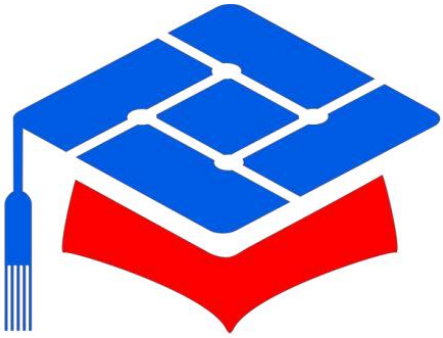


第四届

全国大学生集成电路创新创业大赛

CICIEC

智能 BLDC 控制系统设计报告

报名杯赛： arm 杯

队伍编号： ASC123511

团队名称： ILWT

目录

1 研究价值

1.1 研究背景

电机是工业生产和生活中必不可少的一种机械零件。工业中常用三相感应电机，小功率时也会使用单相感应电机。生活中常见的电机除了感应电机，还包括单相串励电机和有刷永磁直流电机。感应电机没有电刷，结构简单可靠，但功率因数较低，体积笨重，且小功率时效率低；单相串励电机起动转矩大，抗过载能力强，但是功率因数低，运行时噪音大，存在碳刷磨损问题；有刷永磁直流电机调速方便，噪音小，成本低，但是存在碳刷磨损问题，且永磁体在高温时会消磁，致使电机性能变差。

无刷永磁直流电机（BLDC）是一种随着半导体器件（尤其是功率器件）的发展而兴起的电机结构，它没有易磨损的电刷，也不需要巨大的定子以产生足够的电感，具备较低的运行噪声，较高的功率密度，极高的效率（通常大于 90%）和极高的可靠性。因此，BLDC 常常用于电动汽车，高端的家用电器和航空模型上面。我国盛产稀土元素，由钕铁硼材料制成的磁钢磁力强，耐高温，可以用于航空模型电机；而常见的铁氧体磁钢由于造价低廉，可以在家用电器上使用。

但是 BLDC 的缺点也非常显著，即需要复杂的外电路驱动。当前多数方案利用单片机作为控制器，使用中断和通用定时器模块实现换相，CPU 的内核时间大量被中断处理过程占用，效率很低。并且，控制器不够智能，无法根据不同电机自动调整 PWM 频率。另外，用户无从控制器得知电机的转速，温度，电流，电压等信息，人机交互不好。

1.2 研究目的

搭建以智能 BLDC 驱动为核心的 SoC 平台，克服市面上电子调速器只能控制电机而不能回传数据，自适应调节的缺陷，制作智能 BLDC 控制器。使用 BLDC 驱动硬件加速器对电机进行全阶段的控制，并自动处理堵转异常，防止电机烧毁，从而让 CPU 能够有更多空闲时间从事其他的工作，提升电机驱动的稳定性和 CPU 的利用率。另外，通过独立加速器，看门狗，default slave 等 SoC 层面的稳定性设计，加强系统整体稳定性。

我们将智能 BLDC 驱动系统样品与大疆创新公司的更高级别产品进行了对比，价格优势远高于竞品，功能也更加丰富：

	DJI 1240电子调速器	智能BLDC驱动系统
连续电流	25A	60A
动力电池	12s LiPo	3-4s LiPo
自适应PWM频率	不支持	支持，基于GA
重量	90 g	418 g（演示系统）
独立运行	不支持	支持
参数回传	不支持	支持
参数显示	支持，LED指示	支持，LCD显示
参数设定	支持，需另购下载器	支持，在线设定
电机过热保护	不支持	支持
电压异常保护	支持	支持
堵转保护	支持	支持
堵转自动重启动	不支持	支持
转向设定	不支持	支持
售价	650？	230*！！
*使用xc7a50t，不使用LCD显示器，计算最高BOM成本，并保证10元以上净利润		

相比大疆创新的 1240 电子调速器，我们具备自适应 PWM 频率调节，独立运行的功能，在参数回传和系统稳定性方面也更胜一筹。

1.3 应用领域

直流无刷永磁同步电动机（BLDC）驱动器广泛应用于航空航天、电动车辆、医疗器械、轻纺、办公自动化设备和现代家用电器等领域。比亚迪的电动汽车和 Tesla model 3 的主电机均使用了 BLDC 电机控制系统；而在高端家用电器领域，BLDC 控制系统也扮演了举足轻重的作用，比如在变频空调器和变频洗衣机中的应用；同样地，BLDC 在工业 4.0 的进程中也被广泛的应用，如数控机床，组合机床，自动化纺织和各种专用设备；此外，BLDC 在无人机等新兴领域的运用也愈发新潮。

2 系统功能介绍

该智能 BLDC 控制系统是一个完整的电机控制系统。可以实现 BLDC 的全过程驱动和状态检测，并设置有增强系统稳定性的堵转检测，过温保护，电压异常保护，程序异常运行保护等功能。此控制系统能够使用遗传算法（GA）进行自适应 PWM 频率设定，以适配不同的电机，增强了系统的通用性。系统可以独立运行，也可以嵌入其他设备，作为从系统运行，具有很高的灵活性。此外，还装置了 LCD 显示器和按键，方便进行人机交互。

首先，用户通过矩阵键盘设定系统时间，用户可以设定全部时间（年，月，日等），跳过部分时间设定或全部跳过，视用户需求而定。

然后，用户选择是否激活 GUI。如果用户激活 GUI，则控制系统作为从系统运行，矩阵键盘和拨码开关不起作用，系统控制权移交至 UART；如果用户不激活 GUI，则控制系统作为主系统运行，用户通过矩阵键盘和拨码开关控制电机，UART 接收到的数据不会干扰系统运行。

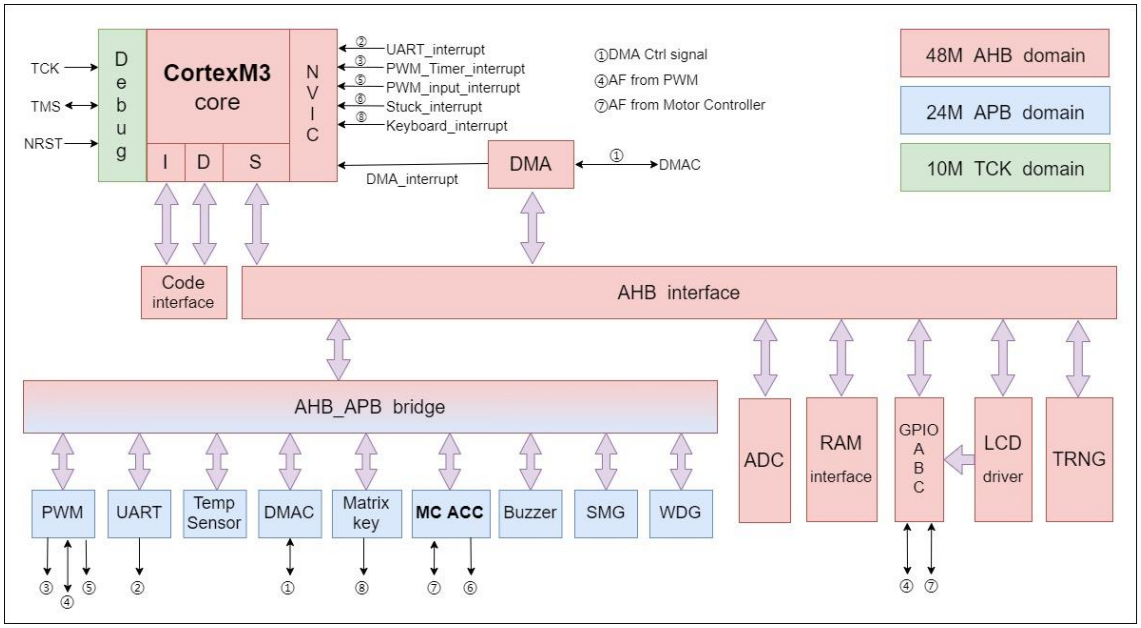
系统独立运行时，用户通过开关和按键设定 PWM 预分频器频率，电机定子极数（pole pair），正反转和堵转自动重启动。设定完成后，用户可以启动电机。如果用户想要改变转速，只需要转动旋钮即可迅速实现输出电压调整，非常方便。用户还可以读取电机当前的转速，供电电压，电机电流，传感器温度，系统时间等数据。当发生电压异常，堵转异常，温度异常时，控制系统能够自动切断电机电源，保障系统安全，防止电机烧毁，器件过

热，起火等事故。当用户设置了堵转自动重启动功能后，系统还能够在堵转后立刻自动恢复电机运行，这在航空模型等应用上是至关重要的。

控制系统作为子系统运行时，用户通过 UART 控制电机。独立运行时设定的参数，在作为子系统运行时均可以设置。此时，控制权属于主控制系统。运行在子系统状态下时，控制器会通过 UART 发送信息，因而可以移除 LCD 显示屏。

除了 BLDC 驱动加速器外，该 SoC 系统还具有多种外设。TRNG 外设能产生随机数，增加遗传算法的随机性；LCD 驱动器可以将 8080 并口的驱动速度提升至少 3 倍（对比 GPIO）；矩阵键盘可以实现电机的参数配置；UART 能够向其他控制芯片发送电机的运行数据，或从其他芯片接收控制命令；数码管可以显示电机的功率输出状态；蜂鸣器可以在出现异常时发出警告声，提示用户需要维护；通用定时器可以捕获来自 555 时基电路的脉宽，控制电压输出级别；ds18b20 驱动器可以实时测量电机的温度，无需 CPU 介入，一旦超过设置阈值，立刻停止电机运行，防止烧毁定子绕组或使得永磁体高温消磁；ADC 控制器能初始化 ADC 并能够自动采集电压和电流。

3 系统框图



4 软硬件功能划分

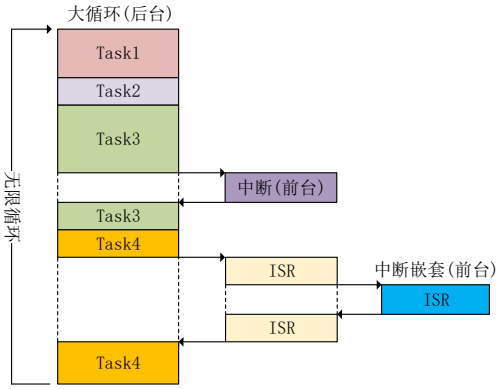
本 SoC 系统大量使用硬件电路控制外设，极大的减轻了 CPU 的负担。硬件模块负责 BLDC 的全过程驱动，LCD 的 8080 并口驱动，随机数生成，数码管动态刷新，蜂鸣器鸣叫，矩阵键盘数值读取，UART 发送和接收，脉宽捕获，电压与电流采集，温度传感器读数等对时序要求较高而对灵活性要求较低的应用。

