ZLG 致远电子

微文摘

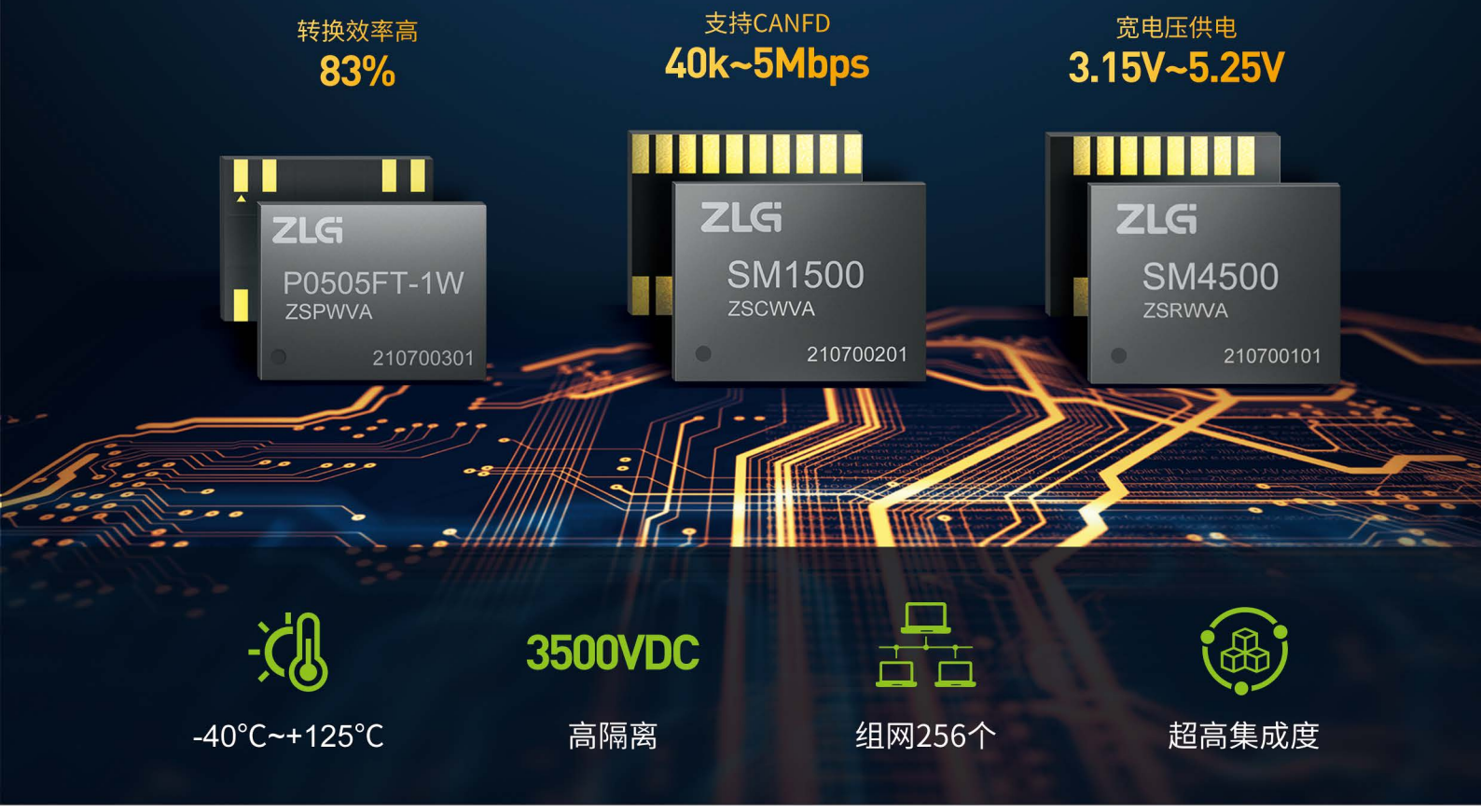
ZLG MICRO DIGEST

2023/1 第1期

月刊

ZLG“芯·艺”之作，开辟总线隔离新格局

全新 SiP 工艺隔离芯片产品族



-40°C~+125°C

3500VDC 高隔离

组网256个

超高集成度

ZLG致远电子超过二十年的电源设计及工艺经验积累，将自主电源IC与成熟SiP工艺结合，推出高集成度总线隔离方案，有“三合一”全隔离芯片以及DC-DC电源隔离芯片不同方案，满足不同客户的需求。

产品名称	产品型号	封装	供电电压	波特率	隔离电压	节点数	环境温度	备注
485隔离芯片	SM4500	DFN16	3.15V~5.25V	10M	3500VDC	256个	-40~125°C	三合一芯片
	SM4510	DFN20	3.15V~5.25V	10M	3500VDC	256个	-40~125°C	
CAN隔离芯片	SM1500	DFN20	4.75V~5.25V	40K-5M	3500VDC	110个	-40~125°C	
DC-DC隔离芯片	P0505FT-1W	DFN14	4.5V~5.5V	--	3000VDC	--	-40~125°C	电源隔离芯片

应用行业

新能源　工业控制　智能制造　煤矿　交通

致远电子官方网站　致远电子官方微信

CONTENTS

目录

【技术分享】Ubuntu上如何使用AWStudio

原创 研发部 ZLG 致远电子 2023-01-30 11:32:58

AWStudio 内的 AWTK Designer 组件是专门用来制作 AWTK 应用程序 UI 界面的实用型工具，只要通过拖拽和点击就可以完成复杂的设计，并且能够随时预览效果图。

AWStudio安装

在 AWStudio 官网下载对应的版本，本文下载 AWStudio Ubuntu 社区版。下载完毕后，在终端切换到 *.deb 文件所在的路径，执行 sudo apt install ./*.deb。安装完毕后，在开始菜单查看是否有 AWStudio 与 AWTK Designer 两个软件，有即安装完毕。如图 1 所示：

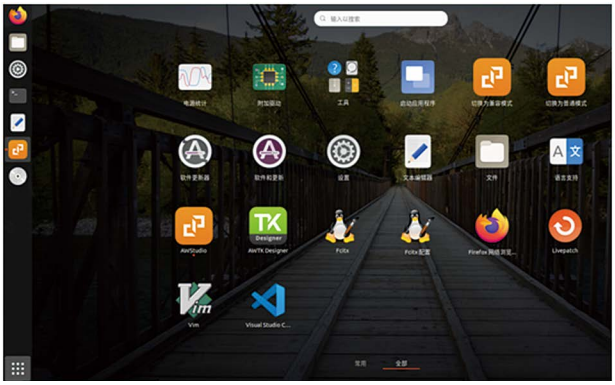


图1 安装完成界面

AWStudio 下载地址为：https://awtk.zlg.cn/awstudio/download.html

VS Code调试环境搭建

我们自己搭建的项目有时候需要调试，可以使用 VS Code，如不需要可以跳过本节。

网上下载 VS Code 并且安装。安装完毕后下载安装 C/C++ 的插件，如图 2 所示：

图2 安装C/C++插件

没有网络的用户可以选择 VSIX 安装插件。

按 ctrl+shift+p 搜索，如图 3 内容所示。

图3 添加配置

在 .json 文件中的 includePath 添加 AWTK 源代码的路径。图 4 可作为参考。

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
                "${workspaceFolder}/**",
                "/usr/share/AWStudio/AWTK/SDK/awtk/src"
            ],
            "defines": [],
            "compilerPath": "/usr/bin/gcc",
            "cStandard": "gnu17",
            "cppStandard": "gnu++14",
            "intelliSenseMode": "linux-gcc-x64"
        }
    ],
    "version": 4
```

图4 json文件添加内容

选择一个 .c 文件，然后点击 VS Code 左边工具栏的“运行和调试”，点击创建 launch.json 文件，点击右下角的“添加配置”按钮，选择 gdb 启动，修改 .json 文件中 program 与 cwd 的路径，具体内容参考图 5。

```
    "configurations": [
        {
            "name": "(gdb) 启动",
            "type": "cppdbg",
            "request": "launch",
            "program": "/home/zlgmcu/AWStudioProjects/Workspace/AwtkApplication/bin/demo
            "args": [],
            "stopAtEntry": false,
            "cwd": "/home/zlgmcu/AWStudioProjects/Workspace/AwtkApplication/bin",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "为 gdb 启用整齐打印",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "将反汇编风格设置为 Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
```

图5 launch.json文件内容

保存文件后，即可添加断点按 F5 调试了。

开发第一个工程

1. 新建工程

打开 AWStudio，点击新建工作区。用户自己设置好自己的名称与路径。完成后新建项目，选择 AWTK Application，修改项目名称。如图 6 所示：

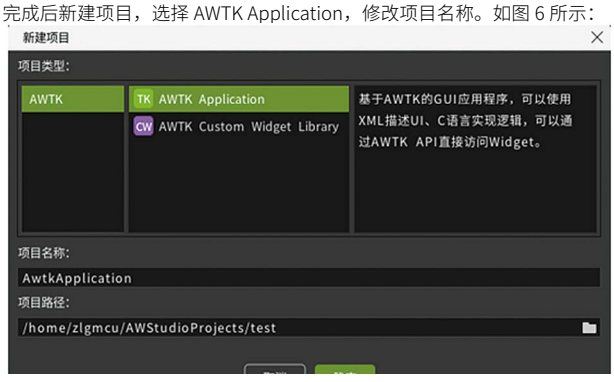


图6 新建项目

右击项目，选择打开，工程便会打开 AWTK Designer。左边有控件列表，下面的代码部分是当前窗口的应用代码，右边是控件对象的属性等（目前没有加入控件所以为空）。如图 7 所示：

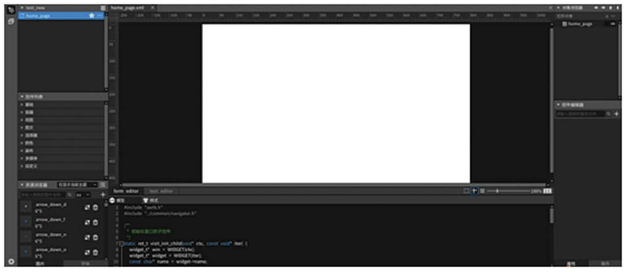


图7 AWTK Designer界面

2. 加入控件

从 AWTK Designer 控件列表拖拽一个按钮到窗口。点击按钮右边便可以修改其属性，例如名称或者大小等。我们在拖拽进一个进度条，进度条可以设置初始值等属性。

3. 配置事件

事件一般用来响应各种行为。大多数事件是作为用户行为的响应而产生的。接下来配置我们的第一个事件。

点击按钮，选择事件，点击右边的 + 号，事件选择 click，就是按钮按下事件。这时，下面的代码区便自动生成了 click 的处理函数，函数功能需要我们补充，代码可参考图 8：

```
static ret_t on_button_click(void* ctx, event_t* e) {
  // TODO: 在此添加控件事件处理程序代码
  widget_t* win = WIDGET(ctx);
  widget_t* progress_bar = widget_lookup(win, "progress_bar", TRUE);
  int8_t value = (PROGRESS_BAR(progress_bar)->value + 5);
  value = value % 100;
  progress_bar_set_value(progress_bar, value);
  return RET_OK;
}
```

图8 按钮处理事件

处理函数的功能是每按下一次，进度条 +5，到 100 重新开始计数。

4. 设置定时函数

定时器可为用户提供一些定时操作。

我们再拖拽进一个进度条，然后在 home_page_init(widget_t* win, void* ctx) 函数添加定时器，如图 9 所示：

```
ret_t home_page_init(widget_t* win, void* ctx) {
  (void)ctx;
  return_value_if_fail(win != NULL, RET_BAD_PARAMS);

  widget_foreach(win, visit_init_child, win);
  widget_t* progress_bar = widget_lookup(win, "progress_bar_time", TRUE);
  timer_add(on_timer, progress_bar, 10);

  return RET_OK;
}
```

图9 添加定时器

其中 progress_bar_time 为我们新添加的进度条，将最大值设置为 1000。再添加一下定时器功能函数，如图 10 所示：

```
static ret_t on_timer(const timer_info_t* timer) {
  widget_t* progress_bar = (widget_t*)timer->ctx;
  int32_t value = (PROGRESS_BAR(progress_bar)->value + 5);
  value = (value + 1000) % 1000;
  progress_bar_set_value(progress_bar, value);
  return RET_REPEAT;
}
```

图10 定时器功能

编译模拟运行后，可以看到进度条会自己增加到 1000，然后重新开始增加。

5. 增加窗体

在左上角的“窗口编辑”界面中，点击“新建窗体”图标，可选择“新建窗体”窗口，如图 11 所示：

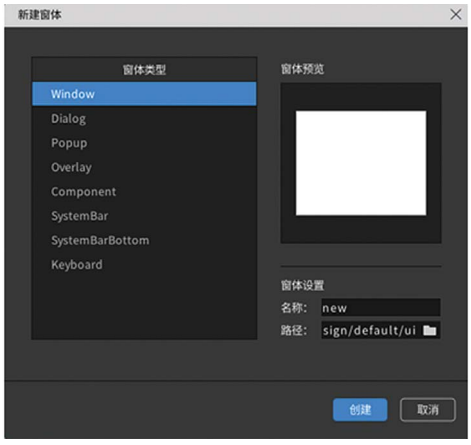


图11 新建窗体

我们在原来的窗口再拖拽一个按钮，并且设置好他的触发事件为 cilck。在事件函数中，添加函数 navigator_to("new")，这样按钮按下就会打开我们新建的新窗体。

在新窗体中，拖拽进一个按钮，设置它的事件为 cilck。在事件函数中添加如下功能：

widget_t* win = WIDGET(ctx);

window_close(win);

即按下按钮“是”，会关闭我们的窗体。

6. 编译，模拟运行

保存我们的工程，点击编译，编译完成后点击模拟运行，就可以看到我们自己搭建的 AWTK 界面了，如图 12 所示：

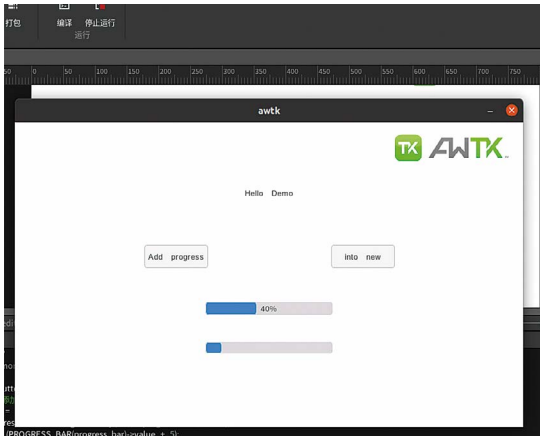


图12 模拟运行

【产品应用】AWorksLP样例详解(MR6450)—PWM(单通道)

原创 研发部 ZLG 致远电子 2023-01-03 11:33:01

AWorksLP 对外设进行了高度抽象化，为同一类外设提供了相同的接口，应用程序可以轻松跨平台。本文以 MR6450（点击了解详情）平台为例，介绍 AWorksLP PWM 外设基本用法。

简介

脉冲宽度调制 (PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。以下简述几个关键的概念：

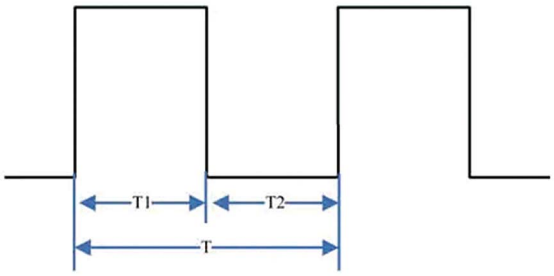


图1

- PWM 周期：指信号从高电平到低电平再回到高电平的时间，如上图 T 所示。
- PWM 频率一秒内 PWM 周期的次数。
- 占空比：一个周期内高电平持续时间所占的比例即（T1 / T）。
- 脉宽时间：高电平时间。

接口介绍

函数原型	简要描述
aw_err_t aw_pwm_enable (int fd);	使能 PWM 设备输出
aw_err_t aw_pwm_disable (int fd);	PWM 设备输出停止
aw_err_t aw_pwm_accurate_output (int fd, uint32_t period_num);	精确输出 period_num 个周期 PWM 波
aw_err_t aw_pwm_config_set (int fd, aw_const aw_pwm_config_t *p_config);	设置 PWM 设备参数配置
aw_err_t aw_pwm_config_get (int fd, aw_pwm_config_t *p_config);	获取 PWM 设备的配置参数
aw_err_t aw_pwm_config_frac_set (int fd, aw_const aw_pwm_config_frac_t *p_config);	以更加精确的形式设置 PWM 设备的参数
aw_err_t aw_pwm_config_frac_get (int fd, aw_pwm_config_frac_t *p_config);	获取 PWM 设备的分数形式配置参数

函数列表：

下表为 PWM 接口相关结构体类型。

结构体类型表：

PWM 配置信息说明：

类型	简要描述
aw_pwm_config_t	PWM 配置参数结构体
aw_pwm_config_frac_t	PWM 配置参数结构体(分数形式)，用分数表示，更精确

1. aw_pwm_config_t

```
typedef struct {
  uint32_t duty_ns;
  uint32_t period_ns;
  uint32_t is_inverse;
} aw_pwm_config_t;
```

PWM 配置参数结构体。

成员详解：

- duty_ns：pwm 周期中高电平的有效时间 ns 为单位。
- period_ns：pwm 周期 ns 为单位。
- is_inverse：输出波形是否反相，0 表示不反相。

2. aw_pwm_config_frac

```
typedef struct aw_pwm_config_frac {

  uint32_t duty_numerator;
  uint32_t duty_denominator;
  uint32_t period_numerator;
  uint32_t period_denominator;
  uint32_t is_inverse;
} aw_pwm_config_frac_t;
```

PWM 配置参数结构体 (分数形式)，用分数表示，更精确。

成员详解：

- duty_numerator：PWM 周期中高电平的有效时间分子部分。
- duty_denominator：PWM 周期中高电平的有效时间分母部分。
- period_numerator：PWM 周期分子部分。
- period_denominator：PWM 周期分母部分。
- is_inverse：PWM 输出波形输出是否反向，0: 不反向，1：反向。

使用样例

AWorksLP SDK 相关使用请参考《AWorksLP SDK 快速入门（MR6450）——开箱体验》一文，本文不在赘述。

1. PWM单通道输出功能

{SDK}\demos\peripheral\pwm 路径下为 PWM 例程，例程关键代码如下：

