PWM接口；
* 2个UART接口，1路从PS接出，为调试接口，一路从PL接出，为用户接口；
* 2个IIC接口，1路给用户编程使用，1路默认连接了EEPROM芯片；
* 4个差分ADC接口，12位精度；
* 1个SPI接口；
* 1个CAN接口；
* 2个正交解码器接口；
* 3个超声波传感器接口；

下图为PMod引脚定义示意图：



下图为实物管脚定义示意图：



# 软件功能

在这次使用的Linux系统中，包含了不少软件功能，例如： C&C++编译环境、OpenCV、Python2.7等。

## C&C++编译环境

XUP发布的Linaro Ubuntu系统中包含了C&C++编译环境，可以在板上编译C或者C++程序，而非通过交叉编译，这将在一定程度上提高开发或者学习的效率。

使用vim编写一个Helloworld程序：

* + - ***bash$*** *vim helloworld.c*

*#include <stdio.h>*

*int main(void){*

*printf("Hello world!\n\r");*

*return 0;*

*}*

编译并执行helloworld.c

* + - ***bash$*** *gcc helloworld.c –o helloworld*
    - ***bash$*** *./helloworld*



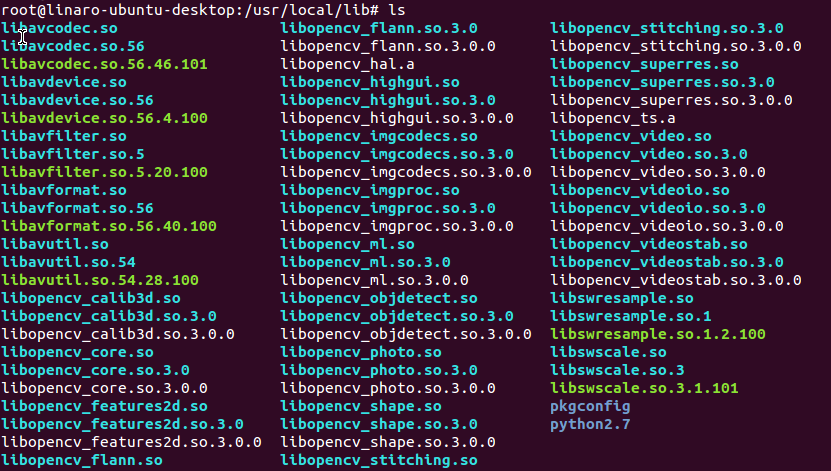
可以看到Hello World！的输出。

## OpenCV

OpenCV是一个开源发行的跨平台计算机视觉库，可以运行在Linux、Windows和Mac OS操作系统上。在这次发布的Linaro系统中已经包含了编译好的最新版openCV3.0。

使用以下命令可以看到OpenCV的版本信息：

* + - ***bash$*** *cd /usr/local/lib/*
    - ***bash$*** *ls*



看到如上图的打印信息就表明OpenCV3.0已经成功安装。

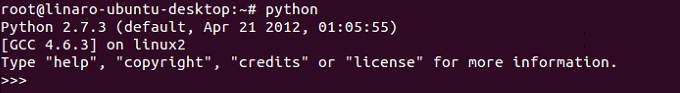
## Python

Python是一种面向对象、解释型计算机程序设计语言。其特色是具有丰富和强大的库，能够轻松的将其他语言制作的各种模块（如C、C++）联结在一起，因此被称为“胶水语言”。在程序的原型设计时，Python的应用颇为广泛。

在本Linaro系统中，已经集成了Python2.7的环境，可以方便的使用Python语言进行编程。使用

* + - ***bash$*** *python*

命令可以进入Python的命令行界面，可以看到安装的Python版本是2.7.3。



# 在Linux Image上进行实验

## 硬件实验

本实验的代码已经在GitHub中上传，在Github上下载Linux\_example，并把这个文件夹拷贝到文件系统用户目录下。

进入该目录，可以看到以下文件下，其中xlib为库文件源码，其他文件夹为具体Demo代码。

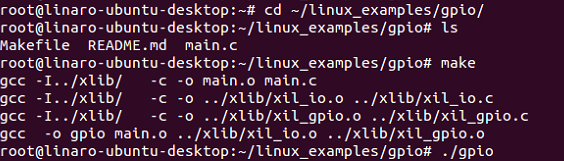
### GPIO

GPIO实验功能为板上LED闪烁，以验证PL和PS端的GPIO功能。进入GPIO目录，在执行程序后，可以看到PL端连接的LD0、LD1同时闪烁两次后熄灭，然后PS端连接的LD4闪烁几次，程序退出。

输入如下命令（#后为注释内容，下同）：

* + - ***bash$*** *cd ~/linux\_examples/gpio/ #进入GPIO程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看GPIO代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./gpio #执行程序*

就可看到程序成功运行，板上LED按照上述的叙述闪烁。



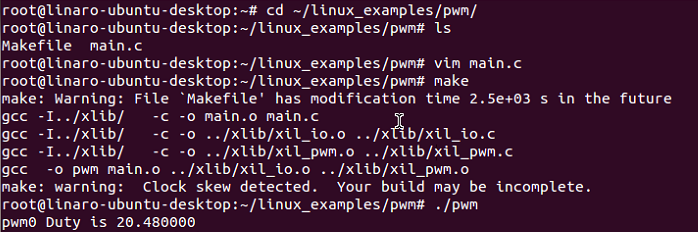
### PWM

PWM实验功能为使用板上的PWM0~3接口输出分别为20%、40%、60%、80%的PWM波，并返回PWM0的占空比的百分比。

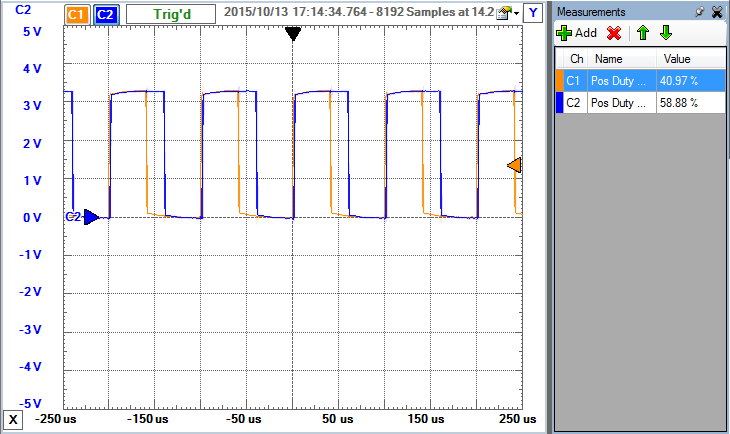
输入如下命令：

* + - ***bash$*** *cd ~/linux\_examples/pwm/ #进入PWM程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看PWM代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./pwm #执行程序*

程序运行后可以看到如下图输出。



可以看到PWM0的占空比为20.48%。同样使用示波器可以看到波形。（这里我使用的是Digilent Analog Discovery，橙色C1为PWM1，蓝色C2为PWM2）



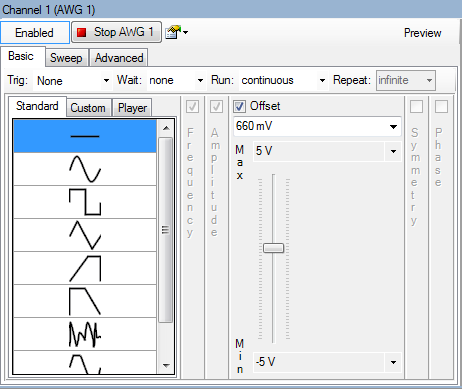
### ADC

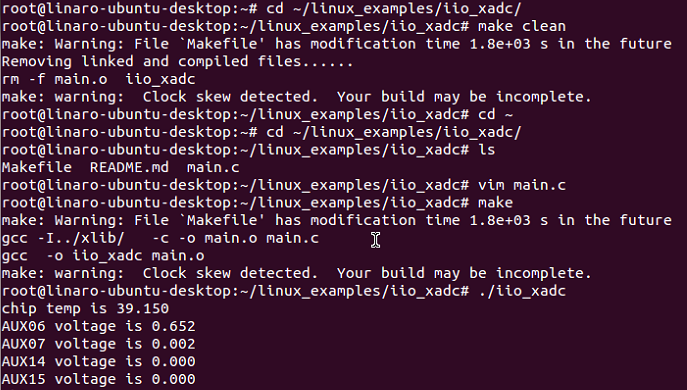
ADC实验主要是完成了4路外部输入的模数转换和芯片内部温度的测量。ADC测量精度为12位，最大测量1V的差分信号。

输入以下命令：

* + - ***bash$*** *cd ~/linux\_examples/iio\_xadc/ #进入ADC程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看ADC代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./iio\_xadc #执行程序*

使用Analog Discovery输出直流电压660mV，如下图：





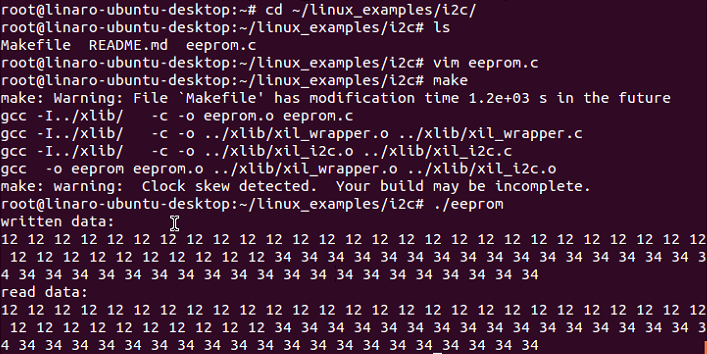
可以看到芯片温度是39.15℃，ADC6通道电压输出为0.652V，其他通道都接地，电压输出为0V。