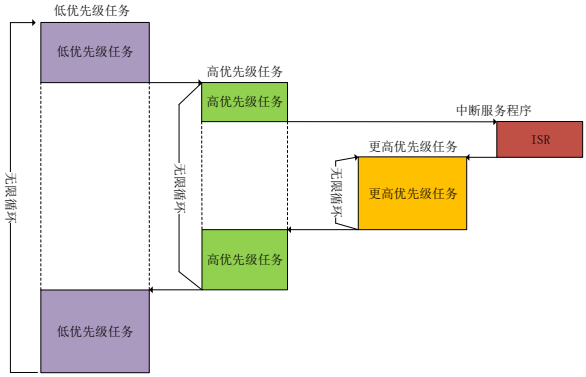
上述外设全部由团队成员使用 Verilog 写成，具有较高的自主性。

软件部分通过 C 语言和汇编语言编程完成，主要用于过程控制和异常响应，是 SoC 的灵魂。功能包括初始化操作系统，初始化各个外设，配置寄存器，进行堵转异常处理，处理 UART 接收数据，处理矩阵键盘数据，LCD 数据可视化，任务调度，运行遗传算法等。

电机属于机械部件，具有较大的惯性，无法向晶体管一样快速改变工作状态，遗传算法的个体需要约 500ms 的时间才能求解个体的适应度。因此，不宜设定过多的代数，每一代也不应产生过多的个体。适应度的求解算法为电机平均转速/电机平均电流，最佳个体为在相同电流下达到最高转速的预分频器数值。遗传算法共有 2 代。第一代在 256-3840 的预分频器解空间内产生 15 个个体，求解每个个体的适应度，将适应度较差的 14 个个体淘汰掉；第二代遗传第一代的预分频器数值，同时在 0-255 的范围内产生变异，生成 8 个新的个体；对 8 个个体的适应度进行求解，最优个体则被作为全局最优解使用。由于 ADC 有效位数仅有 7 位，没有必要进行多次迭代找到真正的全局最优解。

在未加入操作系统之前，我们采用的是单任务系统的编程框架，即裸机的编程框架。裸机编程主要是采用超级循环（super-loops）系统，其应用程序是一个无限的循环，循环机中调用相应的函数完成操作，这部分可以看做后台行为；中断服务程序处理异步事件，这部分可以看做是前台行为。对于前后台系统的编程思路主要有两种编程方式：查询方式以及中断方式。采用查询的中断结合的方式可以解决大部分裸机应用，但随着工程的复杂，裸机方式的缺点就会暴露出来：1、必须在中断（ISR）内处理时间关键运算 2、ISR 函数变得非常复杂，并且需要很长执行时间。





前后台操作系统　　RTOS 操作系统

针对以上情况，可以使用多任务系统（RTOS）来解决应用程序过于复杂带来的一系列问题。我们采用 UCOSIII 实时操作系统来进行任务调度以及异常处理，实现了一个抢占式任务调度器，提供延时、挂起、恢复任务、信号量保护等操作。

本次基于 UCOSIII 的实时操作系统运行步骤有如下 4 个：

1、通过调用系统初始化函数 OSInit()对系统进行初始化，该系统初始化是根据我们配置的宏定义进行的初始化，时间轮转调度等功能不进行配置。在初始化过程中有五个默认任务：空闲任务、时钟节拍任务、统计任务、定时任务以及中断服务管理任务。其中前两个必须创建，后三个根据需要选择性创建。

2、在 main()函数中创建一个 start_task 任务，其余的 3 个任务都在 start_task 任务中创建。

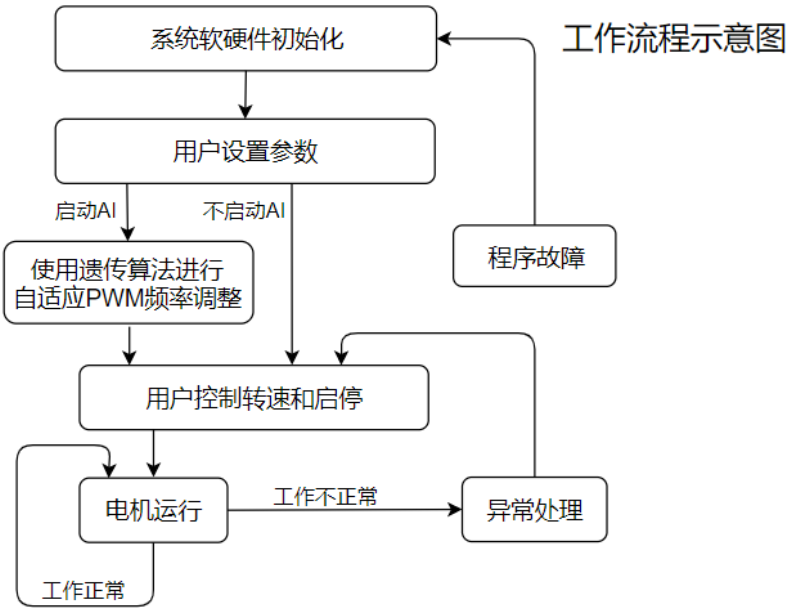
3、在 start_task 任务中创建应用程序所需要的任务：电机控制任务、过采样任务以及显示任务。

4、实时操作系统根据电机控制任务、过采样任务以及显示任务的优先级决定当前由哪个任务抢占 CPU。

以上就是一个抢占式任务调度系统的工作流程，为了保证系统的正常运行我们还要对任务间信号传递变量进行信号量保护、中断调度设置等。

另外，为了加强系统稳定性，我们使用了看门狗外设，在电机控制任务中进行喂狗操作，如果强电磁干扰或射线等不可控导致程序运行异常，看门狗的倒计时计数器便会因为没有运行喂狗程序给它重装载而清零，系统自动复位，有效防止灾难性故障的发生，提升了系统的整体稳定性。

通过软硬件的结合和 SoC 各个部分的密切配合，构成了完整的智能电机驱动系统。此外，由于硬件加速器的使用，用户也可以利用空余的 CPU 内核时间实现其他的功能。工作过程示意图如下所示：

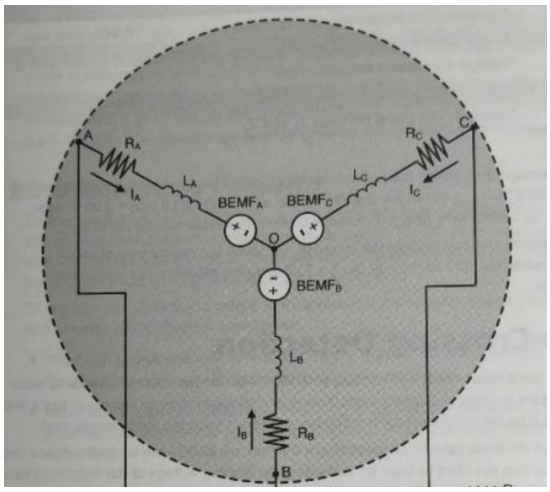


5 BLDC 驱动原理分析

BLDC 和感应电机均为无刷电机，但是 2 者的原理显著不同。感应电机通过交流电产生旋转磁场，一旦交流电的频率给定，感应电机的磁场转速即为恒定值。而 BLDC 使用直流电供电，依赖外部电路实现换相功能。通常，无感 BLDC 只引出 3 个接线端子，但是定子具有很多极数，不同的 BLDC，定子极数（pole pair）不同。

三相 BLDC 没有传统直流电机的换向器，为了防止转子被锁死，必须以特定的方式按顺序为三相中的两相通电来产生旋转磁场，其通电顺序依照转子所处位置的不同分为六步，故称为“六步换向”。比如可按如下的顺序通电：AB->AC->BC->BA->CA->CB。转子的位置可以通过霍尔效应传感器获得，抑或是通过轴角编码器获得。由于成本的限制和减少连线的要求，更有价值的是通过检测反电动势以确定转子的位置。

根据 BLDC 的机械和电子特性，对 BLDC 进行电路建模，如下图所示：



只考虑定子，根据其电学特性可以将其拆分成电感L和电阻R。然后考虑转子和定子构成的电机整体，根据公式：

$$E = NBLV$$

当转子开始旋转后，会切割磁感线产生电动势。换言之，外部电压并不会完全降落在线圈电阻上（电机堵转时除外），降落在线圈电阻上的电压为：

$$v_r = v_{cc} - v_{BEMF}$$

由此形成的电流为：

$$v_r / R$$

因此，在不带载的理想情况下，电机的反电动势等于外部电压，电机定子绕组上是没有电流的。当电机带载时，根据安培力公式和力矩公式：

$$\vec{F} = I\vec{L} \times \vec{B}$$

$$\vec{M} = \vec{L} \times \vec{F}$$

可知，电流大小和电机转矩成正比。

实际的空载电流远非如此理想。由于硅钢片磁滞现象，每一次换相都需要提供额外的电流才能够改变磁场方向。更致命的是，由于 PWM 斩波的实现难度远小于调整外部供电电压，实际应用中使用高频 PWM 控制电机会是更经济的选择，如此一来，外部供电实际是间歇的，这使得定子线圈绕组内的电流也时刻的改变着，产生额外的功耗。这些功耗完全耗散在定子的硅钢片上，变成废热。因此，即便是空载的时候，电机电流也不可能为 0，这不是电机驱动加速器的问题。

对于无感 BLDC 电机，能够方便测量的电学参数不外乎电压和电流。基于电压测量的算法是 6 步换向，基于电流测量的算法是 FOC 驱动。前者不需要使用 ADC 电路，成本较低，可靠性高；后者可以实现转矩的控制，但成本高昂。二者各有优劣，本次使用基于电压测量的 6 步换相算法，后面的计算也基于该算法。

对于 BLDC 控制，最难的是换相。不合时机的换相会极大的降低电机效率；由此产生的瞬态电流会减弱永磁体的磁场，使电机的寿命加速下降；最严重的时候，甚至会使电机失步，产生堵转状态，此时没有反电动势产生，外部电压全部降落在定子绕组线圈和驱动电路上，产生极大的电流和极高的热量，短时间内导致系统崩溃。

假设 3 组绕组的中间节点电压为$\mathrm{v_o}$，那么可以很快捷的得到每一相的电流和电压关系：

$$v_A - v_0 = I_A R_A + L_A \frac{\partial I_A}{\partial t} + v_{BEMFA}$$

$$v_B - v_0 = I_B R_B + L_B \frac{\partial I_B}{\partial t} + v_{BEMFB}$$

$$v_C - v_0 = I_C R_C + L_C \frac{\partial I_C}{\partial t} + v_{BEMFC}$$

在 BLDC 运行过程中，根据 6 步换相算法的过程描述，始终有 2 相导通而另外 1 相浮空。

假设电流保持不变或每个绕组的电感量完全相等，假设每个绕组的内阻完全相等，假设导通的 2 个绕组所产生的电动势大小相等，符号相反。以 B 极和 C 极为例，可以得出：

$$v_B - v_0 = -I_C R_C - L_C \frac{\partial I_C}{\partial t} - v_{BEMFC} = -(v_C - v_0)$$

