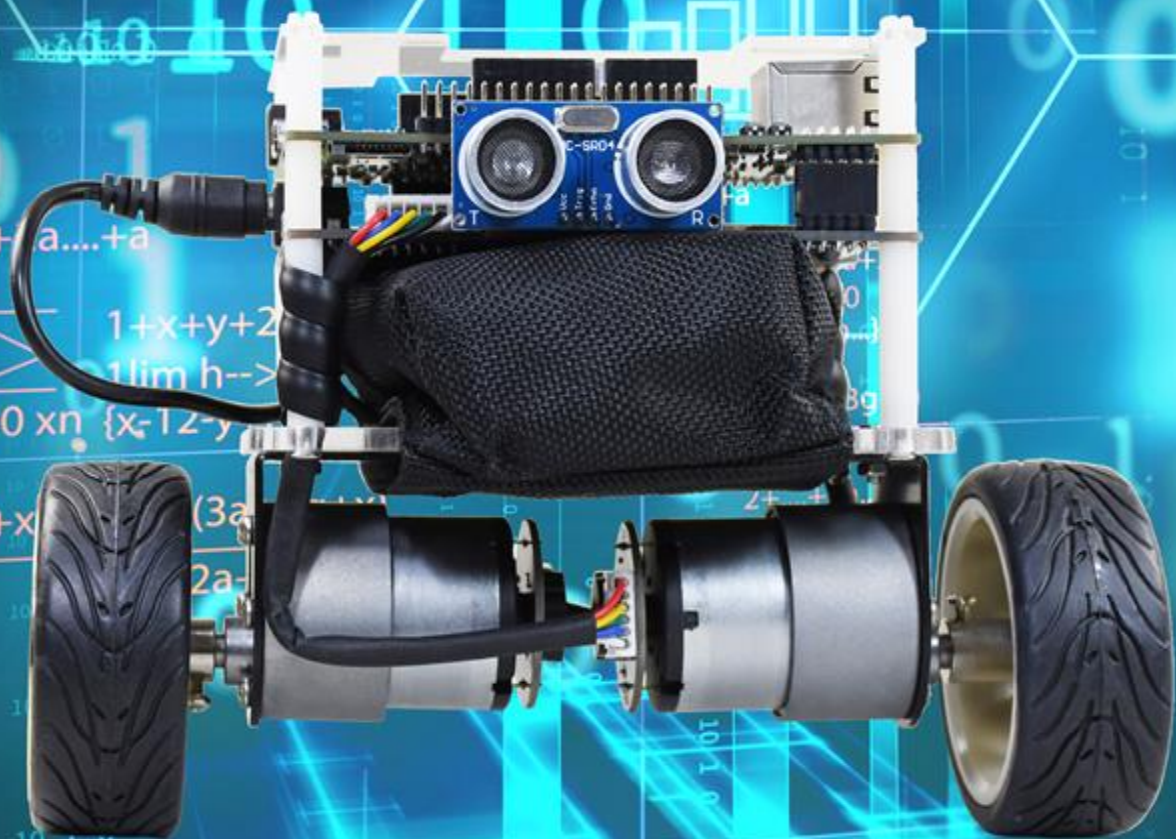


Self-Balancing Robot

User Guide



目录

第 1 章 使用平衡车	2
1.1. 控制电机	2
1.1.1. 转向控制.....	3
1.1.2. 转速控制.....	4
1.1.3. 范例描述.....	4
1.2. 监测电机转速与转向	9
1.2.1. 侦测原理.....	9
1.2.2. 范例描述.....	11
1.3. 获取车身倾斜角	14
1.3.1. 计算倾斜角.....	14
1.3.2. MPU-6500 操作.....	18
1.3.3. 范例描述.....	18
1.4. 获取障碍物距离	20
1.4.1. 原理.....	20
1.4.2. 范例描述.....	22
1.5. 平衡车系统	26
1.6. 使用蓝牙	28
1.7. 使用遥控器	35
1.7.1. 红外遥控器协议.....	35
1.7.2. 范例描述.....	38

第1章

使用平衡车

这份文档主要内容为讲解平衡车的每个接口(如马达转动控制与车身倾斜角度)的工作原理以及对应的控制 hardware/software 代码说明。

用户可以通过这份文件学习整个平衡车的工作原理, 并且依照这些范例代码来修改并组合应用于自己设计。

1.1. 控制电机

要维持平衡车垂直平衡状态,必须要能控制车身上的电机来配合车身倾斜的方向逆向加速转动, 所以需要了解如何控制电机的转动方向以及速度。本节将介绍如何使用电机驱动芯片来驱动电机正转或反转。并同时介绍如何控制电机的转速。因为一般的 FPGA I/O 无法驱动电机, 所以需要额外的电机驱动芯片或电路来驱动电机, 平衡车上使用的电机驱动芯片型号是 Toshiba 的 TB6612FNG。此芯片可以同时控制两个 DC 电机, 如图 1-1 所示。与 FPGA 连接的控制信号为 IN1/IN2/PWM (有 A、B 两组电机控制信号)以及 STBY, 输出到电机的控制信号为 O1/O2。以下将介绍如何控制电机的转向与转速。

注意: 在电机驱动板上, TB6612FNG 通过一个 Photo Coupler 与 FPGA 连接, 所以 FPGA 输出的控制信号需要与 datasheet 内描述的控制逻辑反向。

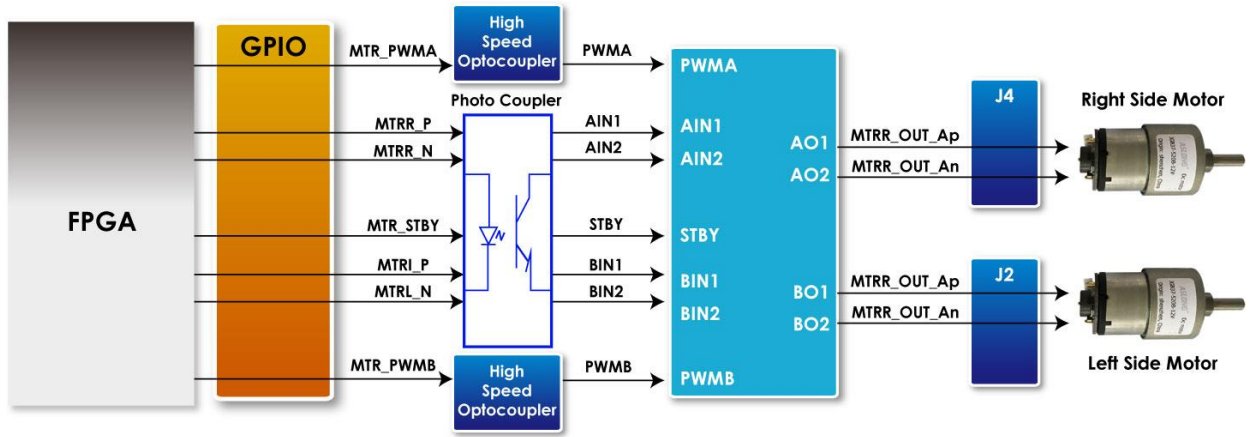


图 1-1 电机与 FPGA 连接图

1.1.1. 转向控制

表 1-1 列出了电机驱动芯片 TB6612FNG 提供的控制功能。

- 控制 IN1 与 IN2 的逻辑值。控制电机正转(IN1=L ; IN2=H)或反转(IN1=H; IN2=L)。
- 若两个控制信号同时为逻辑 0，电机停止转动。
- STBY 相当于 Chip Enable 功能,当它为逻辑 0 时,电机将会停止，待命不动。

用户只需要控制 IN1 与 IN2 的逻辑值，就能简单的改变电机的转动方向。

表 1-1 TB6612FNG 的电机控制菜单

FPGA Control Output				Driver Input				Driver Output		Modes description
MTRX_P	MTRX_N	MTR_PWMX	MTRX_STBY	IN1	IN2	PWM	STBY	O1	O2	--
0	0	1/0	0	1	1	1/0	1	0	0	Short brake
1	0	1	0	0	1	1	1	0	1	CCW
		0	0			0	1	0	0	Short brake
0	1	1	0	1	0	1	1	1	0	CW
		0	0			0	1	0	0	Short brake
1	1	1	0	0	0	1	1	OFF (High Impedance)		Stop
0/1	0/1	1/0	1	1/0	1/0	1/0	0	OFF (High Impedance)		Standby

注意:此表描述的控制逻辑与 TB6612FNG datasheet 中描述的反相, 因为 FPGA 与 TB6612FNG 之间有一个 Photo Coupler。

1. 1. 2. 转速控制

要控制电机的转速,需要搭配控制信号 PWM 的占空比。如图 1-2 所示,PWM 信号的占空比越高(代表逻辑高的正脉冲的持续时间与脉冲总周期的比值越高), 电机转速会越快。用户只需要控制 PWM 讯号的脉冲宽度就可以控制转速。

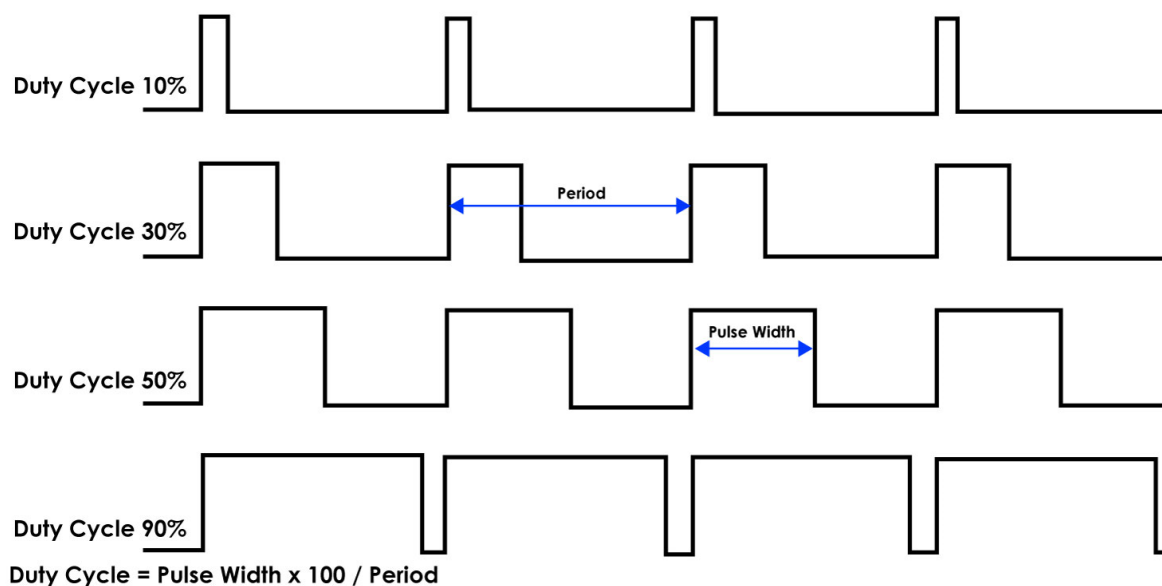


图 1-2 不同占空比示意图

另外, TB6612FNG 提供的最大 PWM 频率为 100KHZ, 平衡车中的 PWM 设置为 7.14KHz.