### IIC

IIC实验主要完成了对片上EEPROM的读写，EEPROM接在IIC1上。程序先完成了对EEPROM写操作，写入了前37位0x12和后37位0x34的数据。

输入以下命令：

* + - ***bash$*** *cd ~/linux\_examples/i2c/ #进入I2C程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim eeprom.c #查看I2C代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./eeprom #执行程序*



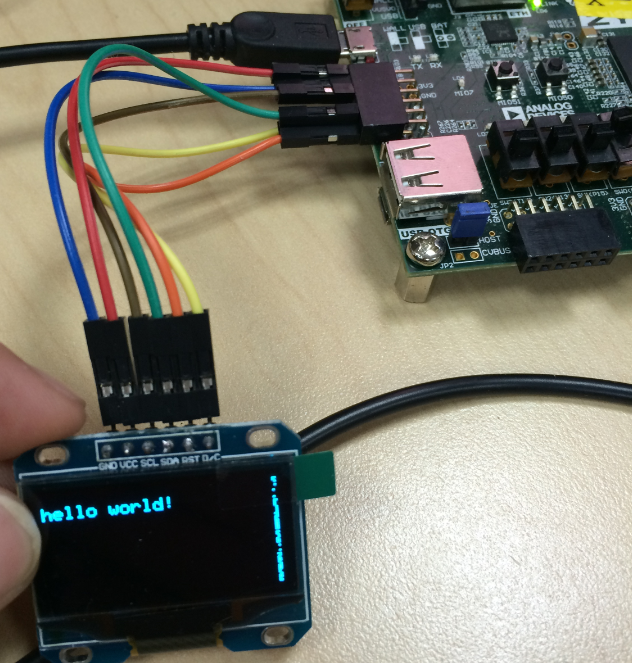
可以看到数据读写都成功了。

### SPI和OLED

SPI实验以OLED屏为例，完成写屏幕操作。硬件上将OLED屏与PMOD口JF对应管脚相连，使用SPI总线协议写OLED屏，在屏上写入hello world！

输入以下命令：

* + - ***bash$*** *cd ~/linux\_examples/oled/ #进入OLED程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看OLED代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./oled #执行程序*

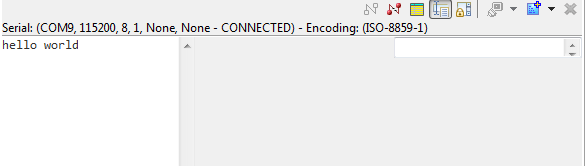


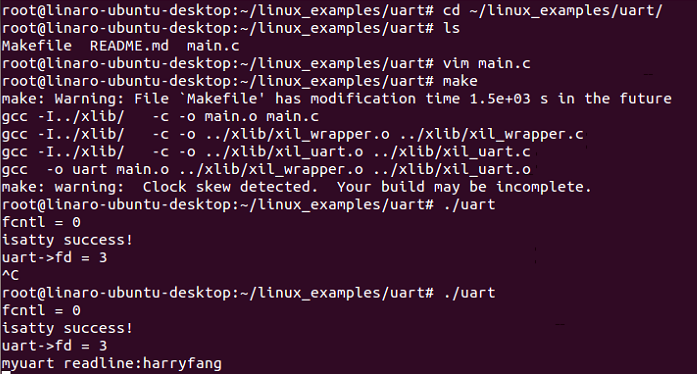
### UART

UART实验主要完成了对用户串口UART0的读写。硬件上使用4针串口线连接UART0和PC。程序先完成了对串口的写操作，先在串口中写入hello world，然后等待读串口。

输入以下命令：

* + - ***bash$*** *cd ~/linux\_examples/uart/ #进入UART程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看UART代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./uart #执行程序*





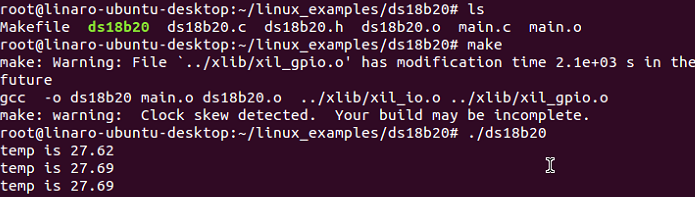
可以看到上图中在电脑终端中写入了hello world，在右边的输入框中输入了harryfang，在板子的终端中打印了收到的信息。

### 温度传感器

温度传感器实验通过ds18b20单总线温度传感器进行温度采集。程序通过GPIO1通道1的0引脚进行单总线通信，并返回当前温度。

将温度传感器data线连接到GPIO1CH1\_0上，然后输入以下命令：

* + - ***bash$*** *cd ~/linux\_examples/ds18b20/ #进入ds18b20程序目录*
    - ***bash$*** *ls #查看当前目录下文件*
    - ***bash$*** *vim main.c #查看ds18b20代码*
    - ***bash$*** *make #编译程序*
    - ***bash$*** *./ds18b20 #执行程序*



在终端窗口中可以看到输出的温度。

## 使用Linux驱动硬件的原理

4.1节中详细叙述了在Linux中如何驱动硬件完成实验效果，而本节中将为读者介绍操作Gpio、SPI等硬件的原理。

### 使用mmap

使用mmap进行硬件驱动与在Standalone下操作寄存器无异。在使用这种方式操作硬件的过程中，其实是将硬件的寄存器基址、偏移以文件的方式传入mmap函数进行内存映射，此后即可如同操作内存一般访问寄存器。

下面来看看mmap函数的定义：

void \*mmap(void \*addr, size\_t len, int prot, int flags, int fd, off\_t offset);

参数fd为即将映射到进程空间的文件描述字，一般由open()返回；

len是映射到调用进程地址空间的字节数，它从被映射文件开头offset个字节开始算起；

prot参数指定共享内存的访问权限。可取如下几个值的或：PROT\_READ（可读），PROT\_WRITE（可写），PROT\_EXEC（可执行），PROT\_NONE（不可访问），操作寄存器时一般设置为可读可写；

flags由以下几个常值指定：MAP\_SHARED, MAP\_PRIVATE, MAP\_FIXED。其中MAP\_SHARED、MAP\_PRIVATE必选其一，而MAP\_FIXED则不推荐使用。如果指定为MAP\_SHARED，则对映射的内存所做的修改同样影响到文件。如果是MAP\_PRIVATE，则对映射的内存所做的修改仅对该进程可见，对文件没有影响。寄存器操作时一般选MAP\_SHARED。

offset参数为映射偏移地址。

在本次的Linux例子下，包含了一个xlib/xil\_io.c的c文件，这个文件中有一个XilAddrMmap函数，对mmap进行了封装，只要用户写入需要操作对象的指针和初始地址，即可将该对象的寄存器映射到内存。然后使用X\_Out32和X\_In32这两个函数可以往寄存器写值。具体可参见xil\_io.c中的详细代码。

以Gpio为例，X\_digitalChipWrite为写整个通道的Gpio状态的函数，其源码如下：

void X\_digitalChipWrite(xil\_gpio \* const me, u8 ch, u32 value)

{

if(ch < 1 || ch > 2)

{

printf("[file:%s\_\_function:%s\_\_linefile:%d]out of memory!\n",

\_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

return;

}

me->xIO->Xil\_Out32(me->xIO, ((ch - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_DATA\_OFFSET, value);

}

其传入值为Gpio指针，通道和写入值，me->xIO->Xil\_Out32(me->xIO, ((ch - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_DATA\_OFFSET, value);这句为操作寄存器，其将value值写入内存地址，该地址为Gpio指针对应的基址加上((ch - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_DATA\_OFFSET地址偏移。

### 调用系统驱动

在linux系统中，可以用调用系统驱动的方式去驱动硬件。

在后文将提到使用devicetree，在devicetree中，可描述的信息包括：

•CPU的数量和类别

•内存基地址和大小

•总线和桥

•外设连接

•中断控制器和中断使用情况

•GPIO控制器和GPIO使用情况

•Clock控制器和Clock使用情况

它基本上就是画一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树，并根据它展开出Linux内核中的platform\_device、i2c\_client、spi\_device等设备，而这些设备用到的内存、IRQ等资源，也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

使用devicetree生成启动文件后，内核会自动生成设备，并以文件形式映射到/dev目录下。

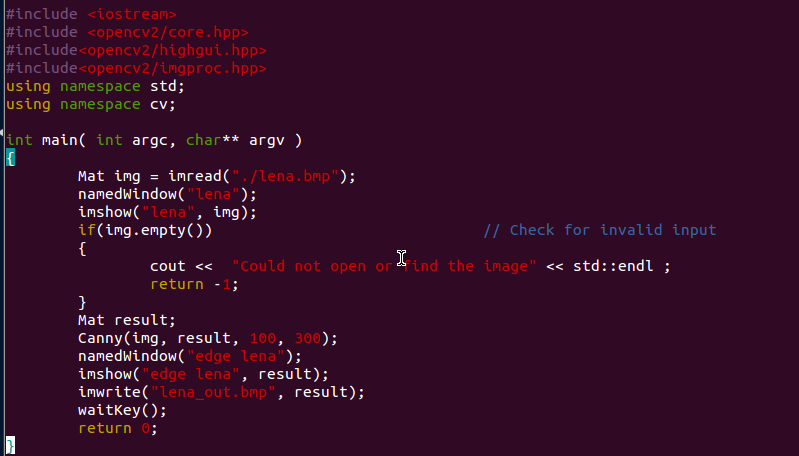
以SPI为例，在/dev可以看到spidev32766.0设备，对应的就是上文中提到的PS端的SPI硬件接口，我们可以以文件方式访问SPI设备。在本次的Linux例子下，包含了一个xlib/xil\_spi.c的c文件，可以看到操作SPI的源码。