```
/**
 * \brief PWM 演示例程入口
 * \return 无
 */
aw_local void* __task_handle (void *arg)
{
  uint32_t  period1 = 2000000;   /* (ns) */
  uint32_t  period2 = 1000000;   /* (ns) */
  int       fd;
  int       ret;
  aw_pwm_config_t      pwm_config;
```

```
    aw_kprintf("\nPWM demo testing...\n");
    fd = aw_open(CONFIG_DEMO_PWM_DEVICE_NAME, AW_O_RDWR,
0);
    if(fd < 0){
      aw_kprintf("pwm open failed \r\n");
      aw_close(fd);
      return 0;
    }
    ret = aw_pwm_config_get(fd, &pwm_config);

    /* period 配置不可以为 0
     * duty 配置为 0，这时可以配置成功：输出一直为低
      * duty 配置为 peroid，占空比为 100%，也可配置成功：输出一直
为高
     * PWM 正在进行输出，不可配置 */
    pwm_config.duty_ns = period1 / 2;
    pwm_config.is_inverse = 0;
    pwm_config.period_ns = period1;
    aw_pwm_config_set(fd, &pwm_config);
    while(1) {

      /* 配置 PWM 的有效时间（高电平时间）50% , 周期 period1*/
      aw_pwm_config_set(fd, &pwm_config);
      aw_pwm_enable(fd);    /* 使能通道 */
      aw_mdelay(250);
      aw_pwm_disable(fd);   /* 禁能通道 */
      aw_mdelay(250);

      /* 输出五个周期 pwm 波 */
      aw_pwm_accurate_output(fd, 5);

      /* 配置 PWM 的有效时间（高电平时间）2% , 周期 period1*/
      pwm_config.duty_ns =  period1 / 50;
      aw_pwm_config_set(fd, &pwm_config);
      aw_pwm_enable(fd);    /* 使能通道 */
      aw_mdelay(250);
      aw_pwm_disable(fd);   /* 禁能通道 */
      aw_mdelay(250);

      pwm_config.duty_ns = period2 / 2;
      pwm_config.period_ns = period2;
      /* 配置 PWM 的有效时间（高电平时间）50% , 周期 period2*/
      aw_pwm_config_set(fd, &pwm_config);
      aw_pwm_enable(fd);    /* 使能通道 */
      aw_mdelay(250);
      aw_pwm_disable(fd);   /* 禁能通道 */
      aw_mdelay(250);


      /* 配置 PWM 的有效时间（高电平时间）2% , 周期 period2*/
      pwm_config.duty_ns = period2 / 50;
      aw_pwm_config_set(fd, &pwm_config);
      aw_pwm_enable(fd);    /* 使能通道 */
      aw_mdelay(250);
      aw_pwm_disable(fd);   /* 禁能通道 */
      aw_mdelay(250);

      pwm_config.duty_ns = period1 / 2;
      pwm_config.period_ns = period1;
    }

    return 0;
}
```

例程默认使用 pwm3_chan4 对应开发板的位置如图 2 所示：

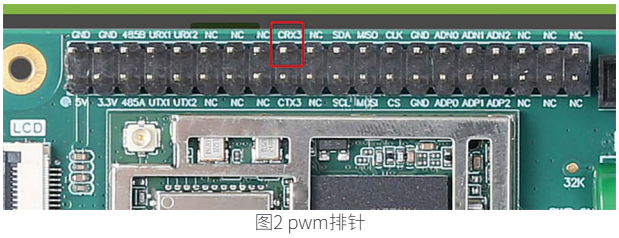


图2 pwm排针

上述代码中使用 aw_pwm_config_get 接口获取 PWM 当前的配置信息，PWM 周期中高电平的有效时间为 1000000ns，PWM 周期为 2000000ns，也就是设置 PWM 的占空比为 50%。使用 aw_pwm_config_set 接口设置 PWM。使用 aw_pwm_enable 接口使能 PWM，使用 aw_pwm_disable 接口关闭 PWM，使用 aw_pwm_accurate_output 接口输出任意个 PWM 波。

在 while 循环中每间隔一段时间通过设置 PWM 的参数，从而输出各种 PWM 的波形，下图为例程中输出的各种 PWM 波形。

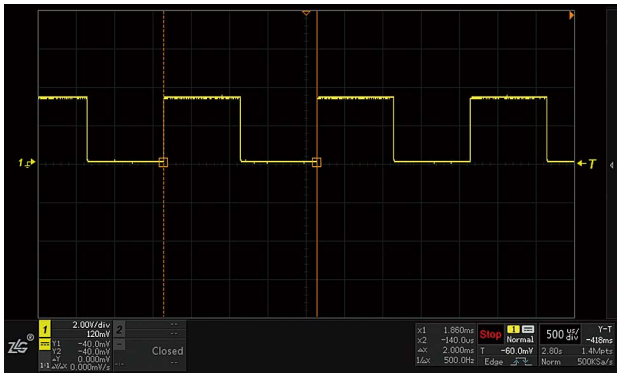


图3 占空比50%，周期2ms

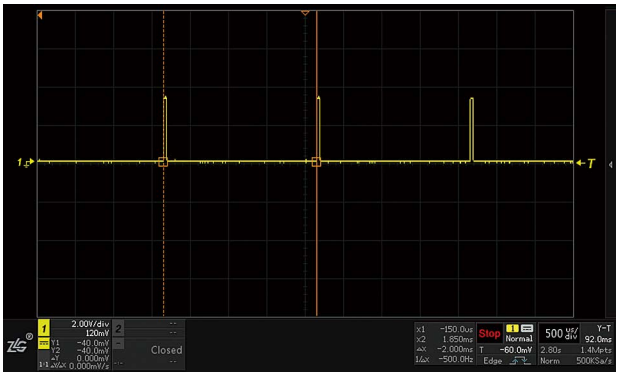


图4 占空比2%，周期2ms

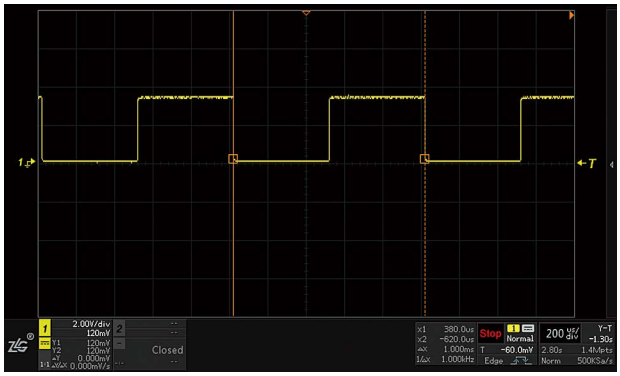


图5 占空比50%，周期1ms

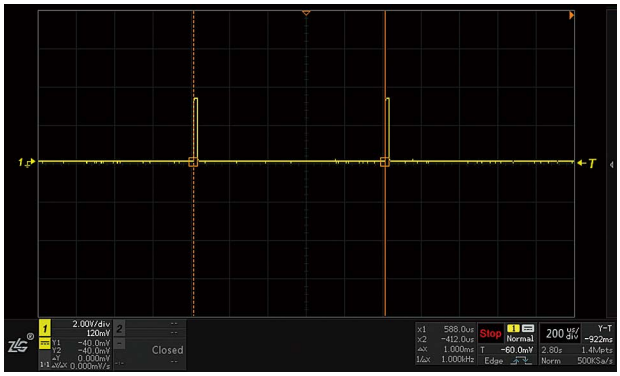


图6 占空比2%，周期1ms

2. 蜂鸣器输出

{SDK}\demos\peripheral\buzzer 路径下为蜂鸣器例程，例程关键代码如下：

```
/**
 * \brief 建立蜂鸣器例程入口
 * \return 无
 */
aw_local void* __task_handle (void *arg)
{
  int         fd;
  fd = aw_open("/dev/Buzzer", AW_O_RDWR, 0);
  if(fd < 0){
    aw_kprintf("Buzzer open failed \r\n");
    aw_close(fd);
    return 0;
  }

  while(1) {
    /* 强度调节设备驱动无源蜂鸣器 */
    aw_buzzer_loud_set(fd, 80); /* 设置蜂鸣器鸣叫强度 */
    aw_buzzer_beep(fd, 500);   /* 启动蜂鸣器延时 500ms */

    /* GPIO 驱动直流蜂鸣器 */
    aw_buzzer_loud_set(fd, 50); /* 设置蜂鸣器鸣叫强度 */
    aw_buzzer_on(fd);      /* 启动蜂鸣器 */
    aw_mdelay(500);       /* 延时 500ms */
    aw_buzzer_off(fd);     /* 关闭蜂鸣器 */
    aw_mdelay(500);       /* 延时 500ms */
  }
  aw_close(fd);
  return 0;
}
```

蜂鸣器在开发板上的位置如图 7 所示。

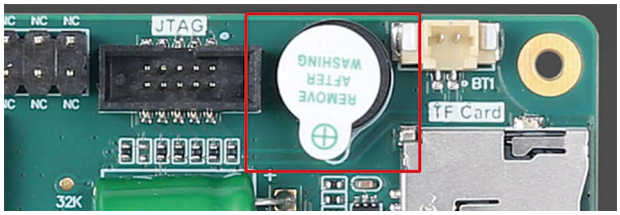


图7 蜂鸣器位置

蜂鸣器引脚所对应的 PWM 通道为 pwm3_chan1，如下图所示。

图8 蜂鸣器对应引脚

```
23  };
24
25  &pwm3_chan1 {
26      pins = <&pin1 PIN_PE07 IOC_PE07_FUNC_CTL_PWM3_P
27  };
28
29  &pwm3_chan4 {
30      pins = <&pin1 PIN_PE05 IOC_PE05_FUNC_CTL_PWM3_P
31  };
32
33  &camera0 {
34      pins = <&pin1 PIN PA07 IOC PA07 FUNC CTL CAM0 D
```

图9 PWM对应引脚

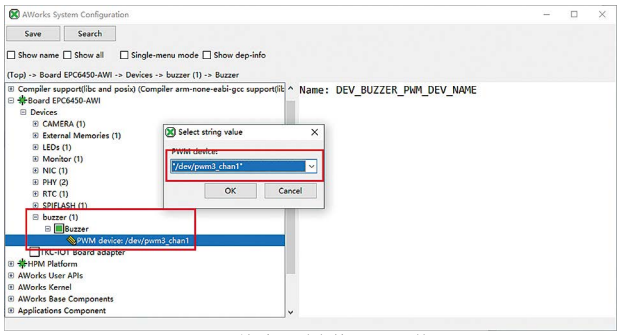


图10 蜂鸣器对应的PWM通道

上述代码中在 while 循中环使用 aw_buzzer_loud_set 接口设置蜂鸣器的鸣叫程度为 80，相当于调节 PWM 的占空比。然后使用 aw_buzzer_beep 接口使蜂鸣器鸣叫一段时间后关闭。再设置蜂鸣器的鸣叫程度为 50，使用 aw_buzzer_on 接口启动蜂鸣器，相当于使能 PWM，延时一段时间后用 aw_buzzer_off接口关闭蜂鸣器，相当于关闭 PWM，然后再延时一段时间。

实验现象为蜂鸣器先以较大的声音鸣叫一段时间后以较小的声音鸣叫一段时间后停止鸣叫，持续一段时间后再次循环。

在 PWM 例程中，PWM 作为设备资源被 Buzzer 引用，可在配置界面中查看所有可引用资源，也可以在界面中查看当前平台所有 PWM 资源，以便在软件设计过程中查看修改。

本文以 PWM 外设通用接口为例，演示了单通道的输出以及接口特性，同时与蜂鸣器进行设备绑定，后续将会更详细的介绍多通道的输出以及相关特性，请持续关注后续推文 ~

【产品应用】
AWorksLP 样例详解(MR6450)— HWTimer

原创 研发部 ZLG 致远电子 2023-01-30 11:32:58

AWorksLP 对外设进行了高度抽象化，为同一类外设提供了相同的接口，应用程序可以轻松跨平台。本文以 MR6450（点击了解详情）平台为例，介绍 AWorksLP HWTimer 外设基本用法。

简介

在 AWorksLP 中将硬件定时器分为了 4 类，即延时型、计数型、周期型和输入捕获型硬件定时器。

- 延时型硬件定时器：

由硬件定时器外设提供的延时功能。

- 计数型硬件定时器：

提供较精确的类似时间戳的功能。

- 周期型硬件定时器：

可设置中断频率的计数器，不仅能提供计数器的功能，也能根据中断频率提供更精确的定时。

- 输入捕获定时器：

可测量脉冲宽度或者测量频率。

接口介绍

延时型硬件定时器：

函数原型	简要描述
aw_err_t aw_hwtimer_delay (int fd, struct aw_timespec *p_tv);	延时
aw_err_t aw_hwtimer_delay_cancel (int fd);	取消延时

计数型硬件定时器：

函数原型	简要描述
aw_err_t aw_hwtimer_count_start (int fd);	启动一个计数型硬件定时器
aw_err_t aw_hwtimer_count_stop (int fd);	停止一个计数型硬件定时器
aw_err_t aw_hwtimer_count_get (int fd, uint64_t *p_count);	读取计数值
aw_err_t aw_hwtimer_count_rate_get (int fd, uint32_t *p_rate);	获取计数时钟频率
aw_err_t aw_hwtimer_count_rate_set (int fd, uint32_t rate);	设置计数时钟频率
aw_err_t aw_hwtimer_count_rate_set_accurate (int fd, uint32_t rate_numerator, uint32_t rate_denominator);	以精确化的方式设置计数时钟频率
aw_err_t aw_hwtimer_count_rate_get_accurate (int fd, uint32_t *p_rate_numerator, uint32_t *p_rate_denominator);	获取计数时钟频率的精确描述

周期型硬件定时器：

函数原型	简要描述
aw_err_t aw_hwtimer_period_wait (int fd, uint32_t wait_ms);	等待定时器周期中断
aw_err_t aw_hwtimer_period_intr (int fd);	打断周期型定时器的等待操作
aw_err_t aw_hwtimer_period_start (int fd);	启动定时器
aw_err_t aw_hwtimer_period_stop (int fd);	停止定时器
aw_err_t aw_hwtimer_period_count_get (int fd, uint64_t *p_count);	读取计数值
aw_err_t aw_hwtimer_period_count_freq_get (int fd, uint32_t *p_rate);	获取周期型定时器的硬件计数频率（不是中断频率）
aw_err_t aw_hwtimer_period_count_freq_get_frac (int fd, aw_hwtimer_rate_t *p_rate);	以更精确的分数形式获取周期型定时器的硬件计数频率（不是中断频率）
aw_err_t aw_hwtimer_period_intr_freq_set (int fd, uint32_t intr_freq);	设置中断频率
aw_err_t aw_hwtimer_period_intr_freq_get (int fd, uint32_t *p_intr_freq);	获取中断频率
aw_err_t aw_hwtimer_period_intr_freq_set_frac (int fd, aw_const aw_hwtimer_rate_t *p_intr_freq);	设置中断频率(以更精确的分数形式)
aw_err_t aw_hwtimer_period_intr_freq_get_frac (int fd, aw_hwtimer_rate_t *p_intr_freq);	获取中断频率(以更精确的分数形式)

输入捕获型硬件定时器：

函数原型	简要描述
aw_err_t aw_hwtimer_cap_start (int fd);	启动输入捕获型硬件定时器
aw_err_t aw_hwtimer_cap_stop (int fd);	停止输入捕获型硬件定时器
aw_err_t aw_hwtimer_cap_read (int fd, uint64_t *p_cap_val, uint32_t timeout_ms);	读取一个捕获到的事件的计数值
aw_err_t aw_hwtimer_cap_intr (int fd);	打断阻塞 read 读操作
aw_err_t aw_hwtimer_cap_config_set (int fd, aw_const aw_hwtimer_cap_config_t *p_config);	配置输入捕获型硬件定时器
aw_err_t aw_hwtimer_cap_config_get (int fd, aw_hwtimer_cap_config_t *p_config);	获取输入捕获型硬件定时器的配置

使用样例

AWorksLP SDK 相关使用请参考《AWorksLP SDK 快速入门（MR6450）——开箱体验》一文，本文不在赘述。

1. 周期型定时器

{SDK}\demos\peripheral\hwtimer 路径下为硬件定时器例程，默认运行的是 demo_hwtimer.c 周期型定时器的代码，例程关键代码如下：

```
/**
 * \brief 硬件定时器中断服务函数。
 * \param[in] p_arg : 任务参数
 */
static void mytimer_isr (void *p_arg)
{
  aw_gpio_toggle((int)p_arg);
  aw_kprintf("enter isr \n\r");
}

/**
 * \brief hwtimer 测试函数
 */
aw_local void* __task_handle (void *arg)
{
  int                fd;
  aw_err_t           ret;
  uint32_t           count = 5;
  aw_hwtimer_rate_t      p_intr_freq;

  p_intr_freq.rate_denominator = 5;
  p_intr_freq.rate_numerator = 1;

   fd = aw_open(CONFIG_DEMO_HWTIMER_PEROID_DEV_NAME, AW_O_RDWR, 0);
  if (fd < 0) {
    aw_kprintf("hwtimer open failed:%d \n\r", fd);
    while(1);
  }
```

```
    ret = aw_hwtimer_period_intr_freq_set_frac(fd, &p_intr_freq);
    while (count) {
      aw_hwtimer_period_wait(fd, 500);
      mytimer_isr(arg);
      count --;
    }