1. 1. 3. 范例描述

平衡车的范例代码提供了电机控制的 IP: TERASIC_DC_MOTOR_PWM.v, 封装在 Qsys 组件中。平衡车的 demo 使用这个 IP 分别控制平衡车的左右电机。用户可以在 CD 内的 Demonstrations\BAL_CAR_Nios_Code\IP\TERASIC_DC_MOTOR_PWM 找到这个 IP。

■ IP Symbol

图 1-3 为 TERASIC_DC_MOTOR_PWM.v 的 Symbol 以及在系统内的方框图。主要的输出为 DC_MOTOR_IN1, DC_MOTOR_IN2 与 PWM, 其他为 Avalon 接口。DC_MOTOR_IN1/ DC

_MOTOR_IN2 是 1.1 节描述的控制电机转向与停止的信号。PWM 信号控制转速。这些端口直接从 FPGA 输出，与电机驱动 IC 连接。

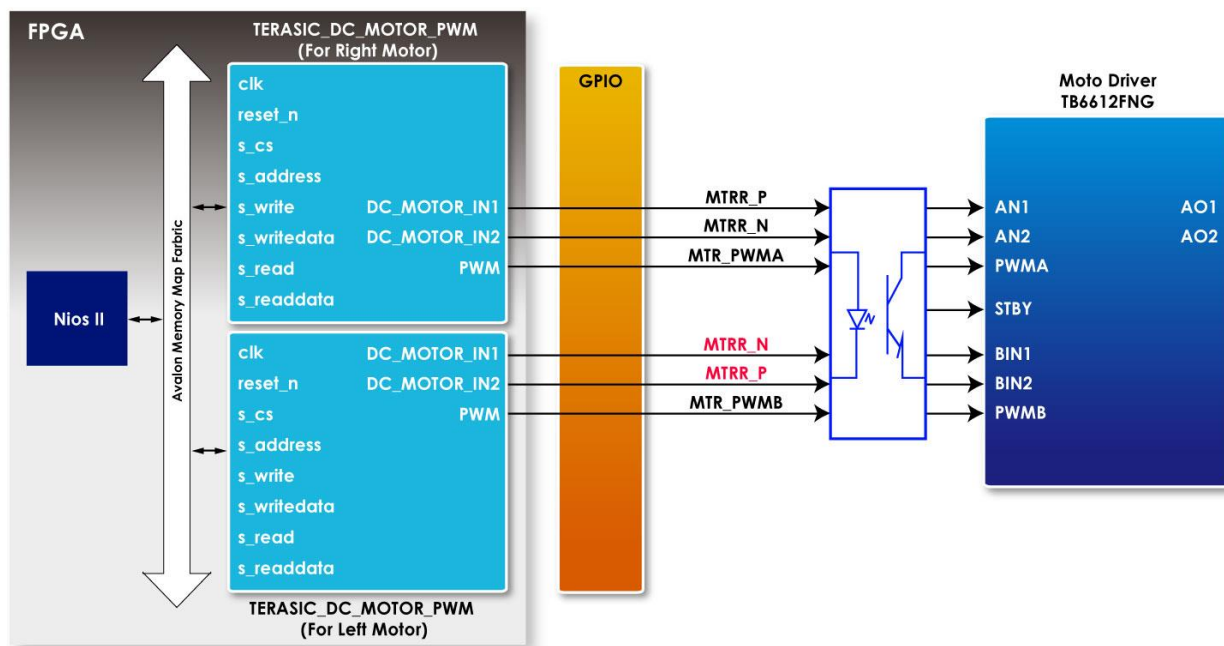


图 1-3 Terasic_DC_MOTOR_PWM.v 的 symbol

■ Register Table

表 1-2 是 IP 的 Register Table。Base Address 1~0 为 PWM 的控制 Register, Base Address 2 为电机运动、转向、减速的控制寄存器。用户可以通过 Nios 或 HPS 读取这些寄存器的值。

表 1-2 Terasic_DC_MOTOR_PWM.v IP 的 Register Table

Reg Address	Bit Field	Type	Name	Description
Base Addr + 0	31:0	R/W	total_dur	PWM total duration value
Base Addr + 1	31:0	R/W	high_dur	PWM high duration value
Base Addr + 2	31:3	-	Unused	Unused bit
	2	R/W	motor_fast_decay	Motor brake control 1 for fast brake 0 for short brake
	1	R/W	motor_forward	Motor direction control : 1 for forward

				0 for backward
	0	R/W	motor_go	Motor enable: 1 for start 0 for stop

■ IP 代码描述

- 转向控制代码

下面一段为转向控制的代码。

```

always @(*)
begin
  if (motor_fast_decay)
  begin
    // fast decay
    if (motor_go)
    begin
      if (motor_forward)
        {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b1, 1'b0, PWM_OUT}; // forward
      else
        {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b0, 1'b1, PWM_OUT}; // reverse
    end
  else
    {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b1, 1'b1, 1'b0};
  end
else
begin
  // slow decay
  if (motor_go)
  begin
    if (motor_forward)
      {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b1, 1'b0, PWM_OUT}; // forward
    else
      {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b0, 1'b1, PWM_OUT}; // reverse
  end
  else
    {DC_MOTOR_IN2, DC_MOTOR_IN1, PWM} <= {1'b0, 1'b0, 1'b0};
  end

```

```
end
```

```
end
```

这段代码将开发者设置的电机控制寄存器值转换为 DC_MOTOR_IN1 和 DC_MOTOR_IN2 实际控制信号，进而控制电机的转向。

用户如果要控制电机前行，首先要设定 motor_go 和 motor_forward 为 "1"。这样，代码"DC_MOTOR_IN2, DC_MOTOR_IN1,PWM}<= {1'b1, 1'b0,PWM_OUT};"//forward"将会被执行。

DC_MOTOR_IN1 和 DC_MOTOR_IN2 会输出逻辑 0 与 1，从 FPGA 送往电机驱动 IC，经过 Photo Couple 反相。TB6612FNG 的 IN1 与 IN2 引脚收到的逻辑为 1 与 0。对照表 1-1，电机将会反转，即前行。

如果用户需要急速刹车，可以设定 motor_fast_decay 为 1，并设定 motor_go 为 0，这样，以下代码将会被执行：DC_MOTOR_IN2, DC_MOTOR_IN1,PWM}<= {1'b1, 1'b1,1'b0};

最终 TB6612FNG 的 IN1 与 IN2 逻辑是 0 与 0，对照表 1-1，电机的状态是 Stop。

平衡车的左右两个电机装配方向相反，所以两个电机的转向也是相反的。因为使用的同一个 IP，所以在工程顶层文件 (DE10_Nano_Bal.v) 内，控制信号特意反相设置，如以下代码所示：

```
Qsys u0 (  
    //clock && reset  
    .clk_clk          (FPGA_CLK2_50),          // clk.clk )  
    .reset_reset_n   (1'b1),                  // reset.reset_n  
  
    //right motor control  
    .dc_motor_right_conduit_end_1_pwm        (MTR_PWMA),          //  
dc_motor_right_conduit_end_1_pwm  
    .dc_motor_right_conduit_end_1_motor_in1  (MTRR_P),              //      .motor_in1  
    .dc_motor_right_conduit_end_1_motor_in2  (MTRR_N),              //      .motor_in2  
    //left motor control  
    .dc_motor_left_conduit_end_1_pwm         (MTR_PWMB),          //  
dc_motor_left_conduit_end_1_pwm  
    .dc_motor_left_conduit_end_1_motor_in1   (MTRL_N),              //      .motor_in  
1
```


.dc_motor_left_conduit_end_1_motor_in2 (MTRL_P),

- 转速控制代码

转速控制部分的代码如下:

```
////////////////////////////////////  
// PWM  
reg      PWM_OUT;  
reg [31:0] total_dur;  
reg [31:0] high_dur;  
reg [31:0] tick;  
always @ (posedge clk or negedge reset_n)  
begin  
    if (~reset_n)  
        begin  
            tick <= 1;  
        end  
    else if (tick >= total_dur)  
        begin  
            tick <= 1;  
        end  
    else  
        tick <= tick + 1;  
    end  
always @ (posedge clk)  
begin  
    PWM_OUT <= (tick <= high_dur)?1'b1:1'b0;  
end
```

Tick 是主要的计数器，total_dur 是表 1-2 中提到的 total_dur register。

当 tick 值达到 total_dur 的设定值，整个计数器会重置并重新计数。PWM_OUT 输出代表一个 PWM 周期结束，所以 total_dur 数值越大，代表 PWM 的周期越长。范例中将 total_dur register 设置为默认值 7000，即计数器计数到 7000 时为一个 PWM 周期。输出的 PWM 频率为 $50\text{Mhz} / 7000 = 7.14\text{KHz}$ 。

如 图 1-4 所示, high_dur register 决定电机的转速。在一个 PWM 周期内, 当 tick 值小于 high_dur 时, PWM 输出为 1, 否则为 0, 由此可以看出 high_dur 控制 PWM 的占空比, high_dur 值越高, 占空比越大, 转速会越快。

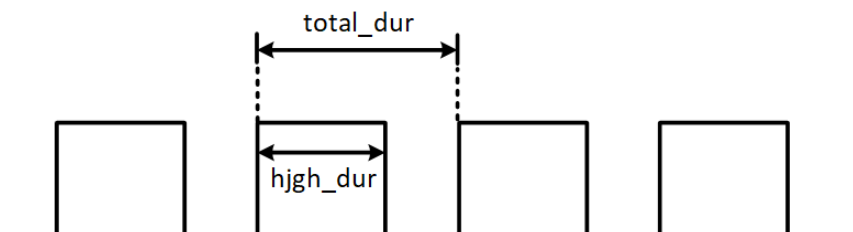


图 1-4 total_dur 与 high_dur 在 PWM 内的关系图

1.2. 监测电机转速与转向

1.1 节介绍了如何控制电机的转速与转向, 本节将介绍如何使用电机上的霍尔效应感应器与译码器, 实时监测电机的转向与转速。获取转速能让用户有效控制平衡车, 另外在相同的转速控制命令下, 车身上的两个电机在实际中不一定有同样的转速。通过读回的转速, 用户可以调整两个电机的误差, 使平衡车在行驶时尽量保持直线。

1.2.1. 侦测原理

如图 1-5 所示是平衡车上的电机外观, 上面有两个霍尔效应传感器以及磁盘。电机转动带动磁盘经过霍尔传感器, 磁力的变化让霍尔效应传感器产生霍尔效应电压, 经过数字电路处理产生方波, 如图 1-6 所示。