## OpenCV

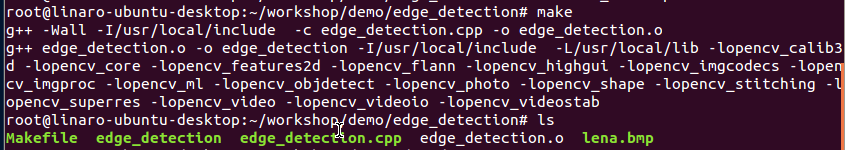
在上一节已经介绍了在Linaro系统中已经集成了OpenCV环境，下面将演示OpenCV的Demo。

### 边缘检测

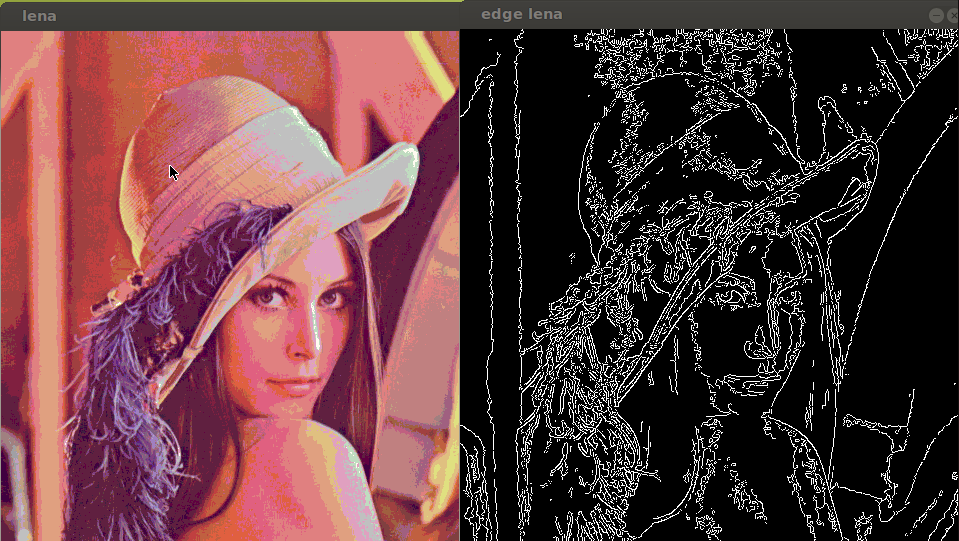
从GitHub上下载OpenCV的Demo代码，可以使用vim命令查看edge\_detection.cpp。



在Makefile文件中定义了编译规则，使用make命令可以编译代码。使用ls命令查看本文件夹下已生成了edge\_detection和edge\_detection.o文件。



生成的edge\_detection为可执行文件，输入./edge\_detection，执行后弹出两个图片框，分别为边缘检测前后的图片，并本文件夹下出现lena\_out.bmp，该图为边缘检测后的输出图片。



## Python实验

使用Python控制GPIO读取温度值，并调用Python\_openCV进行边缘检测。

### ds18b20 Python实验

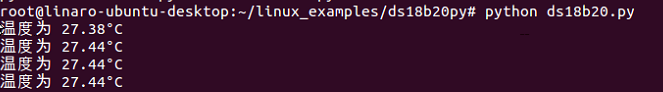
前面4.1.7节中介绍了使用C语言读取ds18b20的数据，本实验将通过Python语言将温度传感器的数据读取打印。

进入~/linux\_examples/ds18b20py目录，看到如下文件：



其中ds18b20.c和ds18b20.h是原先4.1.7中的文件，ds18b20py.c定义了Python模块和C程序的联结方式，ds18b20.so是通过Makefile生成的Python库，而ds18b20.py是最终执行的Python脚本。

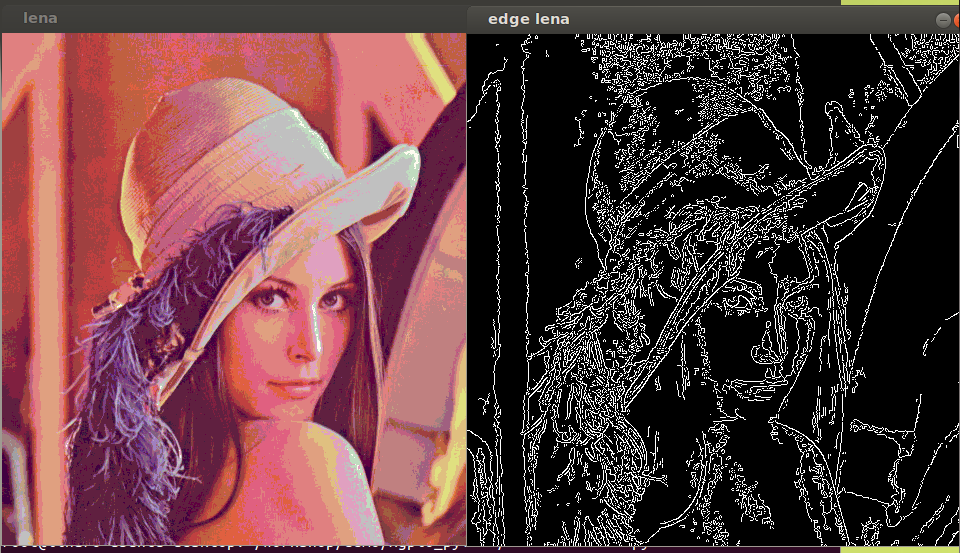
使用python ds18b20.py执行代码，可以看到当前温度输出。



### OpenCV Python实验

和4.2一样，使用Python语言也可以写OpenCV应用。本实验使用Python语言实现边缘检测的效果。

用python edge\_detection.py执行edge\_detection.py文件，可以看到效果和使用C++做的效果完全一样。



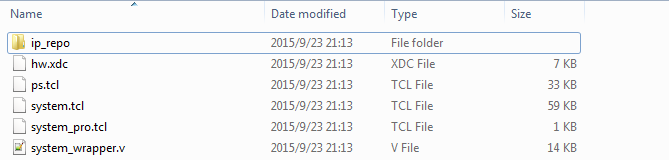
# ZYBO标准硬件搭建

前4节讲到如何使用已经生成的Linux image上进行实验，而在5、6节将会主要讲述如何自己搭建硬件并生成Linux image。

## 通过Tcl建立硬件

（1）Tcl是Xilinx Vivado执行命令的脚本文件，通过Tcl可以方便快捷的生成硬件工程，或者完成约束等。

使用Tcl生成本次ZYBO标准硬件，首先，在Github上下载hardware\_pro文件夹，其中包含了本次实验所需全部的材料。

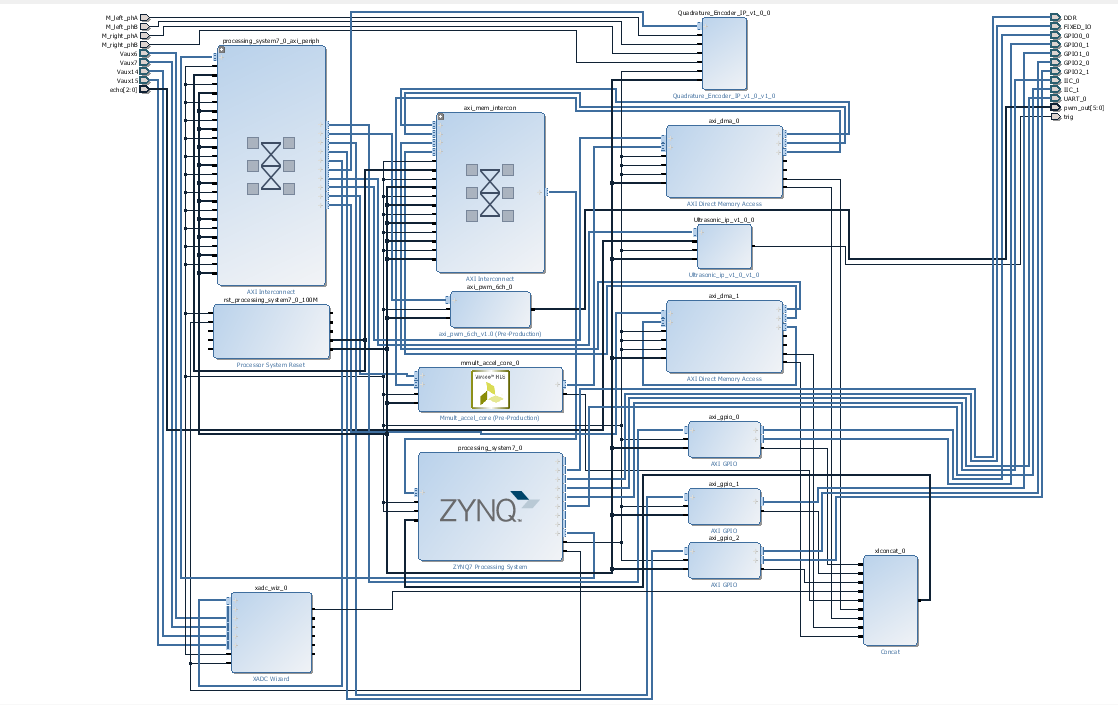


（2）打开Vivado 2014.4，看到最下方的Tcl Console窗口，输入以下命令：

*cd $wordspace #$ wordspace指hardware\_pro文件夹路径*

*source ./system\_pro.tcl #执行Tcl命令生成工程*

（3）等待工程完成后，即可完成Block Design构建、工程中和实现和生成Bitstream。



# 构建Zynq Linaro系统

在第1节中已经讲述了如何在SD卡中安装Linaro系统，而在本节中将会讲述如何构建系统中的boot.bin（引导文件）,devicetree.dtb（设备树文件）,uImage（内核镜像），并说明系统如何启动。