    // 配置每秒中断 2 次
    ret = aw_hwtimer_period_intr_freq_set(fd, 2);

    ret = aw_hwtimer_period_start(fd);
    if (ret != AW_OK) {
      aw_kprintf("Timer allocation fail!\n");
    }

    ret = aw_hwtimer_period_wait(fd, AW_WAIT_FOREVER);

    while (1) {
      aw_hwtimer_period_wait(fd, AW_WAIT_FOREVER);
      mytimer_isr(arg);
    }

    for (;;) {
      aw_mdelay(1000);
    }
    aw_close(fd);

    return 0;
  }
```

在代码中先使用了 aw_hwtimer_period_intr_freq_set_frac 接口，以分数的形式设置中断频率，使用 aw_hwtimer_period_start 接口启动定时器。在循环中使用 aw_hwtimer_period_wait 接口阻塞等待中断的产生、中断产生后继续执行 mytimer_isr 函数使 LED 灯状态翻转，由于设置的频率为五分之一，所以 5 秒 LED 灯的状态翻转一次；循环一定次数后用 aw_hwtimer_period_intr_freq_set 接口设置中断频率为 2HZ，循环等待中断、翻转 LED。

实验现象为 LED 灯先以 5s 的频率闪烁，同时串口打印同时信息。闪烁一定次数后以 0.5s 的频率 LED 闪烁，同时串口打印信息。

下表为使用硬件周期型定时器，在中断中进行引脚翻转，通过逻辑分析仪所测量出的实际数据，在使用设计时可作为部分参考依据。

定时时间（s）	实际时间（s）
0.010000000	0.010000115
0.020000000	0.019999940
0.030000000	0.029999980
0.040000000	0.040000035
0.050000000	0.049999830
0.100000000	0.100000075
0.200000000	0.200000020
0.500000000	0.500000070
1.000000000	1.000000760
2.000000000	1.999999340
3.000000000	3.000002760
4.000000000	4.000001980
5.000000000	5.000004310
10.000000000	10.000008300

2. 计数型定时器

在 config 配置脚本中选择 hwtimer count 计数型定时器测试如图 1 所示。

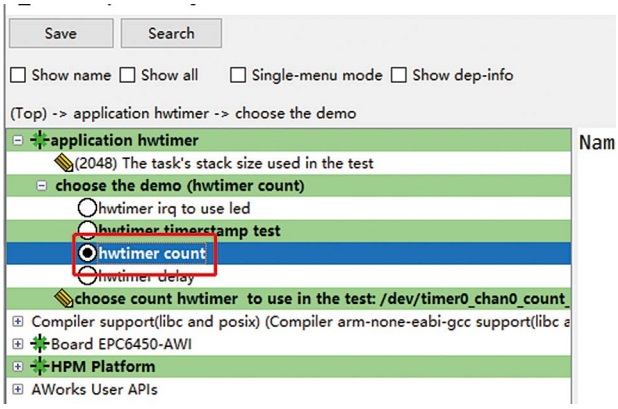


图1 计数型定时器例程

保存后重新 Build 工程，编译好后运行的是 demo_hwtimer_count.c 的代码，例程关键代码如下：

```
  aw_local void* __task_handle (void *arg)
  {
    uint32_t    count = 0;
    int         fd, led_fd;
    int         ret;
    uint32_t    start_count;

     fd = aw_open(CONFIG_DEMO_HWTIMER_PEROID_DEV_NAME, AW_O_RDWR, 0);
    if (fd < 0) {
      aw_kprintf("hwtimer open fail! :%d\n",fd);
      return;
    }
    /* 打开设备会点亮 LED */
    led_fd = aw_open("/dev/led_run", AW_O_RDWR, 0);
    if (led_fd < 0) {
      aw_kprintf("led open fail! :%d\n", led_fd);
      aw_close(fd);
      return;
    }
    ret = aw_hwtimer_count_rate_get(fd, &start_count);
    if (ret != AW_OK) {
      aw_kprintf("Timer count rate get fail!\n");
      aw_close(fd);
      aw_close(led_fd);
      return;
    }
    // 设置时钟频率
    ret = aw_hwtimer_count_rate_set(fd, start_count/2);
    if (ret != AW_OK) {
      aw_kprintf("Timer count rate set fail!\n");
      aw_close(fd);
      aw_close(led_fd);
      return;
    }
```

```
    ret = aw_hwtimer_count_start(fd);
    if (ret != AW_OK) {
      aw_kprintf("Timer start fail!\n");
      aw_close(fd);
      aw_close(led_fd);
      return;
    }
    for (;;) {
      aw_led_toggle(led_fd);
      aw_mdelay(500);
      aw_led_toggle(led_fd);
      aw_hwtimer_count_get(fd, &count);
      aw_kprintf("Count is %d\r\n", count);
    }
    aw_close(fd);
    aw_close(led_fd);
    return 0;
  }
```

在上述代码中使用了 aw_hwtimer_count_rate_get 接口获取改定时器时钟频率，可以在调试模式下查看获取到的参数，为 100M 如图 2 所示。

```
    return;
  }
  ret = aw_hwtimer_count_rate_get(fd,&start_count);
  if (ret != AW_OK) {
    aw_kprintf("Timer count rate get
    aw_close(fd);
    aw_close(led_fd);
    return;
  }
  // 配置每秒中断2次
```

Expression	Type	Value
start_count	uint32_t	100000000

图2 查看参数

使用 aw_hwtimer_count_rate_set 接口设置定时器时钟的频率为 50M，使用 aw_hwtimer_count_start 接口开启定时器，使用 aw_hwtimer_count_get 接口在循环中每延时 500ms 获取一次计数值，并在串口中打印，打印结果如图 3 所示。

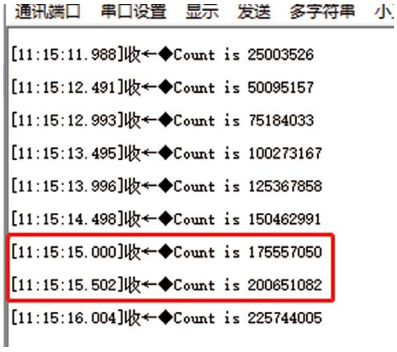


图3 串口打印计数值

打印出的计数值中，相邻两个计数值之差为 25M，是由于设置定时器频率为 50M，每延时 500ms 计数值增加 25M。

3. 延时型定时器

在 config 配置脚本中选择 hwtimer delay 延时型定时器测试如图 4 所示。

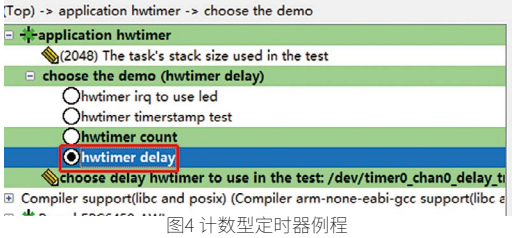


图4 计数型定时器例程

保存后重新 Build 工程，编译好后运行的是 demo_hwtimer_count.c 的代码，例程关键代码如下：

```
  aw_local void* __task_handle (void *arg)
  {
    int           i;
    int           fd;
    aw_err_t      ret;
    aw_timespec_t    timespec;
    aw_timestamp_t   start_timestamp, stop_timestamp;
    aw_timestamp_freq_t timestamp_freq;
    uint64_t      delay_ns, diff;
    uint32_t      ns_numerator = 1000000000;

    timestamp_freq = aw_timestamp_freq_get();
    while (0 == (timestamp_freq % 10)) {
      timestamp_freq /= 10;
      ns_numerator /= 10;
    }

     fd = aw_open(CONFIG_DEMO_HWTIMER_DELAY_DEV_NAME, AW_O_RDWR, 0);
    if (fd < 0) {
      aw_kprintf("hwtimer open failed:%d \n\r", fd);
      while(1);
    }

    delay_ns = 2001000;
    for (i = 0; i < 100; i++) {
      timespec.tv_sec = delay_ns / 1000000000u;
      timespec.tv_nsec = (uint32_t)(delay_ns % 1000000000u);

      start_timestamp = aw_timestamp_get();
      ret = aw_hwtimer_delay(fd, &timespec);
      if (ret !=AW_OK) {
        aw_kprintf("hwtimer delay failed:%d \n\r", ret);
      }
      aw_barrier();
      stop_timestamp = aw_timestamp_get();

      stop_timestamp -= start_timestamp;
      diff = stop_timestamp;
      diff *= ns_numerator;
      diff /= timestamp_freq;

      diff = diff - delay_ns;
      aw_kprintf(
          "hwtimer_delay delay = %u,diff = %u ns\n",
          (uint32_t)delay_ns,
          (uint32_t)diff);
      delay_ns += 100000;
    }
    aw_close(fd);
    return 0;
  }
```

上述代码中在延时开始前使用 aw_timestamp_get 接口记录时间戳，使用 aw_hwtimer_delay 接口进行延时，延时结束后记录结束时间戳，用两个时间戳的差值通过换算，用于对比延时不同时间下与 timestamp 相比的误差，并在串口中打印，打印后增加延时时间，再次循环，串口打印结果如下图所示。

```
[14:48:08.982]收←◆hwtimer_delay delay = 2001000, diff = 69340 ns
hwtimer_delay delay = 2101000, diff = 58120 ns
hwtimer_delay delay = 2201000, diff = 48310 ns
hwtimer_delay delay = 2301000, diff = 43870 ns
hwtimer_delay delay = 2401000, diff = 45060 ns
hwtimer_delay delay = 2501000, diff = 41990 ns
hwtimer_delay delay = 2601000, diff = 40430 ns
hwtimer_delay delay = 2701000, diff = 39960 ns
hwtimer_delay delay = 2801000, diff = 40010 ns
hwtimer_delay delay = 2901000, diff = 39250 ns
hwtimer_delay delay = 3001000, diff = 42380 ns
hwtimer_delay delay = 3101000, diff = 38870 ns
hwtimer_delay delay = 3201000, diff = 38590 ns
hwtimer_delay delay = 3301000, diff = 38110 ns
hwtimer_delay delay = 3401000, diff = 38210 ns
hwtimer_delay delay = 3501000, diff = 37970 ns
hwtimer_delay delay = 3601000, diff = 38160 ns
hwtimer_delay delay = 3701000, diff = 38160 ns
hwtimer_delay delay = 3801000, diff = 37720 ns
hwtimer_delay delay = 3901000, diff = 39930 ns
hwtimer_delay delay = 4001000, diff = 38760 ns
hwtimer_delay delay = 4101000, diff = 38260 ns
hwtimer_delay delay = 4201000, diff = 39370 ns
hwtimer_delay delay = 4301000, diff = 37870 ns
```

图5 串口打印结果

因外设接口调用时代码执行需要时间以及晶振等硬件会导致误差，分析例程打印数据可得，延时性定时器的软件开销在同一硬件以及接口下，其误差基本是一致的。

4. 捕获型定时器

{SDK}\demos\peripheral\cap 路径下为捕获型定时器例程，例程关键代码如下：

```
/* 单边沿触发 */
static void test_cap_single_edge(
    int fd,
    int gpio_cap,
    uint32_t ms,
    aw_hwtimer_cap_config_t *p_config,
    int is_rising)
{
  uint64_t          cap_val1, cap_val2;
  aw_err_t          ret;

  // 制造两次上升沿
  mk_edge(gpio_cap, 5);
  aw_task_delay(ms);
  mk_edge(gpio_cap, 5);

  // 此时应该产生了两次捕获事件
  // 把它们读出来
  ret = aw_hwtimer_cap_read(fd, &cap_val1, AW_WAIT_FOREVER);
  if (AW_OK != ret) {
    aw_kprintf("cap read cap_val1 failed \n");
    return;
  }
  ret = aw_hwtimer_cap_read(fd, &cap_val2, AW_WAIT_FOREVER);
  if (AW_OK != ret) {
    aw_kprintf("cap read cap_val2 failed \n");
    return;
  }
  cap_val2 -= cap_val1;
  cap_val2 *= 1000000;
  cap_val2 /= p_config->sample_rate;

  if (is_rising) {
    aw_kprintf("two rising edge between %u ms \n", ms + 5);
  }
  else {
    aw_kprintf("two falling edge between %u ms \n", ms + 5);
  }
  aw_kprintf("two capture events between %llu us \n", cap_val2);
}

static void demo_cap_base(int gpio_cap)
{
  int               fd;
  aw_err_t          ret;
  aw_hwtimer_cap_config_t   config;

  // 使得测试 GPIO 输出为 0
  aw_gpio_set(gpio_cap, 0);

  fd = aw_open(CONFIG_DEMO_HWTIMER_CAP_DEV_NAME, AW_O_RDWR, 0);
  if (fd < 0) {
    aw_kprintf("cap open failed!\n");
    return;
  }

  // 获取捕获定时器的配置
  ret = aw_hwtimer_cap_config_get(fd, &config);
  if (ret != AW_OK) {
    aw_kprintf("cap config get failed...\r\n");
    aw_close(fd);
    return ;
  }

#if CONFIG_SINGLE_EDGE
  int is_rising;
  // 配置为上升沿触发捕获
  config.cap_edge_flags = AW_CAPTURE_RISING_EDGE;
  is_rising = 1;
  ret = aw_hwtimer_cap_config_set(fd, &config);
  if (ret != AW_OK) {
    aw_kprintf("cap config set failed...\r\n");
    aw_close(fd);
    return ;
  }

  ret = aw_hwtimer_cap_start(fd);
  if (ret != AW_OK) {
    aw_kprintf("cap start failed...\r\n");
    aw_close(fd);
    return ;
```

```
    }
    test_cap_single_edge(fd, gpio_cap, 20, &config, is_rising);
#endif

    aw_close(fd);
}
```

在 CAP 例程中默认使用的是 timer5_chan0，这个通道对应的引脚是 PF08，可以通过查看工程下 timer5_chan0 对应的 .h 文件得知所使用的引脚的编号为 168，通过查看 hpm_pin.h 头文件可知编号 168 对应的引脚为 PF08 如下图所示。

```
/* generated by aworks, do not modify! */
#define CONFIG_DEMO_HWTIMER_CAP_DEV_NAME "/dev/timer5_chan0_cap"
#define CONFIG_DEMO_HWTIMER_CAP_DEV_NAME_value__dev_timer5_chan0_c
#define CONFIG_DEMO_HWTIMER_CAP_TASK_STACK_SIZE 2048
#define CONFIG_DEMO_HWTIMER_CAP_DEMO 1
#define CONFIG_SINGLE_EDGE 1
#define CONFIG_AW_IMG_PRJ_BUILD 1
```

图6 默认通道

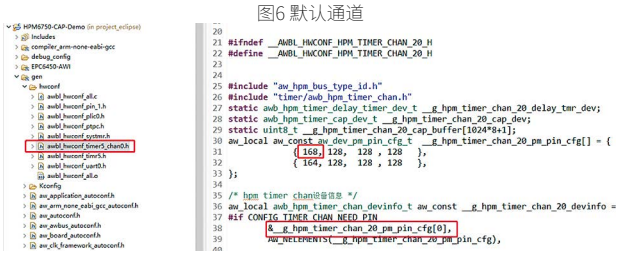


图7 对应引脚编号

```
#define PIN_PF04 (164UL)
#define PIN_PF05 (165UL)
#define PIN_PF06 (166UL)
#define PIN_PF07 (167UL)
#define PIN_PF08 (168UL)
#define PIN_PF09 (169UL)
#define PIN_PF10 (170UL)
```

图8 对应引脚

本实验中还用到了 PF09 这个引脚，用于产生捕获事件，PF09 和 PF08 这两个引脚在开发板上并没有引出来，不利于这次实验，需要修改这两个引脚。

```
#define CAP_GPIO PIN_PF09
int demo_cap_common_entry(void *arg);

int aw_main()
{
    aw_kprintf("\r\nApplication Start. \r\n");
#if CONFIG_DEMO_HWTIMER_CAP_DEMO
    aw_pin_cfg(CAP_GPIO,
               HPM_PIN_MUX(IOC_PAD_FUNC_CTL_ALT_SELECT(0)) | AW_PIN_CFG_GPIO_OUTPUT_HIGH);
#endif
    demo_timer_cap_entry(CAP_GPIO);

    return 0;
}
```

图9 捕获产生引脚

参 考 {SDK} platforms\platform-hpm-aworks-lp\boards\EPC6450-AWI\dts 下的 pins.dts 引脚描述文件，找到 timer4_chan1 如图 10 所示，timer4_chan1 使用的引脚是 PE25, 对应着开发板排针 UTX1 丝印的位置。

```
&timer3_chan1 {
    pins = <&pin1 PIN_PE17 IOC_PE17_FUNC_CTL_TMR3_
};

&timer4_chan0 {
    pins = <&pin1 PIN_PE21 IOC_PE21_FUNC_CTL_TMR4_
           &pin1 PIN_PE22 IOC_PE22_FUNC_CTL_TMR4_
};
&timer4_chan1 {
    pins = <&pin1 PIN_PE25 IOC_PE25_FUNC_CTL_TMR4_
};
```

图10 捕获产生引脚

打开配置界面将 timer5_chan0 修改为 timer4_chan1 如图 11 所示，修改后点击保存，重新 build 工程。

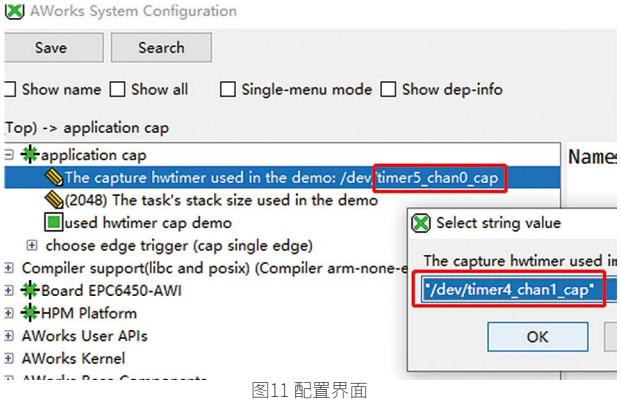


图11 配置界面

将 CAP_GPIO 对应的引脚改为 PIN_PE24，对应着开发板排针 URX1 丝印的位置，如图 12 所示。

```
#include "hpm_pin.h"

#define CAP_GPIO PIN_PE24
int demo_cap_common_entry(void *arg);

int aw_main()
{
```

图12 CAP引脚

将 PE25 , PE24 这两个引脚，也就是排针上 URX1 和 UTX1 短接。

图13 引脚位置

上诉代码中使用 aw_hwtimer_cap_config_get 接口获取捕获定时器的配置信息，配置 AW_CAPTURE_RISING_EDGE 单通道模式后使用 aw_hwtimer_cap_config_set 接口配置捕获定时器。使用 aw_hwtimer_cap_start 接口启动定时器。在 test_cap_single_edge 函数中调用 mk_edge 函数制造两次上升沿，使用 aw_hwtimer_cap_read 接口读取这两次事件捕获到的计数值，计算出差值后在串口上显示。

在 test_cap_single_edge 函数中使用 mk_edge 函数中控制 CAP_GPIO 引脚输出高电平后延时 5ms 再输出低电平。延时 20ms 后再次调用 mk_edge 函数，因此两次上升沿事件间隔应为 25ms。串口打印结果如下图所示。