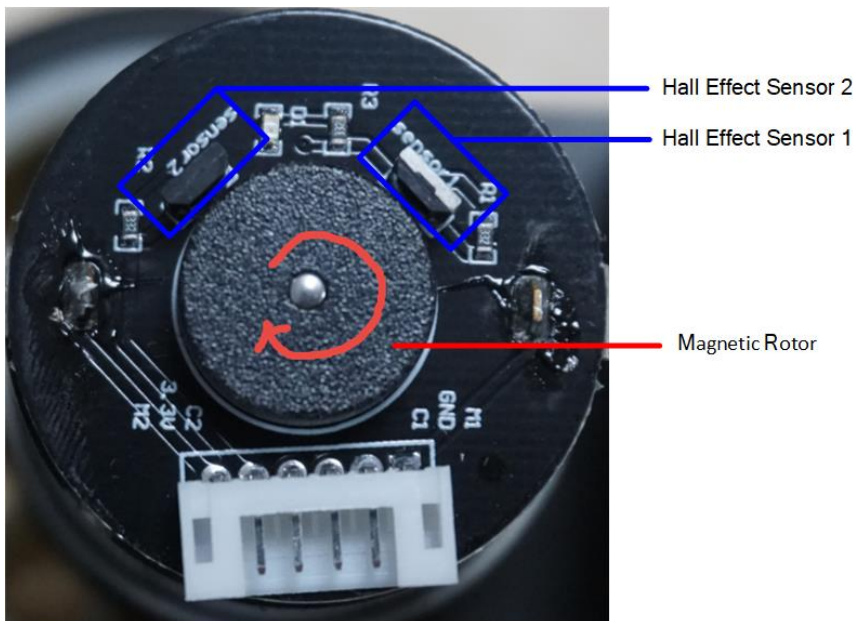


图 1-5 电机上的霍尔效应传感器以及磁盘

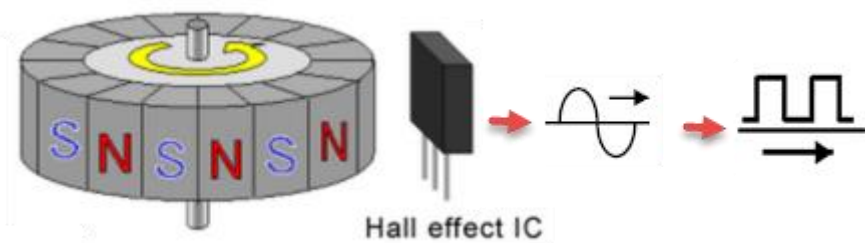


图 1-6 霍尔效应传感器与输出的方波

如图 1-7 所示，因为有两个位置不同的霍尔效应传感器，所以输出两个相位不同的方波(Phase A 和 Phase B)。磁盘在转动时，先被感应的传感器会先输出方波，另一个传感器输出会有延迟，所以两个方波的相位有所不同。由此，用户可以通过方波相位领先计算电机的转向。根据输出的脉冲数，用户也可以计算电机转速。电机转动越快，在固定时间内，脉冲数就越多。

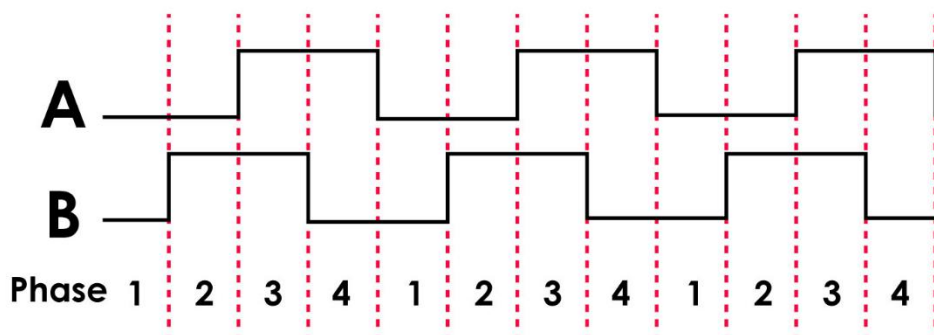


图 1-7 霍尔效应编码器输出的 AB 相波型

图 1-8 是电机输出的相位引脚与 DE10-Nano FPGA 连接的示意图。用户只需要编写代码，去监测这两个脉冲信号的相位与脉冲数，就能获取电机的实时转速与转向。

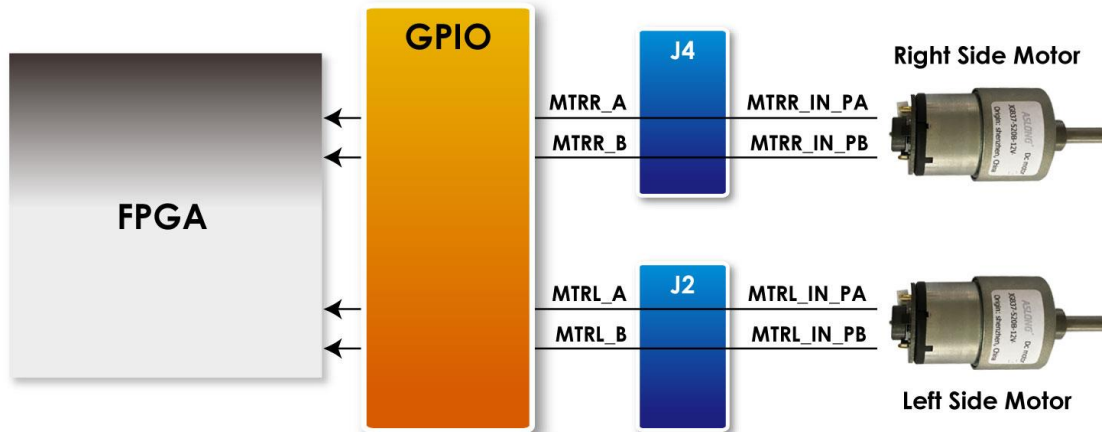


图 1-8 电机输出的相位引脚与平衡车 FPGA 连接图

1.2.2. 范例描述

平衡车的范例提供了一个可以读取电机转向与转速的 Qsys IP，位于 `\Demonstrations\BAL_CAR_Nios_Code\IP\motor_measure\motor_measure.v`

■ IP Symbol

图 1-9 为 `motor_measure.v` 的 Symbol 以及在系统内的方框图，这里只显示了测试右边电机的模块，左边的电机有一个相同的模块来监测转速。这个模块对外接口为 `phase AB[1:0]`，与电机连接，接收霍尔效应感应器输出的波形，监测并判断电机转向与转速，并存入寄存器，使 CPU 通过 Avalon 总线读取这些数据。

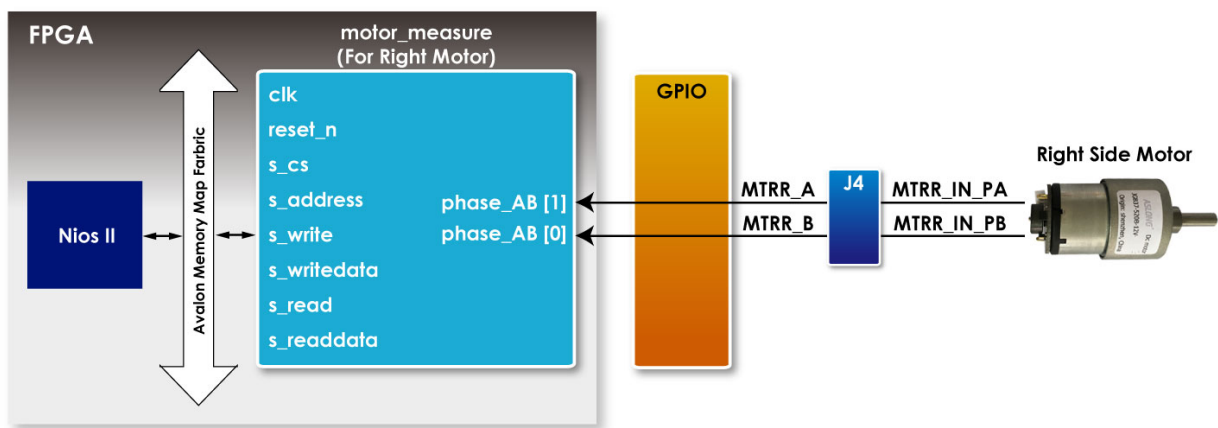


图 1-9 motor_measure.v 的 Symbol 以及在系统内的方框图(右电机)

■ Register Table

表 1-3 列出了 IP 的寄存器设置。主要的寄存器为 Counter，用于计算电机传回的脉冲数量，系统计算单位时间内电机转数，如果 Counter 值是正数，代表电机正转，负数则代表电机反转。Base Addr+0 的 Counter 值仅作读取使用，使 CPU 能读取当前 Counter 数值。Base Addr+2 的 Counter 值为写入设定使用。Base Addr+1 的 count_en 为 Counter 控制用，当设为 1 时，Counter 才开始计数。

表 1-3 motor_measure.v IP 的 Register Table

Reg Address	Bit Filed	Type	Name	Description
Base Addr + 0	31:16	RO	Unuse	-
	15:0	RO	Counter (For Read)	Read Counter value for Motor output pules
Base Addr + 1	31:30	RW	Unuse	-
	1	RW	count_en	Enable Motor pulse counter

Base Addr + 2	31:16	WO	Unused	-
	15:0	WO	Counter (For Write)	Set Counter value

■ 侦测转向与转数

在 motor_measure.v IP 内还有一个子模块代码: TERAISC_AB_DECODER.v, 这个子模块监测电机传来的 Phase A 及 Phase B 信号, 根据相位差异, 判断电机是正转或者反转。并通过 DO_DIRECT 输出到 motor_measure.v IP, 同时输出的还有电机转动的脉冲(DO_PULSE)。

```
TERAISC_AB_DECODER u_decoder
```

```
(
```

```
  .DI_SYSCLK(clk),
  .DI_PHASE_A(phase_AB[0]),
  .DI_PHASE_B(phase_AB[1]),
  .DO_PULSE(conter_pulse),
  .DO_DIRECT(direction)
```

```
);
```

motor_measure IP 内有一个 16bit Counter (初始值为 16'h8000), 只有当 "count_en" register 被设为 1 才会激活。比如以下的代码, 当电机正转(direction=1), Counter 随着电机传回的脉冲数而累加。如果电机反转, Counter 随着电机传回的脉冲数递减, 系统会定时读取 Counter 的值获取当前电机转数。

```
always @( posedge clk)
```

```
begin
```

```
  if(s_cs && s_write && s_address==`CNT_WRITE)
```

```
    counter<=s_writedata[15:0];
```

```
  else if(count_en && conter_pulse )
```

```
  begin
```

```
    if(direction)
```

```
    begin
```

```
      if(counter<16'hffff)
```

```
        counter<=counter+1;
```

```
    end
```

```
  else if(!direction)
```

```
begin
    if(counter>0)
        counter<=counter-1;
    end
    else
        counter<=0;
    end
end
end
```

系统读取 Counter 步骤可以参考 Nios 版本平衡车 demo 内的 Motor.cpp, 路径为:

`\Demonstrations\BAL_CAR_Nios_Code\software\DE10_Nano_bal`

在平衡车系统内, 系统初始化时, 会先设定"count_en" 为 1, 然后每隔 10ms 读取一次 counter 值, 读完后把 IP 内的 counter register 设为初始值 16'h8000, 等待下一个 10ms 去读取 register, 读取的 counter 值减去初始值 16'h8000, 这样如果电机是正转, 计算出来的 Counter 值将会是正数, 如果反转, 计算出的 Counter 值将会是负数。