## BOOT.bin构建

本节介绍如何构建BOOT.bin：

首先去github下载uboot xilinx-v2015.1源码，下载地址为<https://github.com/Xilinx/u-boot-xlnx/tree/xilinx-v2015.1>。解压在工作目录

* ***bash$*** *cd u-boot-xlnx-xilinx-v2015.1/*
* ***bash$*** *make ARCH=arm zynq\_zybo\_config（如果你的开发板是zed请使用make ARCH=arm zynq\_zed\_config）*

修改uboot环境变量sdboot：

* *bash$ vim include/configs/zynq-common.h*

修改为：

*"sdboot=if mmcinfo; then " \*

*"run uenvboot; " \*

*"echo Copying Linux from SD to RAM... && " \*

*"load mmc 0 ${kernel\_load\_address} ${kernel\_image} && " \*

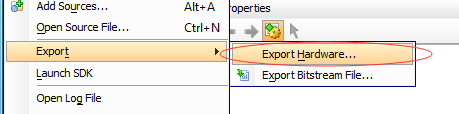
*"load mmc 0 ${devicetree\_load\_address} ${devicetree\_image} &&" \*

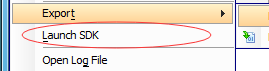
*"bootm ${kernel\_load\_address} - ${devicetree\_load\_address};"\*

*"fi\0" \*

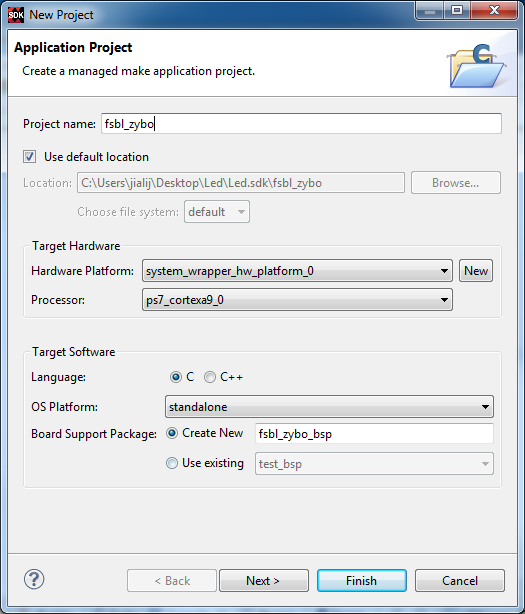
* *make ARCH=arm*
* *cp u-boot /[your-path]/u-boot.elf*

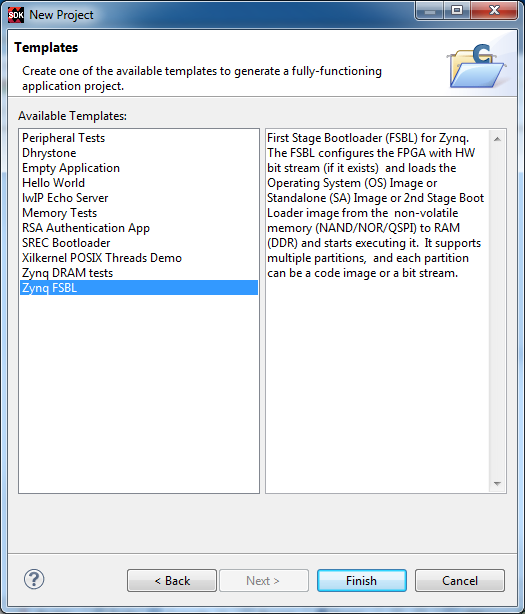
打开已经生成bitstream的vivado工程，导出bitstream，打开SDK：



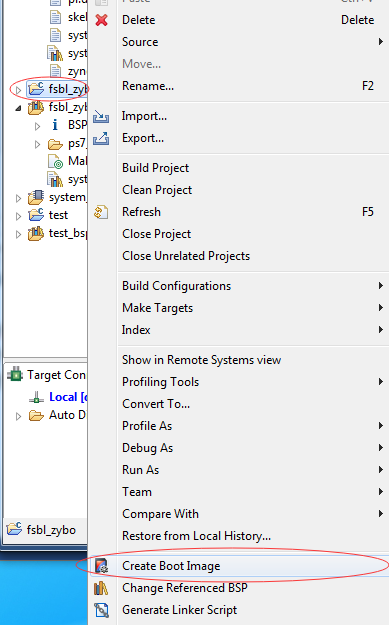


新建一个fsbl工程

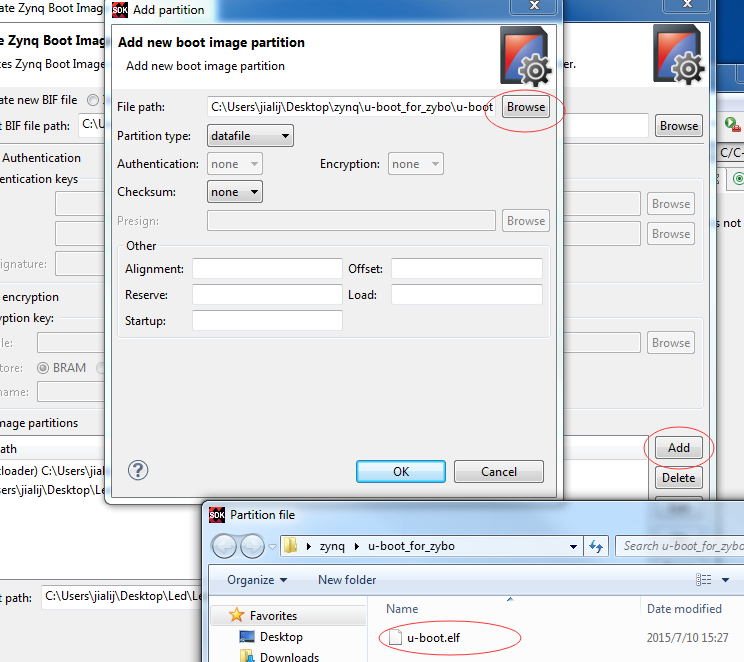


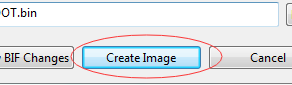


右键fsbl\_zybo工程选择



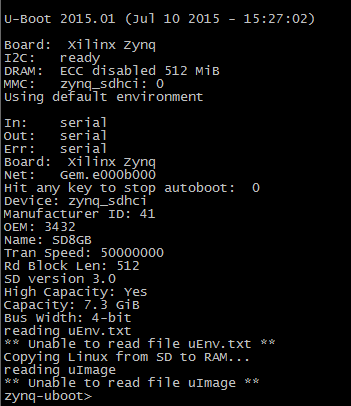
添加刚刚生成的u-boot.elf文件



点击

工程里面就生成了BOOT.bin

将BOOT.bin拷入SD卡测试



## devicetree构建

本节介绍如何构建devicetree.dtb，首先去github下载devicetree xilinx-v2015.1。下载地址为<https://github.com/Xilinx/device-tree-xlnx/tree/xilinx-v2015.1>，解压。

继续使用刚刚的SDK：

*SDK Menu: Xilinx Tools > Repositories > New... (<bsp repo>) > OK*

添加刚刚下载的devicetree xilinx-v2015.1

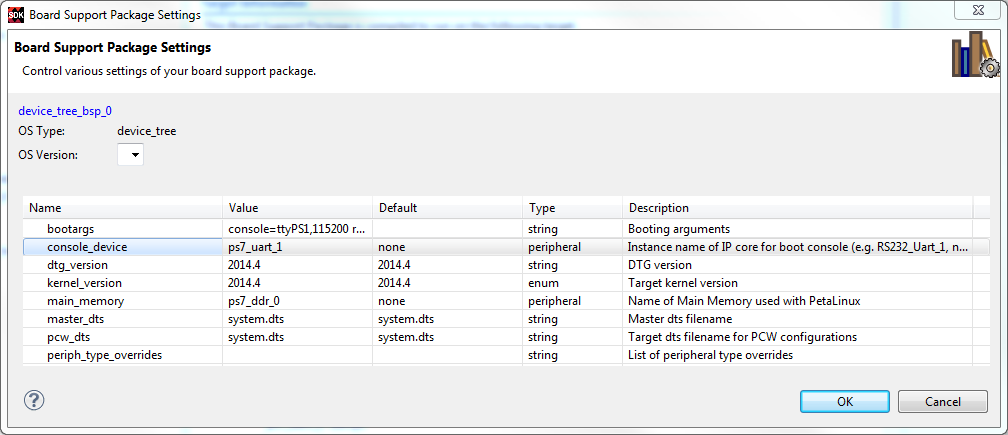
新建一个devicetree工程

*SDK Menu: File > New > Board Support Package > Board Support Package OS: device-tree > Finish*

