3. 第三部分

基于 MIPSfpga 和 Hos 操作系统的蓝牙小车设计与实现

本部分介绍一个 MIPSfpga 硬件系统与 Hos 操作系统综合的应用蓝牙小车，以体现系统综合实践的意义。

为突出操作系统的实用性，我们将对蓝牙小车的应用进行扩展，以下会以小车操作轨迹记录为例进行介绍，如果读者有更好的扩展设计，亦可参照进行实现。

第9章 实验 9：蓝牙模块及马达驱动模块硬件实现

9.1 实验目的

在了解、学习并实践了 MIPSfpga 的搭建和自定制 PMOD 模块之后，我们将对蓝牙小车 2 个重要的硬件 AXI 总线外设接口进行设计和实现，同时将通过一些小程序对其进行基本的运行测试，为后面在操作系统上的蓝牙小车应用的实现完成基础硬件实现。

在本实验中，读者将会完成以下工作：

1.设计并实现以 UART 串口协议为基础的蓝牙芯片总线外设 AXI 总线接口模块。

2.设计并实现以 PMOD 接口协议为基础的 L293D 驱动板总线外设 AXI 总线接口模块。

3.将这 2 个外设接口添加到 MIPSfpga 上，进行综合布线，并烧写到 N4 ddr 开发板上。

4.利用 MIPS sde 交叉编译器编写一个程序下载到 MIPSfpga 中验证并测试以上两个模块的正确性。

同时读者将了解关于无线蓝牙外设的基本工作原理和 L293D 驱动板基本工作原理，以便更好的实现其外设接口的设计。

本实验希望读者能通过对这些外设模块的设计实现加深对 Pmod 接口协议的理解和自定制模块设计方法，同时提高自己的 Verilog 编程能力。

9.2 实验内容

9.2.1 基于 UART 串口的蓝牙外设接口模块硬件设计和测试

该模块是基于串口方式工作，读者可以使用 vivado 提供的 UART 串口 ip 核为基础进行设计，亦可自己完成设计一个串口传输模块。

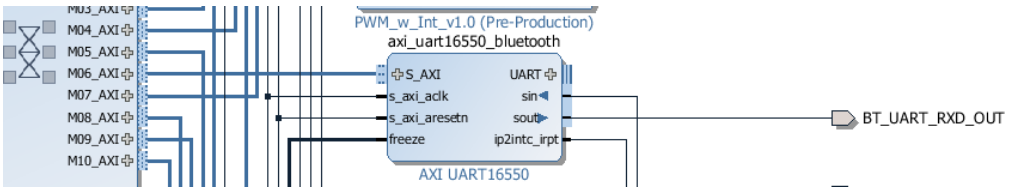


图 第 9-1

读者在完成蓝牙外设接口模块之后，需要通过对 C 主程序进行修改，对该蓝牙设备进行测试。

9.2.2 基于 PMOD 的马达驱动板的硬件设计和测试

该模块需要读者根据 9.3.2 所示的 L293D 马达驱动板原理和实验 3 所学的 PMOD 接口协议原理，自己设计 L293D 马达驱动板接口模块。

读者在完成 L293D 马达驱动板模块之后，需要通过对 C 主程序进行修改，对该蓝牙设备进行测试。

9.3 实验背景及原理

9.3.1 无线蓝牙工作和串口传输原理

MIPS fpga 板上的蓝牙模块采用 UART 串行通信协议。即发送时，发送端在发送时钟脉（TxC）的作用下，将发送移位寄存器的数据按位串行移位输出，送到通信线上；接收时，接收端在接收时钟脉冲（RxC）的作用下，对来自通信线上的串行数据，按位串行移入接收寄存器。这里注意到默认的蓝牙波特率为 9600。

9.3.2 马达驱动板工作原理

马达驱动的转动速度根据所给马达电压的不同而变，电压越大其速度越快。因此采用 PWM 来控制马达驱动的转速。通过 PWM，控制 CPU 发给马达驱动芯片的数据的不同占空比来控制电压的高低，经过马达驱动芯片转化之后给马达驱动相应的电压。L293D 驱动板 PCB 封装引脚如图所示，因此 PWM2A、PWM2B、PWM0A、PWM0B 分别控制四个直流电机的 PWM 调速。

3	Pwm2B	Y2A/B 的 PWM 调速	24	GND	逻辑电源接地
4	Dir_clk	串锁器串行输入时钟	25	VCC_logic	逻辑电源正极
5	Pwm0B	Y3A/B 的 PWM 调速	+	VCC_motor	驱动电源正极
6	Pwm0A	Y4A/B 的 PWM 调速	-	GND	驱动电源接地
7	Dir_enable	串锁器使能端	a/b	Y1A/B	马达 M1 的两端
8	Dir_serial	串锁器串行输入	d/e	Y2A/B	马达 M2 的两端
11	Pwm2A	Y1A/B 的 PWM 调速	f/g	Y3A/B	马达 M4 的两端
12	Dir_latch	串锁器锁存时钟	i/j	Y4A/B	马达 M3 的两端

图 第 9-2 L293D 驱动板 PCB 封装引脚功能图

L293D 驱动板 PCB 封装引脚如图所示。

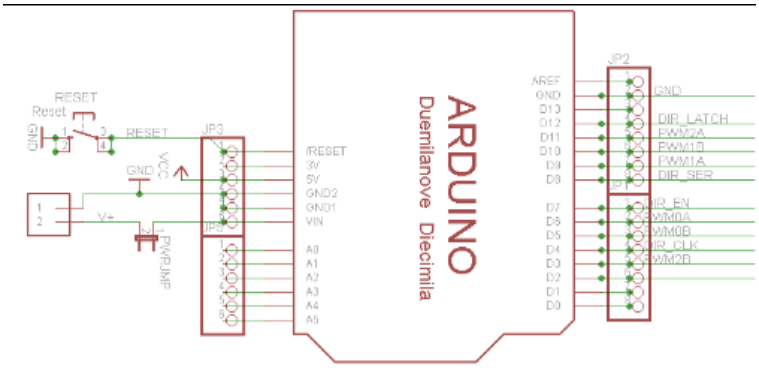


图 第 9-3 L293D 驱动板 PCB 封装引脚图

L293D 驱动板上集成了 74HCT595N 芯片和四个直流电机，74HCT595N 把接收到的串行的信号转为并行的信号，并控制 4 个直流电机，从而能够实现马达驱动的正转和反转。

如图所示，74HCT595N 芯片原理也是串行输入，通过 dir_ser 和 dir_clk 始终控制输入数据到 8 位移位寄存器，时钟频率为 1000Hz。数据传输完成之后给 dir_latch 锁存信号置为 1，则 8 位移位寄存器的内容被送进 8 位存储寄存器中锁存。

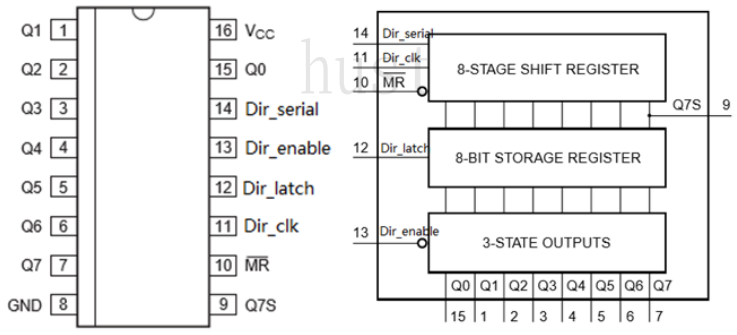


图 第 9-4 74HCT595N 芯片原理图

74HCT595N 芯片各个引脚功能表如图所示，注意 dir_enable 使能信号时低电平有效。

控制信号				输入信号	输出信号	
Dir_clk	Dir_latch	Dir_enable	$\overline{MR}$	Dir_serial	Q_{7S}	Q_n
X	X	L	L	X	L	NC
X	↑	L	L	X	L	L
X	X	H	L	X	L	Z
↑	X	L	H	H	Q_{6S}	NC
X	↑	L	H	X	NC	Q_{nS}
↑	↑	L	H	X	Q_{6S}	Q_{nS}

图 第 9-5 74HCT595N 芯片功能示意图

如图所示，74HCT595N 芯片输出接到 4 个直流电机的 8 个引脚，分别控制马达驱动的

正转反转，因此只要向 74HCT595N 控制模块的寄存器输入 8 位的值就能控制 4 个马达的正转反转。

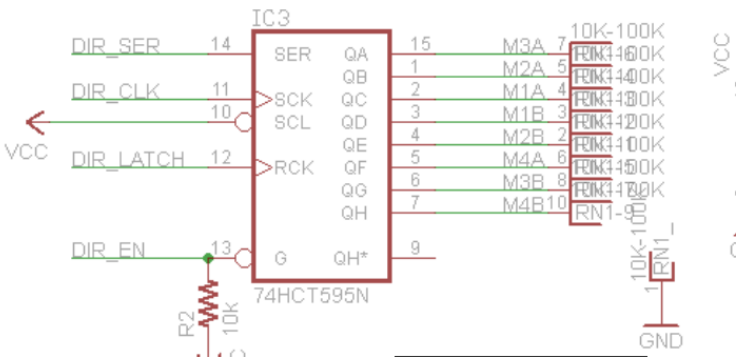


图 第 9-6 74HCT595N 芯片和 4 个直流电机关系示意图

马达驱动的转动速度根据所给马达电压的不同而变，电压越大其速度越快。因此采用 PWM 来控制马达驱动的转速。通过 PWM，控制 CPU 发给马达驱动芯片的数据的不同占空比来控制电压的高低，经过马达驱动芯片转化之后给马达驱动相应的电压。74HCT595N 芯片将根据 01 数据流的占空比不同的电压，最终实现对马达驱动速度的控制。

74HCT595N 芯片将根据 01 数据流的占空比不同的电压，最终实现对马达驱动速度的控制。

9.3.3 3.3.1 PMOD 原理介绍（？）

Pmods™是小尺寸 I/O 接口板，用于扩展 FPGA/CPLD 和嵌入式控制板的功能。Pmod 通过 6 引脚或 12 引脚连接器与系统主板通信。

多年来，元件制造商一直提供开发系统，帮助其客户采用其元件设计应用。对于可编程器件，例如 FPGA 和微控制器，始终存在与其它元件的接口，以便能够与硬件同步或者早于硬件进行软件开发。随着时间推移，涌现出了关于这些“扩展接口”的非常松散的伪标准，其中有些标准的一致性相对较好。Xilinx 等 FPGA 厂商推动这些标准，例如 FMC,使客户尽可能简单地迁移到最新平台。Xilinx 也采用第三方标准，例如 Digilent 制定的 Pmod 标准，用于该接口的外围设备选择较广。微控制器制造商的标准化略慢，许多采用自身的专用接口。然而，制造商动向和 Arduino 平台普及等市场力量正驱使其也向伪标准靠拢。

Pmod 接口是将外设与 FPGA 开发板进行组合和匹配的很好方式，可利用方便、可手工焊接的连接器连接八个引脚以及电源和地。FPGA 的灵活性允许将其八个信号引脚用于几乎所有功能。尽管这提高了其对于 FPGA 的实用性，但也造成该接口难以配合那些外设功能分配给特定引脚的微控制器。为解决这一问题，Digilent 定义了多种不同的 Pmod 引脚排列类型，不同的功能分配给特定的引脚（图 1）。

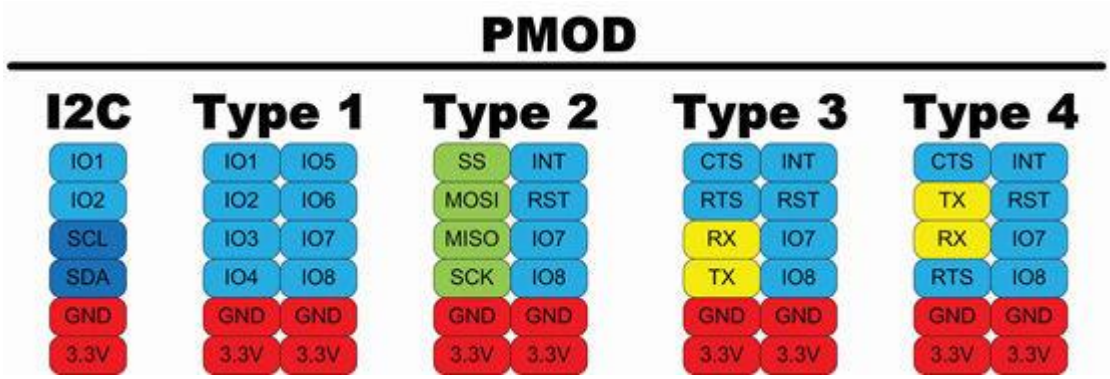


图 第 9-7 Pmod 引脚排列类型将不同的功能分配给特定引脚。

类型定义使得微控制器板较容易使用 Pmod 接口标准，但仍然存在挑战。利用许多微控制器有限的引脚复用能力，难以实现真正的通用接口，已被废弃的 Type 3 UART 接口就是很好的例子。然而，即使存在局限性，对于原型或教育目的，Pmod 接口是一种非常有用的扩展端口。

