### I/O系统

#### **9.1 简介**

I/O系统用于连接计算机与外部设备（peripherals）。在个人计算机里，这些设备一般包括键盘、显示器、打印机和无线网络。在嵌入式系统里，这些设备可能包括吐司机的加热元件、玩偶的声音同步器、引擎的燃料注入器、卫星的太阳能面板定位马达，等等。处理器就像访问内存一样使用地址和数据总线访问I/O设备。

本章提供了I / O设备的具体示例。 9.2节展示了将一个I / O设备连接到处理器并通过程序访问它的基本原理。9.3节 介绍了嵌入式系统环境中的I / O，展示了如何使用基于ARM的Raspberry Pi单板计算机访问板载外设，包括通用，串行和模拟I / O以及定时器。 9.4节给出了与其他常见设备（如字符LCD，VGA监视器，蓝牙无线电和电机）连接的示例。 9.5节描述了总线接口，并阐述了流行的AHB-Lite总线。 9.6节调查了PC中使用的主要I / O系统。

#### **9.2 内存映射I/O**

回忆一下6.5.1节，一部分的地址空间用于I/O设备而非内存。例如，设定0x20000000到0x20FFFFFF范围内的物理地址用于I/O。每一个I/O设备可以指定到这一范围中的一个或者多个地址上。对于特定地址的写入操作就会给设备发送数据，读出操作就将从设备接收数据，这种I/O设备通讯方式称为内存映射I/O（memory-mapped I/O）。

在内存映射I/O系统里，读出和写入可能访问内存，也可能访问I/O设备。图e9.1给出了支持两个内存映射I/O设备所需要的硬件。其中，地址译码器决定处理器与哪个设备进行通讯，根据Address和MemWrite信号来产生对其它硬件的控制信号。ReadData多路选择器在内存和各种I/O设备间选择读出数据。写使能寄存器保存了写入到I/O设备的值。

图e9.1 地址映射I/O所需的硬件

Processor：微处理器

Address Decoder：地址译码器

Memory：存储器

I/O device: I/O设备

例 e9.1 与I/O设备通讯

假设图e9.1中的I/O设备1被指定到了内存地址 0x20001000。写出把值7写入到I/O设备1和从I/O设备1读出输出值的ARM汇编代码。

解：以下汇编代码把值7写入到I/O设备1.

MOV R1, #7

LDR R2, = ioadr

STR R1, [R2]

ioadr DCD 0x20001000

因为地址为0x20001000，地址译码器将设置WE1，同时设置MemWrite 为TRUE。WriteData总线的值（7）被写入到与I/O设备1输入管脚相连的寄存器中。

为了从I/O设备1读出，处理器应执行以下的汇编代码。

LDR R1, [R2]

因为地址译码器发现地址位0x20001000且Mem Write为FALSE，所以它把RDsel1:0设置为01。I/O设备1的输出通过多路选择器传递到ReadData总线，然后被读入到处理器的R1中。

与I/O设备相关联的地址通常称为I/O寄存器，因为它们可能与I/O设备的物理寄存器一致，如图e9.1中所示。

与I/O设备通讯的软件称为设备驱动程序（device driver）。读者可能已经下载或者安装过打印机或者其他I/O设备的设备驱动程序。编写一个设备驱动程序需要详细了解I/O设备硬件知识，包括内存映射I / O寄存器的地址和行为。其他程序调用设备驱动提供的函数来访问设备，而不需知道底层设备的硬件。

#### **9.3 嵌入式I/O系统**

嵌入式系统使用一个处理器来控制与物理环境的交互。嵌入式系统一般围绕着微控制器单元（microcontroller unit，MCU）来构造，MCU结合了一个微处理器与一组容易使用的外围设备，例如通用数字和模拟I/O引脚、串行端口、计时器等。微控制器通常是廉价的，并且通过集成大部分必要部件到单一芯片上来使系统成本和尺寸最小化。大多数嵌入式系统比一角硬币更小、更轻，功率只有几毫瓦，成本从几角到几美元不等。微控制器依据处理的数据量大小来进行分类。8位微控制器是最小和最便宜的，而32位微控制器则提供更多内存和更高性能。

为了具体化，本节将介绍实际系统环境中的嵌入式系统I / O。具体来说，我们将重点放在流行且廉价的Raspberry Pi板，其中包含了一个Broadcom BCM2835片上系统（SoC）和一个700 MHz 32位的实现了ARMv6指令集的ARM1176JZ-F处理器。每个小节在原理之后都跟着在Pi上运行的具体示例。2014年，所有这些例子都在在运行NOOBS Raspbian Linux的Pi上进行了测试。

图e9.2显示了一张Raspberry Pi Model B +板的照片，这是一台完整的Linux计算机，大小与信用卡大小相当，售价35美元。 Pi板从5 V USB电源吸取高达1 A的电流。 它有512 MB的板载RAM和一个SD卡插槽，用于含有操作系统和用户文件的存储卡。 连接器提供视频和音频输出，鼠标和键盘的USB端口，以太网（局域网）端口，以及作为本章主要内容的40个通用I / O（GPIO）引脚。

虽然BCM2835 SoC具有许多功能，而不是典型的廉价微控制器，但在通用I / O方面非常相似。 本章首先介绍Raspberry Pi上的BCM2835，并描述内存映射I / O的设备驱动程序。 本章的其余部分将说明嵌入式系统如何执行通用数字，模拟和串行I / O。定时器也经常用于生成或测量精确的时间间隔。

图e9.2 Raspberry Pi Model B +板

USB Power：USB电源

HDMI Video：HDMI视频

Audio：音频

Ethernet：以太网

##### **9.3.1 BCM2835片上系统**

BCM2835 SoC是Broadcom为移动设备和其他多媒体应用设计的功能强大且价格低廉的芯片。 SoC包括一个称为应用处理器的ARM微处理器，一个用于图形，视频和摄像头的VideoCore处理器，以及许多I / O外设。 BCM2835采用塑料球栅阵列封装，下面有微小的焊球; 它最好由机器人焊接，该机器人将该封装与印刷电路板上匹配的铜焊盘对齐并施加热量。 Broadcom没有发布完整的数据表，但是Raspberry Pi网站上提供了一个简短的数据表，描述了如何从ARM处理器访问外设。 该数据表描述了本章中为简单起见而省略的许多功能和I / O寄存器。

[www.raspberrypi.org/documentation/hardware/](http://www.raspberrypi.org/documentation/hardware/)