```
[13:29:34.200]收←◆
Application Start.
cap demo start

[13:29:34.235]收←◆two riging edge between 25 ms
two capture events between 25053 us
```

图14 串口打印结果

至此，所有类型的硬件定时器样例均已展示完毕，在软件应用设计中可根据实际需求选取不同类型的定时器进行使用。更多其他类型外设的用法介绍，请关注后续同系列推文 ~

【EsDA应用】串口转zws物联网云平台

ZLG 致远电子 2023-01-12 11:30:48

物联网逐渐成为各种行业的一个标配，如何让设备快速接入物联网云平台，将是产品在行业竞争中制胜的关键。

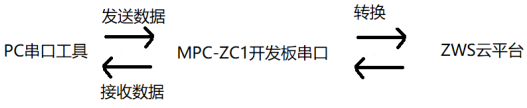
简介

在实际项目中，我们经常会用到串口服务器，以提供串口与网络服务器之间的 双向数据透明传输 为核心业务。其能在不用修改原有产品系统的情况下，为串口设备提供了便捷的联网通道，即扩展了物联网功能，又保障了原有系统的稳定性。

本文以实现 串口转 ZWS(即提供串口与 ZWS 云平台之间的 双向数据透明传输 业务) 为目标，展开讲解，介绍如何通过 EsDA 工具和 MPC-ZC1 平台，进行图形化低代码应用开发，快速完成一个简易的串口服务器。

注 :ZWS 物联网云平台是致远电子推出的物联网 IoT 云平台

本次实验选用 MPC-ZC1 的串口 2 作为目标串口，实验目标如下图：



- pc 机串口对 MPC-ZC1 的串口 2 发送数据，等效对 ZWS 云平台发送数据；
- pc 机串口读 MPC-ZC1 的串口 2 接收到的数据，等效读 ZWS 云平台下发的数据。

前期准备

若是刚开始接触 EsDA MPC-ZC1 ，可先阅读 EsDA MPC-ZC1 系列 文章，从零开始搭建环境和掌握基本开发流程，已有基础的可以跳过：

- EsDA MPC-ZC1 入门（一）—— 软件安装
- EsDA MPC-ZC1 入门（二）—— LED 控制
- EsDA MPC-ZC1 应用——串口服务器（一）
- EsDA MPC-ZC1 应用——串口服务器（二）

1. ZWS云平台入门与相关准备工作

ZWS 物联网云平台是致远电子推出的物联网 IoT 云平台，和阿里云类似，可以接入各种 IoT 设备。 在浏览器上打开 www.zlgcloud.com ， 可自行注册账号，可免费使用 ZWS 云平台提供的设备管理、数据管理、项目管理、触发规则管理等各种功能。

* 可通过 ZWS 物联网云平台上的 ZLG 物联网平台教程深入学习 ZWS 云平台的使用。

1.1 使用 ZWS 云平台的准备工作

1.1.1 点击 www.zlgcloud.com 进入 zws 云平台主页，创建云平台账号，并登录。

1.1.2 在 ZWS 云平台创建自己的设备。

在 ZWS 云平台创建设备，首先要创建设备类型，打开设备类型管理页面。

点击添加类型。

选择 basic 模板，并将新设备类型命名为 aw_flow_test。

点击确定，完成设备类型的创建。

1.1.3 添加设备

有了设备类型，就可以开始创建设备了，打开设备列表页面。

点击添加设备。

设备类型选择刚才新建的 aw_flow_test 类型，并将设备的名称命名为 test 和 ID 命名为 zc1。

点击保存，完成设备添加。

点击返回设备列表。

即可在设备列表中看到新添加的设备。

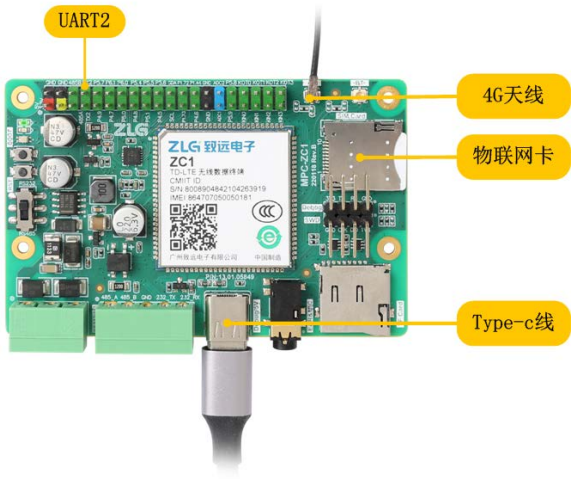
2. 硬件相关准备工作

2.1 准备一个 usb 转 TTL 串口工具 (文中使用的是 ch340 芯片作为主控一款工具，选用其它等效型号亦可)，如下图所示：

2.2 准备好 MPC-ZC1 开发板，并按照下图所示连接好硬件。

将 MPC-ZC1 开发板引出的 TX2 与 RX2 分别与 usb 转 TTL 串口工具的 RX 与 TX 连接，并将 usb 转 TTL 串口工具插到 pc 机 usb 口上。

至此，准备工作已经完成。



节点介绍

1. 串口系列节点介绍

MPC-ZC1 串口通信，需要使用到 AWFlow 如下节点 : serial_ex、serial_in_ex、serial_out_ex。

1.1 serial_ex

串口配置节点，属于隐式节点，不会被显示在画布中，通常用于进行参数的配置，需要和配套对应的功能节点一起使用。

1.1.1 属性

- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 端口（port）： 用于索引串口设备；
- 波特率（baudrate）： 串口波特率参数；

- 字节大小（bytesize）： 数据位参数；
- 奇偶校验（parity）： 串口奇偶校验位参数；
- 停止位（stopbits）： 串口停止位参数；
- 流控（flowcontrol）： 串口流控模式配置；
- 帧超时（frame_timeout）： 收到数据后的总体等待时间；
- 帧长度（frame_length）： 期望收据的数据长度；
- 码间超时（intersymbol_timeout）： 字节间的最大超时时间。

* 其中帧超时、帧长度、码间超时可用于分包应用，3 个参数可同时使用，任意一个条件满足都会触发分包。

配置节点（config 类型）不具备输入输出。

1.2 serial_in_ex

串口接收节点，负责读取指定串口接收到的数据。

1.2.1 属性

- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 配置节点名称（config）： 绑定一个串口配置节点。

1.2.2 输入

pump 类型节点通常不具备数据输入。

1.2.3 输出

- payload： 读取到串口接收的数据，字符串（可按二进制提取）；
- payloadLength： 数据长度，uint32_t 类型；
- payloadType：payload 的数据类型，用于后续节点数据处理；
- istream： 数据流对象，保存着串口接收的原始数据流；

* 当 帧超时、帧长度、码间超时 其中任意参数有效时，输出 payload 格式，否则输出 istream。

1.3 serial_out_ex

串口发送节点，将上级节点输出的数据发送至串口发送接口。

1.3.1 属性

- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 配置节点名称（config）： 绑定一个串口配置节点。

1.3.2 输入

- payload： 负载数据，字符串类型（也可按二进制转换）；
- payloadLength： 负载数据长度，uint32_t 类型；
- payloadType： 指示 payload 的数据类型；
- istream： 数据流对象；

* 支持输入 payload 和 istream 数据，优先使用 istream。

1.3.3 输出

sink 类型节点通常不具备数据输出。

2. zws云系列节点介绍

zws 云数据收发，需要使用到 AWFlow 如下节点 : zws_iot 、zws_iot data_out、zws_iot data_in。

2.1 zws_iot

zws_iot 配置节点，属于隐式节点，不会被显示在画布中，需要和配套的对应功能节点一起使用，主要用于配置连接 zws 云平台的相关参数。

2.1.1 属性

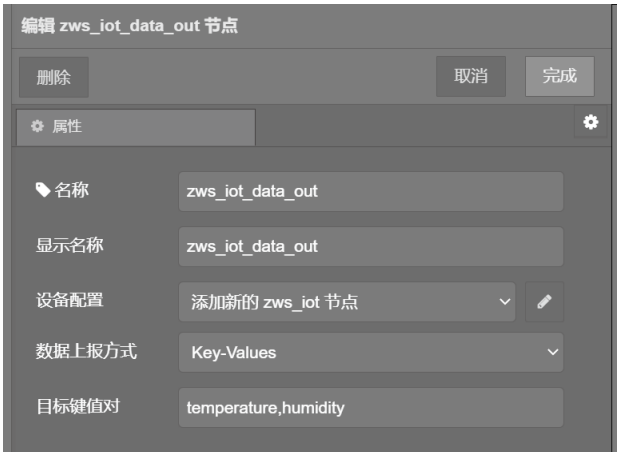
- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 是否连接（connection_status）： 使能立即连接；
- 设备类型（dev_type）： ZWS 三元组 - 设备类型；
- 设备 ID（dev_id）： ZWS 三元组 - 设备 ID；
- 设备密钥（dev_secret）： ZWS 三元组 - 设备密钥；
- 固件版本（firmware_version）： 设备固件版本，产品自定；
- 设备心跳周期（keep_alive_interval）： 心跳周期。

配置节点（config 类型）不具备输入输出功能

2.2 zws_iot_data_out

ZWS 数据上报节点，上报数据到 zws 云平台。

2.2.1 属性



- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 设备配置（config）： 绑定一个 zws_iot 配置节点；
- 数据上报方式（output_type）： 选择上报数据方式；
- 目标键值对（key_names）： 指定上报的数据点名称。

2.2.2 输入

payload： 要上报给 zws 云平台的数据；

其他属性： 当与 key_names 匹配时有效。

2.3 zws_iot_data_in

ZWS 数据接收节点，接收 zws 云平台下发的数据。

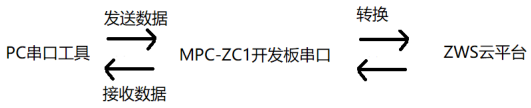
2.3.1 属性

- 名称（name）： 节点名称，用于索引查找本节点；
- 显示名称（displayName）： 用于画布上显示的名称；
- 设备配置（config）： 绑定一个 zws_iot 配置节点。

2.3.2 输出

- payload： 字符串类型，ZWS 云平台的下发的字符串数据。

业务开发



我们主要是通过 EsDA 工具和 MPC-ZC1 平台，实现串口转 zws。即在 pc 机上使用 usb 转串口工具连接 MPC-ZC1 开发板串口就可发送数据到 zws 云平台或读取 zws 云平台下发的数据。

1. 实现串口到ZWS云

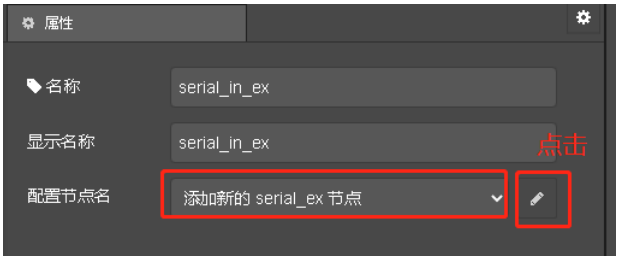
1.1 添加串口节点

1.1.1 添加 serial_in_ex 与 serial_out_ex 到画布上，备用。

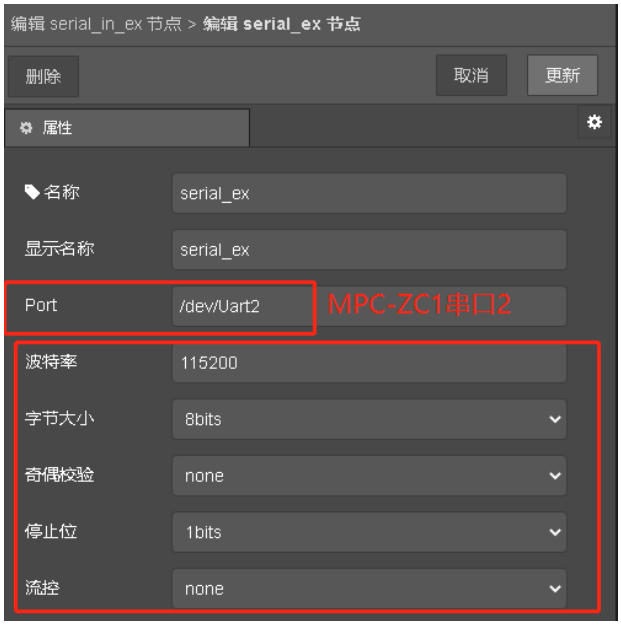
1.1.2 配置串口

双击 serial_in_ex 节点，打开属性面板。

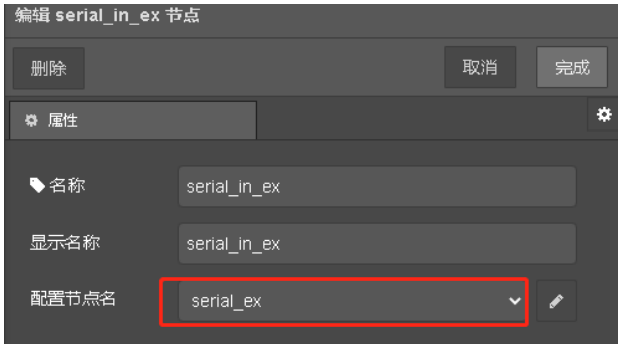
选择 “添加新的 serial_ex 节点” ，点击编辑配置，进入配置节点属性面板。



配置如下图所示，点击右上角添加 / 更新完成配置



可以看到已经创建了一个新的配置节点，名为 serial_ex，选择其作为配置节点，点击完成结束 serial_in_ex 节点的配置。



同样地，双击 serial_out_ex 节点，打开属性面板，直接选择刚刚创建的 serial_ex 节点作为配置节点。

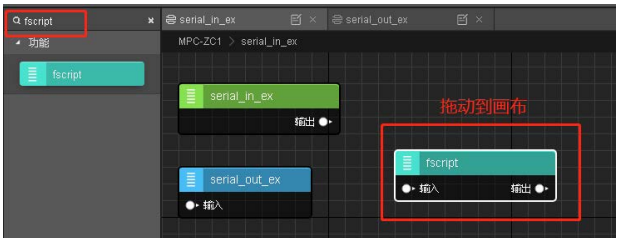
1.2 添加和配置 fscript 脚本节点

fscript 脚本节点可执行一段 fscript 脚本，可以为 initialize、 func 和 finalize 分别指定一段脚本。

关于 fscript 请访问 fscript 教程，可阅读该文档深入了解。

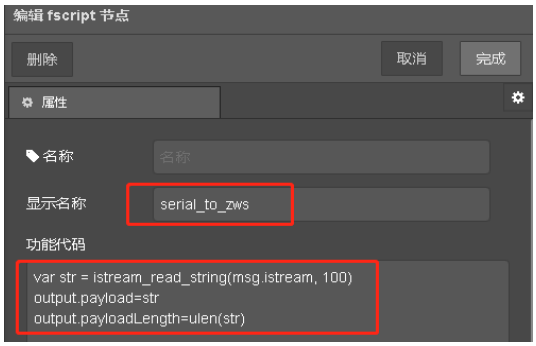
https://github.com/zlgopen/awtk/blob/master/docs/fscript.md

1.2.1 将 fscript 脚本节点拖动到画布。



1.2.2 添加解析脚本

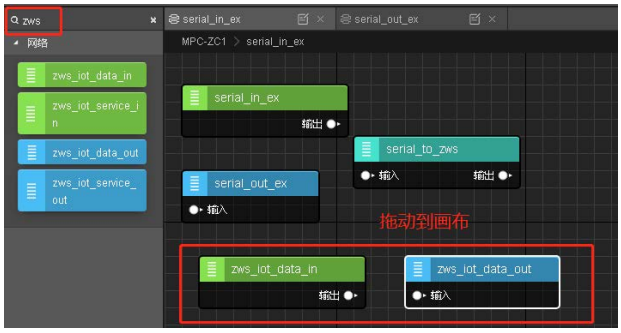
添加脚本，从串口流读取字符串数据，并根据 zws_iot_data_out 节点的数据上报模式 RAW 模式进行组包，同时将 fscript 脚本节点显示名称为 serial_to_zws，点击完成保存。



功能代码如下：