修改bootargs为

*cma=64M console=ttyPS1,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=0*

修改console device为ps7\_uart\_1



使用DTC（在内核源码中的*linux-xlnx/scripts/dtc*）对dts进行编译

修改system.dts 找到&gem0改为

*&gem0 {*

*status = "okay";*

*phy-mode = "rgmii-id";*

*phy-handle = <&ethernet\_phy>;*

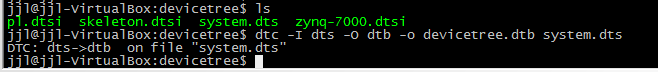
*ethernet\_phy: ethernet-phy@0 {*

*reg = <0>;*

*};*

*};*

* ***bash$*** *dtc -I dts -O dtb -o devicetree.dtb system.dts*



生成devicetree.dtb完毕

## uImage构建

本节介绍如何构建uImage，首先去github下载linux xilinx-v2015.1源码。下载地址为<https://github.com/Xilinx/linux-xlnx/tree/xilinx-v2015.1>。解压到用户目录：

* ***bash$*** *cd linux-xlnx-xilinx-v2015.1/*
* ***bash$*** *make ARCH=arm xilinx\_zynq\_defconfig*

加个usb摄像头驱动，到时候可以玩玩opencv(我共享的rootfs内已经移植了opencv)

* ***bash$*** *make ARCH=arm menuconfig*

*Device Drivers --->*

*Multimedia support --->*

*Media USB Adapters ---> 选择built-in*

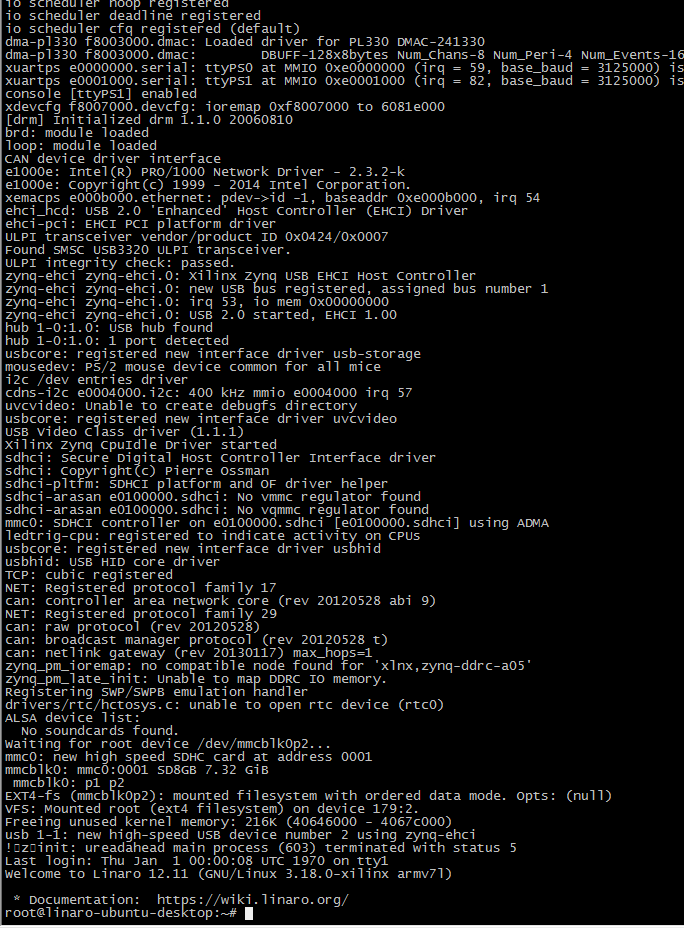
*USB Video Class (UVC) 选择built-in*

*UVC input events device support (NEW) 选择built-in*

* ***bash$*** *make ARCH=arm LOADADDR=0x00008000 uImage*

等待编译完成，将生成的BOOT.bin、devicetree.dtb、uImage拷贝到SD卡FAT分区。

之后按照1.2节中第4步的方法将Linaro根目录文件系统同步到SD卡EXT4分区，整个系统构建完毕，插入板子，确认板子的跳线帽接在SD卡启动选项，就可以登录linaro系统。



## 系统启动流程

Zybo上电后，首先通过读取SD卡FAT分区加载并解析BOOT.bin文件，单个A9核运行通过FSBL(First Stage Bootloader)引导uboot加载，uboot启动后首先加载另一个A9核启动。

随后U-Boot将devicetree.dtb的内存地址作为bootm的第三个参数，kernel通过寄存器R2找到device tree。device tree上的硬件信息将在在Linux启动前加载到内存。内核包含在uImage文件中，通过uboot加载，其缺损配置通过device tree文件加载。

内核加载完毕后引导加载SD卡中的EXT4分区中文件系统。



下图为Linaro Linux系统架构图。

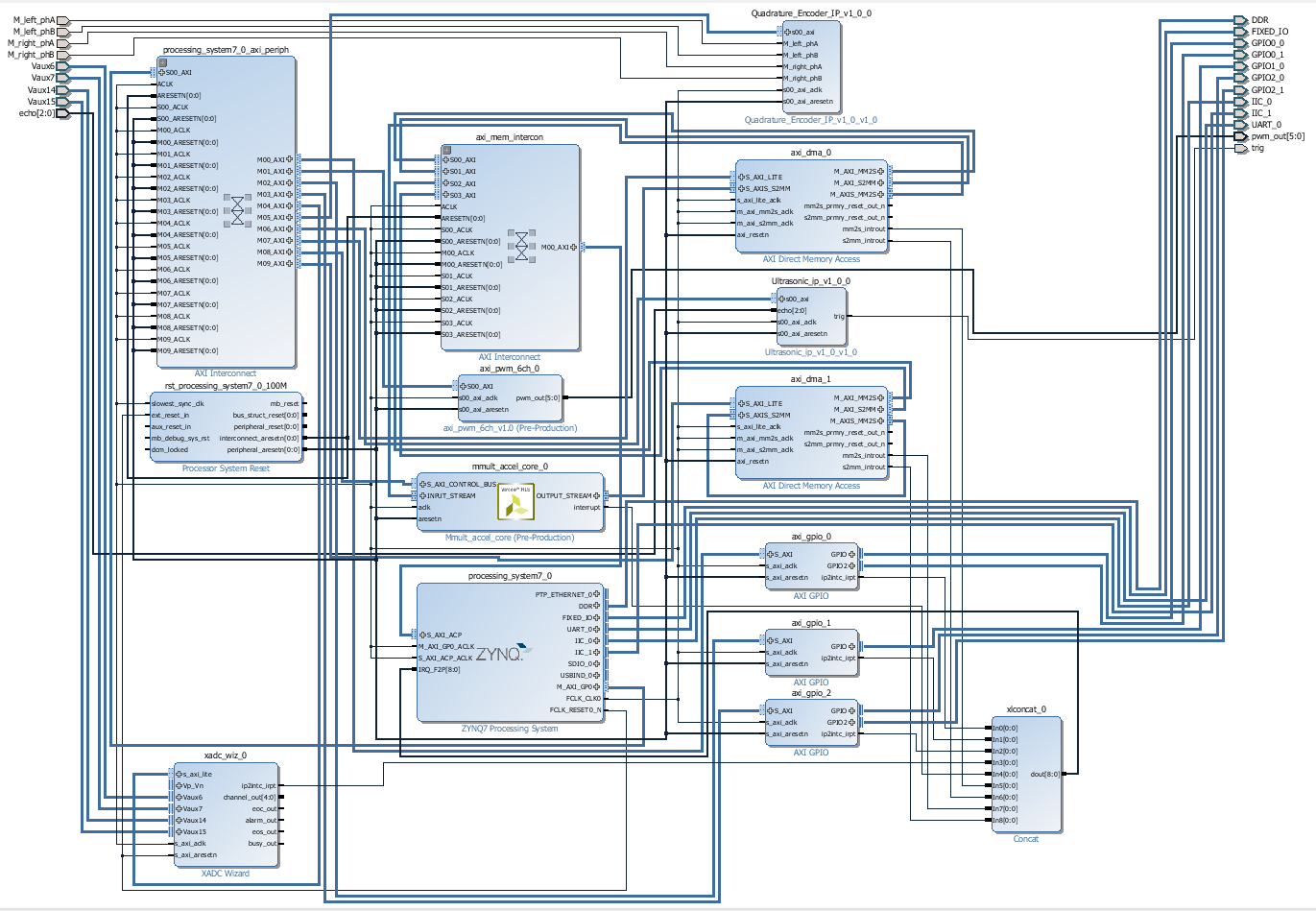


# 修改硬件和软件的方法

如果本次发布的硬件版本不符合应用要求，同样可以通过修改硬件的方法进行调整。

例如在使用的硬件版本中不需要超声波传感器而需要更多的PWM发生器，只需按照如下步骤操作，完成硬件修改。

打开硬件工程，点击左边IP Integrator选项卡中的Open Block Design，打开如下图文件：



图中名为Ultrasonic\_ip\_v1\_0\_0的IP是超声波传感器的IP，将它删除。

## 设计一个IP

1. 下面构建AXI PWM IP，执行Tools->Create and Package IP命令，打开创建用户IP向导，单击Next按钮。
2. 在Choose Create Peripheral or Package IP对话框中选择Create new AXI 4 peripheral单选选项，在IP Definition Location中设置添加的IP核文件存放位置和详细信息。
3. 在Add Interfaces对话框中设置Number of Registers为8，不用修改其他设置，下一步后，选Add IP to repository，点击Finish。
4. 对IP核进行编辑。在IP Catalog中可以看到刚刚添加的IP，右键单击axi\_pwm\_4ch的IP，在弹出的菜单中选择Edit in IP Packager选项，然后单击弹出对话框的OK按钮，这是会打开一个新的Vivado IDE来对IP核进行编辑。
5. 在新打开的IDE中将下载的v文件复制到Design Sources的两个文件中。
6. 选择Package IP窗口，看到一些打勾的选项，在Customization Parameters中点击Merge changes from IP Customization Parameters Wizard链接进行顶层文件更新。然后在Review and Package选项卡中点击Re-Package IP，完成IP设计。