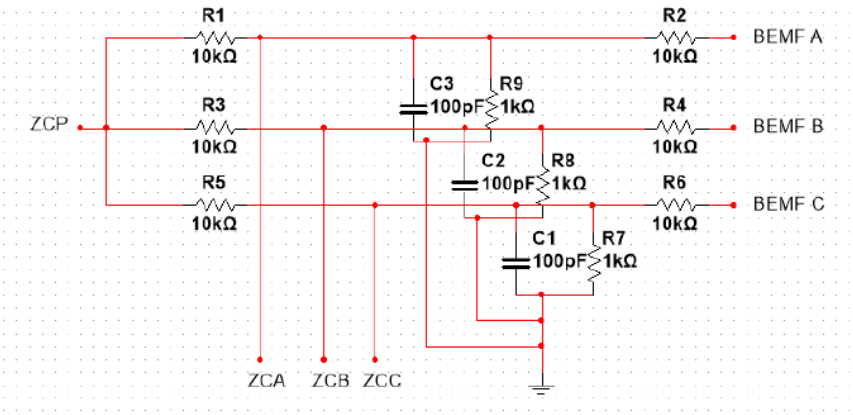
即中点电压：

$$\mathrm{v_o = (v_B + v_C) / 2}$$

上面我们假设 B 极和 C 极通有电流。那么，对于浮空的 A 极来说没有任何电流流过，即：

$$\mathrm{v_A - v_0 = v_{BEMFA}}$$

上面的公式似乎对换相没有帮助。但是，如果添加外围电路辅助，就能够间接地表示出反电动势，从而推断出相位，帮助换相。



令 ZCP 节点电压为v_P，ZCA 节点电压v_A，ZCB 节点电压v_B，ZCC 节点电压v_C，那么，根据 KCL 定律，对于纯阻性外电路，有：

$$\frac{v_A - v_P}{R} + \frac{v_B - v_P}{R} + \frac{v_C - v_P}{R} = 0$$

对该式进行处理可以得到一个实际可供测量的电压

$$v_P = (v_A + v_B + v_C) / 3$$

对上述表达式进行整合与化简，可以得到：

$$v_{BEMFA} = (2v_A - v_B - v_C) / 2$$

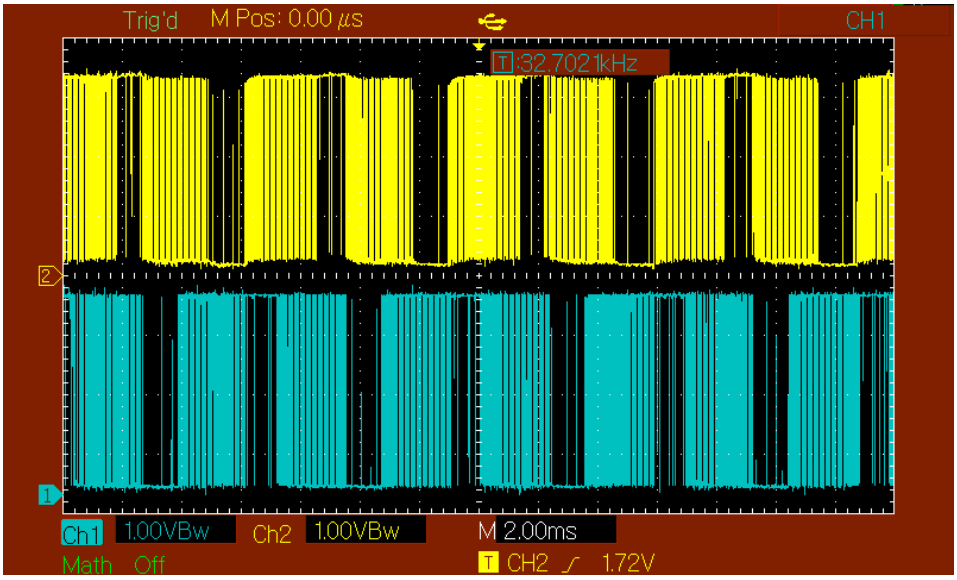
进一步整合与化简可得：

$$v_P - v_A = -\left(\frac{2}{3}\right) v_{BEMFA}$$

亦即反电动势是否经过方向上的变化完全可以仅通过外电路测量得到。而过零点也顺理成章的可以通过比较v_P和其他 3 个端子的电压得到。这种情况下，使用低延时，低成本的比较器代替高延时，高成本 ADC，是一种经济且高效的选择。

然而，反电动势的最终表达方式是在理想情况下得到的。实际情况却是：绕组所能产生的反电动势并不相等，每个绕组的电感也有区别，每个绕组的电阻也不相等。除了电机本身不理想外，外围电路的器件规格也不能保证完全相等。由于使用了 PWM 斩波来间接的调整电压级别，这些电路存在严重的开关噪声。当 PWM 由高到低或由低到高跳变时，会产生振铃现象，EMI 兼容问题等严重影响电路工作的因素，导致比较器输出的波形杂乱无章，一般的低通滤波器完全无法处理这样的信号。所以，必须选择相对稳定的阶段来对比较器的输出结果进行采样。

下图为过零检测电路比较器的输出波形：



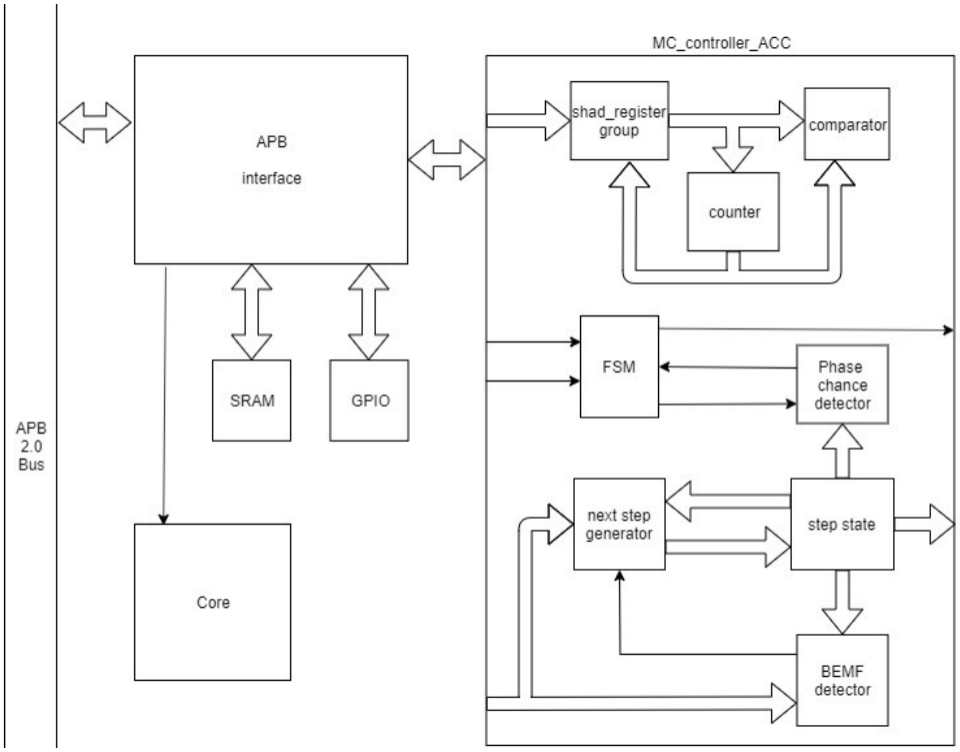
稳定的阶段包括 PWM 的低电平阶段和 PWM 的高电平阶段。为了确保采样稳定，必须对电路进行适当的延时，可以选择在 PWM 为低电平期间延时或是 PWM 为高电平期间延时。前者的缺点在于无法使电机全功率运转，后者的缺点在于无法使电机以超低功率运转。考虑到电机不同绕组的电阻并不相同，为了使上述等式尽可能的接近真实值，故选择在 PWM 为低电平的阶段进行过零检测的采样，同时利用智能开关噪声消除算法进行采样。

至此，所有和电机驱动硬件加速器相关的理论分析全部完成。

6 智能 BLDC 驱动加速器设计细节

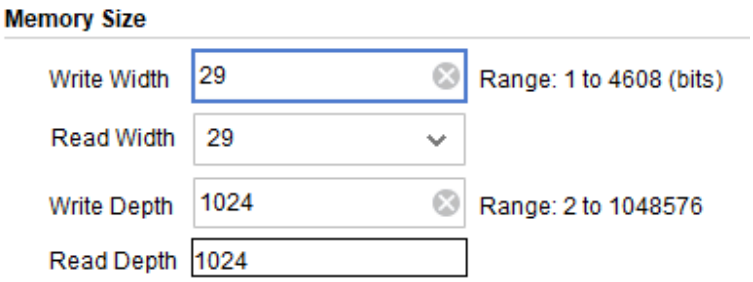
根据 BLDC 驱动的原理分析部分可知，电机驱动可以分阶段进行，类似于卷积神经网络的分层计算。电机驱动可分为转子定向，开环加速，环路控制 3 个阶段。转子定向和开环加速可以整合成为 1 个阶段。而环路控制必须是一个独立的阶段。

因此，该加速器架构使用 mealy 状态机进行设计。为了方便构造加速器，使用模块化的设计，主要分为 BRAM 指令缓存模块，APB_interface 模块和 Motor Controller 模块。BRAM 指令缓存模块用于暂存电机转子定向到开环加速全过程的延时，电压级别和相序；APB_interface 提供了 Motor Controller 到 APB 总线的接口；Motor Controller 用于控制 BLDC 电机。



加速器的原理图

BRAM 指令缓存模块使用了一个 29bits 宽，1024 深度的 BRAM，足够存储电机启动所需的全部序列。



APB_interface 上定义了多组寄存器，如下所示：

0x0000-0x0fff 定义了指令缓存寄存器，总共 1024 个，只能写入，每个寄存器包括 14 位延时，3 位相序和 12 位电压级别。

0x1000 寄存器为闭环状态下的电压输出级别，可读可写，总共 12 位，需要根据预分频计数器来设定，数值越大，则输出电压级别越高。

0x1004 寄存器为预分频计数器，可读可写，总共 12 位，设此寄存器是因为直接使用 24MHz 的 PCLK 时钟会导致 PWM 频率过高，产生涡流，应用于低转速电机是对电能的极大浪费，甚至不能启动。

0x1008 寄存器为闭环状态下的过零检测延时寄存器，可读可写，总共

10 位，用于确保获取结果时电路处于较为稳定的状态，在 0x1000 寄存器数值较大时，该寄存器数值可以相应减小。

0x100c 寄存器为控制寄存器，可读可写，位 1 控制自动重启，位 0 控制闭环状态下的转向。

0x1010 寄存器为相时寄存器，只能读取，总共 14 位，可以表示每一个相序停留的时常，根据该寄存器数值，预分频器数值和定子极数可以算出电机转速。

0x1014 寄存器为使能寄存器，可读可写，只有 1 位，CPU 通过写入该寄存器控制电机驱动加速器的开关。值得注意的是，当堵转现象发生且没有设置自动重启时，电机控制器会自动将该寄存器清零。

0x1018 寄存器为状态寄存器，只能读取，总共 3 位，位 2 表示电机控制器处于执行状态（有 PWM 输出），位 1 表示电机处于闭环状态，位 0 表示电机处于使能状态。高位所示状态是低位所示状态的真子集。