9.4 实验报告及源码附录

辅助材料 A 无线蓝牙测试程序

```
void init_bluetooth(void) {
	*WRITE_IO(BT_UART_BASE + lcr) = 0x00000080; // LCR[7]    is 1
	delay();
	*WRITE_IO(BT_UART_BASE + dll) = 69; // DLL msb. 115200 at 50MHz. Formula is Clk/16/baudrate. From axi_uart manual.
	delay();
	*WRITE_IO(BT_UART_BASE + dlm) = 0x00000001; // DLL lsb.
	delay();
	*WRITE_IO(BT_UART_BASE + lcr) = 0x00000003; // LCR register. 8n1 parity disabled
	delay();
	*WRITE_IO(BT_UART_BASE + ier) = 0x00000001; // IER register. disable interrupts
	delay();
}
char BT_uart_inbyte(void) {
	unsigned int RecievedByte;
	while(!((*READ_IO(BT_UART_BASE + lsr) & 0x00000001)==0x00000001));
	RecievedByte = *READ_IO(BT_UART_BASE + rbr);
	return (char)RecievedByte;
```

```
}
void _mips_handle_irq(void* ctx, int reason) {
	unsigned int value = 0;
	unsigned int period = 0;

	// *WRITE_IO(UART_BASE + ier) = 0x00000000; // IER register. disable interrupts

	*WRITE_IO(IO_LEDR) = 0xF00F;	// Display 0xFFFF on LEDs to indicate receive data from uart
	delay();

	BT_rxData = BT_uart_inbyte();
	if (BT_rxData == 'w') {
		round = 0;
		*WRITE_IO(IO_LEDR) = 0x1;	// Display 0xFFFF on LEDs to indicate receive data from uart
		delay();
	}
	else if (BT_rxData == 's') {
		round = 0;
		*WRITE_IO(IO_LEDR) = 0x2;	// Display 0xFFFF on LEDs to indicate receive data from uart
		delay();
	}
	else if (BT_rxData == 'q') {
		*WRITE_IO(IO_LEDR) = 0x4;	// Display 0xFFFF on LEDs to indicate receive data from uart
		delay();
	}
	else if (BT_rxData == 'e') {
		*WRITE_IO(IO_LEDR) = 0x8;	// Display 0xFFFF on LEDs to indicate receive data from uart
		delay();
	}
	else if (BT_rxData == 'd') {
		round = 0;
		*WRITE_IO(IO_LEDR) = 0x10;	// Display 0xFFFF on LEDs to indicate receive data from
uart
		delay();
	}
	else if (BT_rxData == 'a') {
		round = 0;
		*WRITE_IO(IO_LEDR) = 0x20;	// Display 0xFFFF on LEDs to indicate receive data from
```

```
uart
            delay();
      }
      else if (BT_rxData == '8') {
            *WRITE_IO(IO_LEDR) = 0x20;    // Display 0xFFFF on LEDs to indicate receive data from
uart
            delay();
      }
      else if (BT_rxData == 'h') {
            round = 0;
            *WRITE_IO(IO_LEDR) = 0x20;    // Display 0xFFFF on LEDs to indicate receive data from
uart
            delay();
      }
      else {
            round = 0;
            *WRITE_IO(IO_LEDR) = 0x40;    // Display 0xFFFF on LEDs to indicate receive data from
uart
            delay();
      }
      //

            *WRITE_IO(IO_LEDR) = 0xFFFF;    // Display 0xFFFF on LEDs to indicate receive data from
uart
      return;
}
```


辅助材料 B 驱动板测试程序

```
#define speed1 1024*1024-1
#define speed2 768*1024
#define speed3 384*1024
#define speed4 0
#define rb_f 0x00000020
#define lb_f 0x00000040
#define rf_f 0x00000080
#define lf_f 0x00000004
#define rb_b 0x00000010
```

```
#define lb_b 0x00000008
#define rf_b 0x00000002
#define lf_b 0x00000001
int gear2speed(int *gear) {
	if (*gear == 1) {
		return speed3;
	}
	else if (*gear == 2) {
		return speed2;
	}
	else if (*gear == 3) {
		return speed1;
	}
	else if (*gear == -1) {
		return speed3;
	}
	else if (*gear == -2) {
		return speed2;
	}
	else if (*gear == -3) {
		return speed1;
	}
	else if (*gear >= 3) {
		*gear = 3;
		return speed1;
	}
	else if (*gear <= -3) {
		*gear = -3;
		return speed1;
	}
	else {
		return 0;
	}
}
void set_gear(int lf, int lb, int rf, int rb) {
	gear_lb = lb;
	gear_lf = lf;
	gear_rb = rb;
```

```
	gear_rf = rf;
	_go();
}
void _go(void)
{
	int direction = 0;
	direction = direction | (gear_rf>=0?rf_f:rf_b);
	direction = direction | (gear_rb>=0?rb_f:rb_b);
	direction = direction | (gear_lf>=0?lf_f:lf_b);
	direction = direction | (gear_lb>=0?lb_f:lb_b);
	*WRITE_IO(dir_data) = direction;
	delay();
	*WRITE_IO(WHEEL_RF) = gear2speed(&gear_rf);
	delay();
	*WRITE_IO(WHEEL_LB) = gear2speed(&gear_lb);
	delay();
	*WRITE_IO(WHEEL_LF) = gear2speed(&gear_lf);
	delay();
	*WRITE_IO(WHEEL_RB) = gear2speed(&gear_rb);
	delay();
}
void speedup(void) {
	//小车当前状态不是直行
	if (state != 0) {
		state = 0;
		set_gear(2, 2, 2, 2);
	}
	else {
		set_gear(gear_lf + 1, gear_lb + 1, gear_rf + 1, gear_rb + 1);
	}
}

void slowdown(void) {
	if (state != 0) {
		state = 0;
		set_gear(0, 0, 0, 0);
	}
	else {
```

```
            set_gear(gear_lf - 1, gear_lb - 1, gear_rf - 1, gear_rb - 1);
        }
}
void leftforward(void) {
        state = 1;
        set_gear(0, 1, 3, 3);
}

void rightforward(void) {
        state = 1;
        set_gear(3, 3, 0, 1);
}

void turnright(void) {
        state = 1;
        set_gear(2, 2, -2, -2);
}

void turnleft(void) {
        state = 1;
        set_gear(-2, -2, 2, 2);
}

void mystop(void) {
        state = 0;
        set_gear(0, 0, 0, 0);
}
```

第10章　实验 10：蓝牙小车软件应用实现

10.1　实验目的

在基于 MIPSfpga 系统的 Hos 操作系统实现蓝牙模块和马达模块驱动，并实现蓝牙小车的应用。

在上一个实验中，读者完成了蓝牙小车重要外设模块硬件接口，接下来我们利用包含这些硬件外设的 MIPSfpga 系统，在 Hos 操作系统中实现一个利用手机无线蓝牙控制的蓝牙小车应用，该应用能控制小车前进、倒车、转向等基本功能，有能力的读者可以完成更加复杂的小车动作，例如加速、减速等其他动作。

在这个实验中，读者将会完成以下工作：

1.在 Hos 操作系统中实现无线蓝牙设备和马达驱动板设备驱动程序，并进行测试

2.在 Hos 操作系统用户态中通过设备驱动的调用完成蓝牙小车应用程序的设计实现，并进行测试

以此读者将学习理解 Hos 操作系统外设驱动实现与调用原理，加深对操作系统外设驱动的实现理解，同时读者可以发现操作系统和裸机在外设设备驱动和用户应用的区别。

10.2　实验内容

10.2.1　Hos 操作系统上实现蓝牙模块和马达模块驱动

1.蓝牙串口驱动设备

在实现了蓝牙串口硬件相关的设计之后，读者需要在 Hos 系统中添加蓝牙模块的驱动设备，由于蓝牙硬件接口采用串口实现，需要在内核头文件 arch.h 中注册蓝牙串口硬件模块的分配的地址，同时需要利用第一、二部分所学的硬件和软件中断方式，编写串口中断程序，完善中断处理例程入口函数的中断分发部分的逻辑，将串口中断处理例程注册进去，完成后在 trap 中完善内核态对中断的处理。然后需要在 dev 中设计相应设备，主要的逻辑就是从数据位地址中取出数据放入驱动缓冲区。另外一个需要做的工作就是串口的初始化，初始化的工作最主要的是打开系统对串口中断的响应，通过指明中断号来调用 pic_enable 函数完成。然后在系统调用 syscall 中设计对该串口设备的调用，至此已经完成串口设备驱动的驱动部分。有关串口的中断，读者可以参考内核中的 console.c 实现，由于该设备只读，读者只需完成读设备相关。

2.驱动板接口驱动设备

在实现了驱动板硬件 pmod 接口相关的设计之后，读者需要在 Hos 系统中添加驱动板接口设备,与蓝牙驱动类似需要在内核头文件 arch.h 中注册其硬件模块的分配的地址，同时实现相应的中断例程和系统设备调用。由于该设备只写，读者只需完成写设备相关。

3.直接设备读写

如果读者觉得这种方式较为繁杂，亦可通过直接对硬件所在地址直接对设备进行读写。

10.2.2 蓝牙小车应用实现及测试

在完成两个设备驱动之后，读者需要对蓝牙小车相应的软件应用进行实现，以在 Hos 中实现一个真实的应用：从蓝牙获取控制数据,对驱动板进行控制，使得蓝牙小车按照控制运行。最后读者将让该蓝牙小车在移动电源的控制下脱离主机运行。

10.3 实验背景及原理

10.3.1 Hos 操作系统外设驱动实现与调用原理介绍

读者在第一、二部分了解学习了 MIPS 中断异常相关的 CP0 异常处理寄存器，而在操作系统中一般使用异常中断 trap 函数，进入该函数后又通过调用 trap_dispatch 函数来根据 trapframe 中的异常号进行异常的分发。此异常号是在进入统一入口后根据 cause 寄存器 2~6 位的值装填入 trapframe 的，因此为了能够一一对应地调用正确的函数处理正确的异常，操作系统中的异常号要与软核中的异常号一一对应。

若异常号为 8，则转到系统调用，并将 epc（存放异常处理返回地址）自加一个指令长度（32 位架构为+4），避免返回时又进行系统调用。此后进入系统调用的路径，系统调用路径最终是一系列功能函数，完成一些提供给用户态的基础系统功能，比如打开、关闭、fork 等；

若异常号为 0，则转到中断处理例程中进行中断的分发处理，根据中断号来调用对应的中断处理例程。中断号是与异常号一起从 cause 寄存器的 8~15 位装填入 trapframe 的，因此，中断号同样也有着必须对应的关系。

设备与系统的数据交互在内存控制器中实现。首先在内存控制器中配置新串口的数据、状态字内存地址，以及驱动板的数据内存地址。这里分别选用了无人占用的 0x408 与 0x40C（串口），以及 0x420（驱动板）。注意其所在将是 kernel 段地址所以高位地址没有作用。这里所使用的地址比较重要，必须不能与其他设备接口冲突，比如 VGA 与 PS/2；此外，这个

地址在内核中，编写设备驱动程序的时候要用到，用来对设备进行读写。

在明确了地址之后，需要对内存控制器进行读与写逻辑的完善。由于两个设备都是异步设备，不像 RAM（Random Access Memory，随机存取存储器）需要同步读异步写，不需要 CPU 对读结果进行等待，因此读逻辑只需要将数据从串口缓冲队列通过内存控制器交由 CPU，而写逻辑只需要将数据从 CPU 通过内存控制器交由驱动器硬件接口。

注意，这里的读逻辑是对数据和 CPU 之间的关系的一种描述，而不是指用户态通过系统调用读取串口数据。这里的读是指，数据进入 CPU。

仔细推敲内存控制器的硬件代码，可以发现内存控制器里有负责读、写的部分。以读逻辑为例，内存控制器通过将 CPU 正在访存的地址与现有的注册地址进行比对，来判断是对什么设备进行的操作，这时上文中注册的设备地址起到作用。在判断出操作的设备后，将设备的数据赋予内存控制器的数据输出，CPU 获取这个数据输出完成读逻辑。实际上这就是一个多路选择器，根据访存地址进行选择，选择器的输入是每个设备的输出，对其扩充就是多了一路选择的输入。在这里有对 ROM、RAM、PS/2 等的读取，需要增加对串口的支持，按照上述逻辑在这个多路选择器里注册串口的数据地址以及串口的状态字地址即可。写逻辑的原理与步骤非常接近，对负责写逻辑的多路选择器进行扩充即可。在写逻辑里没有状态字地址，只有一个数据地址，扩充相对容易。

10.4 实验报告

第11章　实例 2：设备驱动方式的蓝牙小车实现

11.1　实验目的

本实验是一个扩展实例，介绍蓝牙小车的扩展应用——设备驱动。

在第 9-10 章的实验中，读者已经完成了一个简单的蓝牙小车应用，接下来我们通过修改内核代码，使读者能通过类似“/dev/stdout”的路径来访问设备文件，也就是在 SFS 层添加设备文件节点，同时利用 Hos 提供的添加设备驱动的机制，添加蓝牙设备驱动以及马达设备驱动，并且在/dev 目录下创建相应的节点

在本实验中，读者将会完成以下工作：

1.在 Hos 操作系统中实现在 SFS 层添加设备文件节点以及 mknod 调用，并进行测试

2.在 Hos 操作系统中实现添加设备驱动，并重新编译 Hos 内核。

3.编写并测试蓝牙和马达的设备驱动程序。

3.以设备驱动调用方式重写蓝牙小车应用并进行测试

11.2　实验内容

11.2.1　VFS 中添加设备文件结点

本次任务主要对以下三个文件进行更改：/kernel/fs/sfs/sfs_inode.c、/kernel/include/fs/sfs.h 以及/kernel/fs/devs/dev.c。