```
var str = istream_read_string(msg.istream, 100)
output.payload=str
output.payloadLength=ulen(str)
```

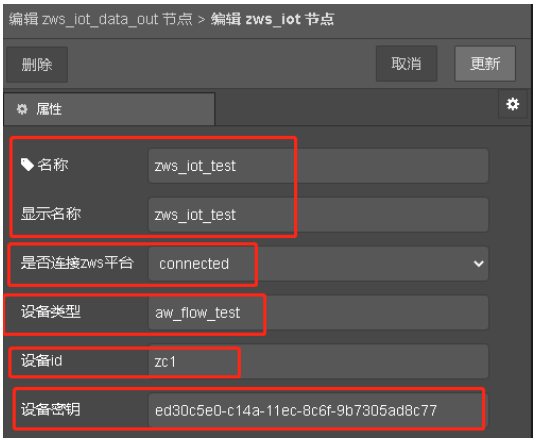
1.3 添加 zws_iot_data_out 与 zws_iot_data_in 节点



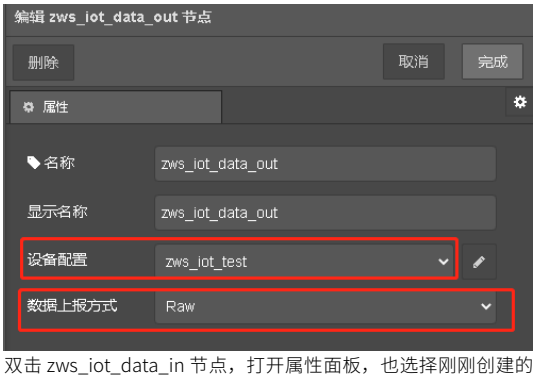
1.3.1 配置连接 zws 云平台的参数

双击 zws_iot_data_out 节点，打开属性面板，选择 RAW 上报方式。选择 “添加新的 zws_iot 节点” ，点击编辑配置。

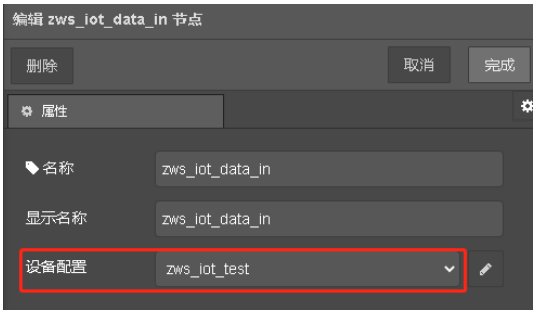
进入 zws_iot 配置节点属性面板（设备类型、设备 id、设备密钥必须与之前在 zws 云平台上创建的设备一致，否则无法登录成功），点击右上角添加 / 更新，完成配置。



可以看到已经创建了一个新的配置节点，名为 zws_iot_test，选择其作为配置节点，点击完成，结束 zws_iot_data_out 节点的配置。



双击 zws_iot_data_in 节点，打开属性面板，也选择刚刚创建的 zws_iot_test 作为配置节点，然后点击右上角完成节点配置。



1.4 绘制流图

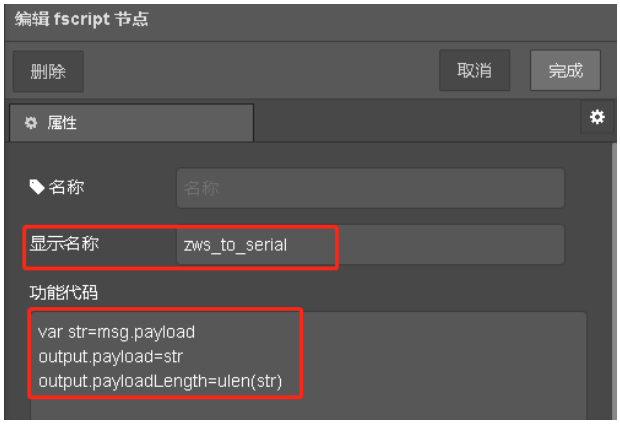
将画布里的 serial_in_ex 节点、serial_to_zws 节点、zws_iot_data_out 节点按照下图所示依次连接起来。

这样，串口到 ZWS 云平台的流图就完成了。

2. 实现ZWS云到串口

2.1 添加 fscript 脚本节点

将一个新的 fscript 脚本节点拖动到画布，并按下图所示进行配置，然后点击右上角完成。



功能代码如下：

```
var str=msg.payload
output.payload=str
output.payloadLength=ulen(str)
```

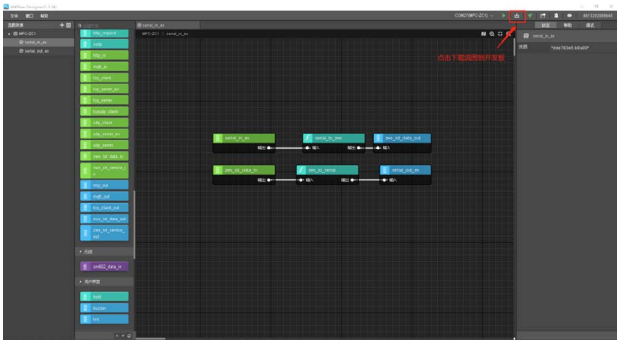
2.2 绘制流图

将画布里的 zws_iot_data_in 节点、zws_to_serial 节点、zws_iot_data_out 节点按照下图所示依次连接起来。

这样，ZWS 云平台到串口的流图就完成了。

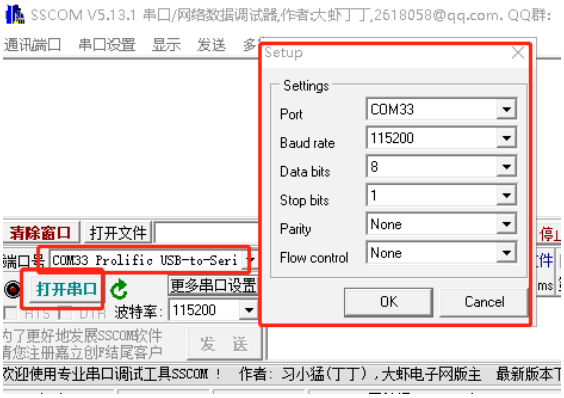
结果验证

将上一小节绘制好的流图下载到 MPC-ZC1 板子里，我们就可以开始验证结果啦。



1. 验证串口到ZWS云平台

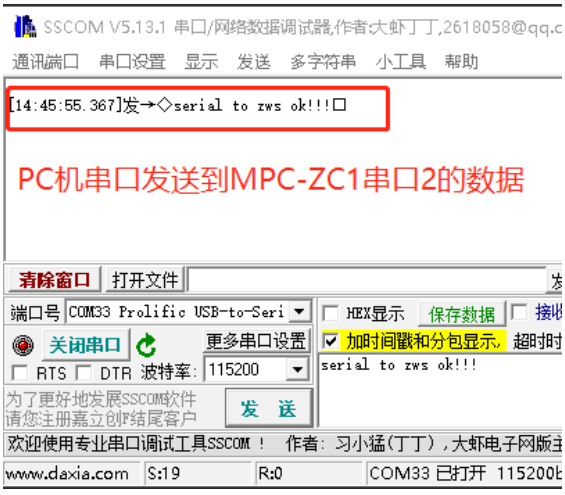
1.1 PC 机打开串口工具（这里以 sscom_v5.13.1 为例），并选择之前接到 PC 机上的 USB 转 TTL 工具的对应 COM 口，波特率选择 115200。



1.2 点击 https://www.zlgcloud.com/ 进入 zws 云平台主页，登录云平台账号，并打开设备列表，选择刚才创建的设备，并点击设备详情。

1.3 点击实时数据，并选择 raw 数据。

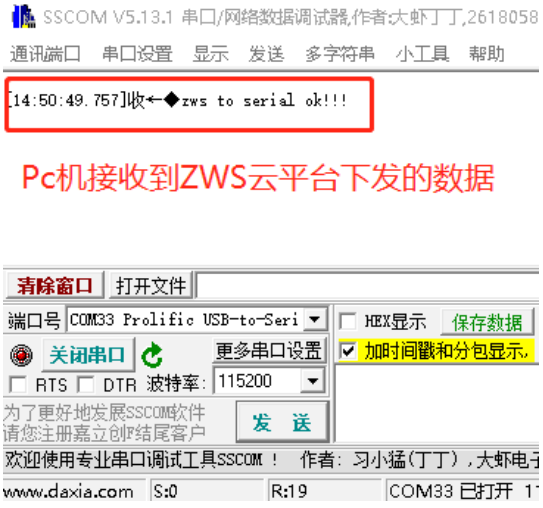
1.4 使用 sscom 发送字符串数据，在 ZWS 云平台实时数据网页就能看到接收到的数据。



2. 验证ZWS云平台到串口

2.1 点击设备控制，根据下图配置，并点击发送，若发送成功网页会弹出“发送成功提示框 "。

2.2 sscom 会接收到来自 zws 云平台字符串数据。



【产品应用】
如何通过物联网云远程维护ZigBee网关?

原创 研发部 ZLG 致远电子 2023-01-28 11:33:38

智能家居、智慧照明市场采用的通讯协议多样化，ZigBee 是主流的通讯协议之一。基于 ZigBee 的智能产品，都无法绕开 ZigBee 网关，因此，对网关的远程管理维护也很重要。

应用场景

智能 IoT 设备的无线通信技术各有千秋，ZigBee 凭借低功耗优势在智能家居、智慧照明等场景中应用广泛。其中，在一些场景下需要对 ZigBee 网关进行管理和调试，以保证用户设备的正常通信。那么，我们来看看如何通过云平台对 ZigBee 网关进行远程管理和维护。

云端赋能ZigBee网关

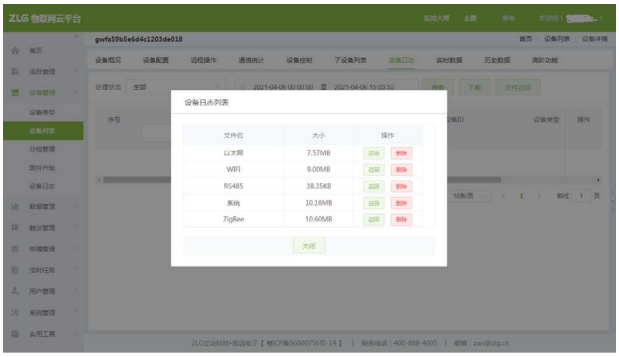
ZWS 云平台是 ZLG 致远电子研发的通用物联网云平台，将 ZigBee 网关接入云平台，就能可视化远程管理维护网关，可对网关进行如下操作。

1. 远程配置网关

可以远程配置 ZigBee 网关的网络号、通道号，自定义设置 ZigBee 节点超时时间，除此之外，还可设置白名单，过滤 ZigBee 节点，允许特定节点加入网关。

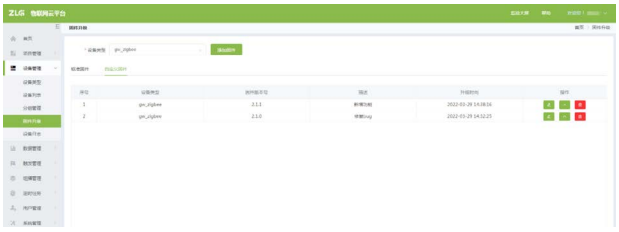
2. 远程控制网关

可以给网关远程下发命令，比如：复位、恢复出厂设置等，还支持对网关子设备以广播、轮询、单播的方式进行命令下发。



3. 远程召回日志

网关发生故障或者 bug，技术人员查看设备日志能快速找到故障原因。可以远程召回存储在网关中的以太网、WIFI、RS485、系统、ZigBee 日志文件。



4. 远程升级固件

网关增加、优化或修复功能，往往会迭代固件版本，而通过串口、JFLASH 烧写等方式进行固件升级，耗时不易集中管理，可以通过云平台对 ZigBee 网关在线远程升级。

GZ32M-I 网关

GZ32 ZigBee 网关是 ZLG 致远电子研发的工业级 ZigBee 物联网网关，内置了 ZWS 云平台服务，完成配网后，支持远程管理、远程升级、网关备份等功能。

【技术分享】如何在嵌入式Linux平台上使用Nginx搭建RTMP流媒体服务器？

原创 研发部 ZLG 致远电子 2023-01-04 11:30:27

RTMP 作为目前主流的流媒体传输协议，广泛应用于音视频领域。那么我们如何快速在嵌入式板子上搭建起自己的 rtmp 流媒体服务器？本篇文章将带大家实践一下。

概述

Nginx 是一个以高效稳定著称的高性能的 HTTP 和反向代理 web 服务器，它同时也是基于事件驱动开发的异步高性能跨平台服务器。Nginx-RTMP 是基于 Nginx 框架的模块开发，很好地继承了 Nginx 的异步高性能以及扩展性好的优点。

RTMP 是 Real Time Messaging Protocol (实时消息传输协议) 的首字母缩写。该协议基于 TCP 协议簇，是 Adobe 公司为 Flash/AIR 平台和服务器之间音、视频及数据传输开发的实时消息传送协议。在 RTMP 协议中，视频必须是 h264 编码，音频必须是 AAC 或 MP3 编码，且多以 flv 格式封包。目前 RTMP 是主流的流媒体传输协议。而 Nginx-RTMP 模块主要是对 rtmp 协议的实现，广泛应用于音视频领域。

那么以下将通过一个简单的视频监控方案带大家了解下 Nginx-RTMP 流媒体服务器的搭建过程。

方案实现

以 M3568 平台为例，在嵌入式 Linux 系统上搭建 Nginx-RTMP 流媒体服务器以及通过 ffmpeg/gstreamer 实现简单的推拉流过程。实现框架如图 1 所示

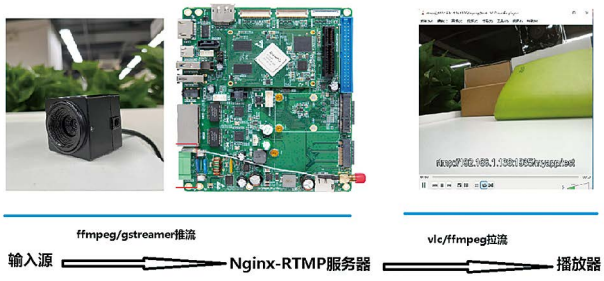


图1

采集 M3568 板子上的摄像头视频流，使用 gstreamer/ffmpeg 进行 RTMP 推流到板子上的 nginx 服务器（nginx 通过 rtmp 模块提供 rtmp 服务），然后在同个局域网内使用 vlc 播放器（或其他客户端）从 nginx 服务器获取该实时视频流。

1. 前期准备

1.1 硬件

M3568 核心板；

M3568-EV-Board 底板；

usb 摄像头（如果没有摄像头可以使用 mp4 视频文件代替做测试）。

1.2 软件

PC 上安装 vlc 播放工具；

带 nginx-rtmp-module 模块编译的 nginx 安装包（nginx-install.tar）。

（备注：本篇文章没有展开讲述 Nginx + nginx-rtmp-module 的交叉编译过程以及具体步骤，但此部分内容已发布在 ZLG 开发者社区上，可通过链接 https://z.zlg.cn/articleinfo?id=853011 进行访问。）

2. 实现步骤

2.1 将移植好的 nginx 安装包拷贝到板子上，然后解压到板子的根目录下。

2.2 修改 /etc/nginx/nginx.conf 配置文件，添加 rtmp 配置。如图 2 所示。

```
#user  nobody;
worker_processes  1;

#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;


events {
    worker_connections  1024;
}

rtmp {
        server {
                listen 1935;
                application myapp {
                        live on;
                        record off;
                }
        }
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    #log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
    #                  '$status $body_bytes_sent "$http_referer" '
    #                  '"$http_user_agent" "$http_x_forwarded_for"';