0x101c 寄存器为相时溢出寄存器，可读可写，总共 14 位，CPU 通过写入该寄存器设置堵转检测的临界值。

以上为 APB_interface 所定义的寄存器。

APB_interface 顶层模块包括 APB 信号的电机驱动硬件加速器的输出，具体信号如下：

```
module APB_MC(
    //APB2 interface
    input PCLK,
    input PRESETn,
    input PSEL,
    input PENABLE,
    input [31:0] PADDR,
    input [31:0] PWDATA,
    input PWRITE,
    output [31:0] PRDATA,
    //MC interface
    input [2:0] BEMF,
    output [2:0] VH,
    output [2:0] VL,
    //ERR interrupt
    output stuck
);
```

定义了多个控制寄存器和控制信号，这些控制寄存器和控制信号用于接收从 APB2 总线发送的数据，或由 MC 主体发送的数据，也可以响应 APB2 总线的读请求，如下图所示：

```
19  //control registers DEF
20  reg [11:0] Prediv;  //PWM predivider
21  reg [13:0] ph_overflow;  //ph_cnt overflow
22  reg [11:0] close_loop_pwm;  //close loop PWM
23  reg [9:0] comp_delay;  //BEMF compare delay
24  reg [1:0] ctrl;  //ctrl register
25  reg en;  //en register
26
27  //control wire DEF
28  wire [2:0] status;  //status indicator
29  wire [9:0] boot_addr;  //max 1024 steps
30  wire [9:0] boot_addr_wr;
31  wire [9:0] boot_addr_rd;
32  wire [28:0] boot_cmd_wr;  //14bits delay, 3bits phase, 12bits PWM
33  wire [28:0] boot_cmd_rd;
34  wire [13:0] ph_cnt;
35  wire boot_wr_en;
36  wire [11:0] Prediv_next = (|PWDATA[11:8])? PWDATA[11:0] : 12'd256;
```

APB 总线相关的控制寄存器如下图所示，按照时序执行可以实现控制寄存器的读取与写入：

```
38  //APB Write and read control block
39  wire write_en = PSEL & (PWRITE) & (~PENABLE);
40  wire read_en = PSEL & (~PWRITE) & (~PENABLE);
41  reg wr_en_reg;
42  reg rd_en_reg;
43  reg [10:0] addr_reg;
44  always@(posedge PCLK or negedge PRESETn) begin
45    if(~PRESETn) wr_en_reg <= 1'b0;
46    else if(write_en) wr_en_reg <= 1'b1;
47    else wr_en_reg <= 1'b0;
48  end
49  always@(posedge PCLK or negedge PRESETn) begin
50    if(~PRESETn) rd_en_reg <= 1'b0;
51    else if(read_en) rd_en_reg <= 1'b1;
52    else rd_en_reg <= 1'b0;
53  end
54  always@(posedge PCLK or negedge PRESETn) begin
55    if(~PRESETn) addr_reg <= 11'b0;
56    else if(write_en | read_en) addr_reg <= PADDR[12:2];
57  end
```

BRAM 读写控制模块，实现 BRAM 的 APB 总线写入和 Motor Controller 读取，如下图所示：

```
//BRAM control block
assign boot_addr = (en)? boot_addr_rd : boot_addr_wr;
assign boot_addr_wr = addr_reg[9:0];
assign boot_wr_en = (wr_en_reg & ~addr_reg[10] & ~en);
assign boot_cmd_wr = {PWDATA[29:16],PWDATA[14:12],PWDATA[11:0]};
```

部分与 Motor Controller 相关的控制寄存器受 APB 总线和 Motor Controller 的双重控制，其他寄存器起到暂存数据的作用。

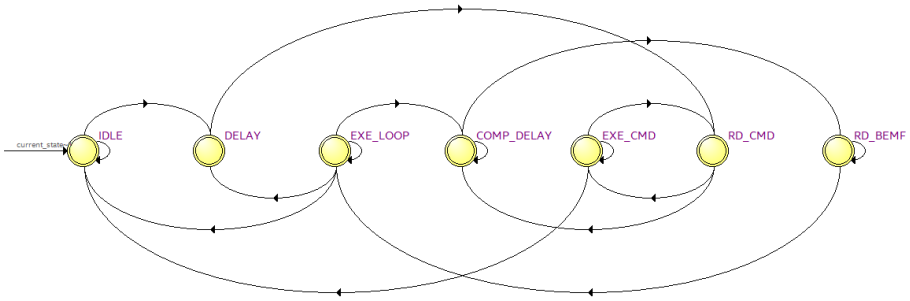
CPU 可以从 MC 加速器中读出闭环 PWM 数值，预分频器数值，输入捕获延时寄存器数值，控制寄存器数值，相时寄存器数值，使能寄存器数值，状态寄存器数值和相时超时寄存器数值。PRDATA 的控制逻辑如下：

```
//APB read
assign PRDATA = (rd_en_reg & addr_reg[10])? (
                {(32){(addr_reg[2:0] == 3'd0)}} & {20'b0,close_loop_pwm} |
                {(32){(addr_reg[2:0] == 3'd1)}} & {20'b0,Prediv} |
                {(32){(addr_reg[2:0] == 3'd2)}} & {22'b0,comp_delay} |
                {(32){(addr_reg[2:0] == 3'd3)}} & {30'b0,ctrl} |
                {(32){(addr_reg[2:0] == 3'd4)}} & {18'b0,ph_cnt} |
                {(32){(addr_reg[2:0] == 3'd5)}} & {31'b0,en} |
                {(32){(addr_reg[2:0] == 3'd6)}} & {29'b0,status} |
                {(32){(addr_reg[2:0] == 3'd7)}} & {18'b0,ph_overflow}
                )
                : 32'b0;
```

CPU 可以向 MC 控制寄存器中写入数据，部分关键数据会进行复位初始值设定，即使 CPU 没有配置某些寄存器，也不会发生堵转等异常，只会降低电机效率。控制寄存器如下：

```
78  //APB write
79  always@(posedge PCLK or negedge PRESETn) begin
80    if(~PRESETn) close_loop_pwm <= 12'b0;
81    else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd0)) close_loop_pwm <= PWDATA[11:0];
82  end
83  always@(posedge PCLK or negedge PRESETn) begin
84    if(~PRESETn) Prediv <= 12'd256;
85    else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd1)) Prediv <= Prediv_next;
86  end
87  always@(posedge PCLK or negedge PRESETn) begin
88    if(~PRESETn) comp_delay <= 10'd30;
89    else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd2)) comp_delay <= PWDATA[9:0];
90  end
91  always@(posedge PCLK or negedge PRESETn) begin
92    if(~PRESETn) ctrl <= 2'b0;
93    else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd3)) ctrl <= PWDATA[1:0];
94  end
95  always@(posedge PCLK or negedge PRESETn) begin
96    if(~PRESETn) ph_overflow <= 14'd255;
97    else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd7)) ph_overflow <= PWDATA[13:0];
98  end
99  always@(posedge PCLK or negedge PRESETn) begin
100   if(~PRESETn) en <= 1'b0;
101   else if(wr_en_reg & addr_reg[10] & (addr_reg[2:0] == 3'd5)) en <= PWDATA[0:0];
102   else if(stuck & ~ctrl[1]) en <= 1'b0;
103 end
```

Motor Controller 是 BLDC 驱动硬件加速器的核心部分，使用状态机进行控制，并具备较为复杂的控制逻辑。IDLE 状态表示加速器处于非使能状态，该状态只负责等待使能信号；DELAY 状态配合自动重启动功能使用，可以确保开始时 BRAM 的指针指向 0x000 地址，防止读取控制指令时出现错误；RD_CMD 状态表示加速器正在读取电机控制指令，在这个状态中，BRAM 的指针会自增，读取到的指令会被缓存到相应的寄存器中，如果没有指令或是已经到达地址边界，则跳转至 COMP_DELAY 状态；EXE_CMD 状态表示正在执行读到的指令，当指令执行完毕后会返回到 RD_CMD 状态；COMP_DELAY 状态表示正在进行延时操作，当 PWM 占空比非满时没有影响，当 PWM 占空比 100%时，COMP_DELAY 就是必须经过的状态；RD_BEMF 状态表示正在读取过零检测的结果，为了使检测结果更加可靠，总共会读取 8 次，如果维持原状态的次数小于等于 4 次，则认为反电动势没有过零点，反之则认为反电动势已经过零点；EXE_LOOP 状态表示正在输出 PWM 波形，当一个周期的 PWM 波形输出完毕后，会跳转至 COMP_DELAY 状态。这些全部都是在正常运行情况下的状态切换。如果在运行期间，CPU 将使能寄存器复位，则处于 EXE_CMD 状态或 EXE_LOOP 状态下的硬件加速器会立刻回到 IDLE 状态。如果在运行期间发生堵转，则硬件加速器根据自动重启动控制寄存器的值判断是返回 IDLE 状态，还是返回 DELAY 状态进行新一轮的电机驱动。状态机的转移关系图如下：




```
209 //FSM logic
210 assign next_state = (((current_state == EXE_CMD) & ~en) | ((current_state == EXE_LOOP) & ((en & phase_overflow & ~shad_restart_en) | ~en)))? IDLE :
211                    (((current_state == IDLE) & en) | ((current_state == EXE_LOOP) & (phase_overflow & en & shad_restart_en)))? DELAY :
212                    ((current_state == DELAY) | ((current_state == EXE_CMD) & en & EXE_CMD_done))? RD_CMD :
213                    ((current_state == RD_CMD) & ~CMD_empty)? EXE_CMD :
214                    (((current_state == RD_CMD) & CMD_empty) | ((current_state == EXE_LOOP) & (en & EXE_LOOP_done)))? COMP_DELAY :
215                    ((current_state == COMP_DELAY) & COMP_DELAY_done)? RD_BEMF :
216                    ((current_state == RD_BEMF) & RD_BEMF_done)? EXE_LOOP :
217                    current_state;
218
```

除了状态机以外，该硬件加速器的复杂之处还包括指令译码系统，多重状态切换系统，影子寄存器，以及可靠性与多功能方面的设计。Motor

Controller 的顶层模块和 APB_interface 相连，还和电机驱动电路的物理接口，NVIC 相连，具体接口如下：

```
module MC(
    input PCLK,
    input PRESETn,
    input [2:0] BEMF,
    output [2:0] VH,
    output [2:0] VL,
    input [28:0] boot_cmd,
    output [9:0] boot_addr,
    input [11:0] Prediv,
    input [11:0] close_loop_pwm,
    input [9:0] comp_delay,
    input [13:0] ph_overflow,
    input orientation,
    input restart_en,
    input en,

    output [2:0] status,
    output [13:0] ph_cnt,
    output stuck
);
```

为了方便设计硬件电路，定义了一些参数，包括状态名和相序，具体如下图所示：

```
//state DEF
parameter [6:0] IDLE        = 7'b0000001;
parameter [6:0] DELAY       = 7'b0000010;
parameter [6:0] RD_CMD      = 7'b0000100;
parameter [6:0] EXE_CMD     = 7'b0001000;
parameter [6:0] COMP_DELAY  = 7'b0010000;
parameter [6:0] RD_BEMF     = 7'b0100000;
parameter [6:0] EXE_LOOP    = 7'b1000000;

//phase DEF
//PH[8:6]=BEMF  PH[5:3]=VL  PH[2:0]=VH
parameter [8:0] PH0 = 9'b000_000_000;
parameter [8:0] PH1 = 9'b100_010_001;
parameter [8:0] PH2 = 9'b010_100_001;
parameter [8:0] PH3 = 9'b001_100_010;
parameter [8:0] PH4 = 9'b100_001_010;
parameter [8:0] PH5 = 9'b010_001_100;
parameter [8:0] PH6 = 9'b001_010_100;
```