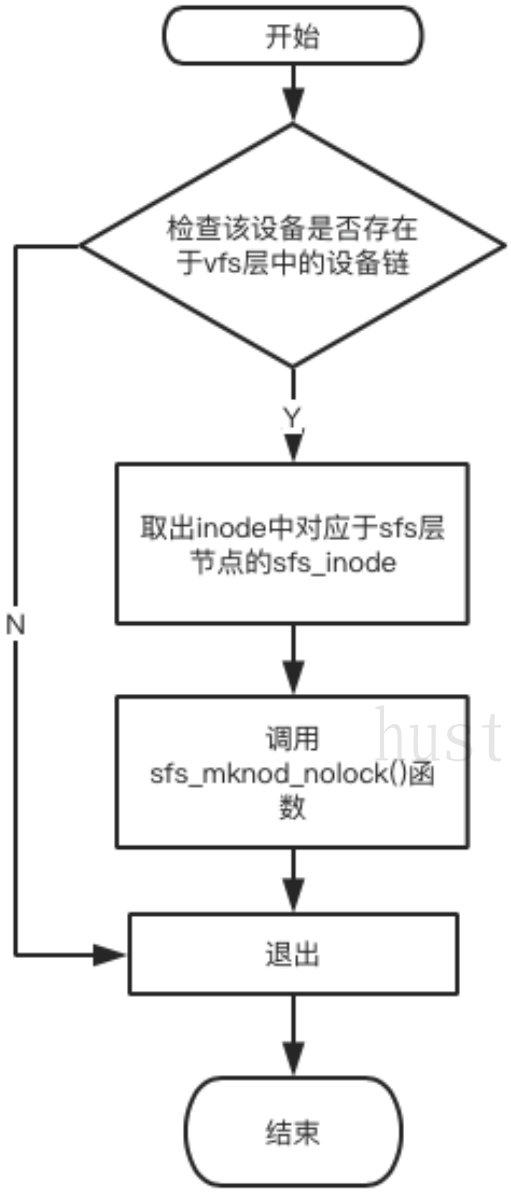
首先，要想添加代表设备的节点，则必须要有代表其的 file type 值，因此我们在 sfs.h 中新增了 SFS_TYPE_DEVICE 类型，并为其赋予一个特殊的值，接着我们要做的就是在磁盘索引节点 sfs_disk_inode 中添加设备文件的数据结构，因此我们新增了记录设备主次设备号的 dev_index 结构体并将其添加进索引节点中的 union 结构中，这样一来，磁盘索引节点中就有了代表设备文件的类型了，也就是说，Hos 除了支持普通文件、链接文件以及目录文件外，现在还可以支持设备文件了，但是根据上一章我们了解到，仅有索引节点还远远不够，我们还需要完善新增索引节点对应的操作函数，同时将其赋值给 inode_ops 结构体中的文件操作函数，只有这样，设备文件才能像其他文件一样向上提供一个统一的函数访问接口。

因此我们接下来要做的就是在 sfs_inode.c 中完成这些工作，首先增加声明 static const struct inode_ops sfs_node_devops，它声明了设备文件节点的函数访问接口。

接着分别编写 sfs_opendev()、sfs_closedev()、sfs_devread()、sfs_devwrite()、sfs_devfstat()、sfs_devioctl()、sfs_devgettype()、sfs_devtryseek()、sfs_devlookup()等函数。它们就是设备文件所对应的文件操作函数，然后在 sfs_node_devops 内完成赋值。这样一来设备文件节点的函数访问接口就完成了。

```
static const struct inode_ops sfs_node_devops = {
  .vop_magic = VOP_MAGIC,
  .vop_open = sfs_opendev,
  .vop_close = sfs_closedev,
  .vop_read = sfs_devread,
  .vop_write = sfs_devwrite,
  .vop_fstat = sfs_devfstat,
  .vop_fsync = NULL_VOP_PASS,
  .vop_mkdir = NULL_VOP_NOTDIR,
  .vop_link = NULL_VOP_NOTDIR,
  .vop_rename = NULL_VOP_NOTDIR,
  .vop_readlink = NULL_VOP_INVAL,
  .vop_symlink = NULL_VOP_NOTDIR,
  .vop_namefile = NULL_VOP_PASS,
  .vop_getdirentry = NULL_VOP_INVAL,
  .vop_reclaim = NULL_VOP_PASS,
  .vop_ioctl = sfs_devioctl,
  .vop_gettype = sfs_devgettype,
  .vop_tryseek = sfs_devtryseek,
  .vop_truncate = NULL_VOP_INVAL,
  .vop_create = NULL_VOP_NOTDIR,
  .vop_unlink = NULL_VOP_NOTDIR,
  .vop_lookup = sfs_devlookup,
  .vop_lookup_parent = NULL_VOP_NOTDIR,
  .vop_mknod = NULL_VOP_NOTDIR,
};
```


但是由于设备文件节点是统一管理在 dev/目录下的，为了方便学生在该目录下创建设备文件节点，参考 mknod 系统调用在 linux 中的实现原理，我们针对目录节点的操作函数访问接口做出了改变，新增函数 sfs_mknod_nolock()以及 sfs_mknod()用来在指定目录下新建设备文件节点，其中前者被后者调用，sfs_mknod()的工作流程如图 3.2 所示，它主要完成的工作就是取出 VFS 层 inode 中代表 sfs 层文件的 sfs_inode 节点，而前者的主要工作就是利用 sfs 层中已有的操作函数在 sfs 层创建一个设备文件节点。



在 sfs_mknod_nolock()函数的执行过程中，会依次调用 sfs 层中的操作函数 sfs_dirent_search_nolock()、sfs_dirent_create_inode()以及 sfs_dirent_link_nolock()，其中 sfs_dirent_search_nolock()是 sfs 层中常用的查找函数。它在目录下查找 name，并且返回相应的搜索结果（文件或文件夹）的 inode 的编号（也是磁盘编号），和相应的 entry 在该目录的 index 编号以及目录下的数据页是否有空闲的 entry。sfs_dirent_create_inode()函数负责创建设备文件对应的磁盘文件索引节点 sfs_disk_inode，并进一步初始化其对应的内存文件索引节点 sfs_inode，最后将 sfs_inode 与 VFS 层 inode 相 link。sfs_dirent_link_nolock()函数则负责创建 dev 目录下的设备文件节点与目录文件节点之间的链接。当完成了这些工作后，该设备文件

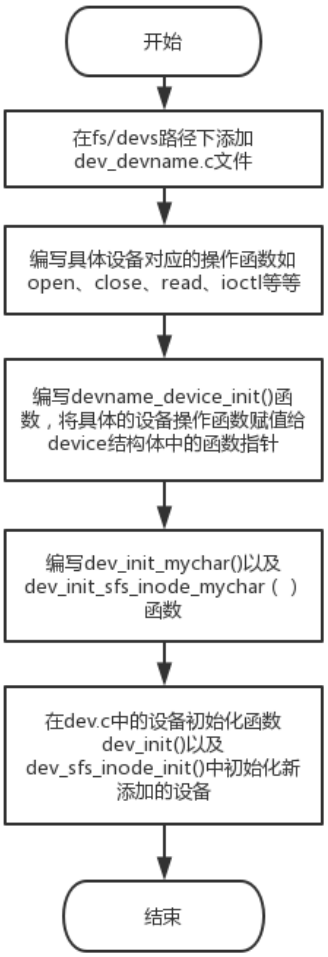
节点就和普通文件节点一样，可以通过文件系统层提供的统一接口函数来访问。

接着我们还需在 sfs_node_devops 中新增一行赋值.vop_mknod=sfs_mknod，这样当用户在 dev/目录下创建设备文件节点时，就会自动的转到目录节点对应的操作函数 sfs_mknod()函数来执行。

当然，仅完成这些还不够，在 dev.c 中也需要做出改变。在添加具体的设备驱动时，我们将通过 dev_make_sfs_inode()函数在 SFS 层添加该设备文件的磁盘节点，在这个函数执行过程中我们调用了 vop_mknod(dir, devname, index, &node)函数，它会进一步的调用上述的 sfs_mknod()函数，这样我们对 SFS 层所做的改变就联系在一起了。

11.2.2 添加设备驱动接口

Hos 提供了添加设备驱动的机制，但却并不是常见的模块化机制，而是通过内核中提供的接口函数，手动的将新建的设备驱动添加进内核，并重新编译内核，这两个函数分别是 dev_init_devicename（）和 dev_init_sfs_inode_devicename（）；前者负责为新建设备驱动创建 VFS（virtual File System）层节点，并将其添加进 Hos 中负责登记所有设备的双向设备链表 vfs_dev_t 中。后者则负责在 SFS 层为新建设备添加文件节点 inode。因为目前只是在设备驱动层有这些函数，还没有向内核登记这些初始化函数，所以最后还需要向上层的设备文件层提供这些函数接口，这样才算是完成了在内核中添加设备驱动的工作。具体的添加设备驱动的流程图如图 3.3 所示:



11.2.3 添加蓝牙和马达驱动函数

从上一章中我们可以了解到，蓝牙硬件接口采用串口实现，所以需要在内核头文件/kernel/include/arch.h 中注册蓝牙串口硬件模块的分配地址，如下所示。

蓝牙串口总线地址

```
#define BT_UART_BASE       0xB0500000

#define rbr         0*4    // In:receive buffer
#define ier         1*4    // Out: Interrupt Enable Register
#define fcr         2*4    // Out: FIFO Control Register
```

```
#define lcr        3*4    // Out: Line Control Register
#define mcr         4*4    // Out: Modem Control Register
#define lsr        5*4    // In: Line Status Register
#define msr         6*4    // Modem Status Register
#define scr        7*4

#define thr        rbr    // Out:Transmitter Holding Register
#define iir        fcr    // In: Interrupt ID Register
#define dll        rbr    // Out: Divisor Latch (Least Significant Byte) Register
#define dlm         ier
```

注册完总线地址后，就需要在/kernel/fs/devs/中设计相应设备了，从上章我们了解到蓝牙设备只有一个 sout 口，我们就是通过它来读取来自手机蓝牙传输的信号。因此在设备驱动中我们要做的就是从蓝牙串口的 receive buffer 中读取数据并放入驱动缓存区 bluetooth_buffer 中。但想要能够访问蓝牙串口，那就必须得按照 uart 通信协议来初始化这个蓝牙设备，这里我们通过 init_bluetooth()函数来完成，具体实现如图 3.5 所示：

```
void init_bluetooth(void) {
    *WRITE_IO(BT_UART_BASE + lcr) = 0x00000080; // LCR[7] is 1
    delay();
    *WRITE_IO(BT_UART_BASE + dll) = 69; // DLL msb. 115200 at 50MHz. Formula is Clk/16/baudrate.
    delay();
    *WRITE_IO(BT_UART_BASE + dlm) = 0x00000001; // DLL lsb. From axi_uart manual.
    delay();
    *WRITE_IO(BT_UART_BASE + lcr) = 0x00000003; // LCR register. 8n1 parity disabled
    delay();
    *WRITE_IO(BT_UART_BASE + ier) = 0x00000001; // IER register. disable interrupts
    delay();
}
```


图 3.4 蓝牙串口初始化函数

要想通过蓝牙串口接收信号，就要按照 uart 协议对蓝牙串口的关键寄存器进行如上所示的初始化工作，这样一来，蓝牙串口就开始工作了，我们就可以从串口的 rbr 中接收数据了。需要注意的是，进行接收数据前要检测蓝牙串口是否准备好与其他蓝牙设备进行通信。具体实现如图 3.5 所示：

```
char BT_uart_inbyte(void) {
    unsigned int RecievedByte;
    while((*READ_IO(BT_UART_BASE + lsr) & 0x00000001) != 0x00000001){
        delay();
    }
    RecievedByte = *READ_IO(BT_UART_BASE + rbr);
    return (char)RecievedByte;
}
```

图 3.5 蓝牙串口读取数据函数

添加完这两个主要函数之后，我们只需按照上一小节中介绍的添加设备驱动的流程完成蓝牙设备对应的 open、close、ioctl 函数即可，而这些函数可以参考 dev.c 中已有的设备驱动来编写，所以难度不大。需要注意的是蓝牙设备是只读设备，因此调用 open 函数时需要传入 open_flag 参数，检测用户是否具有读文件权限。

添加完蓝牙设备驱动并测试后，马达设备驱动的实现就驾轻就熟了。马达驱动的转动速度根据所给马达电压的不同而变，电压越大其速度越快。因此采用 PWM 来控制马达驱动的转速。通过 PWM，控制 CPU 发给马达驱动芯片的数据的不同占空比来控制电压的高低，马达驱动中对应的芯片 74HCT595N 将根据 01 数据流的占空比最终输出不同的电压，最终实现对马达驱动速度的控制。同时需要将 74HCT595N 芯片输出接到 4 个直流电机的 8 个引脚，它们分别控制马达驱动的正转反转，因此只要向 74HCT595N 控制模块的寄存器输入 8 位的值就能控制 4 个马达的正转反转，进而控制小车的前进后退左转右转等操作。

因此我们必须了解前进后退左转右转等操作所对应的写入芯片控制模块寄存器的值，经过测试得到，这些对应关系如表格 3 1 所示。

表格 3 1 操作命令对应的控制模块寄存器值

操作	写入控制模块寄存器的值
前进	0x2e
后退	0xd1
左转	0x9c
右转	0x63

接着，同样在内核头文件/kernel/include/arch.h 中注册马达驱动硬件模块的分配地址，如下所示。

马达驱动总线地址

```
#define CAR_DIR              0xB0700000
#define PWM_B_R_BASE      0xB0C00000
#define PWM_F_L_BASE      0xB0D00000
#define PWM_F_R_BASE      0xB0E00000
#define PWM_B_L_BASE      0xB0F00000
```