## 添加用户IP到工程文件

1. 添加axi\_pwm\_4ch的IP，后点击Run Connent Automation将IP核与系统连接，并将pwm\_out进行Make External，看到pwm\_out\_1[3:0]的接口。
2. 右击sources窗口的Reset Output Products，后Generate Output Products，在system\_wrapper.v中已经看到pwm\_out\_1接口。
3. 修改hw.xdc中的约束。将原先Ultra的部分注释，并添加以下约束：

set\_property PACKAGE\_PIN U20 [get\_ports pwm\_out\_1[0]]

set\_property IOSTANDARD LVCMOS33 [get\_ports pwm\_out\_1[0]]

set\_property PACKAGE\_PIN T20 [get\_ports {pwm\_out\_1[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {pwm\_out\_1[1]}]

set\_property PACKAGE\_PIN Y18 [get\_ports {pwm\_out\_1[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {pwm\_out\_1[2]}]

set\_property PACKAGE\_PIN V20 [get\_ports {pwm\_out\_1[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {pwm\_out\_1[3]}]

1. 保存约束后，在Address Editor中可以看到：

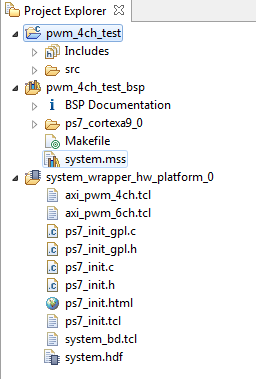


此时axi\_pwm\_4ch的基址为0x43C3\_0000。

1. 点击generate bitstream，完成中和、实现和生成比特流的步骤。
2. 生成比特文件后，选择File->Export->Export hardware和Launch SDK，将Vivado工程导入SDK。

## 设计一个SDK软件程序

1. 在SDK中选择File->New->Application Project命令，在Project Name文本框中输入pwm\_test，单击Next进入下一步。
2. 选择Hello World作为Application模版，单击Finish按钮完成。
3. 在Project Explorer中可以看到已经自动新建了应用程序工程、板级支持包（bsp）和硬件工程。



1. 打开pwm\_4ch\_test下的src目录中helloworld.c文件，写入以下程序：

#include <stdio.h>

#include "platform.h"

#include "axi\_pwm\_4ch.h"

#include "xparameters.h"

#define PWM\_4CH\_BASEADDR XPAR\_AXI\_PWM\_4CH\_0\_S00\_AXI\_BASEADDR

void print(char \*str);

int main() {

u32 high;

init\_platform();

print("Hello World\n\r");

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG0\_OFFSET, 2000000);

high = 2000000 \* 20 / 100;

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG1\_OFFSET, 0x80000000 + high);

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG2\_OFFSET, 2000000);

high = 2000000 \* 40 / 100;

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG3\_OFFSET, 0x80000000 + high);

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG4\_OFFSET, 2000000);

high = 2000000 \* 60 / 100;

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG5\_OFFSET, 0x80000000 + high);

AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG6\_OFFSET, 2000000);

high = 2000000 \* 80 / 100;

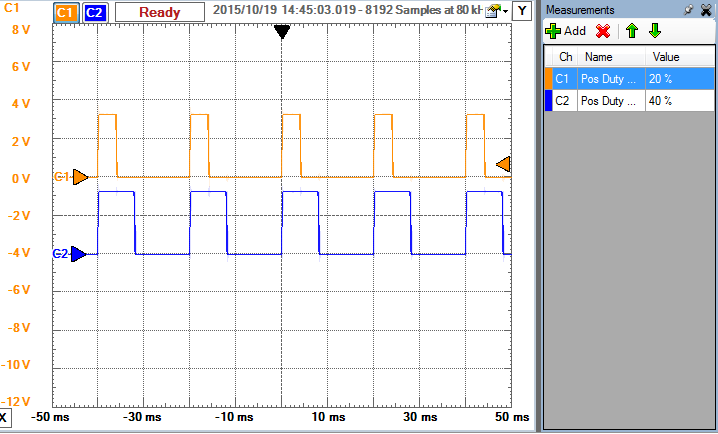
AXI\_PWM\_4CH\_mWriteReg(PWM\_4CH\_BASEADDR, AXI\_PWM\_4CH\_S00\_AXI\_SLV\_REG7\_OFFSET, 0x80000000 + high);

cleanup\_platform();

return 0;

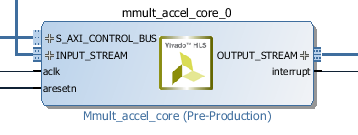
}

1. 将工程编译后运行，PWM已经输出，通过Analog Discovery的示波器功能已经可以看到下图为第一和第二通道的PWM波形。



# 软硬件交互, 使用硬件加速计算

在这次的标准硬件中包含了一个使用HLS编写的IP核，功能是进行浮点矩阵乘法，然后将矩阵乘法IP作为一个逻辑加速器和PS相连，组成一个完整的硬件加速器，并在Linux系统中编写一个测试程序。



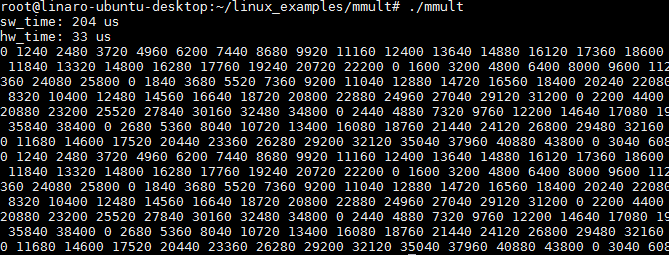
启动ZYBO后，将从GitHub中下载的linux\_dev文件夹下的放入用户目录。

* + - **bash$** cd linux\_dev/xdma
    - **bash$** insmod xdma.ko
    - **bash$** lsmod



可以看到xdma模块已经被挂载。

* + - **bash$** cd ~/linux\_examples/mmult/
    - **bash$** make
    - **bash$** ./mmult



可以看到硬件运行时间是软件运行时间的1/8左右，而且这还是算上了设备IO读写的时间，因此加速效果较为理想。

# 参考设计

以上章节介绍了XUP目前提供给用户的已有环境和各个外设的API，本章主要通过几个示范性的简单的DEMO来介绍如何使用现有的API进行二次开发工作。

## DEMO1 GPIO实验——按键控制LED灯

我们这个参考设计的功能是：能通过ZYBO板卡上的开关来控制对应开关上方的LED灯。

我们只需要使用xlib里面gpio的API，就可以完成整个设计。

设计源码可以在XUP github中下载到。用户需注意与xilb库的相对路径。

#include "xil\_gpio.h"

#define GPIO\_BASE\_ADDR 0x41200000

#define LD0 0

#define LD1 1

#define LD4 7

int main()

{

int i;

xil\_gpio \*my\_gpio = XilGpioCreate(GPIO\_BASE\_ADDR);

my\_gpio->modeChipWrite(my\_gpio,GPIO\_CH1,0x00);

my\_gpio->modeChipWrite(my\_gpio,GPIO\_CH2,0xff);

while(1)

{ my\_gpio->digitalChipWrite(my\_gpio,GPIO\_CH1,my\_gpio->digitalChipRead(my\_gpio,GPIO\_CH2));

}

XilGpioDestroy(my\_gpio);

return 0;

}

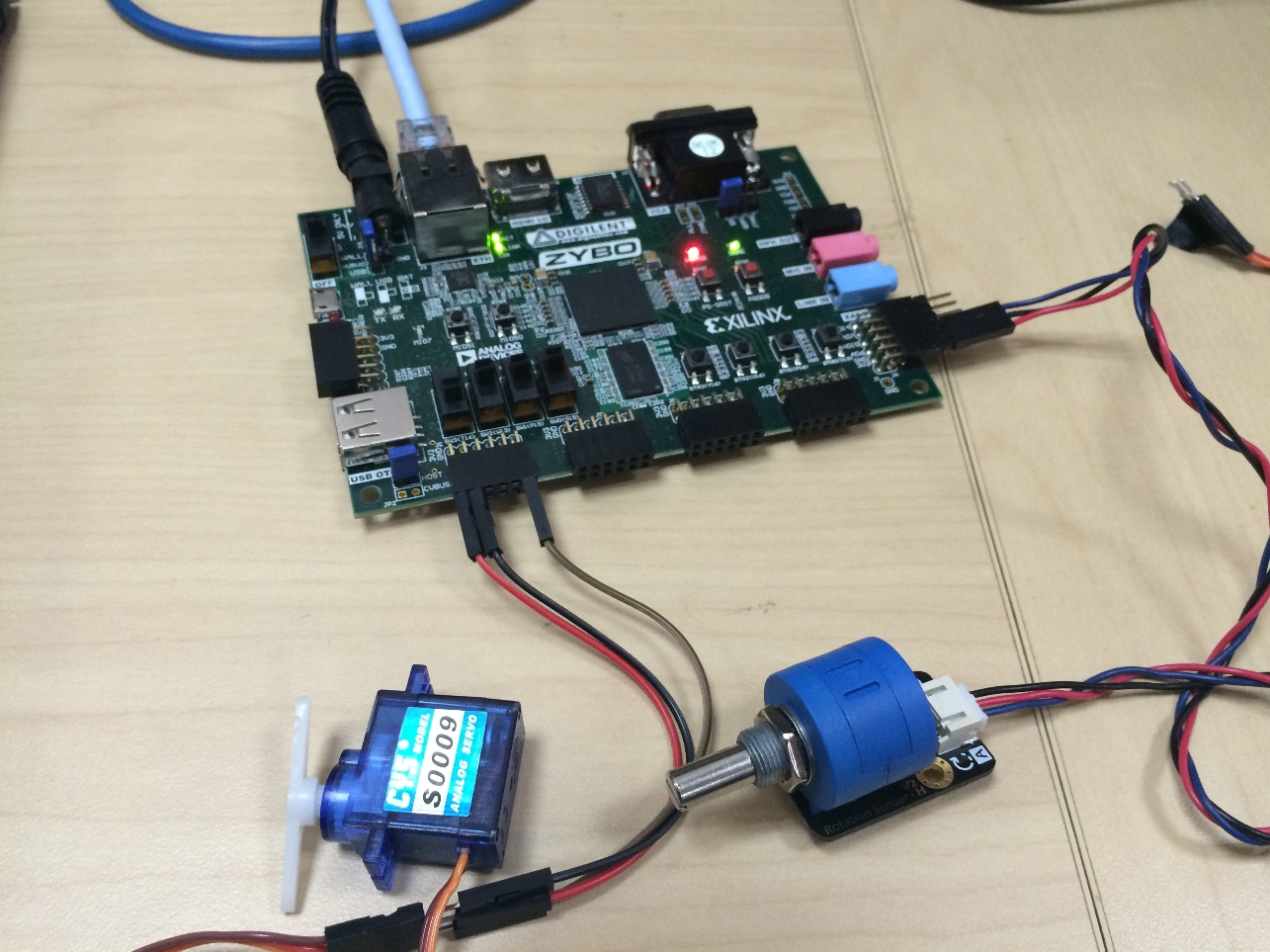
通过我们提供的Makefile make后，你会发现ZYBO板卡上的开关可以控制对应的LED灯的亮灭了，GPIO的API就是这么易用，具体的实验现象可以在github提供的视频中看到。