最终读取左右电机的 Counter 值被传输平衡车所采用的平衡 PID 算法。

1.3. 获取车身倾斜角

本节将介绍如何获取平衡车身的倾斜角度, 提供给系统进行校正, 保持车身平衡。

1.3.1. 计算倾斜角

平衡车的理想状态是与地面保持垂直 90 度, 但实际上因为只有两个车轮支撑, 所以车身会随时的往前或者往后倾斜, 此时车身与垂直面有一个倾斜角度 θ , 如图 1-10 所示。我们的目的是获取这个角度, 然后反馈入平衡系统内, 驱动电机往反方向移动, 使倾斜角度保持理想的 0 度来修正。



图 1-10 平衡车身倾斜角度

要获取车身倾斜角度，需使用车身上的运动跟踪器件 MPU-6500 来实现，MPU-6500 带有三轴加速计(accelerometer)与陀螺仪(gyroscope)。从加速计可以读取三轴的加速度(单位：g)，从陀螺仪可以获取三轴的角速度 (单位：角度/Sec)。平衡车系统采用这两种感应器获取角度。首先需要了解 MPU-6500 在平衡车身上的 XYZ 坐标轴状况，如图 1-11 所示，平衡车身后后倾斜，会造成 X 轴与 Y 轴的加速度会有所变，同时 Y 轴的角速度也会有所变化。

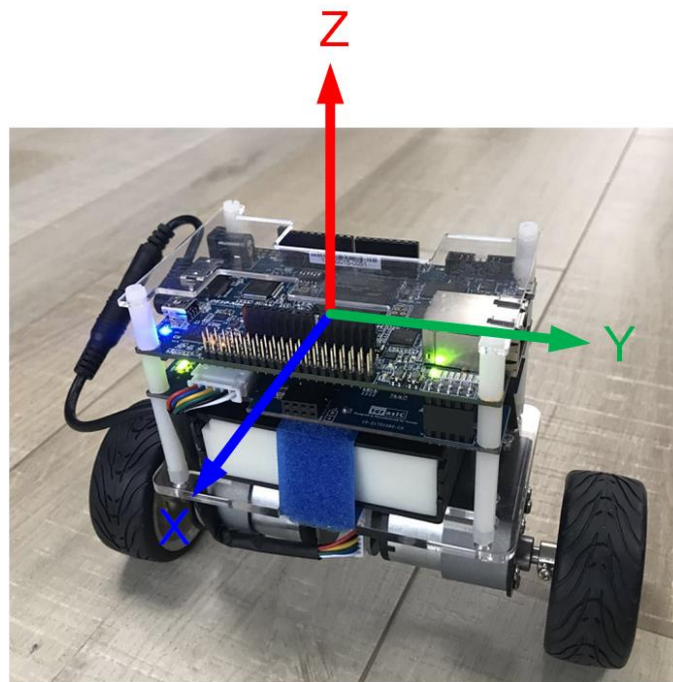


图 1-11 MPU-6500 在平衡车上的 XYZ 轴状态

先介绍如何用加速计来计算平衡车的倾斜角度，如图 1-12，如果不考虑车身运动加速度的状态，仅车身往前或往后倾斜时，平衡车与垂直状态的倾斜角 θ ， g 为重力加速度，将 g 分解为 X Z 两个方向， g_x 和 g_z 分别为 X 轴与 Z 轴分量，倾斜角 θ 为 g_x 、 g_z 的正切角， g_x 与 g_z 可通过 MPU-6500 内的加速计读取，通过公式 $\theta = \arctan(g_x/g_z)$ 可以计算 θ 的度数。

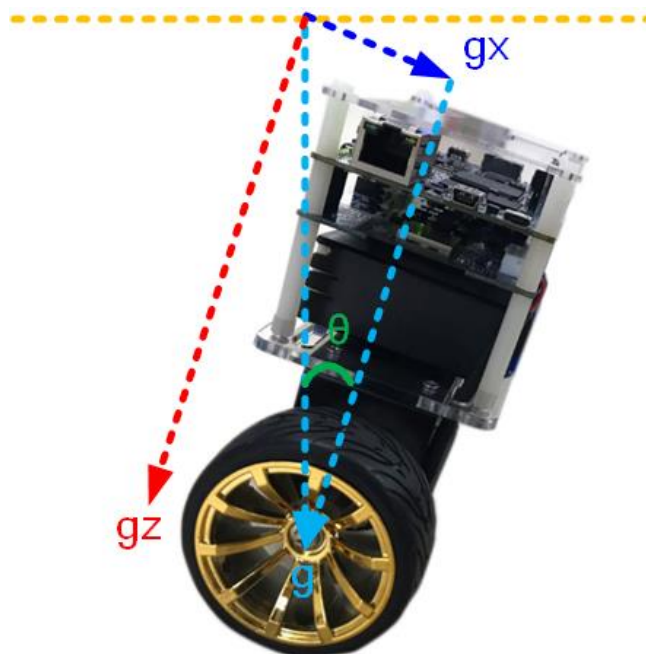


图 1-12 平衡车身倾斜角度

另外还可以通过 MPU-6500 内的陀螺仪所测得的 Y 轴角速度来计算车身倾斜角，如图 1-13，当车身倾斜时，Y 轴的角速度将会变化，倾斜角度可以通过对陀螺仪的角速度进行积分计算获取。

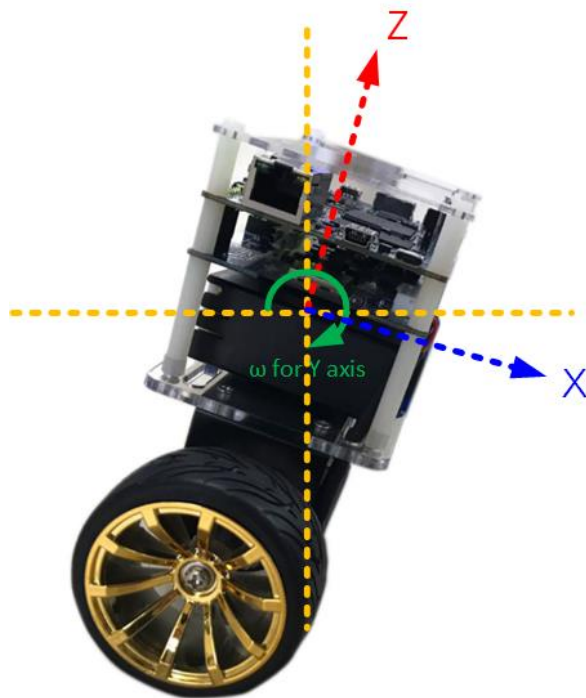


图 1-13 平衡车身倾斜角度

上述两种方法计算出的角度，都会出现误差。加速度计读取到的角度，在受外界干扰时，获取的值误差会变大。使用陀螺仪的角速度积分计算获取的角度，由于积分运算会将误差累积，随时间增大，误差也越来越大。如果平衡系统使用误差大的角度，要使车身稳定将会很困难，所以需要对倾斜角度进行误差校正。方法为一般常见的卡尔曼滤波，将两个感应器(陀螺仪与加速计)的数据融合(data fusion)输入，得到更精确的一个角度。

图 1-14 显示了原始倾斜角度与经过卡尔曼滤波的角度数值对比，可以看到蓝色的未经滤波的角度上下变动幅度相当大，以这样的数据控制车身平衡会相当不稳定，而经过了滤波后的角度值明显的变动幅度小了很多。

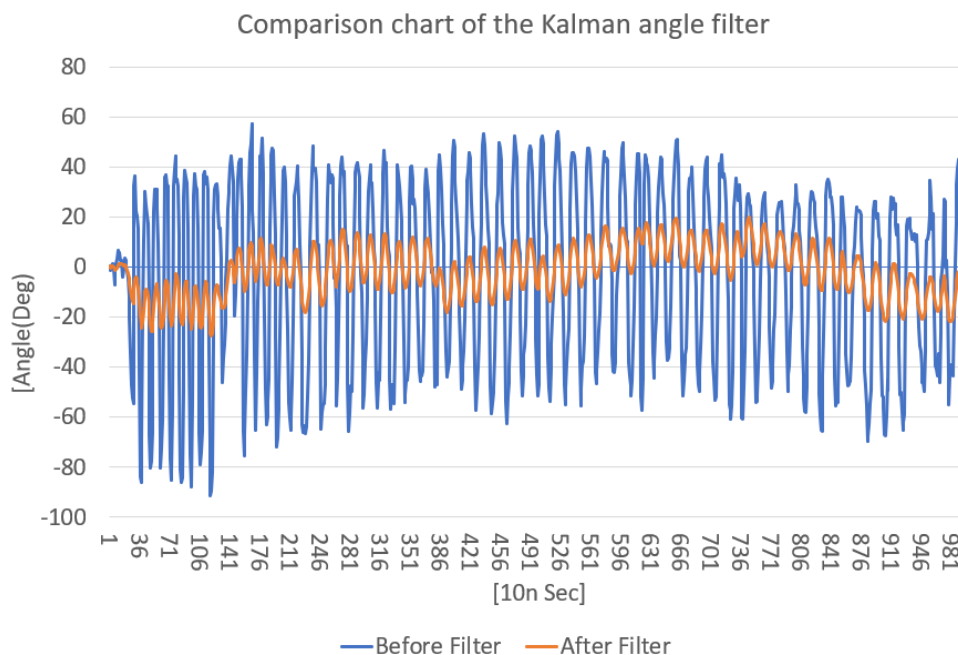


图 1-14 倾斜角度使用卡尔曼滤波前后比较

1. 3. 2. MPU-6500 操作

可以通过 I2C 或者 SPI 接口来完成 FPGA 对 MPU-6500 的控制，我们的范例使用 I2C 接口来读取 MPU-6500 的寄存器值。其 Salve Address 为 7'b1011001，XYZ 轴的加速计与陀螺仪数值的寄存器在地址位置 3B(Hex)~48 间。关于详细的 MPU-6500 数据与 Register map 文件可以在 CD 内的\Datasheet\Sensor\内获取。控制代码可以参考...CD\Demonstrations\BAL_CAR_Nios_Code\software\DE10_Nano_bal\路径下的 MPU.cpp 与 MPU.h。

1. 3. 3. 范例描述

我们提供的 Nios II 平衡车范例,在 Qsys 内使用 Open core I2C module，Nios II 通过这个模块使用 I2C 接口读取 MPU-6500。主要的获取倾斜角度的运算函数可参考路径：
 \BAL_CAR_Nios_Code\software\DE10_Nano_bal\ 的 main.cpp，以下为主要代码。

```

/*****
Function   : Get Angle value (Kalman filter)
parameter  :
return value :
*****/

```

```

void Get_Angle(void)
{
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
    Gyro_Balance=-gy;
    x_angle=atan2(ax,az)*180/PI;
    gy=gy/16.4;
    Angle_Balance=kalman.getAngle(x_angle,-gy);
    Gyro_Turn=gz;
}

```

首先，读取 XYZ 轴的加速度以及陀螺仪的角速度：

```
mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
```

因为陀螺仪读取的角速度与实际车体的角度值正负极性相反，所以要进行取反：

```
Gyro_Balance=-gy;
```

如 1.3.1 节所述，透过计算 X 与 Z 轴的加速度分量的角度获取倾斜角，这里使用 atan2() 函数计算角度：

```
x_angle=atan2(ax,az)*180/PI;
```

还可以透过 Y 轴的陀螺仪角速度来获取倾斜角，但读出的角速度值要先除以精度值：

```
gy=gy/16.4;
```

MPU-6500 是 16 位数据寄存器，最高位是符号位，数据寄存器的输出范围是 -7FFF~7FFF，即 -32767~32767。如图 1-15 所示，如果选择陀螺仪范围为 ±2000，那么 -32767 对应的是 -2000(°/s)，32767 对应是 2000(°/s)，当读取陀螺仪的值为 1000 时，对应的角速度计算如下：32767/2000 = 1000/x，即 x = 1000/16.4(°/s)，可以看出 32767/2000 = 16.4，对应手册中的精度 16.4 LSB/(°/s)，其他范围的也是如此。

3.1 Gyroscope Specifications

Typical Operating Circuit of section 4.2, VDD = 1.8V, VDDIO = 1.8V, T_A=25°C, unless otherwise noted.