图e9.3展示了Raspberry Pi模型板的简化示意图。 电路板通过USB电源接收5 V电源，稳压器产生3.3,2.5和1.8 V电平，用于I / O，模拟和其他功能。 BCM2835还具有内部开关稳压器，可为节能的SoC产生可变的低电压。 BCM2835连接到USB /以太网控制器，也可直接输出视频。 它还具有54个可配置的I / O信号，但由于空间原因，用户只能通过插头引脚访问其中的一小部分。 接头还提供3.3 V和5 V接地，方便为连接到Pi的小型设备供电，但最大总电流在3.3 V时为50 mA，5 V时为~300 mA。B型和B +类似，但B +将I / O插头引脚数量从26个增加到40个，USB端口数量从2个增加到4个。包括Adafruit Pi Cobbler在内的各种电缆可将这些插头引脚连接到面包板上。

Raspberry Pi使用SD卡作为闪存盘。 该卡通常预装了Raspbian Linux，这是一个适用于8 GB SD卡的小型Linux版本。 您可以通过连接HDMI显示器和USB鼠标和键盘将其转变为完整的计算机，或通过以太网电缆从另一台计算机连接到Pi来使用Pi。

图e9.3 Raspberry PI I/O 示意图

USB Power Jack：USB电源插孔

Regulator：调节器

HDMI Video：HDMI视频

RCA Video：RCA视频

SD Card：SD卡

Camera Link：相机连接处

Crystal Oscillator：晶体振荡器

Ethernet：以太网

I/O header：I / O头

Stereo Audio Jack：立体声音频插孔

##### **9.3.2 设备驱动程序**

程序员可以通过读取或写入存储器映射的I / O寄存器来直接操作I / O设备。 但是，调用访问内存映射I / O的函数是更好的编程习惯。 这些功能被称为设备驱动程序。 使用设备驱动程序的一些好处包括：

1. 当代码涉及到明确命名的函数调用而不是在晦涩难懂的内存地址处写入位字段时，代码更容易阅读。
2. 熟悉I / O设备深层工作原理的人可以编写设备驱动程序，普通用户可以调用它而无需了解细节。
3. 代码更容易移植到具有不同内存映射或I / O设备的另一个处理器，因为只有设备驱动程序必须更改。
4. 如果设备驱动程序是操作系统的一部分，则操作系统可以控制对在系统上运行的多个程序之间共享的物理设备的访问，并且可以管理安全性（例如，当您向网络浏览器输入密码时，恶意程序无法读取键盘）。

本节将开发一个名为EasyPIO的简单设备驱动程序来访问BCM2835设备，以便您可以了解设备驱动程序中的内容。 普通用户可能更喜欢WiringPi，一个用于Pi的开源I / O库，其功能类似但不完全对应EasyPIO中的功能。

BCM2835上的内存映射I / O位于物理地址0x20000000-0x20FFFFFF。 表e9.1总结了各种外设使用的物理基址。 外设从其基址开始有多个I / O寄存器。 例如，读地址0x20200034将返回GPIO（通用I / O）引脚31：0的值。 粗体显示的外设将在后续章节中进一步讨论。

Raspberry Pi通常运行使用虚拟内存的Linux操作系统，这进一步使得内存映射I / O复杂化。程序中的加载和存储是指虚拟地址，而不是物理地址，因此程序无法立即访问内存映射的I / O。相反，它必须首先要求操作系统将感兴趣的物理地址映射到程序的虚拟地址空间。例e9.2中EasyPIO的pioInit函数执行此任务。代码涉及C中的一些重型指针操作。一般原则是打开/ dev / mem，这是Linux一种访问物理内存的方法。然后使用mmap函数将gpio设置为指向物理地址0x20200000（GPIO寄存器的开头）的指针。该指针被声明为volatile，告诉编译器此内存映射的I / O值可能会自行更改，因此程序应始终直接读取寄存器而不是依赖其旧值。GPLEV0通过GPIO访问I / O寄存器13个字，例如在0x20200034，其中包含GPIO 31：0的值。为简洁起见，此示例省略了在实际EasyPIO库中进行的错误检查。后续小节定义了更多寄存器和函数来访问I / O设备。

表e9.1内存映射I / O地址

Physical Base Address：物理基址

Peripheral：外部设备

例 e9.2 初始化内存映射I/O

#include <sys/mman.h>

#define BCM2835\_PERI\_BASE 0x20000000

#define GPIO\_BASE (BCM2835\_PERI\_BASE + 0x200000)

volatile unsigned int \*gpio; //指向gpio基址的指针

#define GPLEV0 (\* (volatile unsigned int \*) (gpio + 13))

#define BLOCK\_SIZE (4\*1024)

void pioInit(){

int mem\_fd;

void \*reg\_map;

// /dev/mem 是一个访问Linux内存的伪驱动程序

mem\_fd = open("/dev/mem", O\_RDWR|O\_SYNC);

reg\_map = mmap(

NULL, // 开始本地映射的地址 (null = 不在意)

BLOCK\_SIZE, // 4KB映射内存块

PROT\_READ|PROT\_WRITE, // 启用读取和写入映射内存

MAP\_SHARED, // 非独占访问此内存

mem\_fd, // 映射到 /dev/mem

GPIO\_BASE); // GPIO 外设的偏移量

gpio = (volatile unsigned \*)reg\_map;

close(mem\_fd);

}

##### **9.3.3 通用数字I / O**

通用I/O（general-purpose I/O，GPIO）引脚用于读写数字信号。例如，图e9.4展示了连接到6个GPIO端引脚的三个发光二极管（light-emitting diode，LED）和三个开关。当用1驱动时，LED连线发光，在用0驱动时关闭。限流电阻与LED串联，以设置亮度并避免GPIO的电流能力过载。 开关接线闭合时产生1，打开时产生0。 原理图显示了引脚名称以及相应的标头引脚编号。

至少，任何GPIO引脚都需要寄存器来读取输入的引脚值，写入输出的引脚值以及设置引脚的方向。 在许多嵌入式系统中，GPIO引脚可以与一个或多个专用外设共用，因此需要额外的配置寄存器来确定引脚是通用引脚还是特殊引脚。 此外，当在输入引脚上发生诸如上升沿或下降沿的事件时，处理器可以产生中断，并且配置寄存器可以用于指明中断发生的条件。

回顾一下，BCM2835有54个GPIO。它们由GPFSEL，GPLEV，GPSET和GPCLR寄存器控制。图e9.5显示了这些GPIO寄存器的存储器映射。 GPFSEL5 ... 0确定每个引脚是否为通用输入，输出或专用I / O。这些功能选择寄存器中的每一个都使用3位来指明每个引脚，因此每个32位的寄存器能控制10个GPIO，如表e9.2所示，并且需要6个GPFSEL寄存器来控制所有54个GPIO。例如，GPIO13由GPFSEL1 [11：9]配置。配置总结在表e9.3中；许多引脚具有多个专用功能，将在后续章节中讨论; ALT0是最常用的。读取GPLEV1 ... 0将返回引脚的值。例如，GPIO14读取为GPLEV0 [14]，GPIO34读取为GPLEV1 [2]。引脚不能直接写入；然而，通过置GPSET1 ... 0或GPCLR1 ... 0的相应位可以将位强制为高电平或低电平。例如，通过写入GPSET0 [14] = 1强制GPIO14为1，通过写入GPCLR0 [14] = 1强制为0。