状态切换控制如下图所示：

```
//state ctrl
reg [6:0] current_state;
wire [6:0] next_state;
always@(posedge PCLK or negedge PRESETn) begin
  if(~PRESETn) current_state <= IDLE;
  else current_state <= next_state;
end
```

定义影子寄存器，这些寄存器仅在特定状态下才会装载，可以防止 CPU 在硬件加速器工作期间访问控制寄存器，造成电路失效。该设计通过增加约一倍的控制寄存器，实现了更高的可靠性。寄存器定义如下图所示：

```
49  //shad register DEF
50  reg [9:0]  shad_comp_delay;       //registered when DELAY
51  reg        shad_orientation;      //registered when DELAY
52  reg        shad_restart_en;       //registered when DELAY
53  reg [13:0] shad_ph_overflow;      //registered when DELAY
54  reg [11:0] shad_Prediv;           //registered when DELAY or COMP_DELAY is done
55  reg [11:0] shad_close_loop_pwm;   //registered when COMP_DELAY is done
56  reg [13:0] shad_phcnt;            //registered when phase_change_pulse
57  reg [13:0] shad_cmd_delay;        //registered when RD_CMD
58  reg [11:0] shad_cmd_PWM;          //registered when RD_CMD
```

定义多组计数器，以满足硬件加速器的各种功能，具体定义及其控制方式如下图所示：

```
60  //counter DEF
61  reg [13:0] cmd_delay_cnt;   //command delay counter, cleared when !EXE_CMD, enabled when EXE_CMD and PWM_cnt_done
62  reg [8:0] current_phase;    //phase controller, one-hot encoded. revised when RD_BEMF_done or (RD_CMD & CMD_empty=0)
63  reg [8:0] last_phase;       //phase recorder, one-hot encoded. record <current_phase> when RD_BEMF_done
64                              //Prediv counter, cleared when reaching <shad_Prediv> or !EXE, enabled when EXE
65  //reg [11:0] PWM_cnt;         //PWM counter, cleared when !EXE, enabled when <shad_Prediv>==<Prediv_cnt> & EXE
66  reg [11:0] PWM_cnt;         //PWM counter, cleared when !EXE, enabled when EXE
67  reg [10:0] addr_cnt;        //boot address counter, enabled when RD_CMD, cleared when !(RD_CMD or EXE_CMD)
68  reg [13:0] phase_cnt;       //phase counter, cleared when phase_change or !(3states), enabled when RD_BEMF_done & !phase_change
69  reg [9:0] comp_delay_cnt;   //compare delay counter, enabled when COMP_DELAY, cleared when !COMP_DELAY
70  reg [2:0] BEMF_cnt;         //BEMF counter, enabled when RD_BEMF, cleared when !RD_BEMF
71  reg [2:0] BEMF_HI_cnt;      //BEMF High counter, cleared when !RD_BEMF, enabled when RD_BEMF and BEMF[x]=1
```

定义关键的控制信号，这些信号被用于控制状态切换，PWM 输出，转速检测等对该硬件加速器非常重要的功能。具体信号如下：

```
73  //control signal DEF
74  wire PWM_cnt_done        = (PWM_cnt == shad_Prediv)? 1'b1 : 1'b0;                  //PWM count done
75  wire PWM_CMD             = (PWM_cnt <= shad_cmd_PWM)? 1'b1 : 1'b0;
76  wire PWM_LOOP            = (PWM_cnt <= shad_close_loop_pwm)? 1'b1 : 1'b0;
77  wire PWM_OUT             = (current_state == EXE_CMD)? PWM_CMD :                   //PWM output control
78                             (current_state == EXE_LOOP)? PWM_LOOP :
79                             1'b1;
80  wire CMD_empty           = ((boot_cmd[14:12] == 3'b0) | addr_cnt[10])? 1'b1 : 1'b0;  //cmd_addr overflow or data invalid
81  wire EXE_CMD_done        = (cmd_delay_cnt == shad_cmd_delay)? 1'b1 : 1'b0;          //EXE_CMD state is done
82  wire COMP_DELAY_done     = (comp_delay_cnt == shad_comp_delay)? 1'b1 : 1'b0;        //COMP_DELAY is done
83  wire RD_BEMF_done        = (BEMF_cnt == 3'd7)? 1'b1 : 1'b0;                         //RD_BEMF is done
84  wire EXE_LOOP_done       = (PWM_cnt_done)? 1'b1 : 1'b0;                             //EXE_LOOP is done
85  wire phase_overflow      = (phase_cnt == shad_ph_overflow)? 1'b1 : 1'b0;            //<phase_cnt> overflow
86  wire phase_change        = (current_phase == last_phase)? 1'b0 : 1'b1;              //phase change indicator
87  wire BEMF_HI             = (BEMF_HI_cnt <= 3'd3)? 1'b0 : 1'b1;                        //HI indicator, valid when RD_BEMF_done
```

根据当前状态，当前相序和正反转控制寄存器的值生成下一相序的信号，生成的相序会被送至 current_phase 寄存器中，以实现相序的切换，完成 EXE_CMD 状态和 EXE_LOOP 状态。下一相序控制逻辑如下：

```
wire [8:0] next_phase_cmd  = {(9){(boot_cmd[14:12] == 3'd0)}} & PH0 |                //next phase generator, only for cmd
                             {(9){(boot_cmd[14:12] == 3'd1)}} & PH1 |
                             {(9){(boot_cmd[14:12] == 3'd2)}} & PH2 |
                             {(9){(boot_cmd[14:12] == 3'd3)}} & PH3 |
                             {(9){(boot_cmd[14:12] == 3'd4)}} & PH4 |
                             {(9){(boot_cmd[14:12] == 3'd5)}} & PH5 |
                             {(9){(boot_cmd[14:12] == 3'd6)}} & PH6;

wire [8:0] next_phase_loop_cw  = {(9){((current_phase == PH1) & BEMF_HI) | ((current_phase == PH6) & BEMF_HI)}} & PH1 |
                                 {(9){((current_phase == PH2) & ~BEMF_HI) | ((current_phase == PH1) & ~BEMF_HI)}} & PH2 |
                                 {(9){((current_phase == PH3) & BEMF_HI) | ((current_phase == PH2) & BEMF_HI)}} & PH3 |
                                 {(9){((current_phase == PH4) & ~BEMF_HI) | ((current_phase == PH3) & ~BEMF_HI)}} & PH4 |
                                 {(9){((current_phase == PH5) & BEMF_HI) | ((current_phase == PH4) & BEMF_HI)}} & PH5 |
                                 {(9){((current_phase == PH6) & ~BEMF_HI) | ((current_phase == PH5) & ~BEMF_HI)}} & PH6;

wire [8:0] next_phase_loop_ccw  = {(9){((current_phase == PH6) & BEMF_HI) | ((current_phase == PH1) & BEMF_HI)}} & PH6 |
                                  {(9){((current_phase == PH5) & ~BEMF_HI) | ((current_phase == PH6) & ~BEMF_HI)}} & PH5 |
                                  {(9){((current_phase == PH4) & BEMF_HI) | ((current_phase == PH5) & BEMF_HI)}} & PH4 |
                                  {(9){((current_phase == PH3) & ~BEMF_HI) | ((current_phase == PH4) & ~BEMF_HI)}} & PH3 |
                                  {(9){((current_phase == PH2) & BEMF_HI) | ((current_phase == PH3) & BEMF_HI)}} & PH2 |
                                  {(9){((current_phase == PH1) & ~BEMF_HI) | ((current_phase == PH2) & ~BEMF_HI)}} & PH1;

wire [8:0] next_phase_loop = (shad_orientation)? next_phase_loop_ccw : next_phase_loop_cw;

wire [8:0] next_phase      = (current_state == RD_CMD)? next_phase_cmd : next_phase_loop;
```

繁琐的影子寄存器和计数器按照定义时描述的控制逻辑设计，由于行数过多，不在此进行源码展示。

控制信号输出电路，根据当前状态，当前相序和控制信号生成硬件加速器所需的输出信号，具体电路设计如下：

```
//signal output control
assign stuck = phase_overflow & (current_state == EXE_LOOP) ;
assign ph_cnt = shad_phcnt;
assign status[0] = (current_state == IDLE)? 1'b0 : 1'b1;
assign status[1] = ((current_state == COMP_DELAY) | (current_state == RD_BEMF) | (current_state == EXE_LOOP))? 1'b1 : 1'b0;
assign status[2] = ((current_state == EXE_CMD) | (current_state == EXE_LOOP))? 1'b1 : 1'b0;
assign {VL,VH} = ((current_state == EXE_CMD) | (current_state == EXE_LOOP))? {current_phase[5:3],current_phase[2:0] & {(3){PWM_OUT}}} : 6'b0;
assign boot_addr = addr_cnt[9:0];
```

使用 BLDC 驱动加速器以后，加速效果格外显著。相比使用通用定时器和中断的控制方案，可以将 CPU 空闲时间提升至约 100%，效率大幅提高。

7 辅助加速外设设计

7.1 真随机数生成器

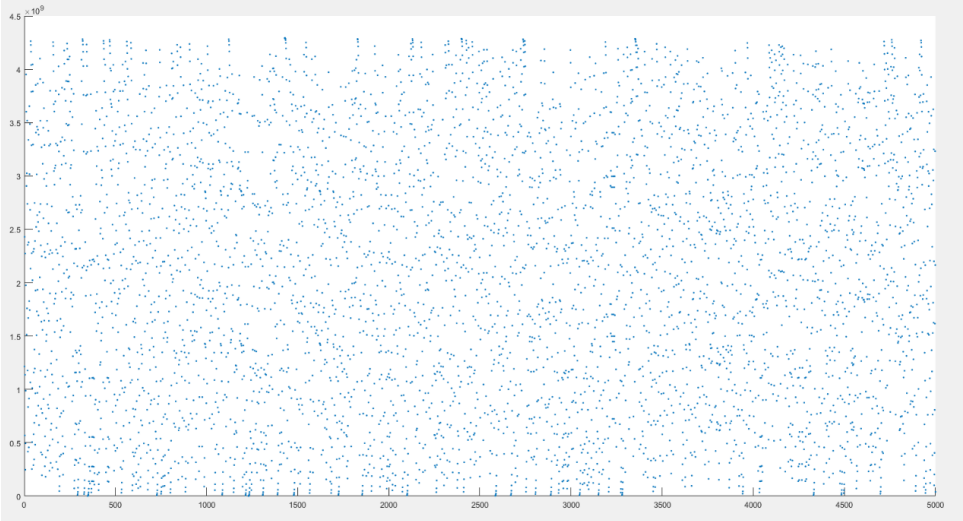
遗传算法需要经过变异运算，才能更好的求解出全局最优值。随机数发生器能够使变异过程更具随机性。随机数生成器包括伪随机数生成器和真随机数生成器，其中真随机数生成器（TRNG）更具有随机性，但设计较为复杂；伪随机数生成器可以使用 LFSR 实现，较为简单。但是，FPGA 无法像 ASIC 一样集成环形振荡器，因此只能通过对伪随机数生成器的 seed 产生方