注册完总线地址后，就需要在/kernel/fs/devs/中设计相应设备了，从上章中我们了解到需要向马达驱动设备写入两个数据，一个控制了小车的速度，另一个控制了小车各个轮胎的正转反转，因此我们编写 writecar()函数来完成这个任务，具体实现如图 3.6 所示：

```
void writeCar(int fl,int fr,int bl,int br,int spi){
        *WRITE_IO(CAR_DIR) = spi;
        delay();
        *WRITE_IO(PWM_F_L_BASE) = fl*100000;
        delay();
        *WRITE_IO(PWM_F_R_BASE) = fr*100000;
        delay();
        *WRITE_IO(PWM_B_L_BASE) = bl*100000;
        delay();
        *WRITE_IO(PWM_B_R_BASE) = br*100000;
        delay();
}
```

图 3.6 马达驱动的写函数

其中，向 CAR_DIR 写入 spi 值，来控制小车四个轮胎的正转反转，向四个 PWM 设备总线地址分别写入速度值，马达驱动中的芯片 74HCT595N 将根据 01 数据流的占空比最终输出不同的电压，最终实现对马达驱动速度的控制。

添加完这个函数之后，同样的，我们只需按照上一小节中介绍的添加设备驱动的流程完成马达驱动设备对应的 open、close、ioctl 函数即可，需要注意的是马达驱动设备是只写设备，因此调用 open 函数时需要传入 open_flag 参数，检测用户是否具有写文件权限。

11.2.4 测试蓝牙小车应用

在完成两个设备驱动之后，我们还需要实现测试蓝牙小车的应用程序，为了在 Hos 中实现一个真实的应用；从蓝牙获取控制数据，对驱动板进行控制，使得蓝牙小车按照控制运行。并让蓝牙小车在移动电源的控制下脱离主机运行。因此，首先需要做的就是在/user/user-ucore/目录下添加测试小车所需的文件 testcar.c，具体实现如下图 3.7 所示。但这还不够，我们还要修改根目录下的 Makefile.config 文件，在“USER_APPLIST := …”一行后增加新添应用的名字。这样一来蓝牙小车应用就完成了。

```
int main(int argc, char **argv)
{
    int spd=9;
    int stat=0x2e;
    char c;
    char buf[100];
    const char *bt_path = "/dev/bluetooth";
    const char *car_path = "/dev/car";
    //open dev : bluetooth && car
    printf("Will open: %s\r\n", bt_path);
    int fd_bt = open(bt_path, O_RDONLY);
    if (fd_bt < 0) {
        printf("failed to open bt_path.\r\n");
        return fd_bt;
    }
    int fd_car = open(car_path, O_WRONLY);
    if (fd_car < 0) {
        printf("failed to open car_path.\r\n");
        return fd_car;
    }
    //operate car
    while(1){
        int rdnum = read(fd_bt,&c,1);
        printf("read from bt : %c, rdnum : %d \r\n", c, rdnum);
        delay();
        switch(c){
            case FRN://1
                stat=0x2e;
                memcpy(buf, &spd, 4);
                memcpy(buf+4, &stat, 4);
                write(fd_car, buf, 10);
                break;
            case BAK://2
                stat=0xd1;
                memcpy(buf, &spd, 4);
                memcpy(buf+4, &stat, 4);
                write(fd_car, buf, 10);
                break;
            case ACL://3
```


图 3.7 测试小车系统应用 testcar

11.3 实验背景及原理

11.3.1 Linux 系统设备驱动概述

Linux 和 Windows 是目前市面上使用最广泛的，也是大家熟悉成都最高的两个操作系统，它们毫无疑问地只吃了设备驱动，不过支持方式各有不同，因为 Hos-Mips 内核实现与 Linux 相类似，故该部分只对 Linux 的设备驱动进行概述。

Linux 是支持 POSIX 的类 UNIX 系统，所以 Linux 在对于设备驱动方面，沿用了与 UNIX 对于设备处理上相类似的架构[1]。即所有的设备都被会看做一个实际存在的，在用户角度看来跟其他普通文件一样的文件节点。这样，设备就被纳入了文件系统的范畴，可以被应用程序以设备无关的形式访问设备，可以像普通文件一样，通过文件 I/O 操作的界面，方便无缝地转为面向设备的操作，因此：

文件系统中有一个或多个文件，即节点代表每一个设备，也都会有文件名，所有的这些设备文件都能唯一确定中的某个设备。上层应用则可以跟访问普通文件一样，直接用文件名来访问那个设备，并且设备也同文件系统中的普通文件无异，受到访问权限机制的保护。[1]

当应用程序需要与特定设备通信，预备与之建立连接时，通常会使用 open 打开对应的设备。对应设备在文件系统中的节点包含了建立连接需要的相关信息。这个与特定设备简历连接的过程，在进程看来就是打开了一个文件。

进程与设备建立连接后（也即打开了对应文件），就可以通过标准库的操作文件的接口，如 read/write/fstat，对其进行操作。对于应用程序而言，设备文件被看做是一个线性空间，内核负责提供从这个线性空间到设备实际的物理空间的映射，从而文件操作与设备驱动则被分隔开来。

与很多操作系统一样，Linux 将设备归为两类：可随机访问，以块（block）为单位进行 I/O 操作的“块设备”；顺序访问，以字符（字节）为单位进行 I/O 操作的“字符设备”。文件系统通常建立在前者之上，但经过多年发展，两类设备的界限逐渐模糊，出现了如网络接口这样建有块设备和字符设备特点的设备。现在，块设备一般只用来表示以块为单位进行 I/O 操作，并且会在其上建立常规文件系统的设备。

Linux 使用主次设备号来标识设备，也就是之前提到的应用程序与目标设备建立连接时，节点中包含的信息之一（还有一个信息时设备文件的类型）。主设备号用于索引驱动程序，次设备号则用于索引同类设备。Linux 下主设备号有 8 位，这意味着块设备和字符设备的种类最多只可能有 2^8 种[1]，次设备号也是 8 位，因此应用程序可以访问 65536 个设备。Linux 通常会把所有的代表设备的文件节点，放在磁盘根目录下的一个名为 dev 目录中。设备文件节点本身比较特殊，它在磁盘上仅占一个索引节点，不像常规文件，它不与存放数据的记录块相联系。

在给一个设备编写好驱动程序后，如何将其装入内核设备驱动层也是一个重要的问题，如果将其静态链接在内核中，不但会让内核体积过大，而且也无法扩展，因此需要能够动态地将驱动程序插入到内核的机制，它既要能在多用户多进程环境下动态安装，同时也能保证内核不受破坏。Linux 采用了可安装模块的机制，module 实际上就是尚未链接的目标文件，可在系统运行时由用户或操作系统管理员动态地安装至内核中[10]，已安装的模块也可被移除，给与内核极大的灵活性。动态安装有两种取到，一是由内核在需要这个模块时自动启动，或是由拥有管理员权限的用户直接安装[11]，驱动程序正是通过模块来实现安装的

11.3.2 Hos 标准输入/标准输出设备

在 fs/devs 目录下，一些关键的设备驱动已经被实现，包括空设备 null、标准输入 stdin、标准输出 stdout、默认磁盘 disk0 等，这部分对标准输入/标准输出设备的实现做一些说明。

stdin 设备文件

stdin 这里实际就是键盘输入，它对自己的 device 结构体做了如下初始化：因为是字符设备，故 d_block 成员被赋值为 0，d_blocksize 为 1，然后将访问操作的实现函数指针赋值给 device 中对应成员，再初始化用于描述缓冲区的读写位置的两个变量，最后初始化了一个等待队列，它被用于等待缓冲区。

stdin 是只读文件，因此其 open 操作将检查 open_flags 参数，如果为 O_RDONLY 外的其他值均返回 E_INVAL 错误。stdin 的读写操作只接受读，否则返回错误，读操作的实现在 dev_stdin_read，利用了一个缓冲区 stdin_buffer 用于存放键盘输入的字符。

该驱动程序与其他驱动程序不同的是定义了一个外部调用的接口 dev_stdin_write，用于写入字符到键盘缓冲区 stdin_buffer，它会在操作系统的键盘中断触发时被调用。

stdout 设备文件

stdout 这里实际就是输出到 console 设备（目前仅实现了串口），它对 device 结构体的初始化工作与 stdin 相同，没有缓冲区的初始化工作。其设备操作的实现也非常简单，主要是 io 操作，它会调用 cputchar 来将 iobuf 的数据逐字符输出到串口。

11.3.3 主要数据结构

首先需要设计的是表示设备的结构体 struct device，它包含具体设备的一些信息，当中包括了设备操作的函数指针。struct device 的字段信息见**错误!未找到引用源。**。

表 3-1 struct device 成员

成员名	描述
d_blocks	表示设备占用数据块的个数
d_blocksize	表示设备数据块的大小
d_open	打开设备操作的函数指针，参数列表为一个指向 device 结构体的指针和文件打开标志位 open_flags
d_close	关闭设备操作的函数指针，参数列表为一个向 device 结构体的指针

d_io	读写设备操作的函数指针，参数列表为一个向 device 结构体的指针，一个指向缓冲区结构体 iobuf 的指针以及一个 bool 值区分读写
d_ioctl	通过 iotcl 的方式控制设备操作的函数指针，数列表为一个向 device 结构体的指针，一个整形变量表示要执行的操作，一个指向用于操作的数据区的 void 型指针

通过这个数据结构，块设备和字符设备的表示得以支持，即通过 d_blocks 成员（设备占用的数据块个数，块设备字符设备则为 0）和 d_blocksize 成员（设备数据块的小，字符设备则为 1）。device 结构体含有四个设备基本操作的指针，编写设备驱动时，只需将合适的实现赋给该指针即可，有了这个 device 结构体，设备的基本信息得以保存，后续如果需要扩展设备的信息，如增加设备操作指针，也可以很方便地修改添加。struct devic 在 include/fs/dev.h 中。同时，还配套定义了四个宏操作 dop_oepn, dop_close, dop_io, dop_ioctl，它们分别接受一个 dev 结构体指针和设备操作需要的参数，以提供一个更便利的执行设备操作的接口。

上述 decice 结构体并没有与文件系统相关联，即将上述信息加入到 inode 中，因此需要修改 inode 结构体的定义，struct inode 在 include/fs/inode.h 中，这个结构体需要修改的字段：

in_type，一个枚举型成员，用于表示该 inode 节点所属的文件系统类型，为了支持在 inode 中加入设备的相关信息，in_type 的枚举类型需要新增一个枚举成员 inode_type_device_info 表示该 inode 是设备节点

in_info，一个 union 成员，它与 in_type 相对应，即 in_type 相应文件系统类型的 inode 信息，上述 device 结构体就保存为这个变量，即在 in_info 的联合体类型中增加一个 device 类型成员__device_info

从代表设备的 inode 获取设备信息即 device 结构体时，也直接通过 vop_info 来操作，该宏会检查 in_type 是否与 in_info 相匹配，上述成员名修改复合该宏的检查机制，故可以正常使用。

接下来考虑的是本文 2.4 提到的，所有的设备信息，即上述 device 结构体的实例，需要通过某种形式组织起来，以便查找设备时获取，因此这里定义了一个设备链表，将所有设备信息成链组织起来[13]。并且，因为是将 device 信息加入到了虚拟文件系统层的 inode 而不是具体文件系统层的文件节点，故还需要定定义一个数据结构，将设备对应的 inode 与设备相关信息绑定在一起，作为设备链表的一个元素。struct vfs_dev_t 的字段信息见表 3 2。

表 3 2 struct vfs_dev_t 成员

成员名	描述
devname	表示设备的设备名
minorindex	表示设备的索引号，这里指此设备号
devnode	表示设备对应的 inode，它是该结构体的关键，通过它来获取到与所查找设备的信息（device 结构体）
fs	表示该设备是否可挂载，这是一个 bool 型变量，在初始化的时候赋值，如果 fs 成员需要被赋值，则它必须是可挂载的，即 moubtable 需要为 true 值

mountable	表示该设备是否可挂载，这是一个 bool 型变量，在初始化的时候赋值，如果 fs 成员需要被赋值，则它必须是可挂载的，即 moubtable 需要为 true 值
vdev_link	用于实现双向循环链表的链表节点，hos-mips 中定义了一个通用的双向循环链表实现，这个成员依照规范加入

通过将上述 vfs_dev_t 结构组织成链[19]，并定义一系列查找及其他操作函数，就能通过设备号来获得对应的设备文件节点（inode）及设备结构体（device）信息，这样设备相关信息就与其对应的 inode 绑定在了一起。

上述 vfs_dev_t 中虽然有一个设备号成员，但是它只被定义为表示此设备号，要实现对主次设备号的支持，这里选择使用十字链表的形式组织设备信息，即定义一个以主设备号作为标识的结构体链表，每个结构体中包含同一主设备号的所有设备组成的链表，也即是上述 vfs_dev_t 类型的链表。该数据结构被定义为 vfs_dev_major，包含的成员列表见表 3 3。

表 3 3 struct vfs_dev_major 成员

成员名	描述
index	表示主设备链的索引号，即主设备号
vdev_list	主设备所包含的设备链头，即同一主设备号的所有设备组成的链表，这是 hos-mips 中通用双向循环链表的的实现规范连表头
vdev_major_link	用于实现双向循环链表的链表节点，hos-mips 中定义了一个通用的双向循环链表实现，这个成员依照规范加入

上述 vfs_dev_t 结构体和 vfs_dev_major 结构体都在 fs/vfs/vfsdev.c 中，其中 vfs_dev_major 主设备链的链表头静态变量也被定义在该文件