BCM2835数据表未指定GPIO的逻辑电平或输出电流能力。 但是，用户根据经验确定，不应尝试从任何单个I / O中抽取超过16 mA或从所有I / O中总共抽取超过50 mA。 因此，GPIO引脚适合驱动小型LED但不适合驱动电机。 这些I / O通常与其他3.3 V芯片兼容，但不具备5 V电压耐受性。

图e9.4 连接到GPIO引脚的LED和开关

Switches：开关

表e9.2 GPFSEL寄存器位字段和GPIO的映射

图e9.5 GPIO内存映射

表e9.3 GPFSEL配置

Pin Function：引脚功能

例e9.3 用于开关和LED的GPIO

使用pinMode，digitalRead和digitalWrite函数增强EasyPIO，以此来配置引脚的方向并读取或写入引脚。 使用这些函数编写C程序以读取三个开关并使用图e9.4中的硬件打开相应的LED。

解：下面给出了额外的EasyPIO的代码。 由于多个寄存器用于控制I / O，因此函数必须计算出需要访问的寄存器以及在寄存器中使用的位偏移量。然后，pinMode清除0位，为预期的3位功能设置1。digitalWrite使用GPSET或GPCLR来处理写入1或0。digitalRead读取所需引脚的值并屏蔽掉其他引脚。

#define GPFSEL ((volatile unsigned int \*) (gpio + 0))

#define GPSET ((volatile unsigned int \*) (gpio + 7))

#define GPCLR ((volatile unsigned int \*) (gpio + 10))

#define GPLEV ((volatile unsigned int \*) (gpio + 13))

#define INPUT 0

#define OUTPUT 1

...

void pinMode(int pin, int function) {

int reg = pin/10;

int offset = (pin%10)\*3;

GPFSEL[reg] & = ~((0b111 & ~function) << offset);

GPFSEL[reg] | = ((0b111 & function) << offset);

}

void digitalWrite(int pin, int val) {

int reg = pin / 32;

int offset = pin % 32;

if (val) GPSET[reg] = 1 << offset;

else GPCLR[reg] = 1 << offset;

}

int digitalRead(int pin) {

int reg = pin / 32;

int offset = pin % 32;

return (GPLEV[reg] >> offset) & 0x00000001;

}

读取开关和写入LED的程序如下。 它先初始化GPIO访问，然后将引脚2-4设置为开关的输入，将引脚7-9设置为LED的输出。 然后它连续读取开关并将其值写入相应的LED。

#include "EasyPIO.h"

void main(void) {

pioInit();

// 将GPIO 4:2设置为输入

pinMode(2, INPUT);

pinMode(3, INPUT);

pinMode(4, INPUT);

// 将GPIO 9:7设置为输出

pinMode(7, OUTPUT);

pinMode(8, OUTPUT);

pinMode(9, OUTPUT);

while (1) { // 读取每个开关并写入相应的LED

digitalWrite(7, digitalRead(2));

digitalWrite(8, digitalRead(3));

digitalWrite(9, digitalRead(4));

}

}

假设程序位于名为dip2led.c的文件中且EasyPIO.h位于同一目录中，则可以在Raspberry Pi命令行上使用以下命令编译和运行程序。gcc是C编译器。 请注意，必须使用sudo以便程序可以访问受保护的I / O内存。 要停止正在运行的程序，请按Ctrl-C。

gcc dip2led.c –o dip2led

sudo ./dip2led

##### **9.3.4 串行I / O**

如果微控制器需要发送比GPIO可用引脚数更多的位数，它必须把消息拆分成多个较小的传输。在每个步骤中，它可以发送任意一个比特或几个比特。前者称为串行I/O（serial I/O），后者称为并行I/O（parallel I/O）。串行I/O的普及是因为它使用少量电线和对很多应用来说足够快。事实上，它的普及程度高到建立了多个串行I/O标准，通过这些标准，微控制器的专用硬件能方便地发送数据。本节介绍串行外设接口（SPI）和通用异步接收器/发送器（UART）标准串行接口。

其它常用串行标准包括内部集成电路（ Inter-Integrated Circuit，I2C）、通用串行总线（Universal Serial Bus，USB）和以太网。 I2C是包含一个时钟和一个双向数据引脚的双线接口；它的使用方式与SPI类似。USB和以太网较为复杂，高性能标准分别在9.6.1节和9.6.4节描述。Raspberry Pi支持所有这五个标准。

###### 9.3.4.1 串行外设接口（SPI）

SPI是一个简单的同步串行协议，易于使用，而且相对速度较快。物理接口由三个引脚组成：串行时钟（Serial Clock，SCK）、主输出从输入（MOSI，也称为SDO）和主输入从输出（MISO，也称为SDI）。SPI连接主设备（master device）和从设备（slave device），如图e9.6(a)所示。主设备生成时钟信号。它通过SCK发送一系列时钟脉冲来启动通信。如果它想将数据发送给从设备，它把数据从最高有效位开始放在MOSI。从设备可能通过将数据放在主设备的MISO同时响应。图e9.6(b)显示了8位数据传输的SPI波形。位在SCK的下降沿发生变化，在上升沿能稳定地采样。 SPI接口还可以发送低电平有效芯片使能，以警示接收器数据即将到来。

BCM2835有三个SPI主端口和一个从端口。 本节介绍SPI主端口0，可在GPIO引脚11：9上的Raspberry Pi上轻松访问。 要将这些引脚用于SPI而不是GPIO，必须将它们的GPFSEL设置为ALT0。 然后，Pi必须配置此端口。 当Pi写入SPI时，数据被串行发送到从端口。 同时，从从端口接收的数据被收集起来，当在传输完成时Pi能读取这些数据。

图e9.6 SPI连接和波形

Master：主

Slave：从

optional：可选

表e9.4 SPI0CS寄存器字段

Bit：位

Name：名称

Function：功能

Meaning for 0：0的含义

Meaning for 1：1的含义

Transfer Done：传输完成

Transfer Active：传输有效

Clock Polarity：时钟极性

Clock Phase：时钟相位

Transfer in progress：传输进行中

Transfer complete：传输已完成

SPI disabled：SPI禁用

SPI enabled：SPI启用

Clock idles low：时钟闲置低

Clock idles high：时钟闲置高

First SCK transition at middle of data bit：数据位中间的第一个SCK转换

First SCK transition at beginning of data bit：数据位开头的第一次SCK转换

图e9.7 SPI主端口0的寄存器

表e9.5 SPI0CLK频率

SPI Frequency：SPI频率

如图e9.7中的存储器映射所示，SPI主端口0与三个寄存器相关联。 SPI0CS是控制寄存器。 它用于打开SPI并设置属性，例如时钟的极性。 表e9.4列出了SPI0CS中与本讨论相关的部分位的名称和功能。 重置时，默认值均为0。 大多数功能，例如芯片选择和中断，都不在本节中使用，但可以在数据表中找到。 SPI0FIFO被写入以发送一个字节，被读取以获取返回的字节。 SPI0CLK通过将250 MHz外设时钟除以寄存器中指定的2的幂来配置SPI时钟频率。 因此，SPI时钟频率总结在表e9.5中。