式进行改善，使结果更具随机性。

该 TRNG 外设使用了 32 位的 m 序列生成器，可以生成除去 0 以外的所有 2^32-1 个序列。随机数生成器的 seed 通过 RTC 生成。RTC 除了能够生成 TRNG 所需的 seed 之外，还能显示当前时间，当系统独立运行时，辅助用户根据时间做出选择。

```
always@(posedge HCLK or negedge HRESETn)begin
    if(HRESETn == 1'b0) sr <= 32'b0;
    else if(wr_en_reg) sr <= HWDATA;
    else sr <= {sr[30:0],sr[31]} ^ {g & {(32){sr[31]}}};
end
```

MATLAB 的验证结果如下，随机生成了 5000 个数据点。可以看出，参数的分布具有较强的随机性：



7.2 LCD 驱动器

为了提高屏幕刷新的速度，减少 CPU 内核的占用时间，提高系统效率，我们使用了 LCD 驱动器外设，将 8080 并口的写数据和写命令操作封装成了写寄存器操作。当 CPU 发起 8080 并口读操作时，使用 GPIO 模拟 8080 并口时序；当 CPU 发起 8080 端口写操作时，将 GPIO 复用到 LCD 驱动器，复杂的时序被映射为写寄存器，由 LCD 驱动器自行完成时序控制，速度相较于单纯使用 GPIO 驱动至少提高 3 倍。

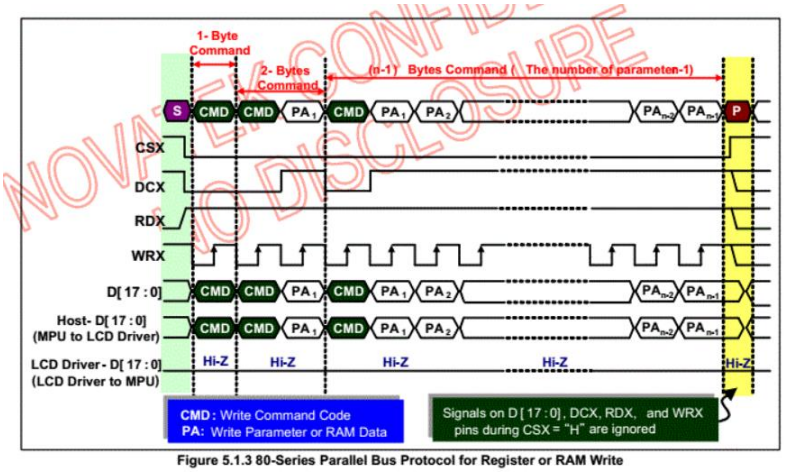


Figure 5.1.3 80-Series Parallel Bus Protocol for Register or RAM Write

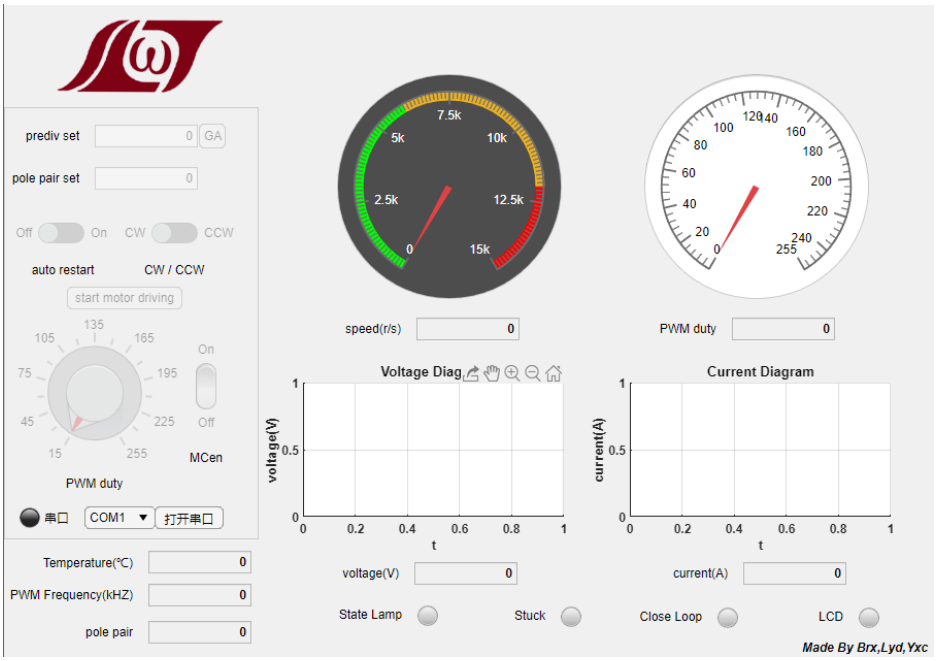
7.3 全双工 UART 和 GUI

智能 BLDC 驱动器可以独立运行，也可以嵌入到其他系统中，作为子系统运行。开机时，控制器会要求用户做出选择，无论是独立运行还是作为子系统运行，UART 都会向外发送运行数据。但是，独立运行时，UART 不会接收来自外部的指令；作为子系统运行时，行列键盘和拨码开关同样不会起作用。这样便能防止控制命令冲突。

UART 采用全双工的工作模式，可以同时进行收发。为了减少 CPU 的等待时间，该 UART 发送端装置了 64 深度的 FIFO，CPU 仅需向 UART 发送端对应的寄存器中写入一串数据，UART 便能够自行将这些数据发送完毕。UART 的接收端未使用 FIFO，而是使用了中断，CPU 在中断服务子程序中接收数据，判断控制命令字段是否发送完毕，增加了系统设计的灵活性。

```
67  wire clk_uart_tx,clk_uart_rx;
68  wire bps_en_rx,bps_en_tx;
69
70  clkuart_pwm TX_pwm(
71          .BPS_PARA(BPS_PARA),
72          .clk(PCLK),
73          .RSTn(PRESETn),
74          .clk_uart(clk_uart_tx),
75          .bps_en(bps_en_tx)
76  );
77
78  clkuart_pwm RX_pwm(
79          .BPS_PARA(BPS_PARA),
80          .clk(PCLK),
81          .RSTn(PRESETn),
82          .clk_uart(clk_uart_rx),
83          .bps_en(bps_en_rx)
84  );
```

当控制器作为 PC 机的子系统（只要控制字段一致，也可以作为其他设备的子系统）运行时，可以通过 GUI 设定全部的参数，GUI 可以将智能 BLDC 驱动系统返回的参数进行图形化地直观显示，GUI 的具体界面如下：



GUI 界面可以分为控制面板和状态监测界面。

在控制面板中，用户可以根据 PC 机与 BLDC 连接端口的串口号进行配置，然后对系统进行初始化配置。初始化包括预分频器数值的设置（若选择 GA 则使用遗传算法进行配置），pole pair 数量的配置，是否开启堵转自启动和确定 BLDC 的转向。初始化完成后，可通过控制面板上的 MCen 开关和 PWM duty 旋钮对电机的开关与转速进行控制。

在状态监测界面中，可以实时地显示当前系统的温度、PWM 频率和 pole pair 的实时数值，转速和 PWM duty 的仪表盘可以直观地显示当前电机的转速。电压和电流图表能够实时反应系统电流和电压的变化，四个状态灯能显示当前系统的工作状态，用户可以更加这些数据清楚地判断系统的运行情况并及时做出反应。

该 GUI 界面使用 MATLAB 的 app designer 进行开发，利用 serial 函数通过 UART 串口实现 PC 机与 BLDC 智能驱动系统之间的通信，通过串口的回调函数在状态监测界面进行显示。具体的代码实现如下：

```
properties (Access = public)
    COM;     % 端口号
    s  ;     %端口设置句柄
    RX_once;%一次接收的数据
    RX_Date;%接收的所以数据
    is_open;%是否打开串口标志位

    vol_globle; % 电压中间量
    cur_globle  % Description
    t_globle    % Description
end
```

串口的定义以及全局变量的定义

```
function pbOpenSerialValueChanged(app, event)
    app.COM=get(app.ppCOM,'Value');
    if strcmp(get(app.pbOpenSerial,'Text'),'打开串口')
        try
            app.s=serial(app.COM);
            app.s.BaudRate=38400;%设置波特率
            app.s.DataBits=8;%设置数据长度
            app.s.StopBits=1;%设置停止位长度
            app.s.InputBufferSize=1024;%设置输入缓冲区大小为1M
            app.s.BytesAvailableFcnMode='terminator';  %串口事件回调设置
            app.s.Terminator='!';
%             app.s.BytesAvailableFcnCount=10; %输入缓冲区存在10个字节触发回调函数
            app.s.BytesAvailableFcn={@app.EveBytesAvailableFcn};%回调函数的指定
            fopen(app.s);%打开串口
            app.is_open=1;
            app.pbOpenSerial.Text='关闭串口';
            app.Lamp.Color=[0 1 0];
            %safety
            app.predivsetEditField.Enable = 1;
            app.GAButton.Enable =1;
            app.polepairsetEditField.Enable = 1;
            app.autorestartSwitch.Enable = 1;
            app.CWCCWSwitch.Enable = 1;
            app.startmotordrivingButton.Enable = 1;
            %
        catch err
            msgbox('打开失败');
        end
```

对串口中各个参数的配置以及回调函数的指定

```
%state lamp
if state_lamp(5)==1
    app.StateLamp.Color = 'r';
elseif state_lamp(1)==2
    app.StateLamp.Color = 'g';
else
    app.StateLamp.Color = [0.8 0.8 0.8];
end
%stuck lamp
if state_lamp(3)==1
    app.StuckLamp.Color = 'r';
else
    app.StuckLamp.Color = [0.8 0.8 0.8];
end
%close loop lamp
if state_lamp(2)==1
    app.CloseLoopLamp.Color = 'g';
else
    app.CloseLoopLamp.Color = [0.8 0.8 0.8];
end
%lcd lamp
if state_lamp(4)==1
    app.LCDLamp.Color = 'g';
else
    app.LCDLamp.Color = [0.8 0.8 0.8];
end
%voltage diagram & current diagram
app.vol_globle(app.cntEditField.Value + 1)=voltage;
app.cur_globle(app.cntEditField.Value + 1)=current;
app.t_globle(app.cntEditField.Value +1)=app.tEditField.Value;
if app.cntEditField.Value == 19
hold(app.UIAxes,'on')
hold(app.UIAxes2,'on')
%   t=[app.tEditField.Value-1 (app.tEditField.Value)];
%   vol_past = app.voltageVEditField.Value;
%   cur_past = app.currentAEditField.Value;
%   vol=[vol_past voltage];
%   cur=[cur_past current];
plot(app.UIAxes,[(app.t_globle(1)-0.05) app.t_globle],[app.vol_pastEditField.Value app.vol_globle],'LineWidth',2)
plot(app.UIAxes2,[(app.t_globle(1)-0.05) app.t_globle],[app.cur_pastEditField.Value app.cur_globle],'LineWidth',2)
axis(app.UIAxes,[app.tEditField.Value-5 app.tEditField.Value+1 voltage-1 voltage+1])
```