11.3.4 虚拟文件系统层

vfs 层查找操作的接口是 vfs_lookup 和 vfs_lookup_parent。vfs_lookup 函数用于查找文件名对应的 VFS 层 inode，接受一个路径参数和一个存放结果 inode 的指针，vfs_lookup_parent 函数用于查找路径所在目录的 VFS 层 inode，接受一个路径参数，一个存放结果目录 inode 的指针以及一个存放路径中最后一级文件名的指针。这两个函数的实现都是通过调用对应具体文件系统层的函数，即通过 vop 系列操作的宏，来得到对应节点，因此实现思路是分别在调用这两个转调到具体文件系统层的宏之前，就在设备链中查找，当传入的路径参数是一个设备的时候，则作相应处理返回适当的 inode，如返回设备根目录的 inode 或传入的路径参数在设备之后的路径对应的 inode。因此，上述在设备链查找的一系列操作可以实现为一个函数供 vfs_lookup 和 vfs_lookup_parent 分别调用，该函数实现为 getdevice，定义在 fs/vfs/vfs_lookup.c 中。该函数原型如下：

```
int get_device(char *path, char **subpath, struct inode **node_store);
```

get_device 函数的作用是根据传入的路径，获得这个路径所在起始目录的 inode，如果该路径中包含设备信息，则在设备链中查找相应设备然后获得设备路径中去掉路径中的设备信

息后剩余部分所在起始目录的 inode。该函数接受一个 path 参数，用于传入要查找设备的全部路径，一个 subpath 参数，用于存放将 path 参数中表示设备的部分路径截断掉之后剩余的子路径，一个 node_store 参数，用于存放查找结果的 inode。

为了便于查找的实现，对设备的打开的形式定义为“device:path”，即扩展了传统 UNIX 路径[17]，分为两部分，前半部分表示要打开的设备名，后半部分则是原来的路径，两者通过冒号分隔，表示 device 设备中的 path 路径。在这种扩展路径下，传统的 UNIX 路径则被当成缺省情况：即将某个挂载了文件系统的设备设置为默认设备，如果路径没有指定 device 就，转为对默认设备的路径，这就是“启动文件系统”bootfs，相关的实现在 vfs.c 中，这里需要对操作系统的另一个部分进程管理做一些修改，在 init 进程对应的进程执行体 init_main 中调用 vfs_set_bootfs，将默认磁盘 disk0 挂载的文件系统设置为 bootfs。

定义了以上对路径的扩展后，get_device 的函数实现流程描述如下：首先对传入的路径字符串进行遍历查找是否有分隔符“：”，如果没有则查找第一个“/”的位置。遍历完路径之后，分几种情况进行处理：

没有分隔符“：”并且第一个“/”不出现在路径第一个字符，表示这个路径是相对路径，那么剩余子路径 subpath 还是原来的路径 path 保持不变，并则转而调用 vfs_get_curdir 函数获得当前进程所在目录的 inode

出现了分隔符“：”并且“：”不出现在路径第一个字符，那么路径 path 分隔为两部分：冒号前为设备名，冒号后为子路径 subpath。这种情况下则获得前半部分设备名设备的根目录节点（如果该设备挂载了文件系统），或者说是表示该设备“入口”的节点，也就是代表这个设备的 inode

没有分隔符“：”并且第一个“/”出现在路径第一个字符，表示这个路径是绝对路径，也即“启动文件系统”根目录的相对路径，调用前述提到的 vfs_get_bootfs 来获取启动文件系统的根目录的 inode

分隔符“：”出现在路径第一个字符，表示这个路径是以当前目录所在设备的文件系统的相对路径，因此先调用 vfs_get_curdir 获得当前进程所在目录的 inode，然后通过 inode 的 in_fs 成员得到它所在的设备的文件系统，最后通过文件系统定义的宏 fsop_get_root 来调用抽象文件系统结构体对应的获取根目录的函数，来得到该文件系统根目录的 inode

通过以上的四种情况，get_device 函数得以实现获得传入路径所在设备起始目录的 inode，后续则是原来 vfs_lookup 和 vfs_lookup_parent 的逻辑，根据取得的 inode 调用其具体文件系统的 lookup 操作获得最终路径需要查找的文件的 inode，例如如果是普通文件，其具体文件系统在 Hos-Mips 是 SFS，则会调用到 sfs_lookup 操作。

剩下的一个问题就是在拿到设备对应的 inode 之后，如何使得宏 vop_lookup 或 vop_lookup_parent 对设备 inode 的调用能够转调到在设备驱动程序中实现的设备操作函数，因为为了遵循抽象以及封装的原则，同时也必须简化实现驱动程序的流程，不能使驱动编写过程直接与虚拟文件接口层相接触或者说为了实现设备驱动而对 vop_lookup、vop_lookup_parent 两个宏，甚至 VFS 层定义的 file_operations 的抽象函数指针数组作特殊的更改，这样会破坏整个文件系统的封装和层次性，造成后续维护、新增功能不变[15]。这个问题的解决方案是提供一个间接调用设备驱动操作函数的包装 file_operations 数组，这个数组

里设备需要实现的函数指针指向的函数实现是已经写好的，这些函数会从 inode 中取出 device 结构体，拿到设备具体信息，而 device 结构体中存放了设备驱动程序中实现的设备操作的函数指针。这样，这些包装函数就能将文件操作中转到调用驱动程序的设备操作，而无需对原本的 VFS 层做任何修改。这些包装函数和设备 inode 的 file_operations 数组被定义在 fs/dev/dev.c 中。目前它实现了 open、close、read、write、fstat、ioctl、gettype、tryseek 和 lookup 操作。

11.3.5 驱动接口

原型	说明
void vfs_cleanup(void)	遍历所有设备，如果其挂载了文件系统，对其执行 cleanup 操作
int vfs_sync(void)	遍历所有设备，如果其挂载了文件系统，对其执行 sync 操作
int vfs_get_root(const char *, struct inode **)	根据设备名字符串查找设备，并获得该设备挂载文件系统的根目录 inode 或该设备本身对应的 inode
const char *vfs_get_devname(struct fs *)	根据传入的抽象文件系统查找挂载它的设备被的名称
int vfs_get_devnode(struct dev_index, struct inode **)	根据设备号获得设备对应的 inode
int vfs_add_dev(struct dev_index, const char *, struct inode *, bool)	将一个新设备加入到设备链中，需要传入设备号、设备名称、设备对应的 inode 以及设备是否可挂载
int vfs_add_fs(struct dev_index, const char *, struct fs *)	将一个文件系统加入到设备链中，这个文件系统不依赖于实体设备，如 emufs 或网络文件系统
struct dev_index vfs_register_dev(unsigned int, const char *)	注册设备，传入主设备号及设备名，获得被填入合适次设备号的完整设备号，必须在新设备加入设备链之前调用
int vfs_mount(const char *, int (*) (struct device * dev, struct fs ** fs_store))	利用传入的函数指针来挂载一个设备的文件系统，通过设备名在设备链中查找设备
int vfs_unmount(const char *)	取消挂载一个设备的文件系统，通过设备名在设备链中查找设备。取消挂载前会首先调用文件系统的同步操作
int vfs_unmount_all(void)	对设备链中所有挂载了文件系统的设备执

	行取消挂载的操作

上述实现中，有几个重要接口的实现需要单独说明。

vfs_get_root 函数，它会被 3.2.1 中提到的 get_device 函数调用，即 get_device 对传入路径参数四种情况处理的第二种——出现了分隔符“:”并且“:”不出现在路径第一个字符，这个情况下路径中的设备名会被分离出来然后传给 vfs_get_root。vfs_get_root 会遍历所有设备，当找到对应的设备之后，会判断该设备是否已挂载文件系统，如果是，则对设备挂载的抽象文件系统结构体执行 get_root 操作获得其根目录，如果该设备未挂载，则返回该设备本身的 inode。

vfs_register_dev 函数和 vfs_add_dev 函数，这两者都是暴露给驱动编写者的，驱动编写者需要在驱动的初始化函数中，首先调用 vfs_register_dev 来注册设备，确定设备的主设备号，该函数会查找对应主设备号的 vfs_dev_major 是否已创建，如果未创建则会创建一个 vfs_dev_major 结构体并加入到 vfs_dev_major 链中，然后返回该 vfs_dev_major 的次设备链的最近一个可用的次设备号，组合成 dev_index 设备号结构体返回。注册好结构体后将其传入 vfs_add_dev，该函数会检查设备名是否唯一，并分配 vfs_dev_t 设备相关信息结构体，将其初始化然后挂到对应设备号的此设备链中。

第12章　实例 1：蓝牙小车 flash 启动的实现

12.1　实验目的

本实验是一个扩展实例，介绍蓝牙小车的扩展应用——flash 自启动。

在上两节的实验中，读者已经完成了基本的蓝牙小车的设计实验，接下来我们将会在基本的蓝牙小车系统上，利用 flash 存储器实现一个可自起的蓝牙小车。在硬件上我们会利用 spi 模块控制 flash 存储器，并固化 bootloader，在 Hos 上实现相应模块

在这个实验中，读者将会完成以下工作：

1.在 MIPSfpga 系统中添加 SPI 模块控制 FLASH

2.修改并添加固化的 bootloader

3.

在基于 MIPSfpga 系统的 Hos 操作系统实现蓝牙模块和马达模块驱动，并实现蓝牙小车的应用。

在上一个实验中，读者完成了蓝牙小车重要外设模块硬件接口，接下来我们利用包含这些硬件外设的 MIPSfpga 系统，在 Hos 操作系统中实现一个利用手机无线蓝牙控制的蓝牙小车应用，该应用能控制小车前进、倒车、转向等基本功能，有能力的读者可以完成更加复杂的小车动作，例如加速、减速等其他动作。

1.在 Hos 操作系统中实现无线蓝牙设备和马达驱动板设备驱动程序，并进行测试

2.在 Hos 操作系统用户态中通过设备驱动的调用完成蓝牙小车应用程序的设计实现，并进行测试

以此读者将学习理解 Hos 操作系统外设驱动实现与调用原理，加深对操作系统外设驱动的实现理解，同时读者可以发现操作系统和裸机在外设设备驱动和用户应用的区别

12.2 实验内容

12.2.1 基于 AXI 的 SPI 接口规范的 flash 设计

1.SPI 模块

复制第 9 章的蓝牙小车工程，在 block Design 选页卡下右键添加 SPI 的 IP 模块，双击进入模块，如，不勾选 XIP 和 Performance 模式，使用标准模式。 SPI 选项中，Mode 模式设置为 Quad，No. of Slave 从器件数为 1，Slave Device 从器件为 Spansion，Transaction Width 传输宽度默认为 8，Frequency Ratio 默认为 2。FIFO Depth 设置为 256，使能 STARTUP2 Primitive

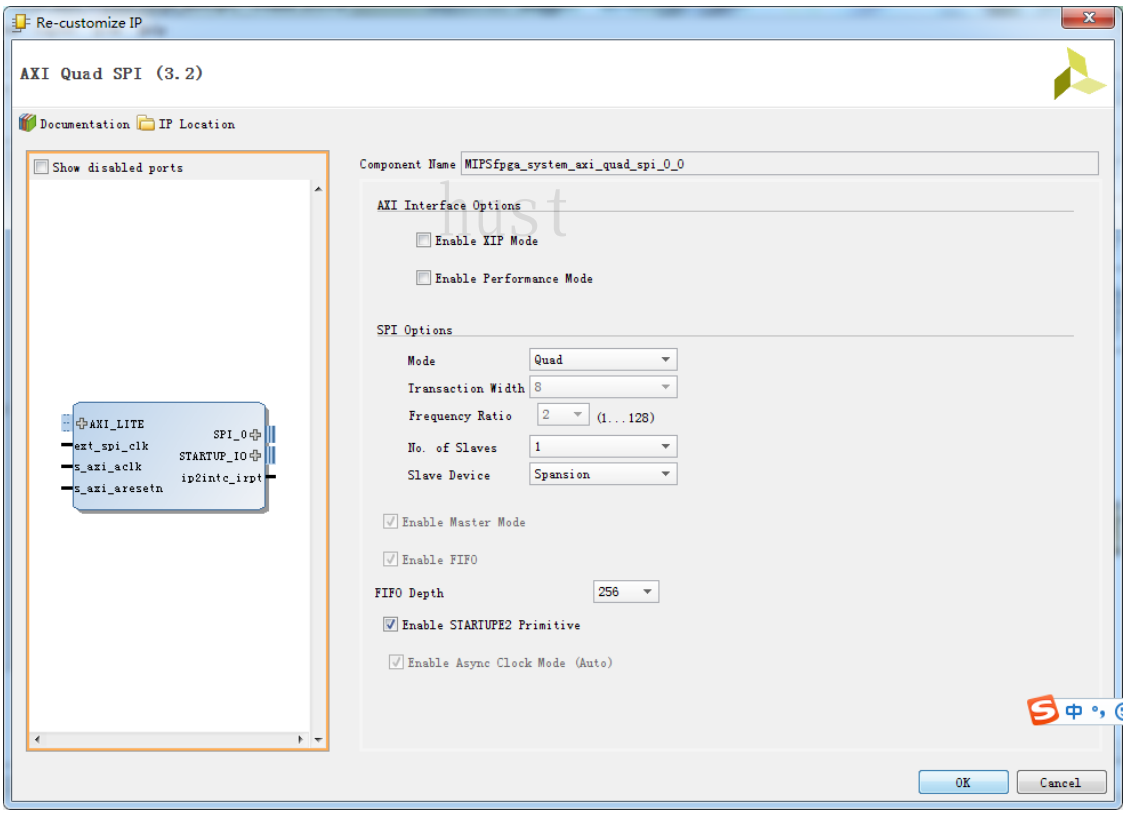


图 第 12-1 AXI Quad SPI IP 核的配置

将 AXI 接口与系统的 AXI4 接口相连，ext_spi_clk 直接与系统时钟连接，s_axi_aclk 和 s_axi_aresetn 分别与对应的系统 aclk 和 aresetn 相连，SPI 输出端信号设置为外部引脚（Make External），如所示。