例e9.4 通过SPI发送和接收字节

设计一个系统，通过SPI在Raspberry Pi主设备和FPGA从设备之间进行通信。 绘制界面的示意图。 编写发送字符'A'并接收字符的Pi的C代码。 在FPGA上为SPI从设备编写HDL代码。 如果从设备只需要接收数据，如何简化？

解：图e9.8展示了使用SPI主端口0的设备间连接。引脚编号从器件数据手册（例如，图e9.3）获得。注意，图上显示了引脚编号和信号名称以标示其物理和逻辑连接。SPI使能时，这些引脚不能用于GPIO。

图e9.8 Pi和FPGA之间的SPI连接

Master：主

Slave：从

EasyPIO.h中的以下代码用于初始化SPI以及发送和接收字符。 设置存储器映射和定义寄存器地址的代码与GPIO类似，不在此处重新打印。

void spiInit(int freq, int settings) {

pinMode(8, ALT0); // CEOb

pinMode(9, ALT0); // MISO

pinMode(10, ALT0); // MOSI

pinMode(11, ALT0); // SCLK

SPI0CLK = 250000000/freq; // 将SPI时钟分频器设置为所需的频率

SPI0CS = settings;

SPI0CSbits.TA = 1; // 打开SPI

}

char spiSendReceive(char send){

SPI0FIFO = send; // 将数据发送到从设备

while (!SPI0CSbits.DONE); // 等到SPI完成

return SPI0FIFO; // 返回收到的数据

}

下面的C代码初始化SPI然后发送和接收字符。 它将SPI时钟频率设置为244 kHz。

#include "EasyPIO.h"

void main(void) {

char received;

pioInit();

spiInit(244000, 0); // 初始化 SPI:

// 244 kHz时钟频率， 为默认设置

received = spiSendReceive('A'); // 发送字母A并接收字节

}

FPGA的HDL代码在下面列出。图e9.9显示了方框图和时序。FPGA采用移位寄存器来保存从主设备接收的数据位和要发送到主设备的剩余数据位。

图e9.9 SPI从设备的电路和时序

在复位后的第一个上升沿和每8个周期之后，来自d的新字节被加载到移位寄存器中。 在每个后续周期中，一个位在mosi上移入，一个位从miso移出。 将miso延迟到sck的下降沿，以便主设备在下一个上升沿进行采样。 在8个周期之后，可以在q中找到接收的字节。

module spi\_slave(input logic sck, // 从主设备

input logic mosi, // 从主设备

output logic miso, // 到主设备

input logic reset, // 系统重置

input logic [7:0] d, // 要发送的数据

output logic [7:0] q); // 要接收的数据

logic [2:0] cnt;

logic qdelayed;

// 发送完整字节时，3位计数器跟踪

always\_ff @(negedge sck, posedge reset)

if (reset) cnt = 0;

else cnt = cnt + 3’b1;

// 可装载的移位寄存器

// 在开始时加载d，在每一步将mosi移动到底部

always\_ff @(posedge sck)

q < = (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

// 将miso与sck的下降沿对齐

// 在开始时加载d

always\_ff @(negedge sck)

qdelayed = q[7];

assign miso = (cnt == 0) ? d[7] : qdelayed;

Endmodule

如果从设备仅需要从主设备接收数据，则它将缩减为以下HDL代码中给出的简单移位寄存器。

module spi\_slave\_receive\_only(input logic sck, //从主设备

input logic mosi,//从主设备

output logic [7:0] q); //已接收的数据

always\_ff @(posedge sck)

q < = {q[6:0], sdi}; // 寄存器进行移位

Endmodule

SPI端口具有高度可配置性，因此可以与各种串行设备通信。 不幸的是，这导致可能错误地配置端口并使数据传输混乱。有时候，有必要改变配置位以跟不同时序期望值的设备进行通信。当CPOL = 1时，SCK反转。当CPHA = 1时，时钟相对于数据提前半个周期切换。 这些模式如图e9.10所示。要知道，不同的SPI产品可能为这些选项使用不同的名称和极性；仔细检查设备的波形。使用示波器检查SCK、MOSI和MISO，对解决通信困难有帮助。

图e9.10 SPI时钟和数据时序配置

图e9.11 异步串行链路

Idle：空闲

start：开始

stop：结束

###### **9.3.4.2通用异步接收器/发送器（UART）**

UART是两个系统之间不发送时钟信号而进行通信的一种串行I/O外设。但这两个系统必须事先就使用何种数据速率达成一致，每个系统必须生成自己的时钟。虽然这些系统时钟可能有小频率误差和未知的相位关系，但UART能设法坐到可靠的异步通信。UART被用于诸如RS-232和RS-485等协议。例如，旧计算机串口使用RS-232C标准，这由电子工业协会（Electronic Industries Association）于1969年提出。该标准的最初设想是连接数据终端设备（Data Terminal Equipment，DTE）和数据通信设备（Data Communication Equipment，DCE），例如计算机主机和调制解调器。尽管与SPI相比，UART相对较慢，并且容易出现配置错误的问题，但该标准已经存在了很久，因而它们现在仍然重要。

图e9.11(a)表示一个异步串行链路。 DTE在TX线上发送数据到DCE，在RX线上接收数据。图e9.11(b)展示出在上述其中一条传输线上以9600波特的数据速率发送一个字符的过程。链路空闲时，其逻辑为‘1’。每个字符传输包括一个起始位（0）、7-8个数据位、一个可选的奇偶位、以及一个或多个停止位（1）。UART检测从空闲状态到起始状态的下降沿，从而将传输锁定在适当的时间。虽然7个数据位足以发送ASCII字符，但通常使用8个数据位，因为这样可以传递任意一个字节数据。

可选的奇偶校验位允许系统检测传输期间是否有损坏的位。它可以被配置为偶校验或奇校验；偶校验意味着传输数据和奇偶校验位的集合共有偶数个1；也就是说，奇偶位是对数据位进行异或（XOR）操作的结果。接收器然后可以检查是否接收到偶数1，如果没有则发出错误信号。奇校验则相反。

一个常见的选择是使用8个数据位、无奇偶校验、1个停止位，总共10个符号来传达一个8位字符信息。因此，信号传输速率单位是波特而不是位/秒（bits/sec）。例如，9600波特表示9600符号/秒（symbols/sec），或960字符/秒（characters/sec）。两个传输系统都必须配置合适的波特率和数据、奇偶校验和停止位的数量，否则数据会出现乱码。这是一个麻烦，尤其对非技术用户而言，这是为什么通用串行总线（USB）在PC系统中取代UART的原因。