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		°/s	3
	FS_SEL=1		±500		°/s	3
	FS_SEL=2		±1000		°/s	3
	FS_SEL=3		±2000		°/s	3
Gyroscope ADC Word Length			16		bits	3
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)	3
	FS_SEL=1		65.5		LSB/(°/s)	3
	FS_SEL=2		32.8		LSB/(°/s)	3
	FS_SEL=3		16.4		LSB/(°/s)	3
Sensitivity Scale Factor Tolerance	25°C		±3		%	2
Sensitivity Scale Factor Variation Over Temperature	-40°C to +85°C		±4		%	1
Nonlinearity	Best fit straight line; 25°C		±0.1		%	1
Cross-Axis Sensitivity			±?		%	1

图 1-15 MPU-6500 datasheet 陀螺仪规格

将加速计与陀螺仪获取的角度值，送入卡尔曼滤波函数，得出误差较小的车体倾斜角度：

Angle_Balance=kalman.getAngle(x_angle,-gy);

在处理车身转弯时，需要利用 Z 轴的角速度来给系统参考：

Gyro_Turn=gz;

以上变量提供给系统的 PID 平衡算法控制使用，以便对车身的实际状况进行控制，达到平衡的状态。

1.4. 获取障碍物距离

本节将介绍如何使用超声波模块侦测平衡车身前方障碍物的距离。

1.4.1. 原理

如图 1-16 所示，平衡车上使用的超声波模块型号为 HC-SR04。除了电源与接地引脚，主要由 TRIG 与 ECHO 两个信号来控制。如图 1-17 所示为超声波模块工作示意图，工作方式描述如下：

- a. 开始侦测距离时，对 TRIG 端输入高电平，保持至少 10us。
- b. 模块内部自动向外发送 8 个 40KHz 的方波，并自动检测是否有信号返回。
- c. 检测到有信号返回后，ECHO 端自动输出高电平，高电平持续时间就是超声波从发射到返回所经历的时间。



图 1-16 超声波模块 HC-SR04

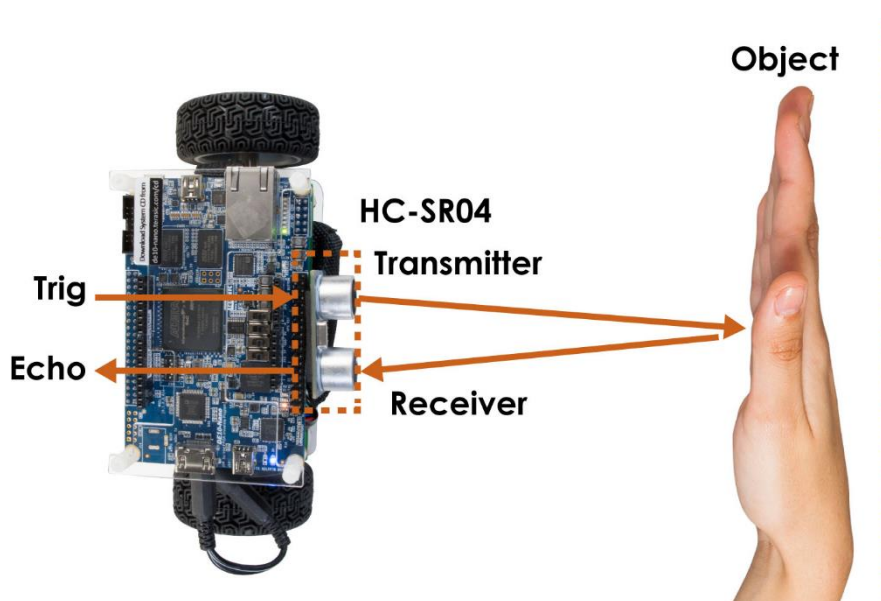


图 1-17 超声波模块工作示意图

■ 距离计算

距离=(高电平时间*声速(340m/s))/2，模块与障碍物间的距离可以通过高电平时间 * 音速 (340M/S)/2 得到。因为信号从发射到反射之间的距离是障碍物距离的两倍，所以除 2。这里的单位是米。其工作的波形图如图 1-18 所示。

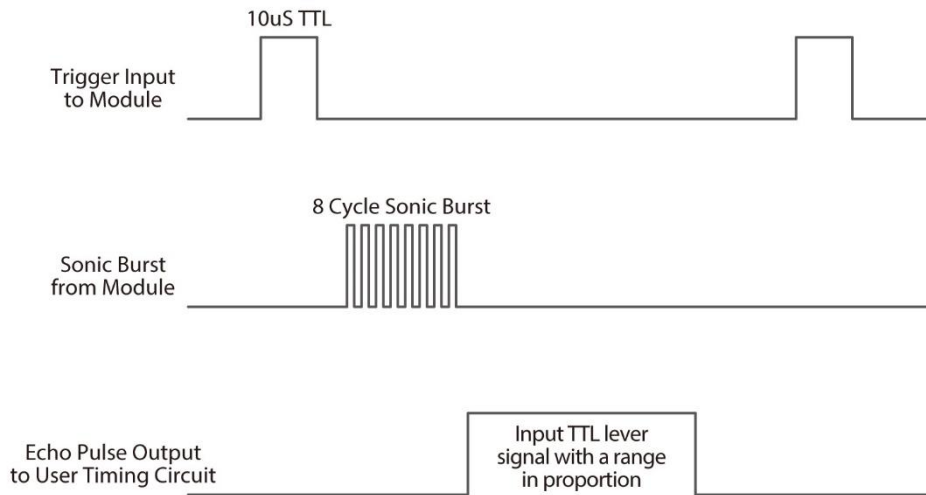


图 1-18 超声波模块工作波形图

1.4.2. 范例描述

平衡车的范例提供了一个通过超声波模块读取距离的 Qsys IP。位于 `\Demonstrations\BAL_CAR_Nios_Code\IP\sonic_distance\sonic_distance.v`

■ IP Symbol

如图 1-19 所示，此 IP 控制 TRIG 管脚，驱动超声波模块开始测距，然后监测 ECHO 端信号，是否有反射信号。并计算 ECHO 引脚信号的高电平持续时间，存入寄存器内，使 CPU 能读取数据。

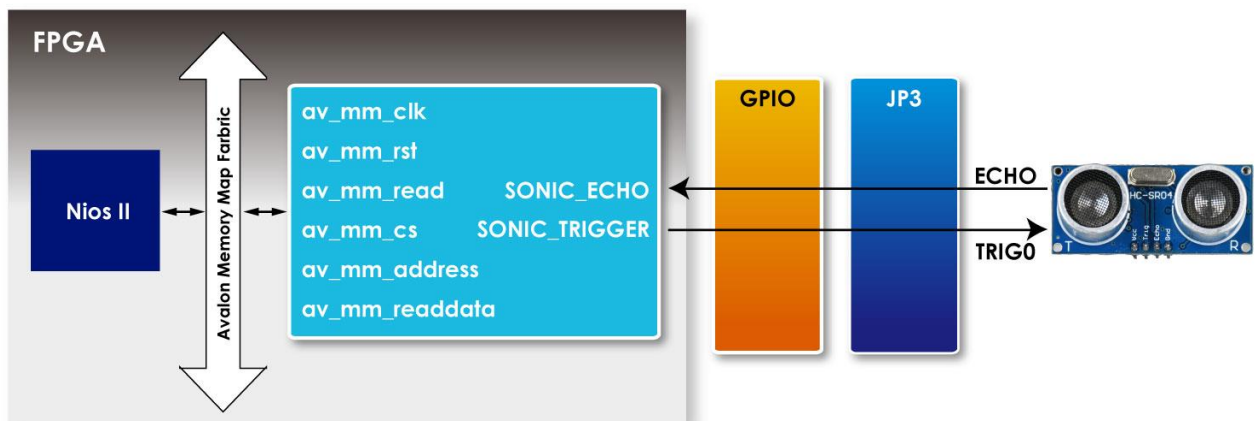


图 1-19 sonic_distance.v 的 Symbol 以及在系统内的方框图

■ Register Table

表 1-4 为本 IP 的 register table, measure_value register 储存每次超声波侦测到与物体距离的声波传输时间。

表 1-4 sonic_distance.v 寄存器

Reg Address	Bit Filed	Type	Name	Description
Base Addr + 0	31:22	RO	Unuse	--
	21:0	RO	measure_value	Sonic wave propagation time

■ IP 代码

此 IP 代码主要由状态机(State Machine)组成, 代码如下:

```
always @(posedge av_mm_clk or negedge count_rst)
if(~count_rst)
begin
    measure_count<=0;
    trig_count<=0;
    state<=0;
end
else
begin
    case(state)
    3'd0:begin
        sonic_trigger<=1;
        state<=1;
    end
    3'd1:begin
        if(trig_count==2000)
        begin
            sonic_trigger<=0;
            state<=2;
        end
    end
end
```



```

        else
        begin
            trig_count<=trig_count+1;
            state<=1;
        end
    end
3'd2:begin
    if(!reg_echo&sonic_echo)
        state<=3;
    else
        state<=2;
    end
3'd3:begin
    if(reg_echo&!sonic_echo)
        state<=4;
    else
        begin
            state<=3;
            measure_count<=measure_count+1;
        end
    end
3'd4:begin
    state<=state;
end
endcase
end