    #access_log  logs/access.log  main;
```

图2

2.3 执行如下命令启动 nginx 服务器。

nginx -c /etc/nginx/nginx.conf

2.4 在板子上执行如下推流命令，采集摄像头的实时视频流推送至板子上的 nginx 服务器上。

gst-launch-1.0 v4l2src device=/dev/video5 ! \

video/x-raw,format=YUY2,width=640,height=480,framerate=30/1 ! \

queue ! videoconvert ! mpph264enc ! \

video/x-h264,stream-format=byte-stream ! queue ! \

h264parse ! flvmux ! rtmpsink location=rtmp://192.168.1.136:1935/myapp/test

如果使用 mp4 视频文件做推流测试，可直接执行如下推流命令。

ffmpeg -re -i ./test.mp4 -c copy -f flv rtmp://192.168.1.136:1935/myapp/test

2.5 此处板子上的 IP 地址配置为 192.168.1.136，此时将 PC 电脑接在跟板子同一个局域网上，然后使用 vlc 播放器打开网络串流 rtmp://192.168.1.136:1935/myapp/test 即可预览视频流画面。

【技术分享】嵌入式核心板开发之ESD静电保护

原创 研发部 ZLG 致远电子 2023-01-11 11:35:35

在电子产品开发中 ESD 静电防护是不可或缺的一环，下面就为大家简单介绍一下，核心板产品开发时有用的 ESD 二极管知识和技巧。

ESD管的介绍

ESD（Electrostatic Discharge Protection Devices），静电保护元器件，又称瞬态电压抑制二极管阵列（TVS Array），是由多个 TVS 晶粒或二极管采用不同的布局设计成具有特定功能的多路或单路 ESD 保护器件，主要应用于各类通信接口静电保护，比如 USB、HDMI、RS485、RS232、VGA、RJ11、RJ45、BNC、SIM、SD 等接口中。

ESD静电二极管特性

1. 低电容：结电容低，通常在 PF 级别；
2. 快速的响应时间：通常在 ps 级；
3. 封装体积小，小型化器件，节约 PCB 空间；
4. 灵活度高，可以根据应用需求设计电容、封装形式、浪涌承受能力等参数。

ESD静电二极管选型指南

1.ESD 静电二极管的截止电压 VRWM 要大于等于电路中最高工作电压；

2. 脉冲峰值电流 IPP 和最大箝位电压 VC 的选择，要根据线路上可能出现的最大浪涌电流来选择合适 IPP 的型号，需要注意的是，此时的 VC 应小于被保护晶片所能耐受的最大峰值电压；

3. 用于高速信号的保护时，还要注意所传输信号的频率或传输速率，当信号频率或传输速率较高时，应选用低电容系列的 ESD 静电二极管；

4.ESD 有单向（A）和双向（C）之分，根据工作的信号进行选择，单极性的信号可以选择单向的 ESD 或双向的 ESD， 双极性的信号要选择双向的 ESD。

选型基本原则

致远电子嵌入式核心板产品 M3568 工控核心板，拥有非常成熟的设计案例可供参考，其中的接口均设计了 ESD 保护电路，并通过了静电测试，保证产品的持续稳定运行，可以放心使用。

1. 如通用外设串口的 ESD 设计，采用通用的低速 ESD 二极管，截止电压 3.3V，钳位电压 12V，结电容 100pF，通过 100 欧串阻配合降低峰值电压，参考设计如下：

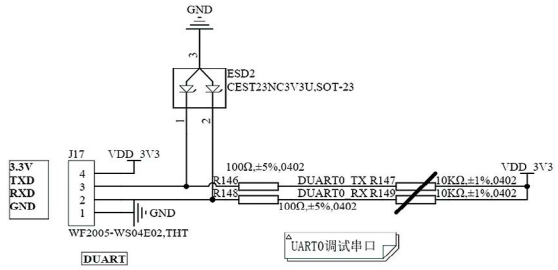


图1 嵌入式核心板M3568 UART口ESD设计参考

2. 如 SDIO 高速信号的 ESD 设计，使用低结电容的高速信号专用 ESD 保护器件，可令 SDIO 的通信质量不受影响，参考设计如下：

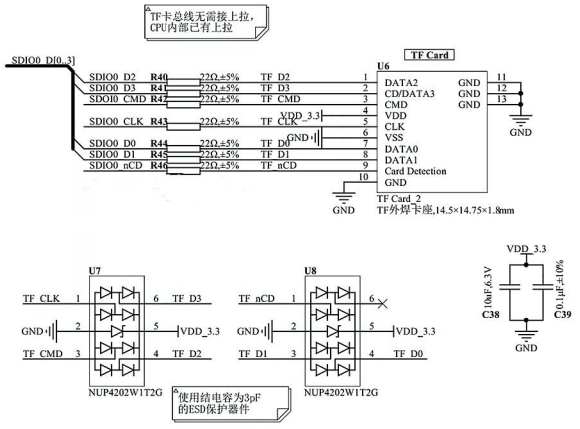


图2 嵌入式核心板M3568 SD/MMC设计参考

3. 如 HDMI 高速信号的 ESD 设计，采用 HDMI 专用的接口保护器件，除了提供ESD防护之外，还能给 EDID 信号提供隔离和电平转换，参考设计如下：

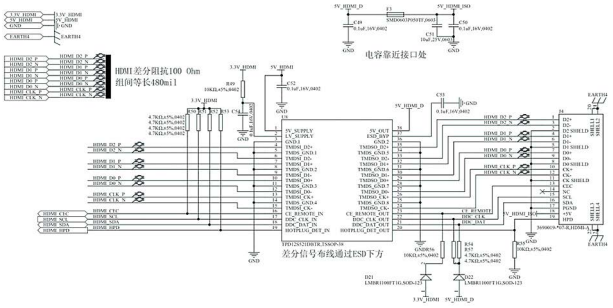


图3 嵌入式核心板M3568 HDMI设计参考

ESD保护设计小结

ESD 二极管广泛应用于通信、安防、工业、汽车、消费类产品、智能穿戴设备等电子产品的通信线及 I/O 口等静电保护。购买致远电子的嵌入式核心板产品，即可获得全面可靠的硬件设计参考和配套软件驱动。致远电子提供的配套评估底板均经过严格的 EMC 测试，可提供测试报告。此外致远电子还会提供贴心的技术支持服务，诸如原理图检查、协助故障调试、特殊需求订制等等，助您扫除产品研发途中的一切障碍。

【产品应用】如何在Coral3568平台快速适配mipi显示屏?

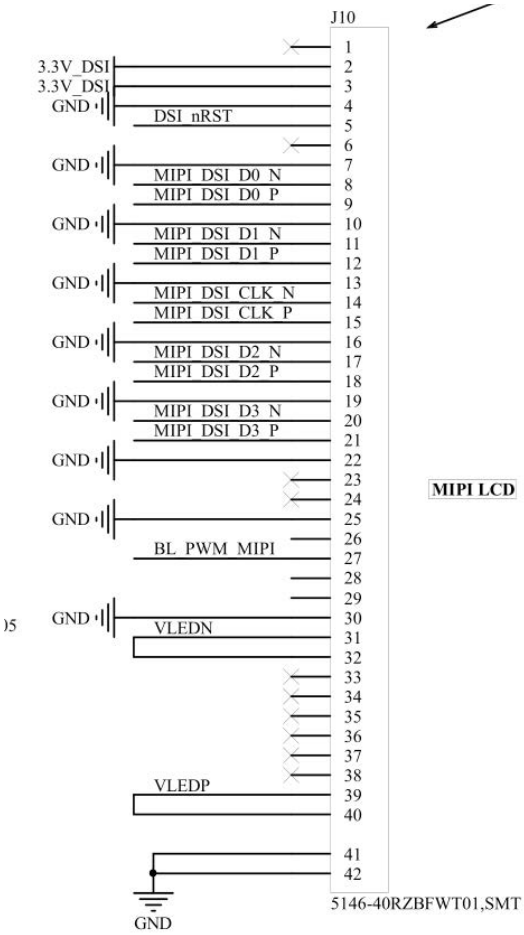
原创 研发部 ZLG 致远电子 2023-01-09 11:33:37

随着工作节奏的变快，如何快速熟悉平台的使用和开发，也是有效工作的重点之一。本文将介绍在 Coral3568 平台上，如何快速适配新的 10.1 寸 mipi 显示屏。

接口一致

Coral-EVa 为致远电子推出的 Coral3568 配套评估底板，同样功能强大，接口丰富。Coral-EVa 评估底板采用适配器供电，更方便实验室和研发办公室使用，HDMI、DP、USB、CAN、RS485、RS232、TTL UART、3.5mm 四线耳麦接口、Micro SD 卡槽、SATA、M.2、LVDS LCD、MIPI_DSI、MIPI-CSI、RTC 时钟、蜂鸣器等功能一应俱全。

Coral-EVa 底板 MIPI_DSI 接口如图所示：



对于大多数 40pin MIPI 显示屏，其接口与上图一致。

格式转换

1. 原厂提供的初始化指令

选用测试 MIPI 屏型号：CC1101I40M-01(分辨率 1280*800)

购买显示屏向商家索要初始化文件，商家现提供的初始化文件为：16424929046_Test_ILI9881C_BOE-B4 TV101WXU-N91_T02_20210423 gamma 优化 (1).txt

初始化文件部分内容：

```
REGISTER,FF,03,98,81,03


REGISTER,01,01,00
REGISTER,02,01,00
REGISTER,03,01,53     //STVA=STV2_4
.....
```

查看初始化文件内容，所有的语句都有统一的格式，即：REGISTER,aa,bb,cc,dd.....

经过对比芯片手册，判定此类的语句意义为：通过 mipi 总线，在地址 aa, 写入数据长度为 bb, 写入数据数值为 cc,dd……的数据。

2. 初始化格式转换

Coral3568 平台 MIPI-DSI 配置文件为：

arch/arm64/boot/dts/rockchip/rk3568-evb1-ddr4-v10-dsi.dtsi

配置文件初始化格式为：

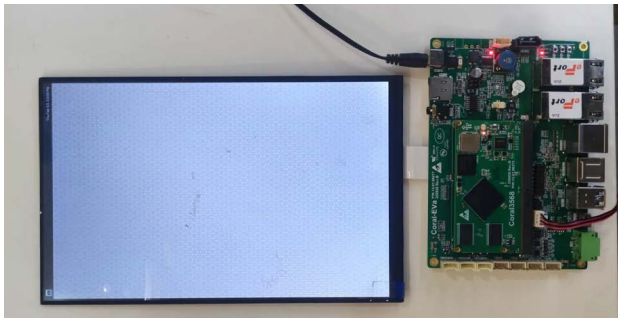
0x15/0x39 | 0x00 | 写入数据长度 | 写入的数据 (1 字节地址 +n 字节数据)

其中，如果写入 1 字节数据，选择 0x15；如果写入大于 1 字节数据，选择 0x39。

格式转换后数据为：

```
panel-init-sequence = [
    39 00 04 FF 98 81 03
    15 00 02 01 00
    15 00 02 02 00
    15 00 02 03 53
    15 00 02 04 D3
    .....
```

至此 ,MIPI 屏初始化配置完成，重新编译内核及固化到 Coral3568 板子，重启即可显示。效果如图：



【产品应用】
SX-3568 + OpenHarmony强强联合

原创 FAE 工程师 ZLG 致远电子 2023-01-09 11:33:37

SX-3568 是 ZLG 致远电子自主设计的一款智慧商显主板，搭配 OpenHarmony 操作系统，在工业领域、商显领域具有极强的稳定性。本文介绍 OpenHarmony 在 SX-3568 上运行情况。

SX-3568智慧商显主板简介

SX-3568 是 ZLG 致远电子设计的一款中高端工控主板，采用国产化高端处理器平台，搭载四核 64 位 Cortex®-A55 处理器，主频高达 2.0GHz，同时配备双核心 GPU+ 高性能 VPU，支持 3D 图像引擎及 4K 高清显示，支持 4K@60fps 视频解码；拥有高能效神经网络 NPU，助力 AI 开发。此外 SX-3568 拥有众多接口资源，显示方面支持 LVDS、HDMI、eDP、MIPI-DSI 显示接口，支持 SATA、USB3.0、miniPCIe 等拓展接口，自带板载 WIFI&BT，另外还有丰富的 CAN、UART、I^2C 等通用接口，目前已经适配 Linux、Debain、Ubuntu、Android、OpenHarmony 操作系统，可广泛应用于智能 NVR、医疗设备、工业控制、车载中控、音视频系统等领域。

OpenHarmony编译准备

1. Ubuntu环境搭建

1.1 开发环境搭建

下载安装 Ubuntu20.04，打开终端，输入一下命令，安装编译环境：

sudo apt-get update -y

sudo apt-get install -y binutils git git-lfs gnupg flex openjdk-11-jdk \

bison gperf build-essential zip curl zlib1g-dev gcc-multilib g++-multilib \

libc6-dev-i386 libncurses5-dev x11proto-core-dev libx11-dev lib32z1-dev ccache \

libgl1-mesa-dev libxml2-utils xsltproc unzip m4 bc gnutls-bin python3 python2 \

python3-pip ruby genext2fs libssl-dev liblz4-tool device-tree-compiler jq libtinfo5

ln -sf /usr/bin/python3 /usr/bin/python

1.2 修改 Ubuntu shell 环境修改为 bash

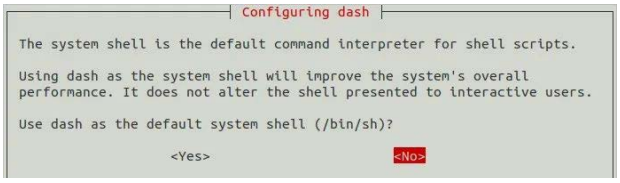
执行以下命令，确认输出结果是否为 bash：

ls -l /bin/sh

```
majicfu@majicfu-VirtualBox:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 12月  14 12:04 /bin/sh -> bash
majicfu@majicfu-VirtualBox:~$
```

如果输出为 dash, 执行以下命令，输入密码，选择 No，修改为 bash：

sudo dpkg-reconfigure dash



1.3 安装 DevEco Device Tool

下 载 DevEco Device Tool 3.0 ReleaseLinux 版 本：https://device.harmonyos.com/cn/ide#download，并解压：

unzip devicetool-linux-tool-3.0.0.401.zip

进入解压后的文件夹，执行如下命令，修改权限并安装：

chmod u+x devicetool-linux-tool-3.0.0.401.sh

sudo ./devicetool-linux-tool-3.0.0.401.sh

当出现 Deveco Device Tool successfully installed 时，软件安装成功。

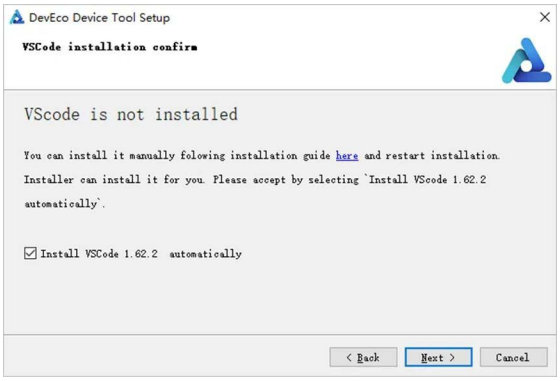
```
[INFO   ] Creating launch script...
[INFO   ] Creating setenv.sh script...
[INFO   ] Updating settings...
[INFO   ] Updating permissions...
[INFO   ] Updating u-dev rules...
[INFO   ] Installing mtd-utils...
Deveco Device Tool successfully installed.
```

2. Windows开发环境搭建

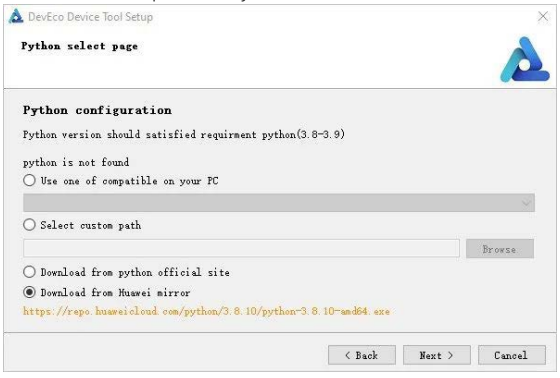
通过 Windows 系统远程访问 Ubuntu 环境，需要先在 Windows 系统中安装 DevEco Device Tool，以便使用 Windows 平台的 DevEco Device Tool 可视化界面进行相关操作。

2.1 安装 DevEco Device Tool 3.0 Release Windows 版：

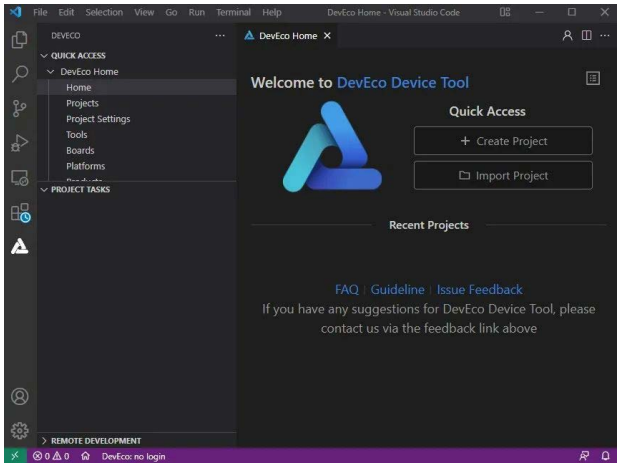
https://device.harmonyos.com/cn/ide#download，注意要安装到非系统盘上，安装时，勾选“Install VScode 1.62.2automatically”



在弹出 Python select page 选择 Download from Huawei mirror，点击 Next（如果系统已安装可兼容的 Python 版本（Python 3.8~3.9 版本），可选择“Use one of compatible on your PC”）。

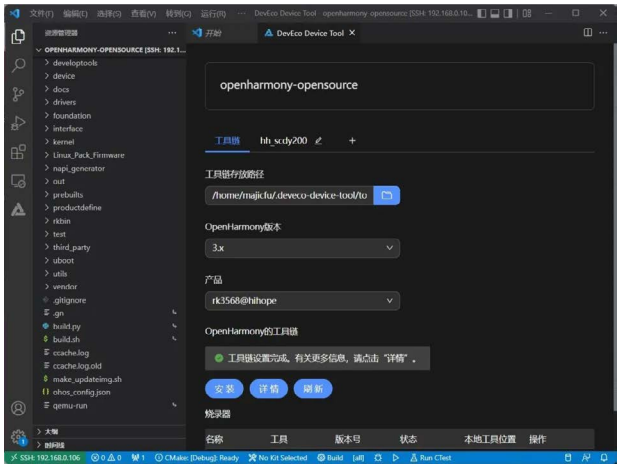


一直点击 Next，直到安装完成，打开 VS code，能进入 DevEco Device Tool 工具界面，软件安装完成。



2.2 通过 VS code SSH 远程连接 Ubuntu

在 VS code 中安装 SSH 服务，通过 SSH 服务远程连接 Ubuntu，连接成功后，电脑会自动下载安装插件，安装完成后，界面如下：



3. 源码编译

3.1 下载源码

我司提供基于 OpenHarmony 3.2 beta 版移植的 SDK 源码，如有需要，可联系 FAE 获取。

3.2 对源码进行分卷校验

md5sum -c ./md5sum.md5

```
→  Ubuntu 20.0  md5sum -c ./md5sum.md5
m3568-ohos-opensource.tar.bz2.000: OK
m3568-ohos-opensource.tar.bz2.001: OK
m3568-ohos-opensource.tar.bz2.002: OK
m3568-ohos-opensource.tar.bz2.003: OK
m3568-ohos-opensource.tar.bz2.004: OK
```

3.3 合包解

cat m3568-ohos-opensource.tar.bz2.0* | tar -jxv -C your_path

3.4 安装相关环境

在源码根目录下，执行以下命令，检查环境是否安装完成，未安装的程序会自动安装：

bash build/prebuilts_download.sh

3.5 固件编译

在源码根目录，执行以下命令，开始编译鸿蒙固件：

./build.sh --product-name rk3568 –ccache

编译时间取决于电脑性能，预计 1-4 小时，编译完成后，提示 build success。