常用的波特率包括300、1200、2400、9600、14400、19200、38400、57600和115200。较低波特率在七八十年代用于调制解调器通过电话线发送一系列音调数据。在现代系统中，9600和115200是两个最常见的波特率；当对速度没有要求时使用9600，115200是最快的标准速率，虽然跟其他现代串行I/O标准相比仍然缓慢。

RS-232标准定义了几个额外的信号。请求发送（Requestto Send，RTS）和清除发送（Clear to Send，CTS）信号，可用于硬件握手（hardwarehandshaking）。它们可以在以下两种模式的任意一种模式下工作。在流控制模式（flowcontrol mode）下，当DTE已准备好从DCE接收数据时，DTE清除RTS为0时。同样，当DCE已准备好从DTE接收数据，DCE清除CTS为0时。一些数据手册使用上划线来表示它们是低电平有效。在较老的单工模式下，当DTE准备好发送数据时，DTE清除RTS为0时。当DCE准备好接收数据时，DCE通过清除CTS回复DTE。

一些系统，特别是那些通过电话线连接的系统，还使用数据终端就绪（Data Terminal Ready，DTR）、数据载波检测（Data Carrier Detect，DCD）、数据集就绪（Data SetReady，DSR）和环指示符（Ring Indicator，RI）来指示设备何时连接到线路。

最初的标准推荐了大块的25针DB-25连接器，但PC精简成一个9针DE-9公连接器它的引脚图如图e9.13(a)所示。电缆线通常如图e9.13(b)所示直接连接。但是，如果直接连接两个DTE，可能需要图e9.13(c) 所示的零调制解调器连线（null modem cable）来交换RX和TX以完成握手。有些连接器是公连接器，而有些是母连接器。总之，可能要使用一大箱电缆和一定量的推测工作才能通过RS-232连接两个系统，再次说明转向USB的必要性。幸运的是，嵌入式系统通常使用一个简化的3-或5-导线设置，由GND、TX、RX组成，可能包括RTS和CTS。

RS-232使用3至15 V电压表示0，使用-3至-15V电压表示1；这就是所谓的双极信号（bipolar signaling）。收发器转换UART的数字逻辑电平为RS-232所期望的正负电平，并且还提供静电放电保护，以在用户插入电缆时保护串口免受损坏。MAX3232E是与3.3V和5 V数字逻辑兼容的流行收发器。它包含一个与外部电容相结合的电荷泵，从单一低电压电源产生±5 V输出。一些用于嵌入式系统的串行外设省略了收发器，只需使用0 V表示0和3.3或5 V表示1; 请查看数据表！

BCM2835有两个名为UART0和UART1的UART。 两者都可以配置为在引脚14和15上通信，但UART0功能更全面，所以此处描述UART0。 要将这些引脚用于UART0而不是GPIO，必须将它们的GPFSEL设置为ALT0。 与SPI一样，Pi必须首先配置端口。 与SPI不同的是，读取和写入可以独立进行，因为任何一个系统都可以在没有接收的情况下传输数据，反之亦然。UART0的寄存器如图e9.14所示。

要配置UART，首先设置波特率。 UART内部具有3 MHz时钟，必须将其分频以产生一个16倍于所需波特率的时钟。 因此，适当的除数BRD是

BRD = 3000000/(16 ×波特率)

BRD的16位整数部分在UART\_IBRD中表示，6位小数部分在UART\_FBRD中表示，因此：BRD = IBRD + FBRD / 64。 表e9.6显示了常用波特率[[1]](#footnote-0)的这些设置。

图e9.13 DE-9公电缆（a）引出线，（b）标准接线，（c）零调制解调器接线

图e9.14 UART0寄存器

表e9.6 BRD设置

Target Baud Rate：目标波特率

Actual Baud Rate：实际波特率

Error：误差

接下来，使用UART\_LCRH线控制寄存器设置数据位，停止位和奇偶校验位的数量。 默认情况下，UART有1个停止位且没有奇偶校验位，但奇怪的是只发送和接收5位长的字。 因此，UART\_LCRH的WLEN字段（位6:5）必须设置为3才能处理8位字。 最后，通过打开UART\_CR控制寄存器的第0位（UARTEN）来使能UART。

使用UART\_DR数据寄存器和UART\_FR成帧寄存器发送和接收数据。 要发送数据，请等到UART\_FR的第7位（TXFE）为1表示发送器不忙，然后向UART\_DR写入一个字节。 要接收数据，请等到UART\_FR的第4位（RXFE）为0表示接收器有数据，然后从UART\_DR读取该字节。

例e9.5 与PC的串行通信

开发PIC32与PC沟通的电路及C程序，通过包含8个数据位、1个停止位和无奇偶校验的串行端口以115200波特率进行通信。PC运行控制台程序，如PuTTY[[2]](#footnote-1)，通过串行端口进行读写操作。该程序应要求用户输入一个字符串，然后应该告诉用户她输入了什么。

解：图e9.15(a)展示了串行链路的基本原理图，说明了电平转换和布线问题。因为很少PC仍有物理串行端口，我们使用可插入式USB到RS-232 DB9串行适配器（参见plugable.com），如图e9.16所示，提供一个到PC的串行连接。适配器连接到一个DE-9母连接器上，该连接器焊接到馈入收发器的电线上，从而将双极RS-232电平转换为Pi的3.3 V电平。Pi和PC都是数据终端设备，因此TX和RX引脚必须在电路中交叉连接。 不使用来自Pi的RTS / CTS握手，并且DE9连接器上的RTS和CTS连接在一起，这样PC就会跟自己握手。 图e9.15(b)显示了使用Adafruit 954 USB转TTL串行电缆的简便方法。 该电缆与3.3 V电平直接兼容，并具有直插入Raspberry Pi公头的母插头引脚。

要配置PuTTY使用串行链路工作，设置连接类型为串行和速度为115200。设置串行线路到由操作系统分配的COM端口为串口转USB适配器（Serial to USB Adapter）。在Windows操作系统中，这可以在设备管理中找到；例如，它可能是COM3。在连接→串口选项卡，设置流控制为NONE或RTS/CTS。在终端选项卡，设置本地回显为强制开，从而在终端显示你键入的字符。

下面列出了EasyPIO.h中的串行端口设备驱动程序代码。 终端程序中的Enter键对应于C中表示为'\r'的回车符，ASCII码为0x0D。 要在打印时前进到下一行的开头，请同时发送'\n'和'\r'（新行和回车）字符。[[3]](#footnote-2)3 uartInit函数如上所述配置UART。类似地，getCharSerial和putCharSerial等待直到UART准备就绪，然后从数据寄存器读取或写入一个字节。

void uartInit(int baud) {

uint fb = 12000000/baud; // 3 MHz UART时钟

pinMode(14, ALT0); // TX

pinMode(15, ALT0); // RX

UART\_IBRD = fb >> 6; // BRD的6 个小数位, 16个整数位

UART\_FBRD = fb & 63;