串口回调函数中对监测界面中各个参数的配置

7.4 通用计时器

该通用定时器有 8 个通道，具有输入捕获，PWM，带死区控制的 PWM，定时器功能。每一个通道均可以独立的配置，互不影响。并且，每一个通道都可以配置为上述 4 种功能。

为了节约硬件资源，每一个通道仅具有一个 16 位的主计数器和一个 8 位的辅助计数器，通过模式寄存器来选择计数方式。由于不需要像智能 BLDC 驱动加速器那样具备极高的可靠性，故没有使用影子寄存器，如此一来减少了约 1/3 的寄存器。此外，每 4 个通道共享 1 个 16 位的预分频器，在保证系统灵活性的条件下，减少了寄存器和查找表的数量。

输入捕获功能可以测定正脉宽或负脉宽的时长，并且具有增强稳定性的毛刺滤除功能，只有信号的高电平时长或低电平时长大于设定的最低采样时长时，信号才会被记录，增加了系统的稳定性。此外，通用计时器还可以在捕获完成后产生中断。

PWM 功能为普通的 PWM 输出，可以编程设定 PWM 的频率和占空比。

带死区控制的 PWM 功能可以输出具有死区的 PWM 信号，防止半桥功率管同时导通而烧毁。当通道配置为该功能时，会占用 GPIOA 的 2 个端口。例如，当通用定时器的通道 3 被使能为该功能时，会占用 GPIOA 的 Pin[3]和 Pin[19]。死区时间可以通过 Deadtime 寄存器配置。该功能可以用来产生具有带载能力的直流电压输出。电压值约为 Vcc*PWM[x]/256。

定时器功能和 CMSDK 中的定时器差别不大，使用方法类似于 systick 计时器，适合在内核定时器已经被占用的情况下使用。

```
module APB_PWM (
    input  wire          PCLK,
    input  wire          PRESETn,
    input  wire          PSEL,
    input  wire   [31:0] PADDR,
    input  wire          PENABLE,
    input  wire          PWRITE,
    input  wire   [31:0] PWDATA,
    output        [31:0] PRDATA,

    output wire   [7:0]  SIGNAL_OUTPUT,
    input  wire   [7:0]  SIGNAL_INPUT,
    output wire   [15:0] ctrl,
    output wire   [7:0]  PWM_out,
    output wire          Timer_Init,
    output wire          Input_Init
);
```

顶层模块

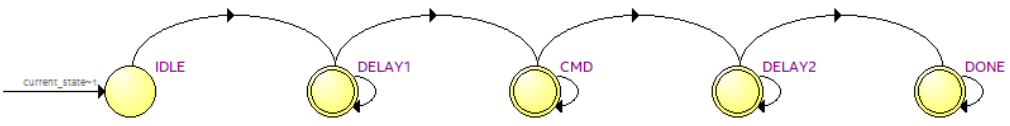
PWM	Mode	0x40010100	32	R/W	0000:捕获高电平时间；0001捕获低电平时间；0010：
	Prediv1	0x40010104	16	R/W	预分频器1
	Prediv2	0x40010108	16	R/W	预分频器2
	Deadtime	0x4001010C	16	R/W	死区时间，单位为CLK周期
	PWM0	0x40010110	8	R/W	0-255可调
	PWM1	0x40010114	8	R/W	0-255可调
	PWM2	0x40010118	8	R/W	0-255可调
	PWM3	0x4001011C	8	R/W	0-255可调
	PWM4	0x40010120	8	R/W	0-255可调
	PWM5	0x40010124	8	R/W	0-255可调
	PWM6	0x40010128	8	R/W	0-255可调
	PWM7	0x4001012C	8	R/W	0-255可调
	Jitter	0x40010130	16	R/W	输入抖动消除，单位为CLK周期
	Timer_Switch	0x40010134	8	R/W	独立开关
	PWM_Switch	0x40010138	8	R/W	独立开关
	Input_Switch	0x4001013C	8	R/W	独立开关
	Timer_InitSet	0x40010140	8	R/W	中断设置
	Input_InitSet	0x40010144	8	R/W	中断设置
	Tim0_LD	0x40010148	16	R/W	定时器装载
	Tim1_LD	0x4001014C	16	R/W	定时器装载
	Tim2_LD	0x40010150	16	R/W	定时器装载
	Tim3_LD	0x40010154	16	R/W	定时器装载
	Tim4_LD	0x40010158	16	R/W	定时器装载
	Tim5_LD	0x4001015C	16	R/W	定时器装载
	Tim6_LD	0x40010160	16	R/W	定时器装载
	Tim7_LD	0x40010164	16	R/W	定时器装载
	Input0	0x40010168	17	R	输入捕获计数器
	Input1	0x4001016C	17	R	输入捕获计数器
	Input2	0x40010170	17	R	输入捕获计数器
	Input3	0x40010174	17	R	输入捕获计数器
	Input4	0x40010178	17	R	输入捕获计数器
	Input5	0x4001017C	17	R	输入捕获计数器
	Input6	0x40010180	17	R	输入捕获计数器
	Input7	0x40010184	17	R	输入捕获计数器
	Timer0_CNT	0x40010188	16	R	定时器读取
	Timer1_CNT	0x4001018C	16	R	定时器读取
	Timer2_CNT	0x40010190	16	R	定时器读取
	Timer3_CNT	0x40010194	16	R	定时器读取
	Timer4_CNT	0x40010198	16	R	定时器读取
	Timer5_CNT	0x4001019C	16	R	定时器读取
	Timer6_CNT	0x400101A0	16	R	定时器读取
	Timer7_CNT	0x400101A4	16	R	定时器读取
	Timer_InitStatus	0x400101A8	8	R	中断标志位，读后清除
	Input_InitStatus	0x400101AC	8	R	中断标志位，读后清除

通用计时器的编程模型

7.5 模数转换器 ADC

为了获取输入电压和电流信号，使用了双通道 ADC 来采集数据。ADC 采用开发板板载的 MAX5864 数据转换器，该转换器具有双通道 8 位 ADC 和双通道 10 位 DAC，最高支持 22MSa 的采样率。该数据转换器使用 SPI 接口控制，驱动较为简单。

上电后，先经过 50us 的延时，然后进入模式配置状态（CMD），配置模式寄存器后，再经过 50us，使得数据转换器的模拟部分正常运转，初始化工作完成。初始化工作完成后，init_done 信号被置高，指示 CPU 初始化已经完成，可以开始读取有效数据。



ADC 的状态转移图

在硬件电路部分，电压采集使用电阻分压将电机驱动所需电压降至 ADC

的量化范围以内；对于电流，则先用霍尔传感器将电流转化成电压信号，通过运算放大器放大后再进行采集。

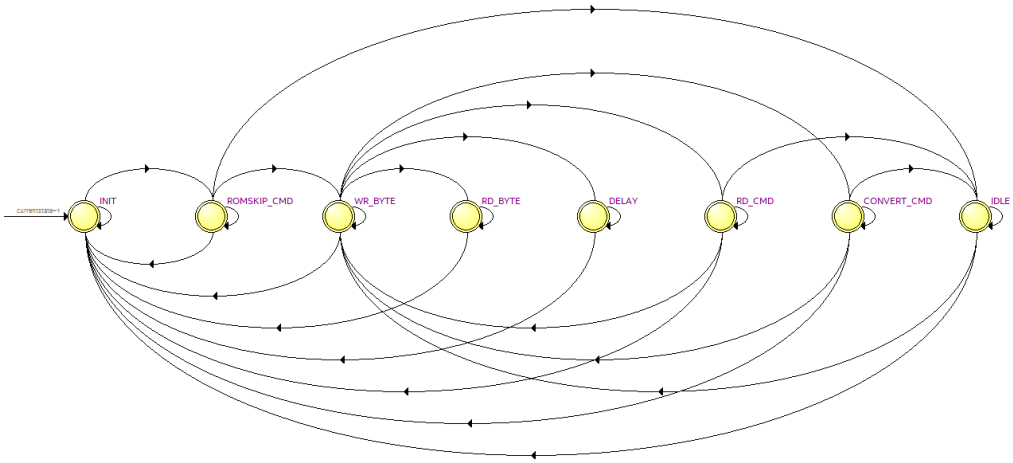
```
module ADC(
    //AHB signal
    input         HCLK,
    input         HRESETn,
    input         HSEL,
    input         HREADY,
    input [1:0]   HTRANS,
    input [31:0]  HADDR,
    output        HREADYOUT,
    output [1:0]  HRESP,
    output [31:0] HRDATA,

    //MAX5864 SC signal
    output        CSn,
    output        SCLK,
    output        DIN,

    //MAX5864 DATA signal
    output        ADC_clk,
    input [7:0]   AD,
    output [9:0]  DA
);
```

顶层模块

7.6 温度传感器

温度传感器能以 500ms 间隔采集定子的温度，将其回传到专用的接口电路中处理，最终以有符号数的形式回传到 CPU。考虑到通用性和可靠性，本次使用了常见的 ds18b20 单总线温度传感器，该传感器仅有一条总线，且频率较低，不容易出现干扰的问题。将温度传感器配置为最高精度，共有 11 位有效数字。

为了减轻 CPU 的负担，使用硬件电路实现 ds18b20 传感器的读取。该外设使用有限状态机控制，总共具有 8 个状态。



ds18b20 的状态转移图

其中，INIT 状态是初始状态，复位时，控制器回到该状态；ROMSKIP_CMD，CONVERT_CMD 和 RD_CMD 状态用于控制向 ds18b20 发送的指令；DELAY 状态用于进行 500ms 延时，使得该器件有足够的时间进行数据采集；WR_BYTE 用于向 ds18b20 写入控制指令；RD_BYTE 状态可以从 ds18b20 读取数据；IDLE 状态表示一次温度采集已经完成，采集完的温度会进行数值转换，然后上传到寄存器中供 CPU 读取。

```
module ds18b20(input sys_clk,
               input rst_n,
               inout dq,
               output reg sign,
               output reg [10:0] data,
               output reg error_en);

parameter rom = 8'hcc;  //控制命令
parameter convert = 8'h44;
parameter read = 8'hbe;

parameter INIT = 8'b00000001;  //定义状态
parameter ROMSKIP_CMD = 8'b00000010;  //0号指令和2号指令
parameter CONVERT_CMD = 8'b00000100;  //1号指令
parameter DELAY = 8'b00001000;
parameter RD_CMD = 8'b00010000;  //3号指令
parameter RD_BYTE = 8'b00100000;
parameter WR_BYTE = 8'b01000000;
parameter IDLE = 8'b10000000;

reg dq_out;  //通过给寄存器变量赋值1，0，Z来决定输出还是输入
assign dq = dq_out;

reg [3:0] div_cnt1;  //分频定义
reg clk_us;

reg [7:0] nextstate;  //定义状态机
reg [7:0] currentstate;

reg [18:0] us_counter;  //定义微秒计数器和毫秒计数器

reg us_counter_clear;  //定义控制信号
reg init_done;
reg [3:0] flow_cnt;  //过程计数器
reg [7:0] write;
reg state_done;
reg [2:0] cmd_cnt;  //记录执行的命令序号，进行状态跳转
reg [3:0] write_cnt;  //写入位数计数器,设置9位，最后一位控制状态跳转
reg [4:0] read_cnt;

reg [15:0] rd_temp;  //数据暂存寄存器
reg [15:0] rd_data;  //原始补码数据
//reg [10:0] data_temp;  //对转成原码的数据
```

ds18b20 驱动器的信号定义

由于该芯片为单总线芯片，必须有 3 态门才能够实现正常的数据读写，因此控制逻辑较为复杂。但是，通信协议非常固定，可以有效减少逻辑资源的占用量。为了使得系统更加稳定，该外设增加了容错设计，当控制器检测到 ds18b20 离线时，会将 error 信号拉高，告知 CPU 温度传感器无法正常工作，从而另电机停转，防止可能的过热带来的严重损失。