```

其状态如图 1-20 所示。

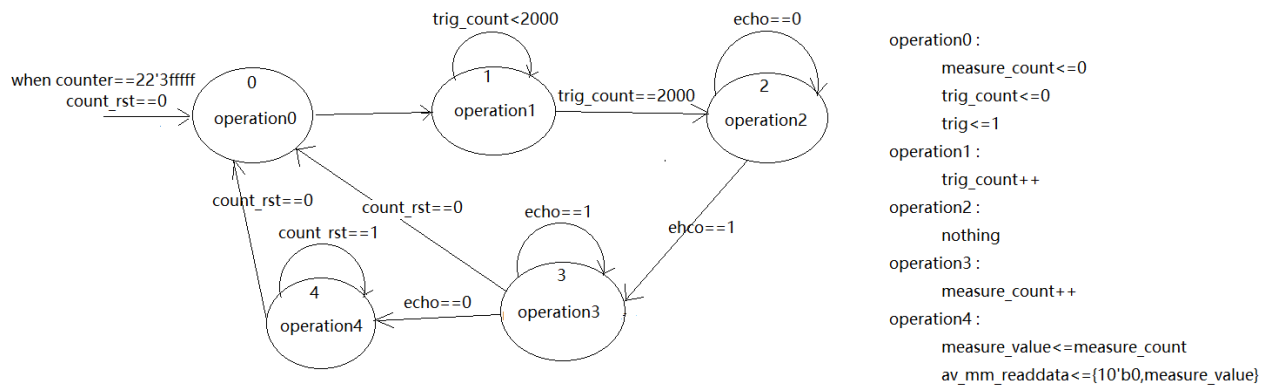


图 1-20 sonic_distance.v 内的 state machine 状态图

当 FPGA 运行时，此 IP 就会开始独立运行，进入 State 0。另外，为了避免状态机卡在某个状态下，这个 IP 设有一个 Counter 自行不停累加。当 Counter 值为 22'h3ffff 时，便会触发 count_rst = 0，使状态机重置。代码如下：

```

always @(posedge av_mm_clk or negedge av_mm_rst)
if(~av_mm_rst)
    counter<=0;
else if(counter==22'h3ffff)
    counter<=0;
else counter<=counter+1;
wire count_rst=(counter==22'h3ffff)?0:1;

```

下面是各个状态的描述：

State 0: 设定 TRIG 引脚为输出高电平，并进入 State 1。

State 1: 进入 State 1，trig_count 开始累加，直到 2000，因为系统 Clock 为 50MHz，所以此过程时间为 10us，此时完成了 TRIG 触发，将 TRIG 信号拉回低电平，并进入 State 2。

State 2: 监测 ECHO 信号是否在上升沿(rising edge)状态，如果是，代表有检测到障碍物，进入 State 3。如果没有，则保持在本状态下，直至 count_rst = 0，使状态机重置。

State 3: 使用 measure_count 计数声波反射时间，当监测到 ECHO 信号有下降沿(falling edge)时，进入 State 4。

State 4: idle 状态，等待 count_rst = 0，重置状态机，进入下一次侦测距离过程。

■ Software Code

添加 IP 到 Qsys，通过读取 IP 的数据寄存器(measure_value)，得到障碍物距离的声波传输时间，然后通过距离计算公式计算距离，Nios 代码路径：\Demonstrations\BAL_CAR_Nios_Code\software\DE10_Nano_bal\main.cpp

```
data = IORD(SONIC_DISTANCE_0_BASE,0x00);  
distance = (float)data*34.0/100000.0;
```

注意：前面提到过 IP 内的 Clock 为 50MHz，计算单位为米，所以计算距离的公式为 $data * 340 * 100 / (2 * 50 * 1000000)$ ，即 $data * 34.0 / 100000$ 。

1.5. 平衡车系统

本节将介绍平衡车的状态系统控制，介绍平衡车如何保持直立，如何控制速度以及转弯等状态。

如 1.3.1 节介绍，平衡车的倾斜角度和旋转角度测量是通过 MPU-6500 测量加速计与陀螺仪实现，平衡车运动速度通过电机的霍尔传感器实现，避障通过超声波传感器测量实现。这些测量值分别作为直立控制、旋转角度控制和速度控制的反馈值。

平衡车的状态控制引入了 PID Controller (Proportional-Integral-Derivative) 概念，采用 PI (比例积分) (Proportional Integral) 或 PD (比例微分) (Proportional Derivative) 来控制状态，分别作用于直立角度控制，旋转角度控制和速度控制。由于这三个控制都是闭环(closed loop)控制，所以又称为直立环(balance loop)，速度环(speed loop)和转向环(turn loop)。其中直立环用 PD 控制，速度环用 PI 控制，转向环用 P 控制。

直立环用 PD 控制，是因为平衡车需要对角度变化迅速做出反应，而微分控制刚好满足这一需求。P 的控制量是小车的倾角(相对于平衡时的角度偏差)，D 的控制量是电机陀螺仪。这个环对应代码里面的 int balance(float Angle, float Gyro) 函数。

速度环用 PI 控制，这是速度控制最常采用的控制方式，它是一种线性控制方式，根据给定值

与实际输出值构成偏差，然后将偏差的比例（P）和积分(I)通过线性组合构成控制量，对速度进行控制。P 的控制量是速度偏差, I 的控制量是位移。这个环对应代码里面的 `int speed(void)` 函数。

小车的转向控制比较简单，demo 通过两个速度编码器差值以及 MPU-6500 Z 轴陀螺仪来测量转向值，这两个分别作为 P 和 D 控制的控制量，从而对转向角进行 PD 控制以保持转向角保持为设定值，同时对 Z 轴陀螺仪控制也能提高小车的响应速度。这个环对应代码里面的 `int turn(float Gyro)`函数。

三个环的综合作用如图 1-21 所示。

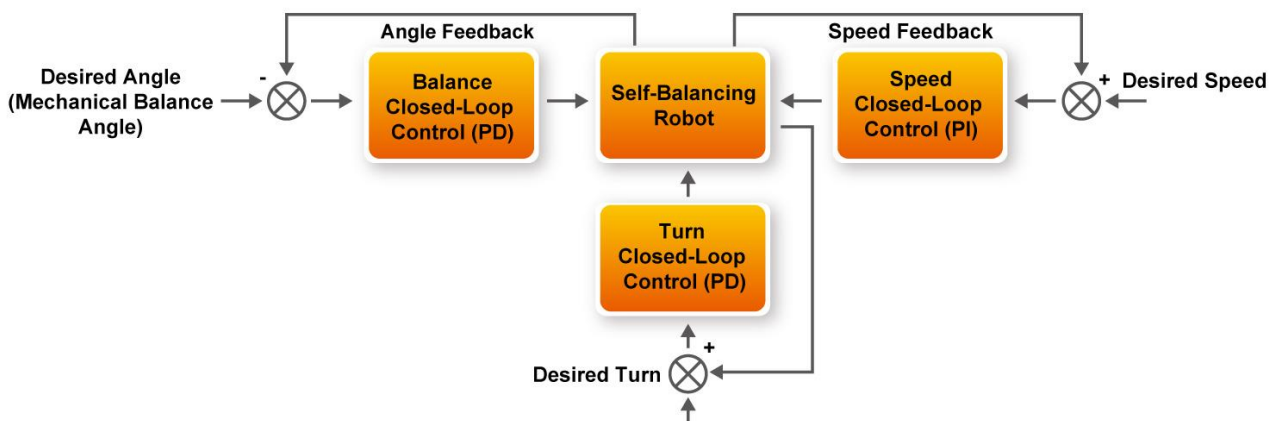


图 1-21 PID 闭环控制

注意: 平衡环，速度环以及转向环的 PID 三个参数都有极性，必须使反馈形成正反馈，从而有利于小车平衡的闭环控制。

为了实现人为(蓝牙/红外)控制，需要在速度环以及转向环中加入固定量以改变平衡车的运行速度(带方向)和转向(带方向)。对应的代码分别为 `int speed(void)`函数中的 `Encoder_Integral=Encoder_Integral-Movement;` 和 `int turn(float Gyro)`函数中的 `Bias+=110` 与 `Bias-=110;` 这样当蓝牙/红外发出控制指令后，速度环和转向环就开始运行，以设定的速度做直线运动，以设定的角速度做旋转运动。

平衡车的状态需要通过固定时间间隔进行采样控制。范例将 MPU-6500 的采样间隔设置为中断间隔，时间是 10ms，执行中断函数 `void MPU_INT_ISR(void * contex, alt_u32 id)`对车子

角度，速度，旋转角度进行采样控制。同时 main 函数中设置闭环轮询蓝牙/红外控制信号，超声波检测障碍物距离进行避障，以及监测电池电量电压。程序流程如图 1-22 所示：

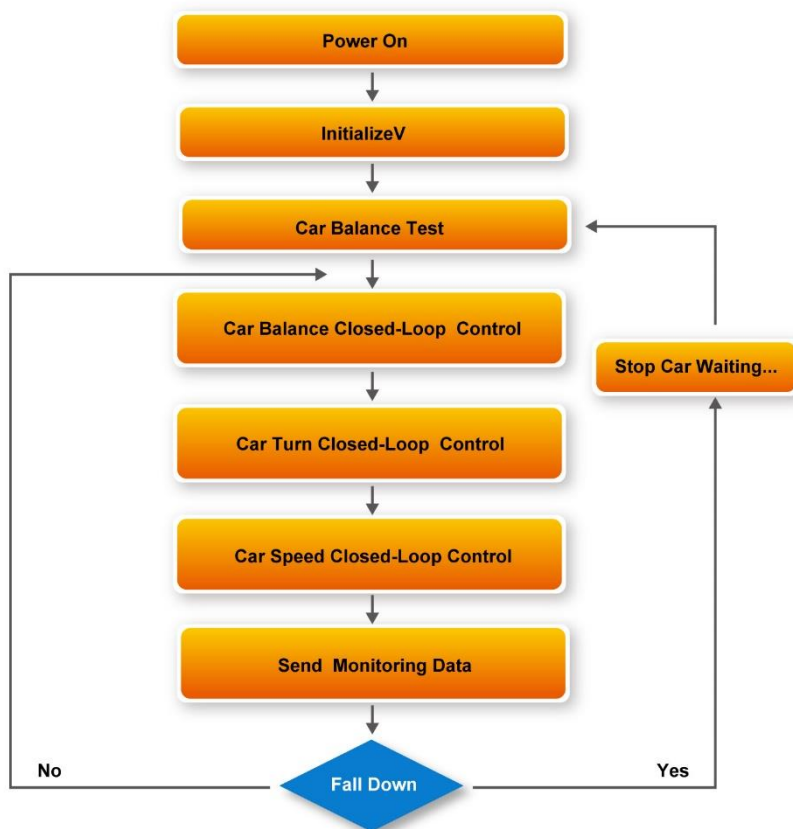


图 1-22 闭环控制程序流程图

1.6. 使用蓝牙

本节将介绍如何使用平衡车上的 ESP32 模块的蓝牙功能，这样用户可以用外部手机通过蓝牙与 ESP32 通信，并转换为串口协议传输至 FPGA，控制平衡车行动。

ESP32 是一款功能较强大 bluetooth+wifi 模块，可开发性较高，出厂的平衡车上的 ESP32 固化有带有平衡车 ID 编号的代码，能够接收手机 APP 通过蓝牙传送的控制指令。

除此之外还有很多其它可扩展功能，比如基于 I2C，WIFI 和 SPI 接口的数据传输等等，但在平衡车内目前只使用蓝牙部分。

图 1-23 为平衡车范例使用 ESP32 蓝牙功能的系统架构，当 ESP32 接收到 APP 通过蓝牙协议传来的字符串指令，便通过 UART 传到 FPGA 内 Qsys 的 UART IP 中。如此 Nios CPU 可以