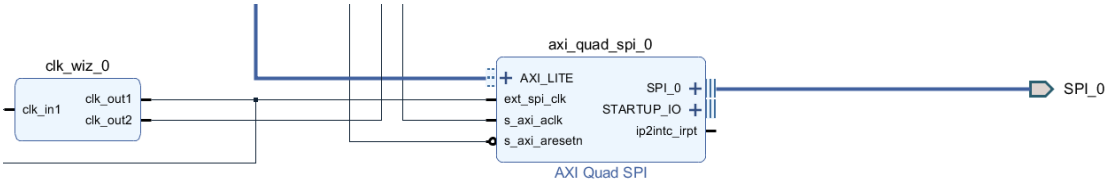


图 第 第 12-2

然后同样在 address 分配相应地址，读者需在蓝牙小车的 xdc 约束文件上添加相应引脚：

```
set_property -dict { PACKAGE_PIN K17     IOSTANDARD LVCMOS33 } [get_ports { spi_0_io0_io }];
#IO_L1P_T0_D00_MOSI_14 Sch=qspi_dq[0]
set_property -dict { PACKAGE_PIN K18     IOSTANDARD LVCMOS33 } [get_ports { spi_0_io1_io }];
#IO_L1N_T0_D01_DIN_14 Sch=qspi_dq[1]
set_property -dict { PACKAGE_PIN L14     IOSTANDARD LVCMOS33 } [get_ports {   spi_0_io2_io }];
#IO_L2P_T0_D02_14 Sch=qspi_dq[2]
set_property -dict { PACKAGE_PIN M14     IOSTANDARD LVCMOS33 } [get_ports {   spi_0_io3_io }];
#IO_L2N_T0_D03_14 Sch=qspi_dq[3]
set_property -dict { PACKAGE_PIN L13     IOSTANDARD LVCMOS33 } [get_ports { spi_0_ss_io }];
#IO_L6P_T0_FCS_B_14 Sch=qspi_csn
```

接着，读者就可以编译生成 Bitstream 了。

2.Flash 存储器烧写

打开 Vivado Tcl 窗口，输入以下 write_cfgmem 命令，创建用于编程闪存的文件。

```
write_cfgmem -format mcs -interface spix4 -size 16 -loadbit "up 0x0 D://MIPSfpga_system.bit" -loaddata "up 0x00400000 ucore-kernel-initrd" -file D://download.mcs
```

-format mcs 指示生成 mcs 格式的文件，spix4 为模式设置，指示接口的宽度为 4，-size 16 为 Flash 大小，单位 Byte（16Byte=128bit）。首先加载硬件设计 bit 文件，从地址 0x00000000 处生成配置比特流，然后加载操作系统数据文件，从 0x0040000 处生成比特流。可得到 download.mcs 和 download.prm 两个文件。

在执行这个命令时，可能出现错误，Tcl 窗口显示如图 3-8 所示。

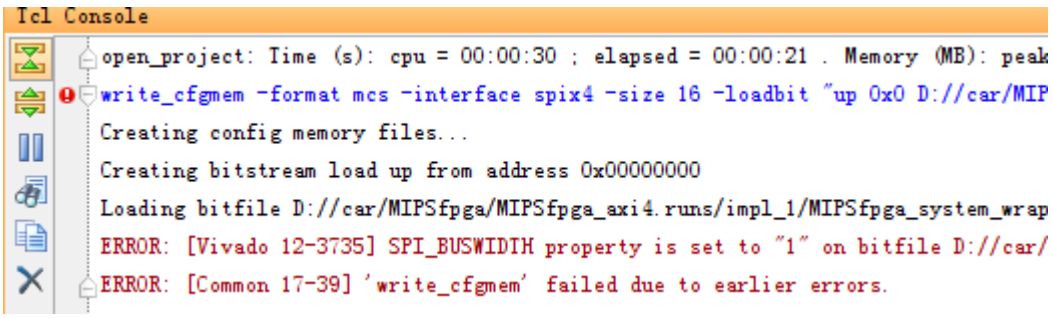


图 3-8 SPI BUSWIDTH 出错

这是因为 Vivado 工程默认 SPI 总线宽度为 1，但是命令要求的模式是 4，所以提示 SPI 的总线宽度矛盾，要在引脚文件中添加这样一句：

```
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
```

将生成的比特流的 SPI_BUSWIDTH 位宽更改为 4bit。将 BIT 位宽更改为 4bit 可以更快地配置 FPGA，有效节省等待时间。

Tcl 窗口显示运行结果如图 3-9，显示了配置闪存文件的信息，详细说明了两个文件在配置文件中各自的起始地址。

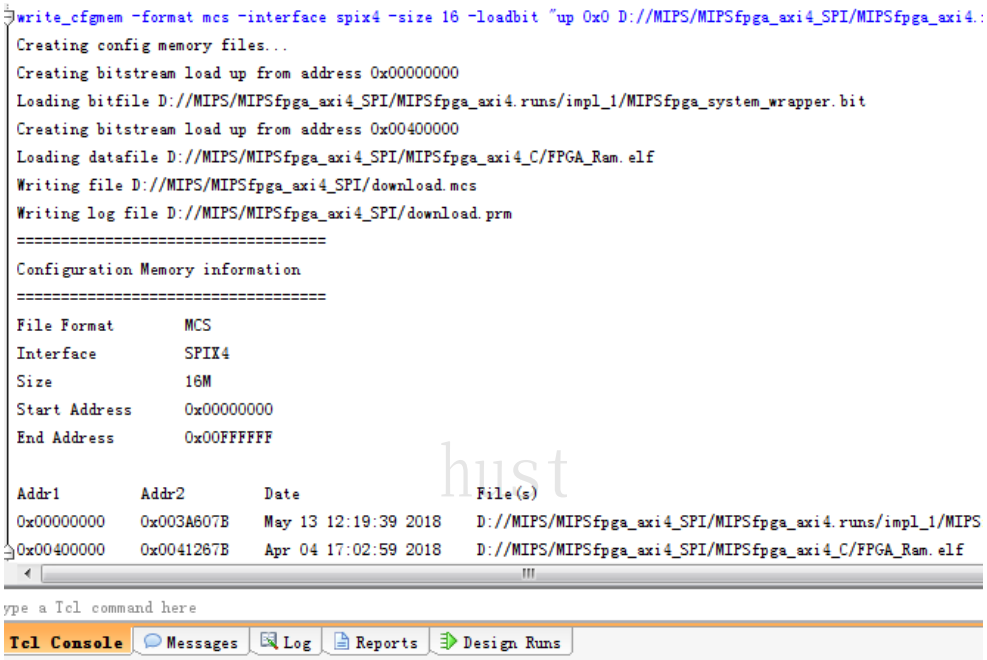


图 3-9 生成 mcs 文件的 Tcl 窗口显示图

用 JTAG 线将 Nexys4 DDR 开发板连接到电脑，打开 Vivado 的硬件管理器 hardware manager，识别出器件。鼠标点击扫描出的器件，右键点击器件，选择添加配置存储器设备 add configuration memory device，选择 flash 型号为 s25fl128sxxxxxx0-spi-x1_x2_x4。

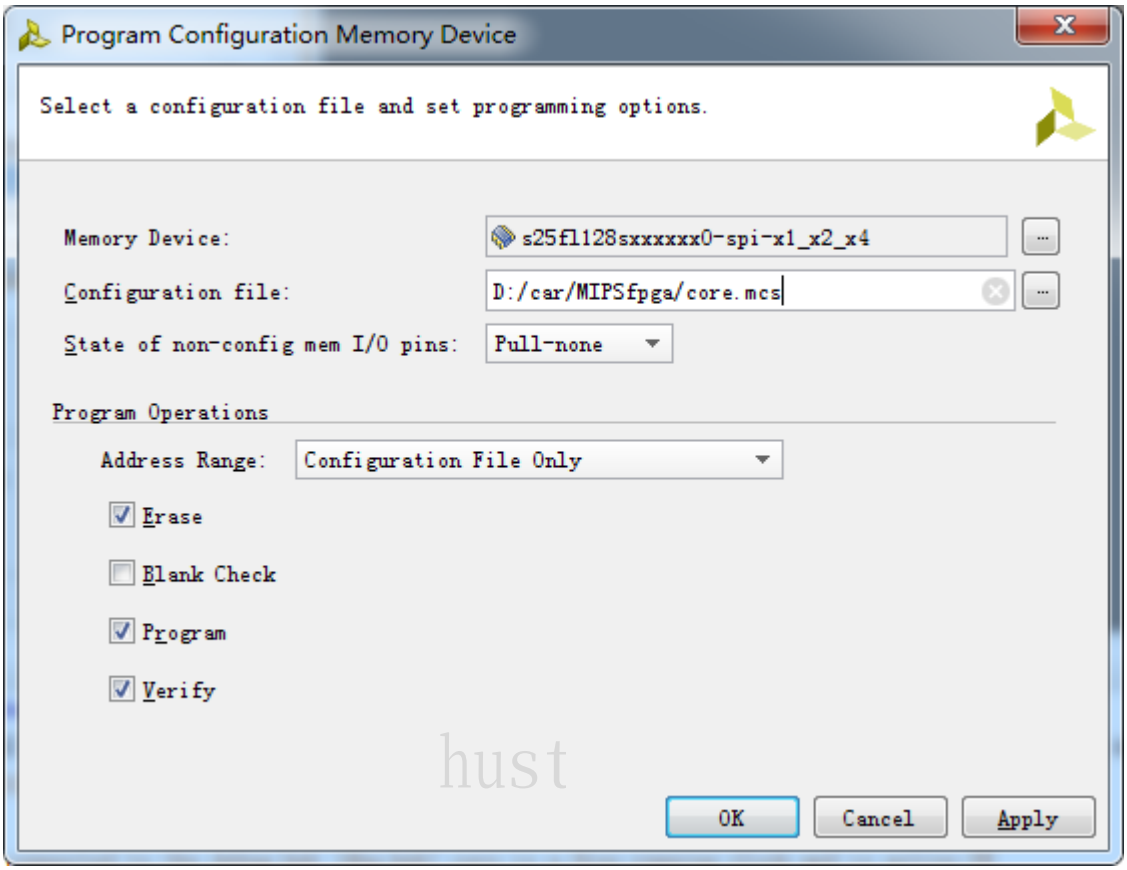


图 3-10 编程配置存储设备

鼠标点击添加的存储设备，右键点击器件，选择编程配置存储器设备，选择相应的 mcs 文件编程即可，如图 3-10 所示。编程过程大概 3~5 分钟，编程成功会弹出提示信息，如图 3-11 所示。

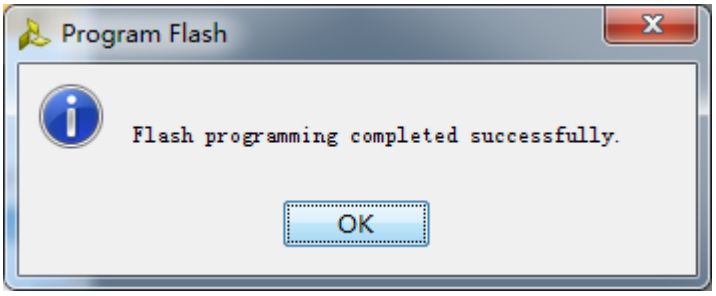


图 3-11 成功编程 Flash

在编程的同时，Vivado Tcl 窗口会显示提示信息，操作对应的命令为 program_hw_cfgmem，下面还显示了配置存储器的各项信息，制造商 ID （Mfg ID）为 1，存储器类型号（Memory Type）为 20，存储容量（Memory Capacity）为 18，设备 ID 1 为 0，设备 ID2 为 0，如图 3-12 所示。

```
program_hw_cfgmem -hw_cfgmem [get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
Mfg ID : 1   Memory Type : 20   Memory Capacity : 18   Device ID 1 : 0   Device ID 2 : 0
Performing Erase Operation...
Erase Operation successful.
Performing Program and Verify Operations...
Program/Verify Operation successful.
INFO: [Labtoolstcl 44-377] Flash programming completed successfully
program_hw_cfgmem: Time (s): cpu = 00:00:02 ; elapsed = 00:02:33 . Memory (MB): peak = 1321.961 ; gain = 0.000
endgroup
```

图 3-12 编程 Flash 的 Tcl 窗口

2.Bootloader

在原始的 MIPSfpga 系统中，我们仅仅在 BRAM 中放置了一段跑马灯输出 fpga 的简单代码用于表示 MIPSfpga 正常，然而由于系统现在使用 Flash 自举方式启动，所以需要在 BRAM 中放置一个 Bootloader 代码，该代码仍为 BRAM 初始化 coe 文件。