```
[OHOS INFO] rk3568 build success
[OHOS INFO] cost time: 0:45:10
=====build  successful=====
2022-12-19 13:05:54
++++++++++++++++++++++++++++++++++++++++
majicfu@majicfu-VirtualBox:~/zlg/sdk/openharmony-opensource$ ls
```

编译完成的固件，存放目录为：

your_path/m3568_openharmony/out/rk3568/packages/phone/images

在此目录下，包含以下固件：

boot.img MiniLoaderAll.bin parameter.txt resource.img

system.img uboot.img updater.img userdata.img vendor.img

3.6 将固件打包成整包烧录

在源码根目录下，执行 make_updateimg.sh 脚本，打包固件

./ make_updateimg.sh

打包完成后，即可在固件存放目录下找到 update.img。

4. 成果展示

固件烧录完成后，开机，查看效果：

【解决方案】
如何给核心板的底板设计电源?

原创 研发部 ZLG 致远电子 2023-01-12 11:30:48

在选型某一款核心板后，硬件工程师将会针对自己的产品应用设计底板，除了外设的功能电路外，底板电源设计也是一款产品好坏的关键。结合 ZLG 致远电子的核心板产品，本文将对底板的电源设计提供一些方法及参考。

确定各供电电源的电压和电流

底板的供电电源可以简单划分为核心板的电源和其它外设的电源，如图 1 所示。ZLG 致远电子的核心板一般只需底板供一路电源，且一般为 5V，电流则各款核心板各有不同，如 M1107 系列核心板要求电流为 0.8A(峰值电流)。其它外设电源的电压、电流则需要根据外设的供电需求确定，如显示屏、摄像头模块、以太网、USB 设备等供电。底板连接核心板 IO 口时是要电平匹配的，所以外设至少包括与核心板 IO 口电平相等的电压，如 M1107 系列核心板的 3.3V，M3568 系列核心板的 3.3V、1.8V。

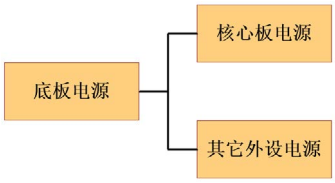


图1 底板供电电源

确定底板电源的架构

输入电源不同、外设不同，底板的电源架构会有所不同，但基本可套用图 2 中的电源架构，具体可根据实际应用增加或变化。图 2 采用的是两级电源架构，一级电源由输入转 5V 供给核心板及其它外设，二级电源由 5V 转各外设需求的电压。

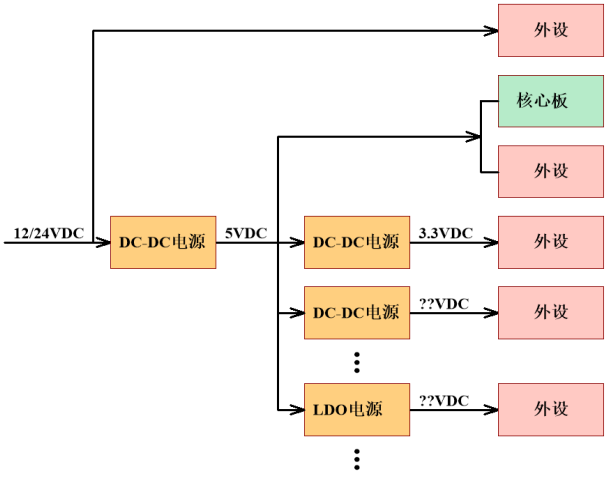


图2 两级电源架构

实际应用时，外设本身需求电流小且要求纹波小通常选用 LDO 供电，如果需求电压还很低，如 1.2V，这时还是 5V 转 1.2V 就不太适合了，3.3V 转 1.2V 或 2.5 转 1.2V 效率更高，这时就得在图 2 后面加第三级电源。同样如果底板的输入需求是交流市电，则需要在图 2 前面加一级电源，如图 3 所示。

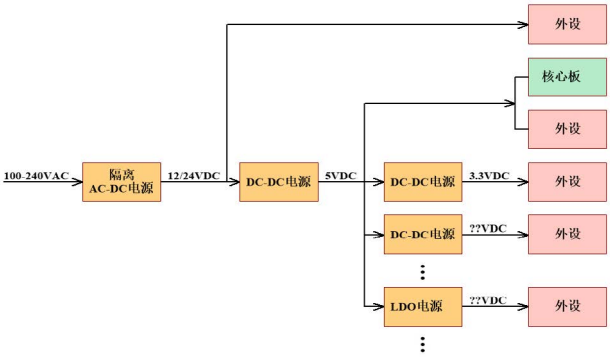


图3 交流市电输入三级电源架构

选型电源芯片，设计电源电路

根据前面确定的各供电电源的电压、电流及电源架构，可以选型满足需求的电源芯片，设计电源电路。DC-DC 电源具有效率高、转换压差大、电流输出能力强的优点，但纹波和噪声较大；LDO 电源具有噪声小、电路简单的优点，但效率较低、输出电流能力较弱。下面针对图 2 的电源架构简单讲解电源电路的设计。

1. 图 2 中一级电源需要将高电压转换到低电压，同时输出电流较大，一般采用 DC-DC 电源，可根据输入电压、输出电压和负载电流选择合适的 DC-DC 电源芯片，设计电源电路。图 4 为 M1107 系列评估底板的电源设计，一级电源采用的是非隔离降压 DC-DC 芯片 , 该芯片的输入电压范围为 4.5V-28V, 输出电流 3A，符合 12V 输入和 5V/2A 输出的要求。

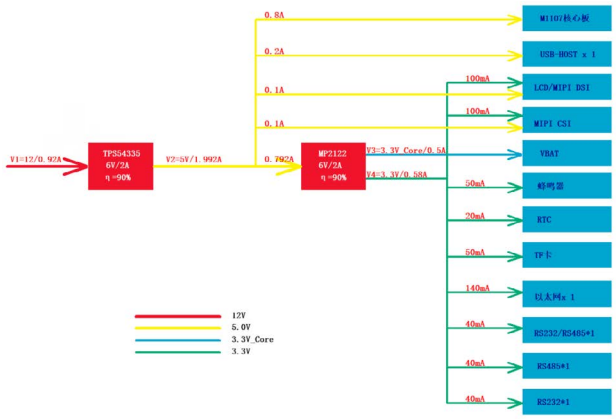


图4 M1107系列评估底板的电源设计

在一些干扰比较大或底板本身对干扰要求比较严格的应用场景，一级电源可以考虑采用隔离电源方案，图 5 中的一级电源使用的是 ZLG 致远电子的 E_UHBD-15W 系列隔离电源的 E2405UHBD-15W, 隔离电压 1500VDC, 可有效解决系统因静电、浪涌而导致供电不稳定的问题。

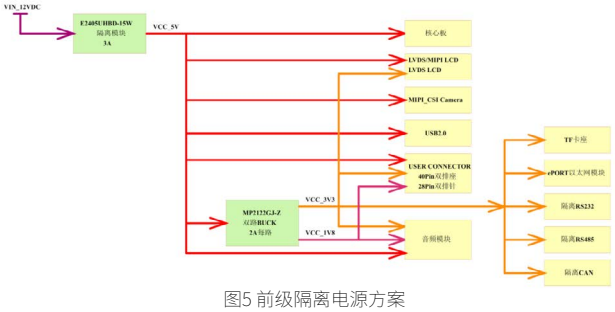


图5 前级隔离电源方案

2. 图 2 中的二级电源主要是考虑用非隔离的 DC-DC 电源还是 LDO 电源，一般数字信号、压差大、载流较大的选 DC-DC 电源，而模拟信号、压差小、载流小的选 LDO 电源。如果是采用 DC-DC 电源和一级电源的设计方法一样，如果是 LDO 则应考虑 (压差 X 载流) 是否满足热阻、温升及功耗等参数。

底板和核心板组合上电时序的设计

ZLG 致远电子的核心板一般是 5V 供电的，底板的 5V 上电后核心板的其它电源会按照设计的上电时序依次上电，这些电源包括了 IO 口电平，IO 口是跟底板的外设相连的，为了电平匹配，底板也同时设计了相等的电压。那问题来了，是核心板的 I0 口电平先上电还是底板的 IO 口电平先上电？底板的 IO 口电平能不能早于核心板的 IO 口电平上电？如果底板的 IO 口电平更早上电，底板的 IO 口电流有可能通过 IO 口倒灌回核心板的 CPU 或其它外设，进而导致核心板的上电时序逻辑混乱，或者启动模式引脚状态检测错误导致无法启动。所以底板设计也需要考虑上电时序的问题，但也相对简单，设计时底板的外设供电比核心板的 IO 口电平上电迟就行，一般有两种设计方法，如下。

1. RC延时使能

一般核心板的 IO 口电平为 3.3V 或 1.8V, 它们是经过 5V 降压转换得来，而底板的 IO 电平也是 5V 降压转换得来，既然是同一路电源转换得到，可以在底板设计时用 RC 延时电路接到电源芯片使能脚，计算 RC 的延时时间大于核心板的 IO 口电平上电时间。图 6 为 M1107 系列评估底板 3.3V 的设计，根据下面公式算得延时时间 td 为 13.28ms，公式中 VEN 为电源芯片的高电平使能阈值电平，该芯片为 1.2V。同时该芯片的软启动时间为 0.5ms，所以总延时为 13.78ms，查看 M1107 系列核心板，3.3V 的延时为 0.77ms，满足底板 IO 口电平上电时间大于核心板 IO 口电平上电时间的要求。

$$
\begin{aligned}
t_d &= RC \cdot \ln \frac{V_{IN}}{V_{IN} - V_{EN}} \\
&= 22 \times 2.2 \times \ln \frac{5}{5\text{-}1.2} \\
&= 13.28\text{ms}
\end{aligned}
$$

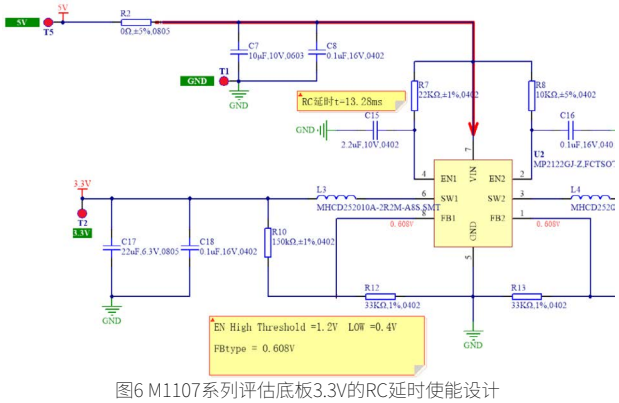


图6 M1107系列评估底板3.3V的RC延时使能设计

2. 利用核心板的PWR_OK引脚使能电源芯片

ZLG 致远电子的一些核心板会引出 PWR_OK(或 PGOOD) 引脚，设计该引脚的目的就是为了底板的上电时序，该引脚在核心板 IO 口电平上电完成后会输出高电平，可以将该引脚连到底板的电源芯片使能引脚，使底板的 IO 口电平后上电。图 7 为 M3568 系列核心板的评估底板 3.3V 的电源设计。

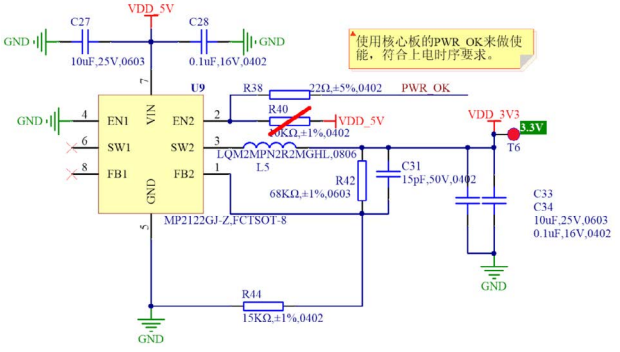


图7 M3568系列评估底板3.3V的PWR_OK使能设计

【技术分享】
深入解读无线通信中的天线① — 初识天线

原创 研发部 ZLG 致远电子 2023-01-10 11:31:00

天线作为无线信号辐射和接收的重要器件，在无线通信中起着关键作用。天线究竟是如何实现信号从有线到无线的转换的？天线都有哪些关键参数，又该如何评估一款天线的性能？本系列文章带你深入了解。

在无线电设备中，天线就是用来辐射和接收无线电波的装置，是一种电与磁的能量转换器。

按方向性分类，天线分为全向天线和定向天线两种。全向天线将能量信号平均辐射到所有方向上，由于能量被分散了，传输距离也较短。而定向天线则将能量信号辐射到特定的方向上，由于能量更集中，因此在该方向上传输距离会更远。

图1 定向天线和全向天线

按材质或结构，天线又可以分为许多种类，常见的是：PCB 天线（板载天线）、陶瓷天线、棒状天线等。致远电子推出的 ZLG52810 蓝牙模块，使用的就是 PCB 天线，这类天线集成在产品内部，可以大大减小对客户产品尺寸的要求。

蓝牙5.0系列透传模块/评估套件ZLG52810系列

那么，要如何评估一款天线性能的优劣？下面介绍天线的几个主要参数：

1. 工作频率

工作频率是天线最基本的参数，代表该天线能够辐射或接收的信号频率。天线的工作频率一般是某个范围，这个范围称为天线的带宽。例如某个天线的带宽是 2.3GHz~2.5GHz，则它能够将该频段内的信号有效辐射出去或接收进来，而该频段外的信号例如 2GHz，则无法通过该天线辐射或接收。

不同技术的产品，需要选择相应工作频段的天线，才能正常工作，例如：

- 蓝牙是 2.402~2.480GHz；
- Wi-Fi 是 2.412~2.472GHz；
- Lora 是 470~510MHz。

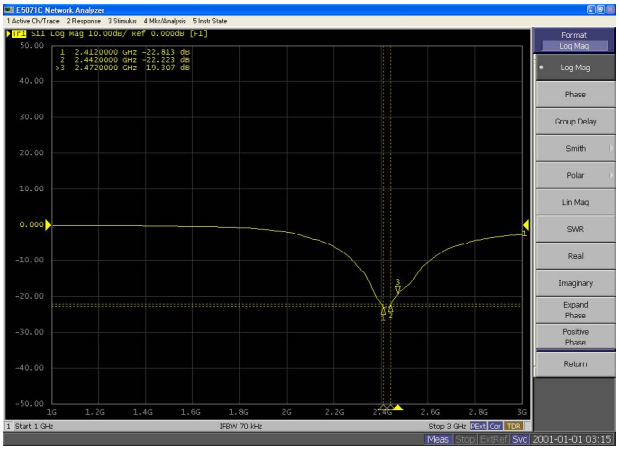


图2 Wi-Fi天线的工作频率测试

2. 增益

天线是无源器件，它并不会增大信号强度。和 PA 的增益不同，天线的增益通常指最大辐射方向的功率增益值，可以理解为天线在特定方向上的辐射能力，增益越大，天线辐射的能量也越集中，在相应方向上辐射能力越强，信号传输距离越远。

广州致远电子推出的 ZM602 系列 Wi-Fi 模块所设计的 PCB 天线增益达到了 3.3dBi，空旷环境下最远通讯距离达到了 450m，传输距离优于市场上绝大部分的 Wi-Fi 产品。

3. 电压驻波比

电压驻波比（VSWR）是表征端口阻抗匹配程度的一个量，它是衡量射频功率从功率源通过传输线到负载（天线）的效率，是驻波中最大电压与最小电压之比。

驻波比等于 1 时，表示馈线和天线的阻抗完全匹配，此时高频能量全部被天线辐射出去，没有能量的反射损耗；驻波比为无穷大时，表示全反射，能量完全没有辐射出去。VSWR 在无线模块设计中，应小于或等于 2.0。

天线馈电点两端感应的信号电压与信号电流之比，称为天线的输入阻抗，通信领域中标准阻抗值为 50Ω。

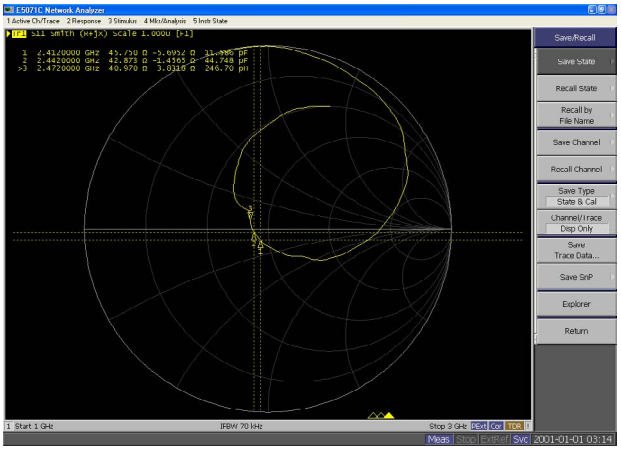


图3 天线的输入阻抗测试

【产品应用】LoRa网关与二次开发终端的神仙搭配

原创 研发部 ZLG 致远电子 2023-01-31 11:31:54

LoRa 项目开发时间紧？驱动开发困难？二次开发终端 ZSL420 提供了丰富的 API 接口，与 LoRa 网关搭配可以快速实现无线数据转 MQTT，有效降低产品成本、缩小开发周期，快速投入市场。

ZLGLink SDK二次开发简介

ZLGLink SDK 包是广州致远电子开发的 LoRa 智能组网二次开发包。该 SDK 主要包含有芯片外设 demo、LoRa 裸驱动 demo、ZLGLink 智能组网协议 demo。提供 eclipse 和 keil 两种开发环境。支持本地升级与远程无线升级。丰富的示例 demo 可以帮助开发者快速上手，缩短 LoRa 终端的开发周期。

表1 ZLGLink智能组网协议demo简介

序号	Demo 类别	说明
1	burst_report	突发上报型应用，主动组网，MCU 低功耗
2	period_report	分时上报型应用，主动组网，MCU 低功耗
3	period_wake	周期唤醒型应用（主机可空中唤醒休眠终端），主动组网，MCU 低功耗
4	lpuart_wake	低功耗串口唤醒型应用（低功耗串口唤醒休眠终端），主动组网，MCU 低功耗
5	join_white	入网白名单型应用（主机过滤非白名单入网设备），主动组网，MCU 低功耗
6	period_wake_burst_report	按键唤醒型应用（休眠终端支持主机空中唤醒和终端按键唤醒），主动组网，MCU 低功耗