UART\_LCRHbits.WLEN = 3; // 8 数据位, 1 停止位, 没有奇偶校验位, no FIFO, no Flow

UART\_CRbits.UARTEN = 1; // 使能uart

}

char getCharSerial(void) {

while (UART\_FRbits.RXFE); // 等到数据可用

return UART\_DRbits.DATA; // 从串口返回char

}

void putCharSerial(char c) {

while (!UART\_FRbits.TXFE); // 等到准备好传输

UART\_DRbits.DATA = c; // 将char发送到串口

}

main函数演示了如何使用putStrSerial和getStrSerial函数打印到控制台和从控制台读取。

#include "EasyPIO.h"

#define MAX\_STR\_LEN 80

void getStrSerial(char \*str) {

int i = 0;

do { // 读完整个字符串直到

str[i] = getCharSerial(); // 回车

} while ((str[i++ ] ! = '\r') && (i < MAX\_STR\_LEN)); // 寻找回车

str[i-1] = 0; // 字符串以空结束

}

void putStrSerial(char \*str) {

int i = 0;

while (str[i] ! = 0) { // 迭代字符串

putCharSerial(str[i++ ]); // 发送每个字符

}

}

int main(void) {

char str[MAX\_STR\_LEN];

pioInit();

uartInit(115200); // 使用波特率初始化UART

while (1) {

putStrSerial("Please type something: \r\n");

getStrSerial(str);

putStrSerial("You typed: ");

putStrSerial(str);

putStrSerial("\r\n");

}

}

图e9.15 Raspberry Pi到PC的串行链路(a)可插拔电缆，(b)Adafruit电缆

MAX3232E Transceiver：MAX3232E收发器

Female DE-9 Connector：母DE-9连接器

Plugable USBtoRS-232 Serial Adapter：可插拔USBtoRS-232串行适配器

To PC USB Port：到PC的USB端口

Adafruit 954 USB toTTL Serial Cable：Adafruit 954 USB转TTL串口线

white：白

green：绿

black：黑

在PC上用C程序与串口进行通讯是有点麻烦的，因为串口驱动程序库没有跨操作系统规范。而其他编程环境，如Python、MATLAB或LabVIEW，可顺利进行串行通信。

##### **9.3.5 计时器**

嵌入式系统通常需要测量时间。例如，微波炉需要一个计时器来跟踪日期时间，另一个计时器来测量烹饪时间。它还可能使用再一个计时器产生马达脉冲以旋转食物盘，以及第四个计时器通过仅在每秒的一小部分时间里激活微波能量以控制功率设置。

BCM2835有一个系统定时器，带有一个64位自由运行计数器，每微秒增加一次（即1 MHz）和4个32位定时器比较通道。 图e9.17显示了系统定时器的内存映射。 SYS\_TIMER\_CLO和CHI包含64位计数器值的低32位和高32位。 SYS\_TIMER\_C0 ... C3是32位比较通道。 当任何比较通道与SYS\_TIMER\_CLO匹配时，SYS\_TIMER\_CS的后四位中的相应匹配位（M0-M3）被置位。 通过向SYS\_TIMER\_CS的该位写入1来清除匹配位。 这似乎违反直觉，但它可以防止无意中清除其他匹配位。 可以通过将该时间添加到CLO并将其放入C1，清除SYS\_TIMER\_CS.M1，然后等待直到设置SYS\_ TIMER\_CS.M1来测量特定的微秒数。

不幸的是，Linux是一个多任务操作系统，可以在没有警告的情况下在进程之间切换。 如果您的程序正在等待计时器匹配，然后另一个进程开始执行，则您的程序可能直到匹配发生后很长时间才会恢复，这样您可能会测量错误的时间。 为避免这种情况，您的程序可以在关键时序循环期间关闭中断，使得Linux不会切换进程。 完成后，请务必重新打开中断。 EasyPIO定义了noInterrupts和interrupts函数，分别用于禁用和启用中断。 当禁用中断时，Pi不会在进程之间切换，甚至无法响应用户按Ctrl-C来终止程序。 如果您的程序挂起，则需要关闭电源并重新启动Pi才能恢复。

图e9.17系统定时器寄存器

例e9.6 闪烁的LED

编写一个程序，使Raspberry Pi上的状态LED每秒闪烁5次，持续4秒。

解：EasyPIO中的delayMicros函数使用定时器比较通道1创建指定的微秒数的延迟。

void delayMicros(int micros) {

SYS\_TIMER\_C1 = SYS\_TIMER\_CLO + micros; // 设置比较寄存器

SYS\_TIMER\_CSbits.M1 = 1; // 将匹配标志重置为0

while (SYS\_TIMER\_CSbits.M1 == 0); // 等到匹配标志被设置

}

void delayMillis(int millis) {

delayMicros(millis\*1000); // 每1 ms有1000μs

}

GPIO47驱动Pi B +上的活动LED。 此程序将此引脚设置为输出并禁用中断。 然后通过一系列数字写入将LED关闭和打开，重复率为200 ms（5 Hz）。 此程序最终重新启用了中断。

#include "EasyPIO.h"

void main(void) {

int i;

pioInit();

pinMode(47, OUTPUT); // 状态led作为输出

noInterrupts(); // 禁用中断

for (i = 0; i<20; i ++ ) {

delayMillis(150);

digitalWrite(47, 0); // 关闭led

delayMillis(50);

digitalWrite(47, 1); // 打开led

}

interrupts(); // 重新启用中断

}

##### **9.3.6 模拟I/O**

真实的世界是模拟信号的世界。许多嵌入式系统需要通过模拟输入和输出与外界交互。它们使用模拟—数字转换器（analog-to-digitalconverter，ADC）将模拟信号量化成数字数值，使用数字—模拟转换器（digital-to-analog-converter，DAC）做相反的事情。图e9.18 展示了这些组件符号。转换器的特征由它们的分辨率、动态范围、采样频率和精确度来决定。例如，一个ADC可能有N = 12位分辨率，范围Vref-—Vref+为0—5V，采样频率为fs =44KHz，精确度为±3最低有效位（LSB）。采样频率单位也可写成样本/秒（samples per second，sps），其中1sps=1Hz。模拟输入电压Vin（t）和数字样本X[n = t / fs]之间的关系是