启动代码的流程大致如下：

步骤一，初始化硬件；

步骤二，把操作系统内核镜像文件从 FLASH 设备复制到内存；

步骤三，解析操作系统头文件，获取入口地址；

步骤四，跳转到操作系统入口地址处，开始执行操作系统。

范例代码见后

3.Hos 操作系统部分

12.3 实验背景知识及原理介绍

1.1.1 AXI Quad SPI

AXI Quad SPI IP 核将 AXI4 和 AXI4-Lite 接口连接到支持标准双路或四路 SPI 协议指令集的 SPI 从器件。如**错误!未找到引用源。**所示为 AXI Quad SPI IP 核的顶层框图。

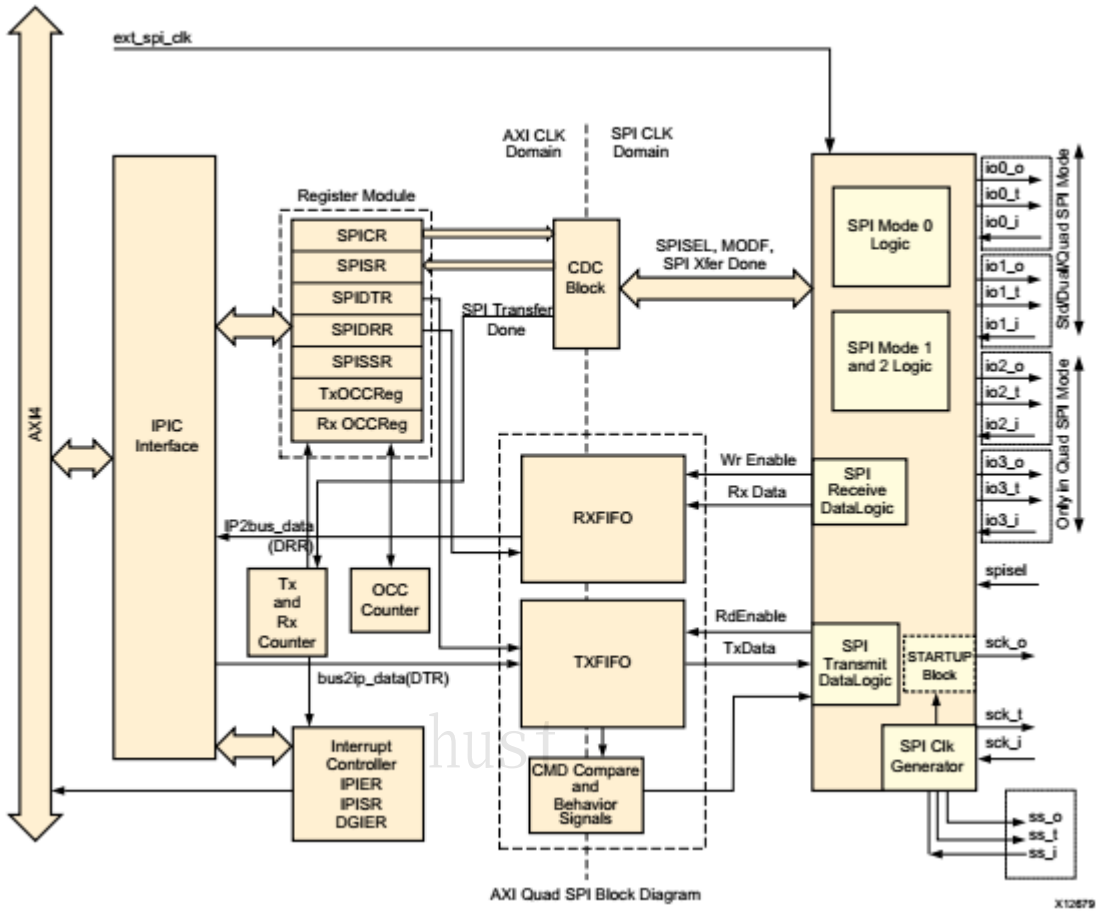


图 3-13 AXI Quad SPI 顶层框图

错误!未找到引用源。列出了传统模式的一组寄存器，这些寄存器可通过 AXI4-Lite 接口以 32 位的形式进行配置和访问。

表 3-2 AXI Quad SPI 寄存器表

地址空间偏移	寄存器名称	访问类型	默认值(hex)	描述
40h	SRR	Write	N/A	Software reset register
60h	SPICR	R/W	0x180	SPI control register
64h	SPISR	Read	0x0a5	SPI status register
68h	SPI DTR	Write	0x0	SPI data transmit register. A single register or a FIFO

地 址空间偏移	寄存器名称	访问类型	默认值(hex)	描述
6Ch	SPI DRR	Read	N/A	SPI data receive register. A single register or a FIFO
70h	SPISSR	R/W	No slave is selected 0xFFFF	SPI Slave select register
74h	SPI Transmit FIFO Occupancy Register	Read	0x0	Transmit FIFO occupancy register
78h	SPI Receive FIFO Occupancy Register	Read	0x0	Receive FIFO occupancy register

关于各个寄存器更加详细的描述可以参考 Xilinx IP 核产品手册 LogiCORE IP AXI Quad Serial Peripheral Interface Product Guide。建议详细了解寄存器。

1.1.2 FPGA 配置与 Quad-SPI Flash

打开电源后，Artix-7 FPGA 必须先配置，然后才能执行功能。有以下四种方式进行配置：

1.编程主机通过 Digilent USB-JTAG（portJ6，标记为“PROG”，**错误!未找到引用源。**中的 2）连接到 FPGA，任何时候都可以进行编程。

2.存储在非易失性串行闪存器件中的文件可以通过 SPI 端口传输到 FPGA。

3.从微型 SD 卡传送到 FPGA。

4.从连接到 USB HID 端口的 USB 记忆棒传输。

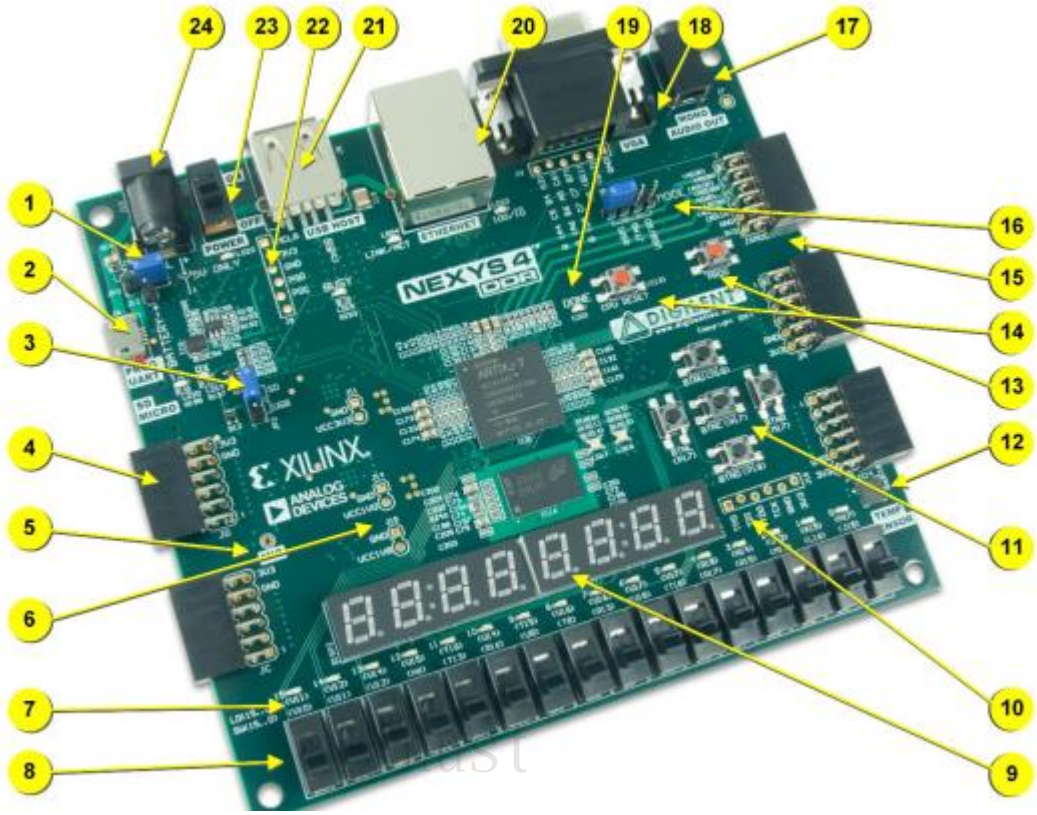


图 3-14 Nexys4 DDR 开发板

FPGA 配置数据存储在比特流文件中，比特流被存储在 FPGA 内基于 SRAM 的存储单元中，这些数据定义了 FPGA 的逻辑功能和电路连接。成功编程后，“编程完成”指示灯（**错误!未找到引用源。**中的 19）点亮。按下“PROG”按钮（**错误!未找到引用源。**中的 13），FPGA 内的配置将会复位，并通过编程模式跳线（**错误!未找到引用源。**中的 16）选择的方法重新编程。

由于 Nexys4 DDR 上的 FPGA 是易失性的，因此它依靠 Quad-SPI 闪存来存储电源周期之间的配置。空白 FPGA 作为主设备，在上电时从闪存设备读取配置文件。为此，需要首先将配置文件下载到闪存。在对非易失性闪存器件进行编程时，一个比特流文件将以两个步骤传输到闪存。首先，FPGA 用可对闪存器件进行编程的电路进行编程，然后通过 FPGA 电路将数据传输到闪存器件。这被称为间接编程。闪存器件编程完成后，可以在随后的上电或复位事件中自动配置 FPGA。存储在闪存设备中的编程文件将一直保留，直到它们被覆盖。

Quad-SPI Flash 器件内部包含多个地址空间。大部分命令在主闪存阵列上执行，还有的在其它地址空间上运行，这些地址空间使用完整的 32 位地址，但可用地址空间可能只定义了地址的一小部分。

闪存阵列：主闪存阵列被划分为称为扇区的擦除单元。这些扇区的组织方式可以是 4 kB 和 64 KB 扇区的混合组合，也可以是统一的 256 KB 扇区，取决于所选的设备模式。

ID-CFI 地址空间：RDID 命令（9Fh）从单独的闪存地址空间中读取设备标识（ID）和

公共闪存接口（CFI）信息的信息。ID-CFI 地址空间由赛普拉斯编程，主机系统只读。

OTP 地址空间：One Time Program ，OTP 存储空间旨在提高系统安全性， OTP 值（例如赛普拉斯编程的随机数）可用于将闪存组件与系统 CPU / ASIC“配对”，以防止器件替代。

表 3-3 Flash 寄存器表

寄存器	缩写	类型	Bit 位
Status Register 1	SR1[7:0]	Volatile	7:0
Configuration Register 1	CR1[7:0]	Volatile	7:0
Status Register 2	SR2[7:0]	RFU	7:0
AutoBoot Register	ABRD[31:0]	Non-volatile	31:0
Bank Address Register	BRAC[7:0]	Volatile	7:0
ECC Status Register	ECCSR[7:0]	Volatile	7:0
ASP Register	ASPR[15:1]	OTP	15:1
ASP Register	ASPR[0]	RFU	0
Password Register	PASS[63:0]	Non-volatile OTP	63:0
PPB Lock Register	PPBL[7:1]	Volatile	7:1
PPB Lock Register	PPBL[0]	Volatile	0
PPB Access Register	PPBAR[7:0]	Read Only	7:0
DYB Access Register	DYBAR[7:0]	Non-volatile	7:0
SPI DDR Data Learning Registers	NVDLR[7:0]	Volatile	7:0
SPI DDR Data Learning Registers	VDLR[7:0]	Non-volatile	7:0

寄存器：寄存器是用于配置 S25FL-S 存储器设备工作方式或报告设备操作状态的一组存储单元，寄存器可以通过特定的命令访问。**错误!未找到引用源。**为寄存器的详细描述。这些寄存器在本设计中的主要作用是读取 Flash 设备的状态，或者改变其工作方式，具体对 Flash 设备中数据的访问，最直接的接口是 AXI Quad SPI IP 核。

要了解关于各地址空间的详细信息，可以参考芯片手册 S25FL128S, S25FL256S 128 Mbit and 256 Mbit 3.0V SPI Flash Memory Datasheet。大致了解即可，对于软件编程不是关键。

主机系统与 S25FL128S 存储设备之间的所有通信均采用命令的单元形式，命令以选择信息传输或设备操作类型的指令开始，命令还可能具有地址、指令修改部分、等待时间段、传输到存储器的数据或从存储器传输出的数据。所有指令、地址和数据信息在主机系统和存储器设备之间串行传输。

命令的结构大致如下：

- 每个命令以片选型号 CS＃变为低电平开始，以 CS＃变为高电平结束。存储器件由主机在整个命令中驱动芯片选择（CS＃）信号为低电平来选择。

- 串行时钟（SCK）标记主机和存储器之间每一位或一组位的传输。
- 每个命令以 8 位（字节）指令开始。该指令总是以串行输入（SI）信号上的单个位串行序列的形式呈现，并且在每个 SCK 上升沿将一位传输到存储器件，该指令选择要执行的信息传输或设备操作的类型。
- 指令可以是独立的，也可以后跟地址位，用于选择设备中几个地址空间中的一个位置。由指令确定使用的地址空间。地址可以是 24 位或 32 位字节边界地址。
- 指令之后数据传输的宽度由发送的指令决定，之后的传输可能继续为串行输出（SO）的单比特信号，也可能作为两个比特组在 IO0 和 IO1 的中传输，也可能以 4 个比特组在 IO0-IO3 中传输。
- 地址或模式位之后可以是将要存储在 Flash 设备中的写入数据，或读取的数据被返回到主机之前的延迟周期。

