第三部分 蓝牙小车应用的设计与实现

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。

第九章 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。

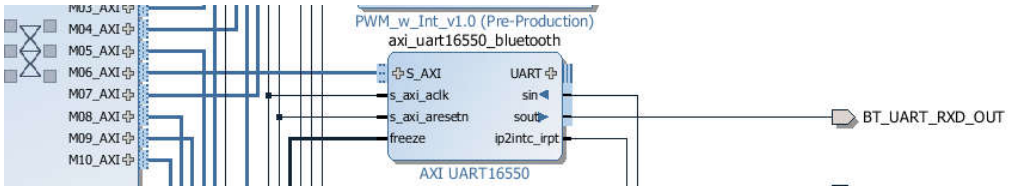
本实验希望读者能通过对这些外设模块的设计实现加深对 Pmod 接口协议的理解和自定制模块设计方法，同时提高自己的 Verilog 编程能力。

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。



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。

表 3-1 L293D 驱动板 PCB 封装引脚功能

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 2- 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。

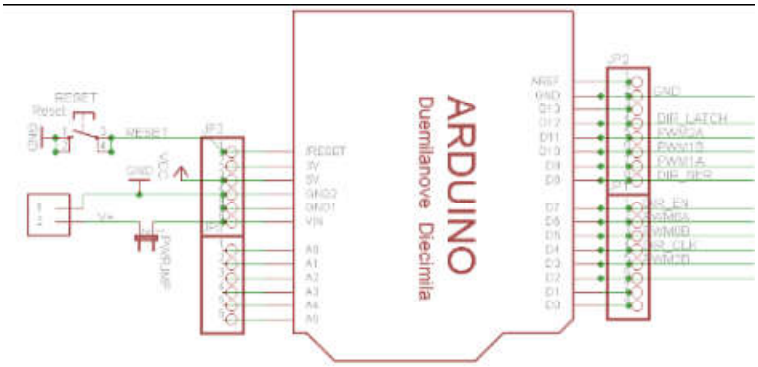


图 2- 3 L293D 驱动板 PCB 封装引脚图

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。

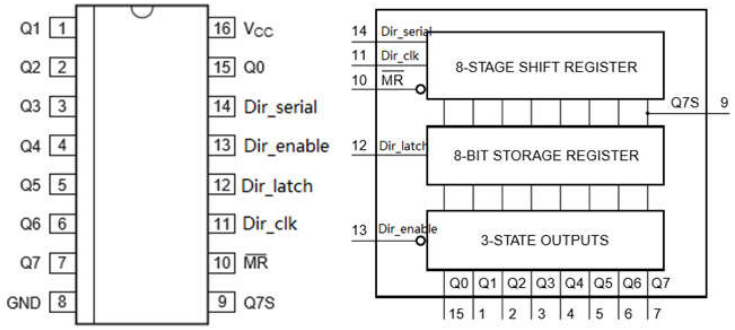


图 3-4 74HCT595N 芯片引脚图

图 2-4 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。

表 3-3 功能表（H 为高电平，L 为低电平，X 为不相关，
Z 为高阻，NC 为不变，↑为上升沿）

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_{nS}
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 2-5 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。

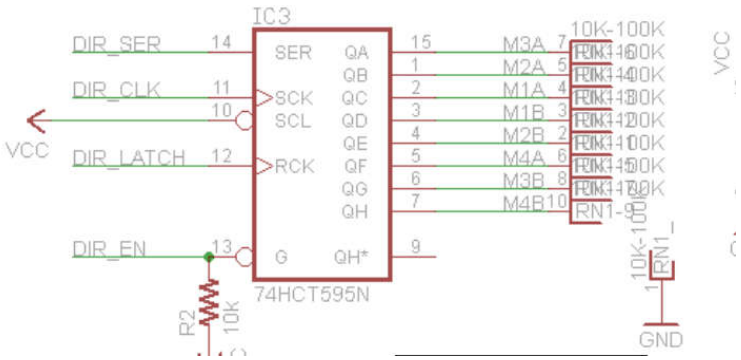


图 2-6 74HCT595N 芯片和 4 个直流电机关系示意图

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 PMOD 原理简介

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）。

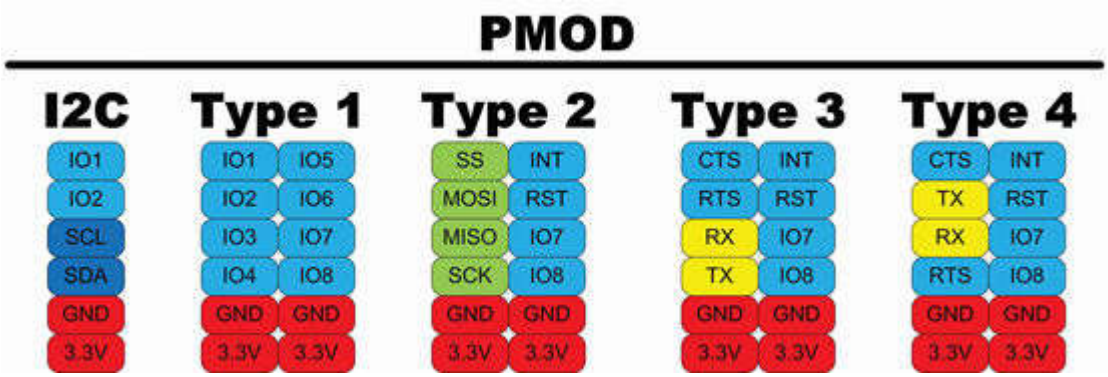


图 1. Pmod 引脚排列类型将不同的功能分配给特定引脚。

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

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);
}
```

第十章 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.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

第十一章 实验 11：挑战：蓝牙小车运行轨迹记录及还原

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，并能根据记录的轨迹自动控制小车按照该轨迹重新运行。

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