图e9.18 ADC和DAC符号

例如，2.5V输入电压对应于1000000000002=80016的输出，不确定性高达3个LSB。

同样地，Vref = 2.56V时，DAC可能具有N = 16位分辨率。它产生的输出为

许多微控制器具有内置的中等性能ADC。对于更高性能（例如，16位分辨率或超过1MHz的采样频率），则通常需要使用连接到微控制器的独立ADC。较少微控制器具有内置DAC，所以可能使用独立芯片。然而，微控制器经常使用一种称为脉冲宽度调制（pulse-width modulation，PWM）技术产生模拟输出。

###### **9.3.6.1 数字/模拟转换**

BCM2835具有用于复合视频输出的专用DAC，但没有通用转换器。 本节介绍使用外部DAC的D / A转换（数字/模拟转换），并描述了通过并行和串行端口将Raspberry Pi连接到其他芯片的过程。 下一节使用脉冲宽度调制实现相同的结果。

一些数模转换器通过具有N条连线的并行接口接收N位数字输入，而另一些则通过串行接口如SPI接收数据。有些DAC要求正负电源电压供应，而其它的则使用单电源供电操作。一些DAC支持弹性的电源电压范围，而另一些则需要特定的电压。输入逻辑电平应当与数字源兼容。有些DAC生成与数字输入成正比的电压输出，而另一些则产生电流输出；可能需要一个运算放大器来将此电流转换为在所需范围内的电压。

在本节中，我们使用Analog Devices AD558 8位并行DAC和Linear Technology LTC1257 12位串行DAC。这两种DAC产生电压输出，都使用一个5-15V电源，使用VIH = 2.4 V，因而与3.3 V的I/O兼容，采用DIP封装使它们易于装在试验电路板上，并且易于使用。AD558产生范围在0-2.56V的输出，功耗75mW，采用16引脚封装，并有1μs稳定时间从而允许1M样本/秒的输出速率。数据手册可从analog.com获得。LTC1257产生范围在0-2.048V的输出，功耗低于2mW，采用8引脚封装，并有6μs稳定时间。它的SPI运行最高频率为1.4MHz。数据手册可从linear.com获得。

例e9.7 结合外部DAC的模拟输出

画出简单信号发生器的电路图和编写相应软件程序，利用Raspberry Pi、AD558和LTC1257生成正弦波和三角波。

解：电路如图e9.19所示。 AD558通过GPIO14,15,17,18,22,23,24和25连接到Pi。它将Vout Sense和Vout Select连接在到Vout以设置2.56V满量程输出范围。LTC1257通过SPI0连接到Pi。两个ADC都采用5V电源供电，并有一个0.1μF去耦电容以降低电源噪声。DAC上的低电平有效芯片使能和负载信号指示何时转换下一个数字输入。在加载新输入时，它们被驱动为高电平。

图e9.19 DAC到Raspberry Pi的并行和串行接口

该程序如下。 pinsMode和digitalWrites类似于pinMode和digitalWrite，但操作的是引脚阵列。 该程序将8个并行端口引脚设置为输出，并将GPIO2配置为输出，以驱动芯片使能和加载信号。 它将SPI初始化为1.4 MHz。initwavetables预计算正弦和三角波的样值数组。正弦波被设定为12位规模，而三角波被设定为8位规模。每个波一个周期有64个采样点；改变这个数值是在用采样精度交换采样频率。genwaves重复通过采样样本。对于每个样本，它禁用了到DAC的CE和LOAD信号，通过并行和串行端口发送新样本，重新启用DAC，然后等待直到计时器指示到了下一个样本的时间。spiSendReceive16传输两个字节，但LTC1257只关心发送的最后12位。 最大频率超过1000 Hz（64 Ksamples / sec），由genWaves函数中发送每个点的时间设置，SPI传输是genWaves函数的主要组件。

#include "EasyPIO.h"

#include <math.h> // 使用正弦函数所必需的

#define NUMPTS 64

int sine[NUMPTS], triangle[NUMPTS];

int parallelPins[8] = {14,15,17,18,22,23,24,25};

void initWaveTables(void) {

int i;

for (i = 0; i<NUMPTS; i ++ ) {

sine[i] = 2047\*(sin(2\*3.14159\*i/NUMPTS) + 1); // 12位规模

if (i<NUMPTS/2) triangle[i] = i\*511/NUMPTS; // 8位规模

else triangle[i] = 510-i\*511/NUMPTS;

}

}

void genWaves(int freq) {

int i, j;

int microPeriod = 1000000/(NUMPTS\*freq);

noInterrupts(); // 禁用中断以获得准确的时序

for (i = 0; i<2000; i ++ ){

for (j = 0; j<NUMPTS; j ++ ) {

SYS\_TIMER\_C1 = SYS\_TIMER\_CLO + microPeriod; // 设定样本之间的时间

SYS\_TIMER\_CSbits.M1 = 1; // 清除计时器匹配

digitalWrite(2,1); // 更改输入时无加载

spiSendReceive16(sine[j]);

digitalWrites(parallelPins, 8, triangle[j]);

digitalWrite(2,0); // 将新点加载到DAC中

while (!SYS\_TIMER\_CSbits.M1); // 等到计时器匹配

}

}

interrupts();

}

void main(void) {

pioInit();

pinsMode(parallelPins, 8, OUTPUT); // 将连接到AD558的引脚设置为输出

pinMode(2, OUTPUT); // 使引脚2成为输出以控制LOAD和CE

spiInit(1400000, 0); // 1.4MHz SPI时钟，默认设置

initWaveTables();

genWaves(1000);

}

###### **9.3.6.2 脉冲宽度调制**

另一种产生模拟输出方的式是脉冲宽度调制（pulse-width modulation ，PWM），其中产生一个周期性脉冲输出，输出一部分为高电平，其余部分为低电平。占空比是脉冲为高电平部分在一个周期里所占比例，如图e9.20所示。输出平均值正比于占空比。例如，如果输出在0和3.3V之间摆动，并具有25％的占空比，则电压平均值为0.25×3.3=0.825V。对PWM信号进行低通滤波可消除振荡使信号得到所需平均值。因此，如果脉冲频率远高于感兴趣的模拟输出频率，则PWM是产生模拟输出的有效方式。

BCM2835具有一个PWM控制器，可以产生两个同步输出。PWM0位于GPIO18，作为引脚功能ALT5，而两个PWM输出均可在立体声音频插孔上使用。图e9.21显示了PWM单元及其所依赖的时钟管理器的内存映射。

PWM\_CTL寄存器用于打开脉冲宽度调制。必须将位0（PWEN1）置1以使能输出。还应设置位7（MSEN1：标记空间使能）以产生图e9.20形式的脉冲宽度调制，其中一周期内的部分输出为高电平，而其余输出为低电平。