## 9.2 DEMO2 ADC、PWM、GPIO实验——角度传感器控制舵机旋转

本参考设计需要的外设是：角度传感器、驱动频率为50HZ的舵机，设计源码可以在XUP github中下载到。用户需注意与xilb库的相对路径。

在这个参考设计中，我们主要使用ADC和PWM还有GPIO的API，最终我们将要实现的是通过角度传感器开控制舵机相应的角度并且LED灯指示角度的大小。

硬件结构如下图：

源码如下：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <sys/time.h>

#include "../xlib/xil\_pwm.h"

#include "xil\_gpio.h"

#define IIO\_XADC\_PATH "/sys/bus/iio/devices/iio:device0/%s"

#define XSM\_SEQ\_CH\_AUX06 "in\_voltage8\_raw"

#define XSM\_SEQ\_CH\_AUX07 "in\_voltage9\_raw"

#define XSM\_SEQ\_CH\_AUX14 "in\_voltage10\_raw"

#define XSM\_SEQ\_CH\_AUX15 "in\_voltage11\_raw"

#define XADC\_SCALE 4095.0

#define PWM\_BASE\_ADDR 0x43C00000

#define GPIO\_BASE\_ADDR 0x41200000

int read\_sysfs\_temp(float \*pValue)

{

int i;

float value[3];

char filepath[3][200];

sprintf(filepath[1], IIO\_XADC\_PATH, "in\_temp0\_offset");

sprintf(filepath[0], IIO\_XADC\_PATH, "in\_temp0\_raw");

sprintf(filepath[2], IIO\_XADC\_PATH, "in\_temp0\_scale");

for(i=0;i<3;i++)

{

FILE \*fd = fopen(filepath[i], "r");

if(!fd)

{

printf("not found %s!\n", filepath[i]);

return -1;

}

fscanf(fd, "%f", &value[i]);

fclose(fd);

}

\*pValue = (value[0] - value[1]) / value[2];

return 0;

}

int read\_sysfs\_u32(char \*filename, int \*pValue)

{

char filepath[200];

sprintf(filepath, IIO\_XADC\_PATH, filename);

FILE \*fd = fopen(filepath, "r");

if(!fd)

{

printf("not found %s!\n", filepath);

return -1;

}

fscanf(fd, "%d", pValue);

fclose(fd);

return 0;

}

int main()

{

int value;

int i = 0;

float temp;

xil\_pwm\* mypwm = XilPWMCreate(PWM\_BASE\_ADDR);

mypwm->setPeriod(mypwm, 0, 2000000);

xil\_gpio \*my\_gpio = XilGpioCreate(GPIO\_BASE\_ADDR);

my\_gpio->modeChipWrite(my\_gpio,GPIO\_CH1,0x00);

while(1)

{

if(read\_sysfs\_u32(XSM\_SEQ\_CH\_AUX06, &value) == 0)

{

printf("AUX06 voltage is %.3f\n", value / XADC\_SCALE);

}

if(i == 20)

{

i = 0;

my\_gpio->digitalChipWrite(my\_gpio,GPIO\_CH1,(value / XADC\_SCALE)\*36);

}

mypwm->setDuty(mypwm, 0,(value / XADC\_SCALE)\*30);

usleep(2000); i ++;

}

return 0;

}

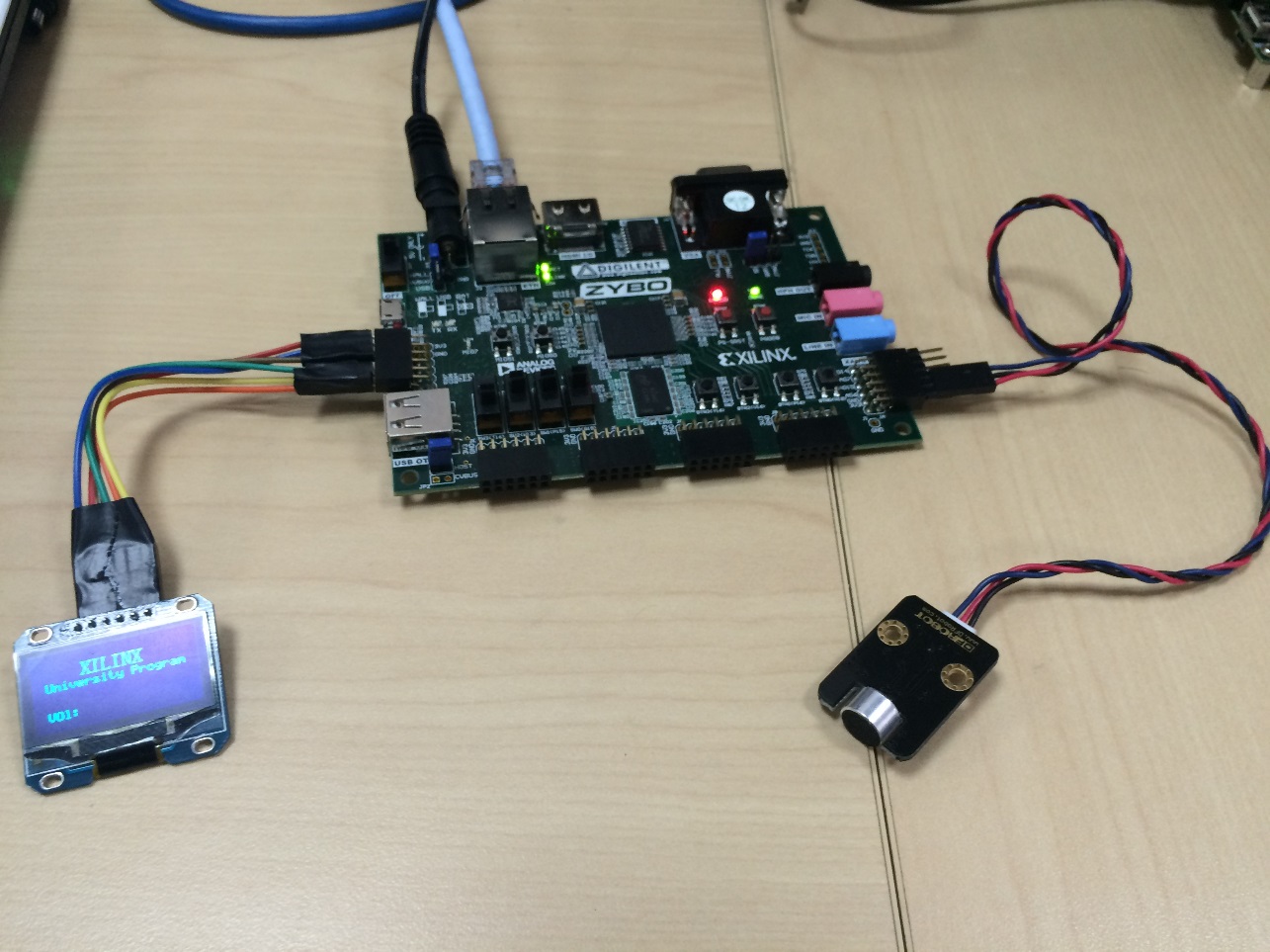
可以看出，只是外设驱动代码的简单拼接，我们就可以完成这个DEMO，最终编译后，可以达到实验的要求。实验现象请见视频。

## 9.3 DEMO3 ADC、SPI 实验——oled显示声音强度

本设计需要的外设有：声音传感器、SPI协议的oled液晶屏。主要使用了xilb中提供的oled和ADC的API。设计时可参考各个外设测试的代码，之后进行简单“组装”，并加入用户算法即可完成设计。

设计源码可以在XUP github中下载到。用户需注意与xilb库的相对路径。

硬件结构如下图：



工程源码如下：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <sys/time.h>

#include "../xlib/xil\_pwm.h"

#include "xil\_gpio.h"

#include "../oled/xil\_oled.h"