很方便的读取 IP 中数据寄存器的值，然后与定义的指令比较，得到有效指令并控制平衡车运动。在 Quartus 工程内预留有一个 Qsys PIO 模块，使 ESP32 通过 I/O 通信,此范例内并没有使用。

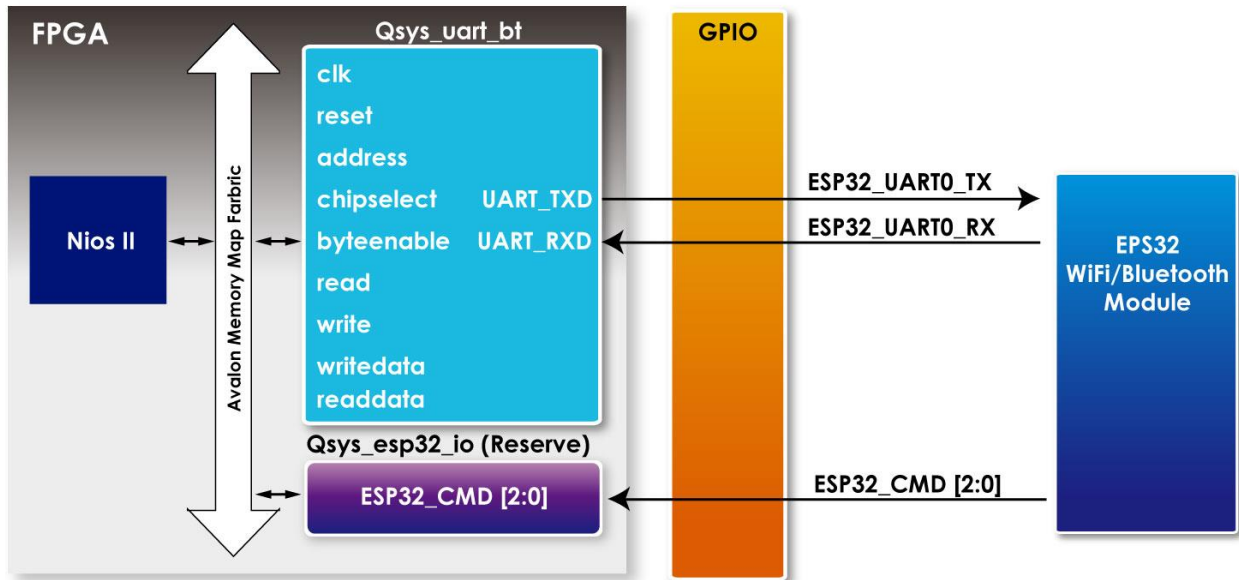


图 1-23 EPS32 与 FPGA 通讯的功能框图

范例内使用的 UART IP 是 Qsys 内嵌的组件，可以通过下面路径获取 User guide。

<Quartus install path>\<Quartus version ex:16.1 >\ip\altera\university_program\communication\
altera_up_avalon_rs232\doc\RS232.pdf

目前在范例内的设定如图 1-24 所示，Baud Rate 设定为 115200。

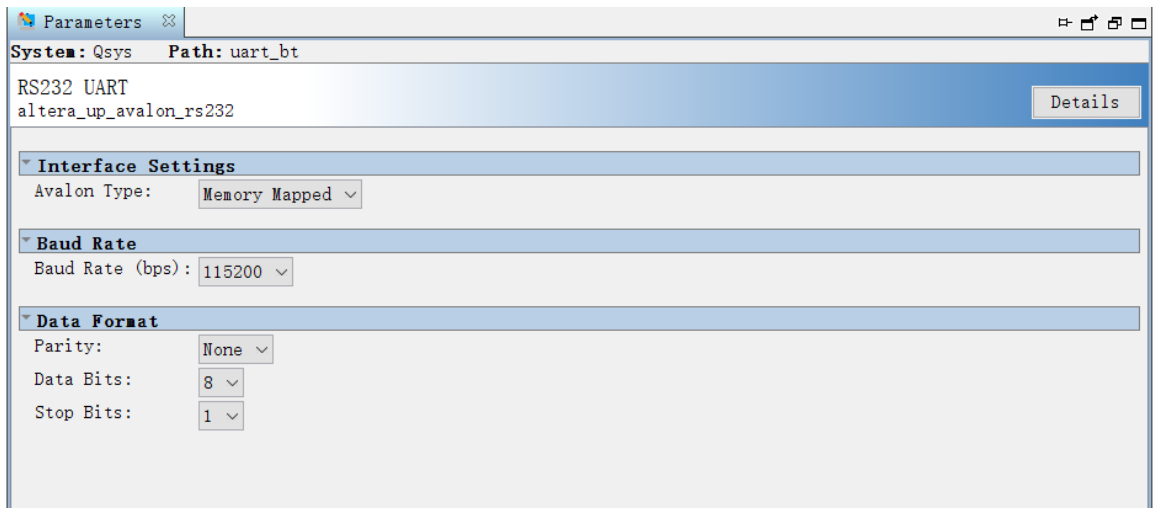


图 1-24 Qsys UART IP 设定

如表 1-5 所示，UART IP 内主要有两个寄存器，可以通过数据寄存器访问读写 FIFOs，通过手机蓝牙传输数据将被存储在此。RS232 UART Core 中断和读状态信息由 Control 寄存器控制。

表 1-5 UART IP 寄存器映射

Offset in bytes	Register Name	R/W	Bit description										
			31..24	23..16	15	14..11	10	9	8	7	6..2	1	0
0	data	RW	(1)	RAVAIL	RVALID	(1)		PE	(2)	(2)	DATA		
4	control	RW	(1)	WSPACE	(1)			WI	RI	(1)		WE	RE

表 1-6 为 Data 寄存器的格式。最后 8bit 是传输的数据，bit 23~16 用于显示还有多少存在读 FIFO 内等待读取，用户可以从这个位置了解到数据传输是否结束。

表 1-6 Data 寄存器位

Bit number	Bit/Field Name	Read/Write	Description
8...0	DATA	R/W	The value to transfer to/from the RS232 UART Core. When writing, the DATA field is a character to be written to the write FIFO. When reading, the DATA field is a character read from the read FIFO.
9	PE	R	Indicates whether the DATA field had a parity error.
15	RVALID	R	Indicates whether the DATA field and PE fields contain valid data.
23...16	RAVAIL	R	The number of characters remaining in the read FIFO (including this read).

下面将介绍如何用 Nios 读取 ESP32 传送过来的数据，并转换为控制命令。

在 main.cpp (路径:\BAL_CAR_Nios_Code\software\DE10_Nano_bal)中，有蓝牙指令的检测。

```
// Bluetooth control
temp=IORD(UART_BT_BASE,0x00);
number=temp>>16;
if(number!=0)
{
    szData[i]=temp&0xff;
    i++;
    if((temp&0xff)==0x0a)
    {
        i=0;
        if(CommandParsing(szData, &Command_EPS32, &Param)){
            switch(Command_EPS32){
                case CMD_FOWARD: //Forward
                    if(cmd_ut){
                        if(distance>15.0){
                            led3=0x01;
                            flag=0x01;
                            demo=false;
                            Car.Set_TurnFORWARD();
                        }
                    }
                else{
                    led3=0x01;
                }
            }
        }
    }
}
```



```

        flag=0x00;
        demo=false;
        Car.Set_TurnFORWARD();
    }
    break;
case CMD_BACKWARD: //Backward
    led3=0x02;
    demo=false;
    Car.Set_TurnBACKWARD();
    break;
case CMD_LEFT: //Left
    led3=0x04;
    demo=false;
    Car.Set_TurnLEFT();
    break;
case CMD_RIGHT: //Right
    led3=0x08;
    demo=false;
    Car.Set_TurnRIGHT();
    break;
case CMD_STOP: //Stop
    led3=0x00;
    demo=false;
    Car.Pause();
    break;

```

首先读取 UART IP 的接收数据寄存器（接收数据寄存器偏移地址为 0）：

```
temp=IORD(UART_BT_BASE,0x00);
```

当有蓝牙指令发送过来时，接收数据寄存器共有 32bits，但 UART IP 每次只传输 8bits 数据，所以一个指令需要接收多次才能完整收到，还有多少字符尚未被读取可以检查寄存器数据的 bit 23~16。将 temp 数据右移 16 位就可得知还有多少剩余未读的字符数：

```
number=temp>>16;
```

若读取的数据 number 不为 0，则数据有效。取最后 8 位存入数组，然后循环接收下一个 8bits，直到接收到数据为 0x0a（设定的结束符），0x0a 是我们自定义的传输结束符，在手机 APP 内

设定传完一个控制命令时，传送这个值代表指令已经发送完毕。

接下来进入指令比较阶段，所有蓝牙传过来的命令被定义在 `command.h` 内：

```
/*
 * Command.h
 *
 */
#ifndef COMMAND_H_
#define COMMAND_H_
#include "terasic_includes.h"
typedef enum{
    CMD_FOWARD=1,
    CMD_BACKWARD,
    CMD_LEFT,
    CMD_RIGHT,
    CMD_STOP,
    CMD_AKBT,
    CMD_ATDM,
    CMD_ATUTON,
    CMD_ATUTOFF,
}COMMAND_ID;
typedef struct{
    char szCommand[10];
    int CommandId;
    bool bParameter;
}COMMAND_INFO;
COMMAND_INFO gCommandList[] = {
    {"ATFW", CMD_FOWARD, false},
    {"ATBW", CMD_BACKWARD, false},
    {"ATTL", CMD_LEFT, false},
    {"ATTR", CMD_RIGHT, false},
    {"ATST", CMD_STOP, false},
    {"ATAB", CMD_AKBT, false},
    {"ATDM", CMD_ATDM, false},
    {"ATUTON", CMD_ATUTON, false},
    {"ATUTOFF", CMD_ATUTOFF, false},
};
#endif /* COMMAND_H_ */
```

将 ESP32 传输过来的命令字符与 `command.h` 内的定义对比，解析手机 APP 发过来的控制命令，代码如下(in `main.cpp`):

```

/*****
Function      : Bluetooth Command Parsing
parameter     : Command 、 Command ID
return value  : Command Parsing data
*****/

bool CommandParsing(char *pCommand, int *pCommandID, int *pParam){
    bool bFind = false;
    int nNum, i, j , x=0;
    bool find_equal = false;
    char Data[10]={0};
    nNum = sizeof(gCommandList)/sizeof(gCommandList[0]);
    for(i=0;i<nNum && !bFind;i++){
        if (strcmp(pCommand, gCommandList[i].szCommand,
strlen(gCommandList[i].szCommand)) == 0){
            *pCommandID = gCommandList[i].CommandId;
            if (gCommandList[i].bParameter){
                /*pParam = 10; ???
                //for(j=0;pCommand[j]!=0x0a;j++){
                for(j=0;pCommand[j]!=0x0d;j++){
                    if(find_equal==true){
                        Data[x] = pCommand[j];
                        x++;
                    }
                    else if(pCommand[j]=='=')
                        find_equal=true;
                }
                *pParam=atoi(Data);
            }
            bFind = true;
        } // if
    } // for
    return bFind;
}