图e9.20 脉冲宽度调制信号

Pulse width：脉冲宽度

Duty cycle：占空比

Period：周期

图e9.21 PWM和时钟管理器寄存器

PWM信号源自BCM2835时钟管理器生成的PWM时钟。 PWM\_RNG1和PWM\_DAT1寄存器分别通过指定整个波形和HIGH部分的PWM时钟周期数来控制周期和占空比。 例如，如果时钟管理器产生25 MHz时钟且PWM\_RNG1 = 1000且PWM\_DAT1 = 300，则PWM输出将在（25 MHz / 1000）= 25 kHz下工作，占空比将为300/1000 = 30％。

使用CM\_PWMCTL配置时钟管理器，并使用CM\_PWMDIV寄存器设置频率。 表e9.7总结了CM\_PWMCTL寄存器的位字段。 PWM时钟的最大频率为25 MHz。 它可以从Pi上的500 MHz PLLD时钟获得，如下所示：

* CM\_PWMCTL：将0x5A写入PASSWD，将1写入KILL以停止时钟发生器
* CM\_PWMCLT：等待BUSY清零以指示时钟已停止
* CM\_PWMCTL：将0x5A写入PASSWD，将1写入MASH，将6写入SRC，以选择没有音频噪声塑形的PLLD
* CM\_PWMDIV：将0x5A写入PASSWD，将20写入23:12使得PLLD除以20，从500 MHz降至25 MHz
* CM\_PWMCTL：将0x5A写入PASSWD，将1写入ENAB以重新启动时钟发生器
* CM\_PWMCTL：等待设置BUSY以指示时钟正在运行

表e9.7 CM\_PWMCTL寄存器字段

Bit：位

Name：名称

Description：描述

Must be set to 5A when writing：写入时必须设置为5A

Audio noise shaping：音频噪音塑形

Clock generator running：时钟发生器运行

Write a 1 to stop the clock generator：写1以停止时钟发生器

Write a 1 to start the clock generator：写1以启动时钟发生器

Clock source：时钟源

例e9.8 结合PWM的模拟输出

写一个analogWrite(val)函数，使用PWM和外部RC滤波器生成模拟输出电压。该函数应接受0（0V输出）到255（全3.3V输出）之间的输入。

解：使用PWM0在GPIO18产生78.125KHz信号。图e9.22的低通滤波器具有如下转角频率以消除高速振荡并传递平均值。

图e9.22 使用PWM和低通滤波器的模拟输出

EasyPIO中的PWM函数如下所示。 pwmInit如上所述初始化GPIO18上的PWM模块。 setPWM设置PWM输出的频率和占空比。 占空比应介于0（始终为OFF）和1（始终为ON）之间。 analogWrite函数根据满量程255设置占空比。

// 默认PLLD值为500 [MHz]

#define PLL\_FREQUENCY 500000000

// 最大pwm 时钟为25 [MHz]

#define CM\_FREQUENCY 25000000

#define PLL\_CLOCK\_DIVISOR (PLL\_FREQUENCY / CM\_FREQUENCY)

void pwmInit() {

pinMode(18, ALT5);

// 配置时钟管理器以生成25 MHz PWM时钟。

// 数据表中缺少时钟管理器的文档

// 但在G.J van Loo 2013年2月6日的“BCM2835 Audio and PWM Clocks”中能找到。

// PWM时钟的最大工作频率为25 MHz。

// 写入时钟管理器寄存器需要同时写入

// 顶部位的“password”为5A，以降低意外写入的风险。

CM\_PWMCTL = 0; // 在更改之前关闭PWM

CM\_PWMCTL = PWM\_CLK\_PASSWORD|0x20; // 关闭时钟发生器

while (CM\_PWMCTLbits.BUSY); // 等待发生器停止

CM\_PWMCTL = PWM\_CLK\_PASSWORD|0x206; // Src =未滤波的500 MHz CLKD

CM\_PWMDIV = PWM\_CLK\_PASSWORD|(PLL\_CLOCK\_DIVISOR << 12); // 25 MHz

CM\_PWMCTL = CM\_PWMCTL|PWM\_CLK\_PASSWORD|0x10; // 启用PWM时钟

while (!CM\_PWMCTLbits.BUSY); // 等待发生器启动

PWM\_CTLbits.MSEN1 = 1; // 标记/空间模式中的通道1

PWM\_CTLbits.PWEN1 = 1; // 启用PWM

}

void setPWM(float freq, float duty) {

PWM\_RNG1 = (int)(CM\_FREQUENCY / freq);

PWM\_DAT1 = (int)(duty \* (CM\_FREQUENCY / freq));

}

void analogWrite(int val) {

setPWM(78125, val/255.0);

}

main函数通过将输出设置为半量程（1.65 V）来测试PWM。

#include "EasyPIO.h"

void main(void) {

pioInit();

pwmInit();

analogWrite(128);

}

###### **9.3.6.3 模拟/数字转换**

BCM2835没有内置ADC，因此本节介绍使用类似于外部DAC的外部转换器进行A / D转换（模拟/数字转换）。

例e9.9 使用外部ADC进行模拟输入

使用SPI将10位MCP3002 A / D转换器连接到Raspberry Pi并打印输入值。设置3.3 V的满量程电压。有关操作的详细信息，请在Web上搜索数据表。

解：图e9.23展示了连接示意图。MCP3002使用VDD作为其满量程引用。它接受3.3-5.5 V电源，我们选择3.3 V。ADC有两个输入通道，我们将通道0连接到一个电位计，我们可以旋转它来调节输入电压在0到3.3 V之间。

图e9.23 使用外部ADC的模拟输入

Pi代码初始化SPI并重复读取和打印样本。 根据数据表，Raspberry Pi必须通过SPI发送16位数0x6000来读取CH0，并将在16位结果的最后10位中接收10位结果。 转换器还需要芯片选择信号，方便SPI芯片使能。

#include "EasyPIO.h"

void main(void) {

int sample;

pioInit();

spiInit(200000, 0); // 200 kHz SPI时钟，默认设置

while (1){

sample = spiSendReceive16(0x6000);

printf("Read %d\n", sample);

}

}

##### **9.3.7 中断**

到目前为止，我们一直依赖于轮询，其中程序不断检查直到一个事件发生，例如数据到达UART或计时器达到其比较值。 这可能会浪费处理器的功能，并且难以编写执行感兴趣工作的同时等待事件发生的程序。

大多数微控制器都支持中断。当一个事件发生时，微控制器可以停止正常的程序执行并跳转到响应中断的中断处理程序，然后无缝地返回到它停止的位置。

Raspberry Pi通常运行Linux，它在进入程序之前拦截中断。 因此，目前编写基于中断的程序并不简单，本文未提供关于Pi的示例。

1. 波特率并不都能均匀地划分3 MHz，因此一些除数会产生频率误差。 UART具有异步特性，只要误差足够小，就可以容纳它。 [↑](#footnote-ref-0)
2. PuTTY 可在[www.putty.org](http://www.putty.org)免费下载。 [↑](#footnote-ref-1)
3. 即使省略了\ r，PuTTY也能正确打印。 [↑](#footnote-ref-2)