#define IIO\_XADC\_PATH "/sys/bus/iio/devices/iio:device0/%s"

#define XSM\_SEQ\_CH\_AUX06 "in\_voltage8\_raw"

#define XSM\_SEQ\_CH\_AUX07 "in\_voltage9\_raw"

#define XSM\_SEQ\_CH\_AUX14 "in\_voltage10\_raw"

#define XSM\_SEQ\_CH\_AUX15 "in\_voltage11\_raw"

#define XADC\_SCALE 4095.0

#define PWM\_BASE\_ADDR 0x43C00000

#define GPIO\_BASE\_ADDR 0x41200000

int read\_sysfs\_temp(float \*pValue)

{

int i;

float value[3];

char filepath[3][200];

sprintf(filepath[1], IIO\_XADC\_PATH, "in\_temp0\_offset");

sprintf(filepath[0], IIO\_XADC\_PATH, "in\_temp0\_raw");

sprintf(filepath[2], IIO\_XADC\_PATH, "in\_temp0\_scale");

for(i=0;i<3;i++)

{

FILE \*fd = fopen(filepath[i], "r");

if(!fd)

{

printf("not found %s!\n", filepath[i]);

return -1;

}

fscanf(fd, "%f", &value[i]);

fclose(fd);

}

\*pValue = (value[0] - value[1]) / value[2];

return 0;

}

int read\_sysfs\_u32(char \*filename, int \*pValue)

{

char filepath[200];

sprintf(filepath, IIO\_XADC\_PATH, filename);

FILE \*fd = fopen(filepath, "r");

if(!fd)

{

printf("not found %s!\n", filepath);

return -1;

}

fscanf(fd, "%d", pValue);

fclose(fd);

return 0;

}

int main()

{

int value;

int i = 0;

float temp;

LED\_Init();

LED\_P8x16Str(40, 1, "XILINX");

LED\_P6x8Str(12, 3, "University Program");

LED\_P6x8Str(12, 6, "VOl:");

xil\_gpio \*my\_gpio = XilGpioCreate(GPIO\_BASE\_ADDR);

my\_gpio->modeChipWrite(my\_gpio,GPIO\_CH1,0x00);

while(1)

{

if(read\_sysfs\_u32(XSM\_SEQ\_CH\_AUX06, &value) == 0)

{

printf("AUX06 voltage is %.3f\n", value / XADC\_SCALE);

}

LED\_Set\_Pos(40, 6);

for (i = 0; i < (int) value / 37; i++)

{

LED\_WrDat(0xFF);

}

for ( ; i < 90; i++)

{

LED\_WrDat(0x00);

}

usleep(20000);

}

return 0;

}

Make之后，我们对着声音传感器说话，在oled的进度条上就可以看到声音的强度了。

# 附录

## 附录A Linux API文档

### GPIO API

初始化GPIO：

xil\_gpio \*XilGpioCreate(u32 BaseAddress)

BaseAddress：传入GPIO的地址；

返回值：返回ＧＰＩＯ的指针。

设置单个引脚输入输出状态：

void (\*modeWrite)(xil\_gpio \* const me, u8 ch, u8 pin, boolean value)

me：传入GPIO指针；

ch：GPIO通道；

pin：GPIO引脚；

value：GPIO状态设定值，输入输出分别为XIL\_INPUT和XIL\_OUTPUT。

读取单个GPIO输入输出状态：

boolean (\*modeRead)(xil\_gpio \* const me, u8 ch, u8 pin)

me：传入GPIO指针；

ch：GPIO通道；

pin：GPIO引脚；

返回值：当前引脚的输入输出状态。

设置单个GPIO引脚的值：

void (\*digitalWrite)(xil\_gpio \* const me, u8 ch, u8 pin, boolean value)

me：传入GPIO指针；

ch：GPIO通道；

pin：GPIO引脚；

value：GPIO设定值，分别为0和1。

读取单个GPIO引脚的值：

boolean (\*digitalRead)(xil\_gpio \* const me, u8 ch, u8 pin)

me：传入GPIO指针；

ch：GPIO通道；

pin：GPIO引脚；

返回值：当前引脚的值。

设置整个GPIO通道引脚输入输出状态：

void (\*modeChipWrite)(xil\_gpio \* const me, u8 ch, u32 value)

me：传入GPIO指针；

ch：GPIO通道；

value：GPIO状态设定值，应为32位无符号整型数。

读取整个GPIO通道输入输出状态：

u32 (\*modeChipRead)(xil\_gpio \* const me, u8 ch)

me：传入GPIO指针；

ch：GPIO通道；

返回值：当前通道的输入输出状态。

设置整个GPIO通道的值：

void (\*digitalChipWrite)(xil\_gpio \* const me, u8 ch, u32 value)

me：传入GPIO指针；

ch：GPIO通道；

value：GPIO设定值，应为32位无符号整型数。

读取整个GPIO通道的值：

u32 (\*digitalChipRead)(xil\_gpio \* const me, u8 ch)

me：传入GPIO指针；

ch：GPIO通道；

返回值：当前通道的值。

释放GPIO指针：

void XilGpioDestroy(xil\_gpio \*me)

me：传入需要销毁的GPIO指针。

### PWM API

初始化PWM：

xil\_pwm \*XilPWMCreate(u32 BaseAddress)

BaseAddress：传入PWM的地址；

返回值：返回xil\_pwm类型的结构。

设置PWM占空比

void (\*setDuty)(xil\_pwm\* const me, u8 ch, float value)

me：传入PWM指针；

ch：PWM通道；

value：PWM占空比的设定值，用0~100的浮点数表示占空比的百分数。

设置PWM频率

void (\*setPeriod)(xil\_pwm\* const me, u8 ch, u32 value)

me：传入PWM指针；

ch：PWM通道；

value：PWM频率的设定值。

获取PWM占空比

float (\*getDuty)(xil\_pwm\* const me, u8 ch)

me：传入PWM指针；

ch：PWM通道；

返回值：PWM占空比的值，用0~100的浮点数表示占空比的百分数。

获取PWM频率

u32 (\*getPeriod)(xil\_pwm\* const me, u8 ch)

me：传入PWM指针；

ch：PWM通道；

返回值：PWM频率的值。

释放PWM指针：

void XilPWMDestory(xil\_pwm \*me)

me：传入需要销毁的PWM指针。

### UART API

初始化UART：

xil\_uart \*XilUartCreate(void)

返回值：返回xil\_uart类型的结构。

配置UART通道：

int (\*open)(xil\_uart\* const me, xil\_uart\_config xconfig)

me：传入UART指针；

xconfig：传入UART配置信息结构体；

返回值：是否成功配置UART；

关闭UART通道：

int (\*close)(xil\_uart\* const me)

me：传入UART指针；

返回值：是否成功关闭UART；

UART读入n个字节：

int (\*readn)(xil\_uart\* const me, void \*buf, ssize\_t count)

me：传入UART指针；

buf：读取串口信息存放的地址；

count：串口读取的字节数；

UART读入一行：

int (\*readline)(xil\_uart\* const me, void \*buf)

me：传入UART指针；

buf：读取串口信息存放的地址；

向UART写n个字节：

int (\*writen)(xil\_uart\* const me, const void \*buf, ssize\_t count)

me：传入UART指针；

buf：写入串口信息存放的地址；

释放UART指针：

void XilUartDestroy(xil\_uart \* const me)

me：传入需要销毁的UART指针；

### SPI API

初始化SPI：

xil\_spi \*XilSPICreate(xil\_spi\_config config)

config：传入的SPI配置信息；

返回值：返回SPI指针。

设置SPI传输速度：

int (\*setSpeed)(xil\_spi\* const me, u32 speed)

me：传入SPI指针；

speed：SPI传输速度；

返回值：0；

SPI传输单字节：

u8 (\*RWOneByte)(xil\_spi\* const me, u8 buf, u8 cs\_change)

me：传入SPI指针；

buf：传入SPI发送内容；

cs\_change：选择是否片选；

返回值：返回SPI接受的字节；

SPI传输N字节：

int (\*RWNbyte)(xil\_spi\* const me, u8 \*txBuf, u8 \*rxBuf, u32 len, u8 cs\_change)

me：传入SPI指针；

txBuf：发送信息缓存地址；

rxBuf：接收信息缓存地址；

len：发送信息字节数；

cs\_change：选择是否片选；

返回值：传输是否成功；

释放SPI指针：

void XilSPIDestroy(xil\_spi\* me)

me：传入SPI指针；

### IIC API

初始化IIC：

xil\_i2c \* XilI2CCreate(char \*devname, u8 devaddr)

devname：IIC设备名；

devaddr：IIC设备地址；

返回值：返回IIC指针；

读IIC：

int (\*Read)(xil\_i2c\* const me, void \*addr, int addrsize, u8 \*buf, int len)

me：传入IIC指针；

addr：IIC读取地址；

addrsize：IIC读取地址长度；

buf：IIC读取内容指针；

len：IIC读取内容长度；

返回值：读取是否成功；

写IIC：

int (\*Write)(xil\_i2c\* const me, void \*addr, int addrsize, u8 \*buf, int len)

me：传入IIC指针；

addr：IIC写入地址；

addrsize：IIC写入地址长度；

buf：IIC写入内容指针；

len：IIC写入内容长度；

返回值：写入是否成功；

释放IIC：

void XilI2CDestroy(xil\_i2c\* const me)

me：传入IIC指针。

### XADC API

读取单通道ADC的值：

int read\_sysfs\_u32(char \*filename, int \*pValue)

filename：ADC通道名；

pValue：ADC转换结果存放指针；

返回值：读取是否成功。