```

最后将控制命令转换对应的控制函数, 控制平衡车, 如下面的倒退命令:

```
case CMD_BACKWARD: //Backward
    led3=0x02;
    demo=false;
    Car.Set_TurnBACKWARD();
    break;
```

1.7. 使用遥控器

除了能用手机 APP 通过控制平衡车外, 也可以使用红外遥控器来控制。随包装附赠的红外遥控器采用 NEC 协议, 并使用 38KHZ 频率, 能发射控制信号到平衡车上的红外接收器。经过 FPGA 内的解码 IP 将命令解码出来, 控制平衡车运动。

1.7.1. 红外遥控器协议

NEC 的格式由前置码, 16-bit 客户码和 8-bit 按键码组成, 先传输的前置码包含 9ms 载波和 4.5ms。接着传输 16-bit 客户码, 之后 8bit 按键码一次, 最后 8-bit 为反向按键码(Inversed Key Code), 即按键码的反向值, 目的是让 IR 接收端可以验证数据, 如图 1-25 所示。

逻辑判断采用时间长短来区分, 560us 的载波加上 1690us 的 0 代表传输逻辑 1, 560us 的载波加上 560us 的 0 代表逻辑 0。

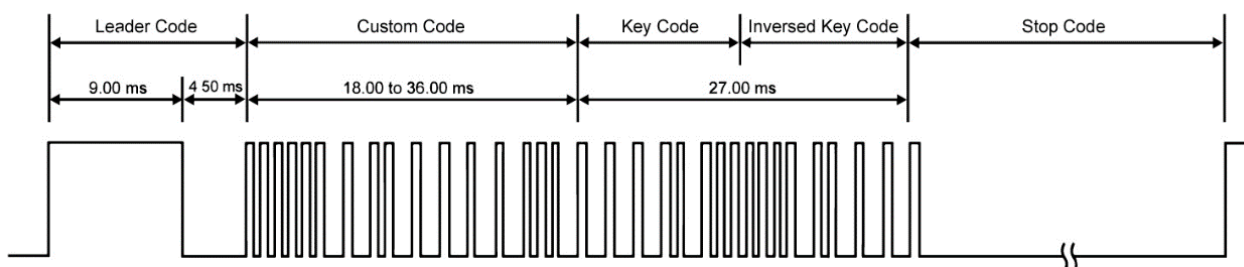


图 1-25 NEC 协议前置码与变动长度示意图

平衡车上的 IR 接收器可以解 38kHz 载波(carrier frequency), 并把接收到的讯号反向.所以须注意, FPGA 内处理的讯号会跟发射端相反, 如图 1-26 所示。

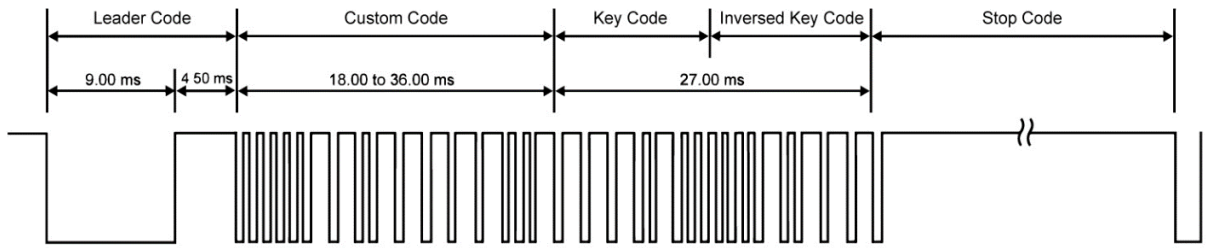


图 1-26 红外线接收器所接收的信号

平衡车动作按键定义与对应的 Key code 编码如图 1-27 所示。

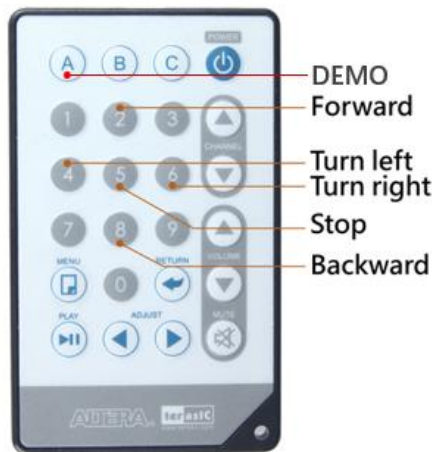






















图 1-27 红外遥控器按钮功能图

遥控器按钮对应的键码如表 1-7 所示。

表 1-7 遥控器按键码信息

IR Controller key	Custom Code		Key code	Inversed Key Code
	D[3:0] D[7:4]	D[11:8] D[15:12]	D[19:16] D[23:20]	D[27:24] D[31:28]
A	68	B6	F0	0F

	68	B6	31	CE
	68	B6	01	FE
	68	B6	00	FF
	68	B6	10	EF
	68	B6	20	DF
	68	B6	30	CF
	68	B6	40	BF
	68	B6	50	AF
	68	B6	60	9F
	68	B6	70	8F
	68	B6	80	7F
	68	B6	90	6F
	68	B6	21	DE
 Channel	68	B6	A1	5E

 Channel	68	B6	E1	1E
 Volume	68	B6	B1	4E
 Volume	68	B6	F1	0E
	68	B6	C0	3F
	68	B6	11	EE
	68	B6	61	9E

1.7.2. 范例描述

平衡车内的范例提供了一个解码红外遥控器编码的 Qsys IP。位于 `\Demonstrations\BAL_CAR_Nios_Code\IP\TERASIC_IRM\TERASIC_IRM.v`

图 1-28 为 FPGA 内使用 TERASIC_IRM.v 来解码红外信号的功能框图。红外接收器收到的信号会传入这个 IP 内。IP 主要提供 Avalon 接口, 主要解码部分由 `irda_receive_terasic.v` 子模块实现。解码出的 custom code 以及 key code 等信息将回传到 TERASIC_IRM.v, 并存入寄存器内。同时发出一个中断信号, 告诉 CPU 来读取解码出的按键值, 由 Nios CPU 来读取, 其寄存器格式如表 1-8 所示。

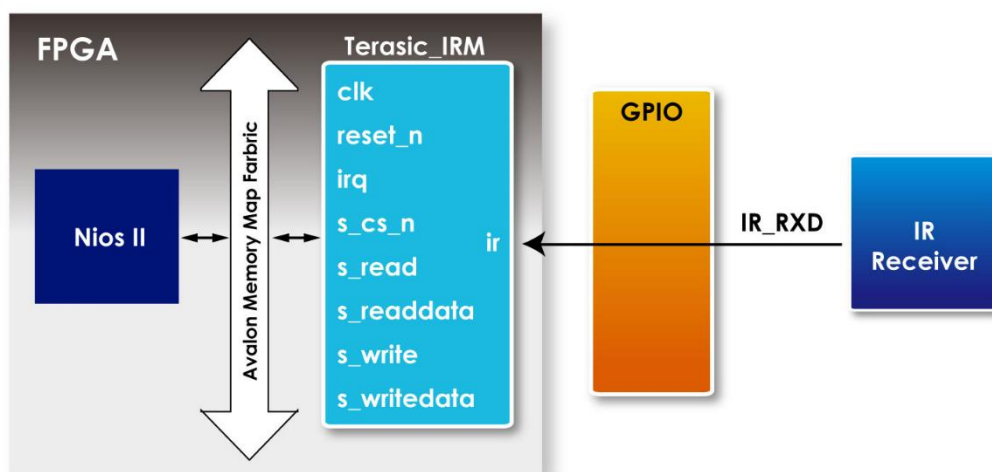


图 1-28 使用 Terasic_IRM.v 解码红外信号的功能框图

表 1-8 寄存器设定格式

Reg Address	Bit Filed	Type	Name	Description
Base Addr + 0	31:24	RO	Inversed Key Code	Inversed Key Code
	24:16	RO	Key Code	Key Code
	15:0	RO	Custom Code	Custom Code: 16'h6b86

如果按下遥控器的"2"按钮，寄存器改为 32'hfd026b86。其中 "6b86" 为 Custom code，"02" 为 Keycode，"fd"为 Inversed Key Code，也就是"02"的反向。

Nios 接收到 IR 中断后会读取寄存器的值，然后与定义好的码表比较，判断出指令的含义，如下的代码，在 IrRx.h 中定义有 IR 的 32bits 数据与 IR 遥控键位相对应的码表。

```
typedef enum{
    IR_POWER =      0xed126b86,
    IR_CH_UP =      0xe51a6b86,
    IR_CH_DOWN =    0xe11e6b86,
    IR_VOL_UP =     0xe41b6b86,
    IR_VOL_DOWN =   0xe01f6b86,
```



```

IR_MUTE =          0xf30c6b86,
IR_ADJ_LEFT =     0xeb146b86,
IR_ADJ_RIGHT =    0xe7186b86,
IR_PLAY_PAUSE = 0xe9166b86,
IR_NUM_0 =        0xff006b86,
IR_NUM_1 =        0xfe016b86,
IR_NUM_2 =        0xfd026b86,
IR_NUM_3 =        0xfc036b86,
IR_NUM_4 =        0xfb046b86,
IR_NUM_5 =        0xfa056b86,
IR_NUM_6 =        0xf9066b86,
IR_NUM_7 =        0xf8076b86,
IR_NUM_8 =        0xf7086b86,
IR_NUM_9 =        0xf6096b86,
IR_NUM_A =        0xf00f6b86,
IR_NUM_B =        0xec136b86,
IR_NUM_C =        0xef106b86,
IR_RETURN =       0xe8176b86,
IR_MENU =         0xee116b86
};

```

在 main.cpp 中有检测 IR 的部分，具体如下：

```

// IR Remote control
    if (!IR.IsEmpty()){
        Command_IR = IR.Pop();
        //Command_IR = IORD(IR_RX_BASE,0x00);
        //printf("%04xh\r\n", Command_IR);
        switch(Command_IR){
        case CIrRx::IR_NUM_5:    //Stop
            led3=0x00;
            demo=false;
            Car.Pause();
            break;
        case CIrRx::IR_NUM_2:    //Forward
            if(mode==0x02){
                if(distance>15.0){
                    led3=0x01;
                    flag=0x02;
                }
            }
        }
    }

```

```
demo=false;
Car.Set_TurnFORWARD();
}}
else{
led3=0x01;
flag=0x00;
demo=false;
Car.Set_TurnFORWARD();
}
break;
```

通过 IR.IsEmpty 检测是否收到 IR 数据，再通过 IR.Pop 读取寄存器 DATA_BUF 的值，然后比较是哪个按键，对应控制小车前进或者后退。

获得帮助

当您遇到问题时，请通过以下信息联系我们：

- Terasic Inc.
9F, No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, Taiwan 300-70
Email : support@terasic.com
Web : www.terasic.com

版本历史

日期	版本	修改记录
2018.03.16	First publication	
2018.07.11	V1.1	修改图 1-21 和图 1-22