如图 3-15 所示为单独指令命令，命令只包含一个指令，不含地址、数据、延迟周期，指令传输完成后 CS 信号变为高电平。

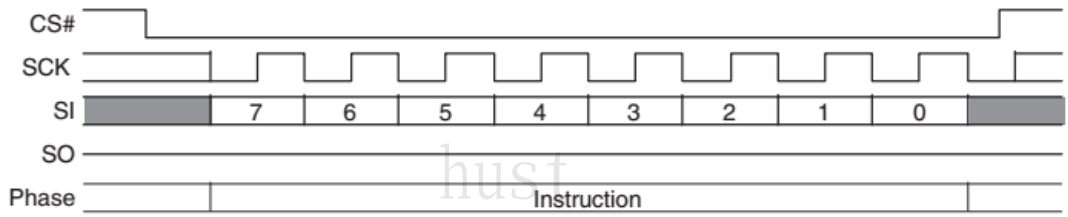


图 3-15 单独指令命令

如图 3-16 所示为单通道带延迟的 IO 命令，命令只包含一个指令，32 位地址数据，一个延迟周期，一个 8 位的数据。这些数据传输完成后 CS 信号变为高电平。

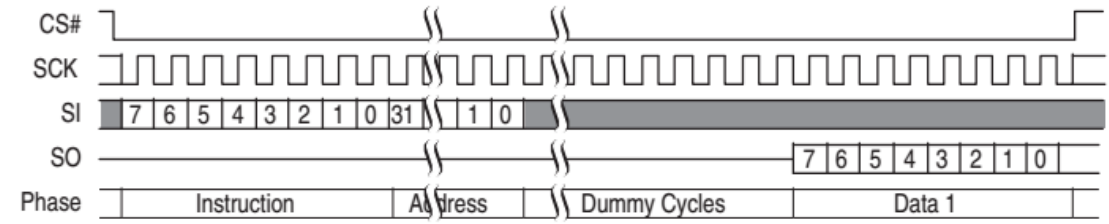


图 3-16 单通道带延迟的 IO 命令

关于 Flash 设备命令的详细信息，可以参考芯片手册 S25FL128S, S25FL256S 128 Mbit and 256 Mbit 3.0V SPI Flash Memory Datasheet。编程之前必须了解所使用的命令时序。

处理器对 Flash 设备的访问是通过 AXI Quad SPI IP 模块间接实现的，编写访问 Flash 设备的程序需要详细了解两个方面的知识：一是 AXI Quad SPI IP 模块的寄存器（发送的接口），二是相关的 Flash 命令（发送的内容）。

首先，向 Flash 发送一个字节的接口函数代码如下。

```
unsigned char SPI_Send_Byte(unsigned char data)
```

```
{
    while((*READ_IO(SPI_AXI_LITE + sr) & 0x4)==0);
    *WRITE_IO(SPI_AXI_LITE + dtr) = data;
    while((*READ_IO(SPI_AXI_LITE + sr) & 0x1)==1);
    return *READ_IO(SPI_AXI_LITE + drr);
}
```

该函数的参数是要向 Flash 中写入的字节数据 data，返回 Flash 中取出的字节数据。必须先写后读，因为需要先向从机发送数据，才能驱动 Flash 设备的 SCK 时钟，主机发送的这个数据可以是任意的，只用于触发 SCK。这个函数接口是与 Flash 设备通信的基本接口，写入和读取数据都要调用这个接口。

对 Flash 的具体访问遵从 Flash 的命令格式，需要按照命令的时序发送数据。从 Flash 主闪存阵列读取数据的接口函数示例代码如下：

```
#define FLASH_SPI_CS_LOW    *WRITE_IO(SPI_AXI_LITE + ssr) = 0x0
#define FLASH_SPI_CS_HIGH    *WRITE_IO(SPI_AXI_LITE + ssr) = 0x1
void SPI_FLASH_ReadMEM(unsigned int memAddr,unsigned int numRead)
{
   unsigned int j;
   FLASH_SPI_CS_LOW;
   SPI_Send_Byte(FAST_READ_CMD);
   SPI_Send_Byte((unsigned char) ((memAddr>>16) & 0xff));
   SPI_Send_Byte((unsigned char) ((memAddr>>8) & 0xff));
   SPI_Send_Byte((unsigned char) (memAddr & 0xff));
   SPI_Send_Byte(Dummy_Byte);
   for(j=0;j<numRead;j++)
   {
      buff[j] = SPI_Send_Byte(Dummy_Byte);
   }
   FLASH_SPI_CS_HIGH;
}
```


该函数的参数是读取 Flash 主闪存的起始地址 memAddr 和字节数 numRead ，将读取的数据复制到一个字节缓冲区 buff 中。

对于 Flash 设备的测试代码，具体还可以编写 RDID 9Fh 读识别命令，读取制造商标识、设备标识和通用闪存接口信息，等等。

12.4 Bootloader 参考代码

main.c

```
#include "flash.h"
#include "fpga.h"
#define debug
#ifdef debug
#include "debugFunctions/uart.h"
#endif
//extern struct flash flashInstance;
//struct flash *pFlashInstance = &flashInstance;
int main()
{
#ifdef debug
	uart_init();
	uart_print("serial port is working during the boot process\r\n");
#endif
	unsigned int entryAddress = 0;
	elfGetEntryAddress(&entryAddress);
#ifdef debug
	uart_print("entryAddress");
	uart_printHex(entryAddress);
	uart_print("\r\n");
#endif
	//now we analysis the elf header.
	int textVirtualAddress = 0;
	int textFlashAddress = 0;
	int textLength = 0;
	int dataVirtualAddress = 0;
	int dataFlashAddress = 0;
	int dataLength = 0;
	elfGetTextAndDataSectionInformation(&textVirtualAddress,&textFlashAddress,&textLength,
&dataVirtualAddress,&dataFlashAddress,&dataLength);
#ifdef debug
	uart_print("textVirtualAddress");
	uart_printHex(textVirtualAddress);
```

```
	uart_print("\r\n");
	uart_print("textFlashAddress");
	uart_printHex(textFlashAddress);
	uart_print("\r\n");
	uart_print("textLength");
	uart_printHex(textLength);
	uart_print("\r\n");
	uart_print("dataVirtualAddress");
	uart_printHex(dataVirtualAddress);
	uart_print("\r\n");
	uart_print("dataFlashAddress");
	uart_printHex(dataFlashAddress);
	uart_print("\r\n");
	uart_print("dataLength");
	uart_printHex(dataLength);
	uart_print("\r\n");
#endif
	textLength = textLength/4;//do the divide because we copy the section data 4 bytes by 4 bytes.
See below.
	dataLength = dataLength/4;//do the divide because we copy the section data 4 bytes by 4 bytes.
See below.
	int cnt = 0;
	unsigned int data = 0;
	//read the text section
	necessaryOperationBeforeReadBytes(textFlashAddress);
	for(cnt = 0 ; cnt < textLength ; cnt ++)
	{
		data = readFourBytes();
		*WRITE_IO(textVirtualAddress + cnt*4 ) = data;
	}
	operationsAfterReadBytes();
	//read the data section
	necessaryOperationBeforeReadBytes(dataFlashAddress);
	for(cnt = 0 ; cnt < dataLength ; cnt ++)
	{
		data = readFourBytes();
		*WRITE_IO(dataVirtualAddress + cnt*4 ) = data;
	}
```

```
    operationsAfterReadBytes();

    asm volatile("addu $t0, $0, %0;\n"
                 "jr $t0;\n"
                 "nop;\n"
                 "nop;\n"
                 :
                 :"r"(entryAddress)
                 :"t0"
                );
#ifdef debug
    uart_print("never here!\r\n");
#endif
    return 0;
}

    //jump to the kernel entry point.
    /*asm ("addu $t0 ,$0, $0\r\n"
        "lui $t0 ,0x8000\r\n"
        "addiu $t0, 0x1000\r\n"
        "jr $t0\r\n"
        "nop\r\n"
        "nop");"addu $t0, $t0, %1\n"*/
```


flash.c

```
#include "flash.h"
//this function is not included in header file.
static unsigned char SPI_Send_Byte(unsigned char data)
{
    // 检查并等待 TX 缓冲区为空
    while((*READ_IO(SPI_AXI_LITE + sr) & 0x4)==0);
    // 缓冲区为空后向缓冲区写入要发送的字节数据
    *WRITE_IO(SPI_AXI_LITE + dtr) = data;
    // 检查并等待 RX 缓冲区为非空
    while((*READ_IO(SPI_AXI_LITE + sr) & 0x1)==1);
    // 数据发送完毕，从 RX 缓冲区接收 flash 返回的数据
    return *READ_IO(SPI_AXI_LITE + drr);
```

```
}
void necessaryOperationBeforeReadBytes(unsigned int address)
{
	*WRITE_IO(SPI_AXI_LITE + srr) = 0x0000000a;		//software reset
	*WRITE_IO(SPI_AXI_LITE + cr) = 0x164;		//clear Tx and Rx
	*WRITE_IO(SPI_AXI_LITE + cr) = 0x6;		//enable spi and master mode
	FLASH_SPI_CS_LOW;		// 低电平片选有效，SPI 通讯开
始
	SPI_Send_Byte(FAST_READ_CMD);		// 发送读取 ID 指令
	SPI_Send_Byte((unsigned char) (address >>16));
	SPI_Send_Byte((unsigned char) (address >>8));
	SPI_Send_Byte((unsigned char) (address ));
	SPI_Send_Byte(Dummy_Byte);
	return;
}
unsigned int readFourBytes()
{
	unsigned int data4Bytes = 0x0;
	unsigned int data = 0x0;
	int i = 0;
	for( ; i < 4 ; i++)
	{
		data = (unsigned int) SPI_Send_Byte(Dummy_Byte);
		data4Bytes = data4Bytes | data << (i * 8);//carefull here. Do not modify the endianess.
		data = 0x0;
	}
	return data4Bytes;
}
unsigned short readTwoBytes (void)
{
	unsigned char data1 = 0x0;
	unsigned char data2 = 0x0;
	data1 =	SPI_Send_Byte(Dummy_Byte);
	data2 =	SPI_Send_Byte(Dummy_Byte);
	unsigned short data2Bytes = data1 | data2 << 8;//carefull here. Do not modify the endianess.
	return data2Bytes;
}
unsigned char readOneByte (void)
```

```
{
	return SPI_Send_Byte(Dummy_Byte);
}
void operationsAfterReadBytes()
{
	FLASH_SPI_CS_HIGH;                                  // 停止 SPI 通讯
	return;
}
void skipNBytes(int n)
{
	int cnt = 0;
	for(cnt = 0; cnt < n ; cnt ++)
	{
		readOneByte();
	}
	return;
}
```

analyseELF.c


```
#include "flash.h"
#include "analyseELF.h"
#define convert2FlashAddress(x) (READ_ADDR + x)
void elfGetEntryAddress(unsigned int *entryAddress)
{
	necessaryOperationBeforeReadBytes(convert2FlashAddress(0x18));
	(*entryAddress) = readFourBytes();
	operationsAfterReadBytes();
	return;
}
void elfGetTextAndDataSectionInformation(unsigned int *textVirtualAddress, unsigned int *textFlashAddress, unsigned int *textLength, unsigned int *dataVirtualAddress, unsigned int *dataFlashAddress, unsigned int *dataLength)
{	//We assume the ELF is 32 bits address mode and little endian.
	//There are NO error handling when dealing with 64 bits address mode or big endian.
	int SectionHeaderAmount = -1;
	necessaryOperationBeforeReadBytes(convert2FlashAddress(0x30));
	SectionHeaderAmount = readTwoBytes();
```

```
    operationsAfterReadBytes();
    unsigned int sectionHeaderFlashAddress = 0;
    necessaryOperationBeforeReadBytes(convert2FlashAddress(0x20));
    unsigned int tempOffset = readFourBytes();
    operationsAfterReadBytes();
    necessaryOperationBeforeReadBytes(convert2FlashAddress(tempOffset));
    int cnt = 0;
    int dataOrTextSection = 0;
    for(cnt = 0 ; cnt < SectionHeaderAmount ; cnt ++)
    {
        if(dataOrTextSection == 2)
        {
            break;
        }
        skipNBytes(8);
        unsigned int sh_flags = readFourBytes();//sh_flags : this terminology comes from ELF format specification.
        if(sh_flags == 0x1 + 0x2)
        {//this is data section because it is writable and allocatable
            dataOrTextSection++;
            *dataVirtualAddress = readFourBytes();
            *dataFlashAddress = convert2FlashAddress( readFourBytes() );
            *dataLength = readFourBytes();
            skipNBytes(16);
        }
        else if(sh_flags == 0x2 + 0x4)
        {//this is text section because it is allocatable and executable
            dataOrTextSection++;
            *textVirtualAddress = readFourBytes();
            *textFlashAddress = convert2FlashAddress( readFourBytes() );
            *textLength = readFourBytes();
            skipNBytes(16);
        }
        else
        {//this is neither text section nor data section
            skipNBytes(28);//note : this is decimal , not hex.
        }
    }
```

```
    operationsAfterReadBytes();// do not forget to "close" the "file".
    return;
}
```

第13章　实验 11：挑战：蓝牙小车应用扩展——小车操作轨迹记录

11.1 实验目的

利用 FAT32 文件系统在 SD 卡中记录蓝牙小车操作轨迹记录，并能于

11.2 实验内容

11.2.1 SD mirco 的 ucore 操作系统驱动设计

11.2.2 蓝牙小车应用改进及 FAT32 文件系统的读写

11.3 实验背景及原理

11.3.1FAT32 文件系统

11.4 实验报告