表2 常用ZLGLink API简介

序号	API	说明
1	aw_ntl_dev_type_set()	通过该接口可以设置主机、从机和中继等设备类型，以区分星型网络中不同的设备
2	aw_ntl_dev_work_mode_set()	根据不同的功耗需求，可以通过该接口设置从机的工作模式（一般模式，周期唤醒模式和深度休眠模式）
3	aw_ntl_dev_networking_create()	手动创建网络接口，根据用户指定通信信道创建网络
4	aw_ntl_dev_networking_auto_create()	自动创建网络接口，主机自动选择空闲信道创建网络
5	aw_ntl_dev_networking_joined_enable()	主机允许从机入网接口
6	aw_ntl_dev_networking_join()	从机发起入网接口
7	aw_ntl_sendto()	数据发送接口
8	aw_ntl_recvfrom()	数据接收接口

MQTT客户端采集二次开发终端温度数据

MQTT 客户端向终端发布温度采集指令，终端在接收到温度采集指令后上报温度数据，MQTT 客户端采集终端温度数据示意图，如图 1 所示。

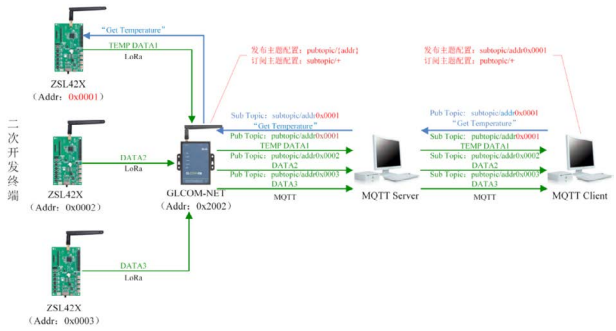


图1 MQTT客户端采集终端温度数据示意图

1. 网关配置

网关配置主要包括协议转换配置，MQTT 配置和网关 ZLGLink 配置三个部分。

1.1 协议转换配置

打开网关的网页配置页面，左边栏选择【协议转换】，在转换选择页面里选择【ZLGLink 转 MQTT】，然后点击【保存】，网关自动重启，如图 2 所示。

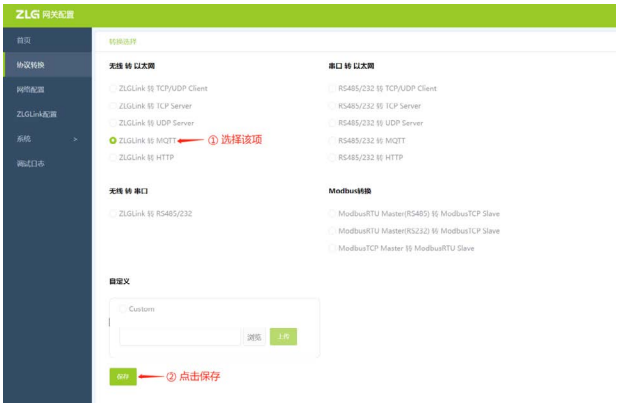


图2 选择ZLGLink转MQTT

1.2 MQTT 配置

左边栏选择【网络配置】，在网络配置里选择【MQTT】标签页。

在【基本配置】里配置网关需要连接的 MQTT 服务器的地址、端口号、MQTT 用户名、密码和客户端 ID。

在【订阅主题】里配置网关订阅的主题和服务质量。

在【发布主题】里配置网关发布消息的主题和服务质量，如图 3 所示。

图3 配置MQTT

1.3 ZLGLink 配置

左边栏选择【ZLGLink 配置】标签页。

在【基本配置】里配置 ZLGLink 的基本配置参数。

在【组网控制】里使能自组网并允许入网，如图 4 所示。

图4 ZLGLink配置

2. ZLGLink 终端二次开发配置

2.1 从机应用选择使用 ZLGLink SDK 中的突发上报从机 demo 并设置温度采集指令，如图 5 所示。

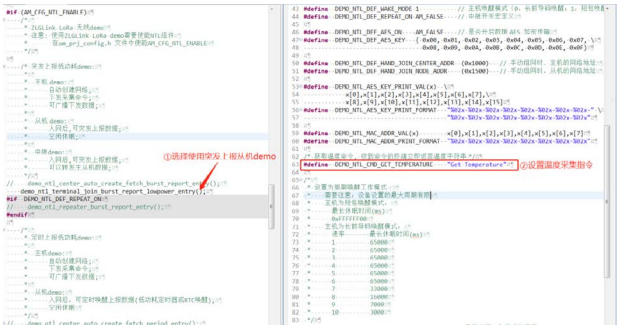


图5 从机应用选择

2.2 在网关开启允许入网功能后 ZSL420-EVB Demo 板通过按键 SW1 开启入网，如图 6 所示。

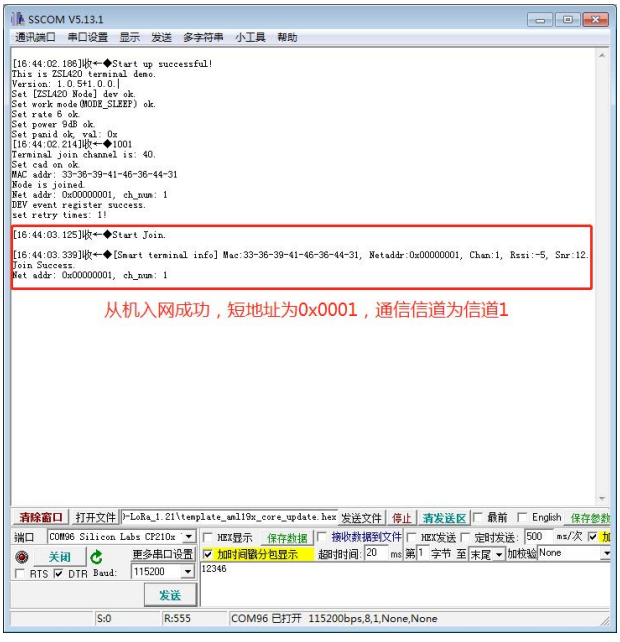


图6 从机入网到网关

2.3 MQTT 客户端配置

配置 MQTT 客户端需要连接的 MQTT 服务器的地址、端口号、MQTT 用户名、密码和客户端 ID。本文以 MQTT.fx 客户端上位机做演示，如图 7 所示。

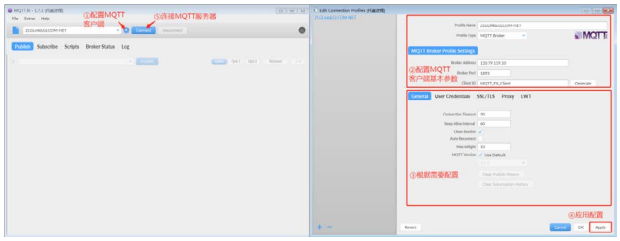


图7 配置MQTT客户端

2.4. 成果展示

1.MQTT 客户端订阅主题客户端数据主题：pubtopic/+；

2.MQTT 客户端向终端（0x0001）发布温度采集指令主题：subpotic/addr0x0001，温度采集指令“Get Temperature”；网关接收后下发至终端（0x0001），终端再将温度数据上报给网关，MQTT 客户端就能收到 ZSL420-EVB Demo 板温度数据，如图 8 所示。

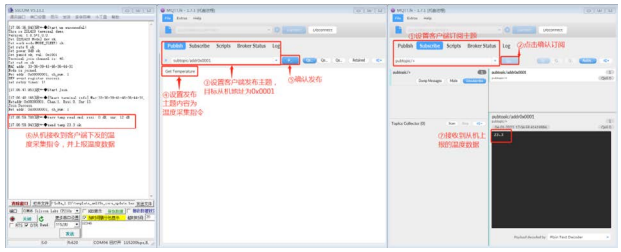


图8 MQTT客户端采集LoRa终端温度数据

【技术分享】RS-485自动收发应用异常怎么办?

原创 FAE 工程师 ZLG 致远电子 2023-01-05 11:30:58

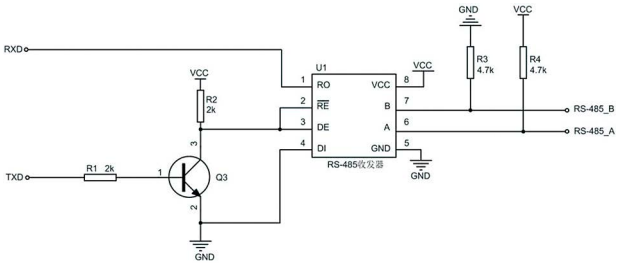
一般 RS485 收发器会有两个引脚来控制数据的收发方向，如果添加外围电路将其设计成自动收发状态，会有什么常见问题？有没有一款产品带自动收发功能，又可以避免这些问题？

RS485自动收发的原理

在各种通讯方式中，RS485 总线是较为常见的一种，因其接口简单，组网方便等特点，在工业控制、仪器、仪表、多媒体网络、机电一体化产品等诸多领域得到广泛应用。

MCU 通信一般使用 TTL 电平，如果外接设备使用的是 485 电平，那么两者是无法直连进行通讯的，必须通过 485 收发器，进行电平转换。由于 485 通信是半双工通信，也就是说，数据不能同时进行收发，所以 485 收发器通常会有控制收发方向的引脚。

下面我们来看一下，485 收发器实现自动收发的外围电路设计。



从原理图中可以看出，自动收发主要是通过 NPN 三极管开关电路来实现，具体的数据收发过程是怎样的呢?

发送数据时

发送数据时，使用的是 MCU 的 TX 引脚，假设我们想要发送数据 0x55，那么转换成二进制就是 0b01010101，即在 TX 引脚上就体现为高、低电平之间的相互切换。

当 TX 引脚为 0 时，三极管不导通，DE 为高电平，进入发送模式。因为 DI 引脚接地，那么此时 AB 之间的差分电平逻辑就为 0;

当 TX 引脚为 1 时，三极管导通，RE 为低电平，进入接收模式。此时收发器的 A、B 引脚进入高阻态，因为上拉电阻 R4、下拉电阻 R3 的作用，此时 AB 之间的差分电平逻辑为 1。

所以保证了 TX 引脚输出什么电平，AB 之间的差分电平逻辑也保持一致。

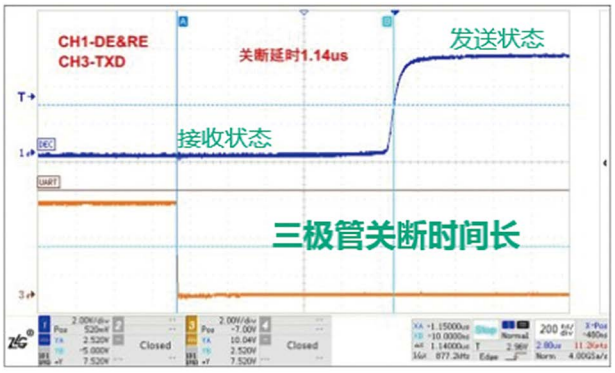
接收数据时

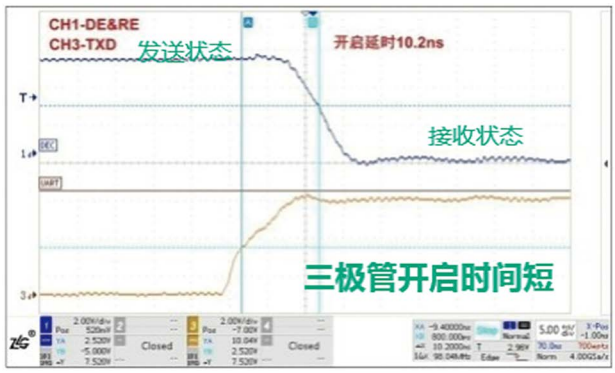
接收数据时，使用的是 MCU 的 RX 引脚。在接收数据过程中，TX 引脚保持高电平，三极管导通，RE 为低电平，进入接收模式，RX 引脚会接收 AB 传输过来的数据。

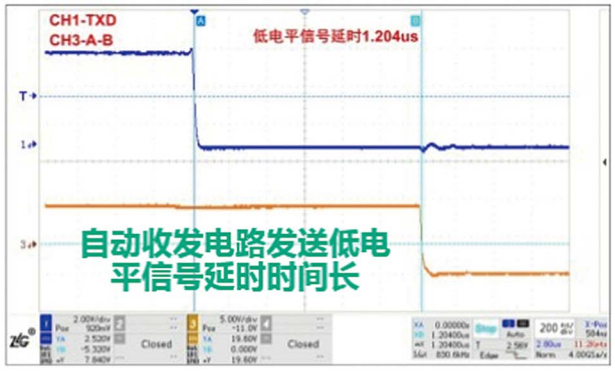
自收发485电路常见问题

1. 通信速度慢

三极管的开启延时为 ns 级别，关断延时为 us 级别，会导致收发电路发送低电平的延时时间较长。其次高电平的发送是通过外部上下拉电阻驱动的，电阻越大，上升沿越缓慢。

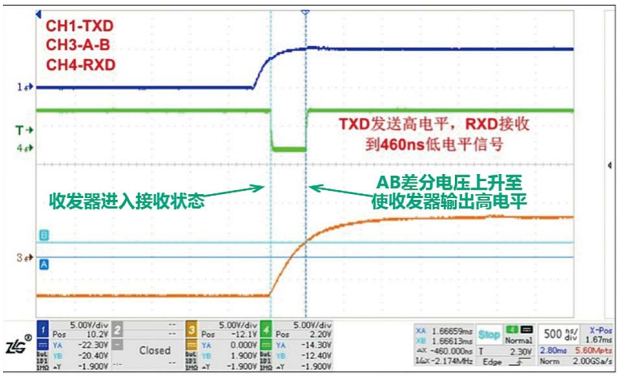






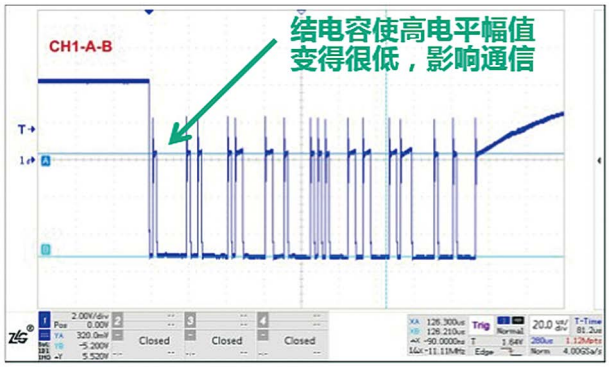
2. 高波特率通信时存在通讯风险

假设 TX 引脚上一个发送的 bit 为 0，即将发送的 bit 为 1，由于高电平的发送是通过外部上下拉电阻驱动的，收发器会切换到接收状态。此时 AB 线从低电平切换到高电平需要几百 ns，RX 引脚在这段时间内会接收到 0。如果波特率太高，RX 引脚接收到的低电平会被误认为是接收的起始位，导致通讯异常。



3. 外围电路接结电容影响收发器通讯稳定性

高电平的发送是通过外部上下拉电阻驱动，高电平输出缓慢，如果外部保护电路的结电容又较高，会导致 AB 差分电压幅值较低，当幅值低于门限电平时，会导致通讯异常。



致远电子解决方案

那么有没有自带“自动收发切换”且能克服以上常见使用问题的产品呢?

致远电子的 RSM485M、RSM(3)485PHT 给你答案。

致远电子 RSM 系列隔离收发器是一款应用于工业 RS-485 总线传输及隔离的模块产品，能有效解决总线干扰、通信异常等问题。与传统的设计相比， RSM 系列产品内置完整的隔离 DC-DC 电路、信号隔离电路、 RS-485 总线收发电路以及总线防护电路， 具备高集成度与可靠性，能够有效帮助用户提升总线通信防护等级。其中 RSM485M、RSM(3)485PHT 带有自动流控功能。

隔离RS-485收发器RSM485M

- 自动流控
- 超小体积
- 带隔离输出电源脚
- 最多可连接 64 个节点
- 最大波特率 500kbps
- 电磁辐射 EME 极低
- 电磁抗干扰 EMS 极高
- 集成电源隔离和信号隔离

隔离RS-485收发器RSM(3)485PHT

- 自动流控
- 单一输入电源供电
- 具有隔离输出电源脚
- 自动收发数据功能
- 最多可连接 128 个节点
- 电磁辐射 EME 较低
- 电磁抗干扰 EMS 较高
- 集成电源隔离、信号隔离和总线 ESD 保护功能
- 通过 IEC62368、UL62368、EN62368 认证

全工况隔离DC-DC电源芯片
P0505FT-1W

点击购买

【产品应用】匠“芯”升级，为客户降低制造成本而生

原创 研发部 ZLG 致远电子 2023-01-11 11:35:35

P 系列产品推出基于 SiP 工艺封装技术的产品，打造出全工况隔离 DC-DC“芯片级”电源模块，实现更高的集成度；DFN 表贴封装，为客户提供更多的设计空间，同时降低客户的生产制造成本。

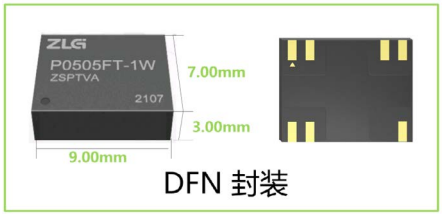
打破常规，“芯片级”电源模块

在以往的定压系列产品的升级迭代过程，都是从改良电路性能、升级生产工艺技术水平为手段，而产品的封装工艺仍然采用传统的灌封形式，产品的结构与外观没有突破性的改变，基本以 SIP 与 DIP 封装形式为主。

SIP封装		
尺寸大小	11.60x6.00x10.10	19.65×7.05×10.10
DIP封装		
尺寸大小	12.70×10.00×7.70	20.00×10.60×8.10

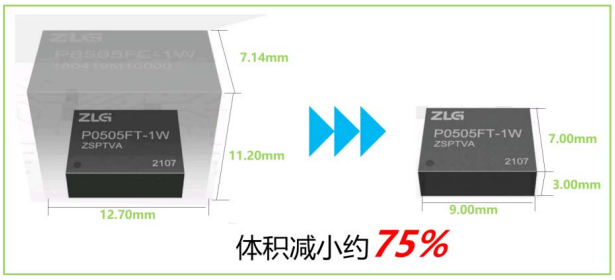
“芯片级”电源模块是基于新一代自主开发、具有完全知识产权的 IC 芯片而设计的产品，解决了行业内小功率电源模块的难题：容性负载能力差、转换效率低、无短路保护功能、静态功耗高等等。重大的技术创新突破点，就是彻底摒弃传统的灌封工艺，采用 SiP（System In Package）系统级封装技术，在体积与封装结构上与传统灌封电源模块完全不同，产品外观由原来的简单粗糙的灌封产品往芯片级精细化、小型化转变，是集电路技术、工艺技术、材料技术于一体的结晶。

参数	条件	P0505FT-1W
技术应用	--	自主研发IC
输入电流(mA)	满载/空载	235/10
轻载效率(%)	10%负载	69
满载效率(%)	Min/Typ	81/83
线性调整率	输入电压变化±1%范围	±1.2
负载调整率(%)	负载从10%~100%变化	±7
短路保护功能	--	可持续短路保护
容性负载(μF)	输出满载	1800
工作环境温度(℃)	--	-40~105
封装尺寸	9.00x7.00x3.00mm	



体积微型化，为设计提供更多的空间

自主的 IC 方案与成熟 SiP 工艺的整合，推出了高度集成化的超小、超薄的 DFN 封装的供电解决方案。比传统的灌封工艺产品，体积减少 75%，厚度仅 3.00mm，占板面积也减小了 55% 以上，这对于正在为体积受限而烦恼的设计人员来说，无疑是雪中送炭，例如手持设备、便携设备都是对体积要求非常高的行业。



DFN封装，为客户降低制造成本

传统灌封的电源模块为直插型，客户在产品生产制造环节既要有 SMD 工艺回流焊制程，又要有插件工艺波峰焊制程，造成生产制造工艺的复杂性，这不仅导致生产周期变长、增加制造成本，甚至还会引发产品品质异常风险，降低产品的可靠性。

P_FT-1W 系列产品采用 SiP 工艺技术，塑封产品为 DFN 封装，支持全自动贴片生产，更便捷的应用方式，有效地提升客户的生产效能，同时大幅降低由人工干预造成的应用问题，提升客户整体产品的稳定性，从而降低客户的生产制造成本。

产品的应用示例

P系列全工况电源应用示例

- 运算放大器电源；
- 继电器驱动电路；
- 医学、手持、便携仪表；
- 大功率IGBT驱动；
- 纯数字电路、模拟前端隔离电路；
- 一般低频模拟电路；
- 数据交换电路；
- 定制类供电解决方案；
- ……

2023/1 第1期

微文摘

ZLG MICRO DIGEST

ZLG致远电子官方微信

广州致远电子股份有限公司

更多详情请访问 www.zlg.cn

欢迎拨打全国服务热线 400-888-4005