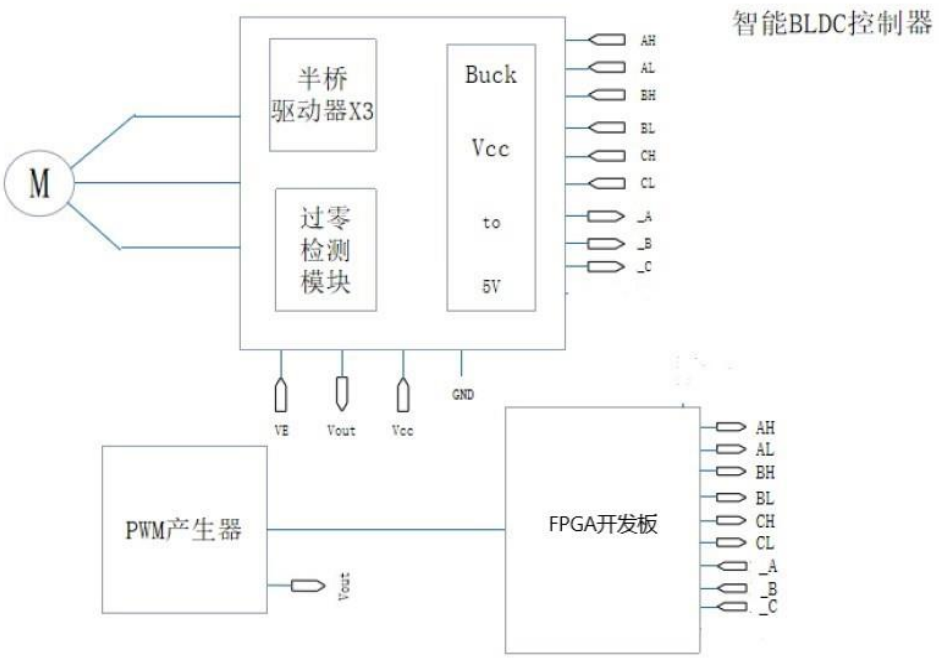
```
module APBTemp(
    //APB interface
    input PCLK,
    input PRESETn,
    input PSEL,
    input PENABLE,
    output [31:0] PRDATA,
    //ds18b20
    input dq,
    output dq_out,
    output dq_ctrl
);
```

APB 接口 ds18b20 驱动器的顶层模块

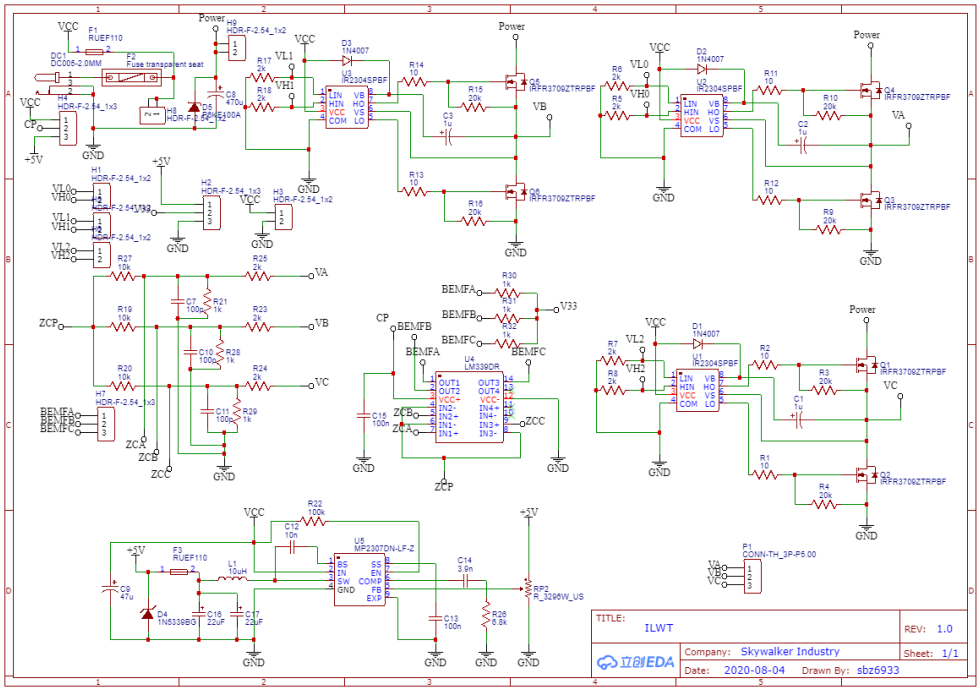
8 硬件原理图

硬件部分主要包括，比较器，555 时基电路，MP2307 开关电源，3 对半桥驱动电路，6 个 NMOS 功率管，过零检测电路，电压与电流采集电路。

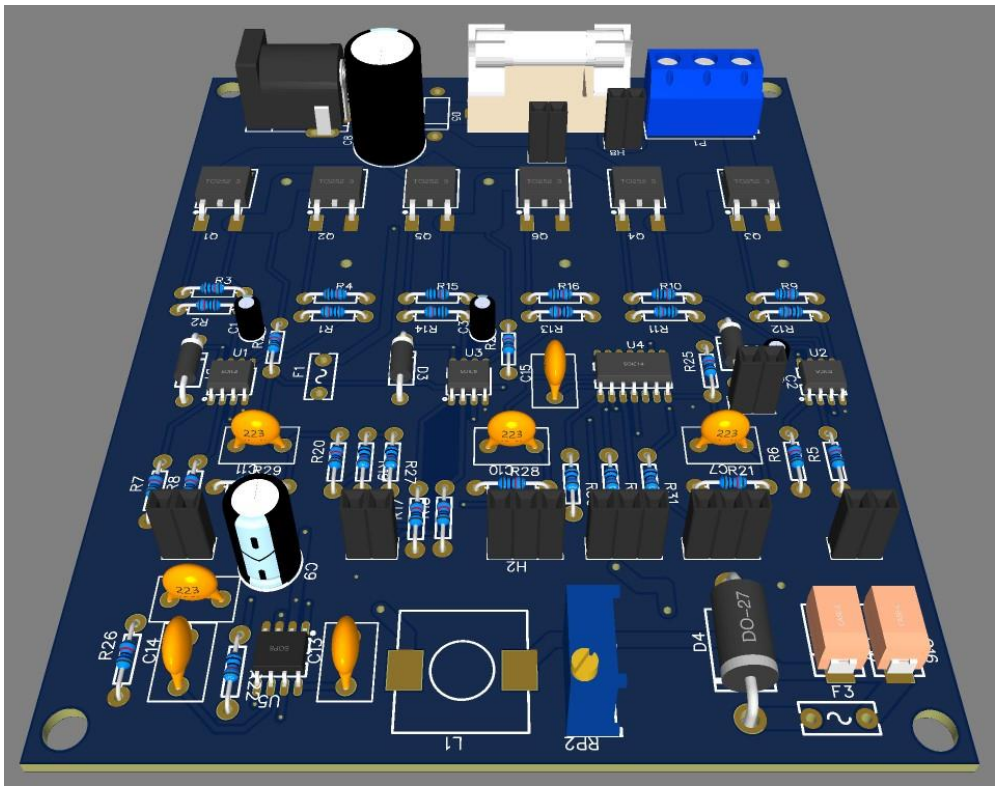
硬件总体框图：



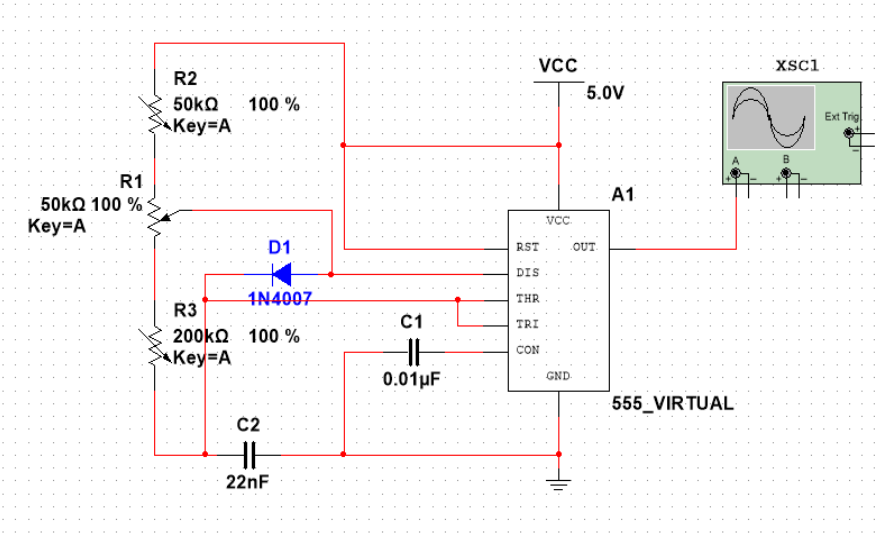
硬件原理图（开关电源，半桥，过零检测）：



硬件 3D 版图（开关电源，半桥，过零检测）：

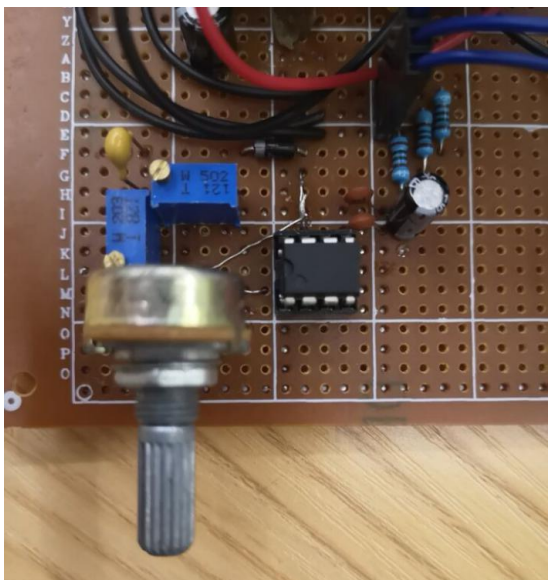


555 时基电路原理图：

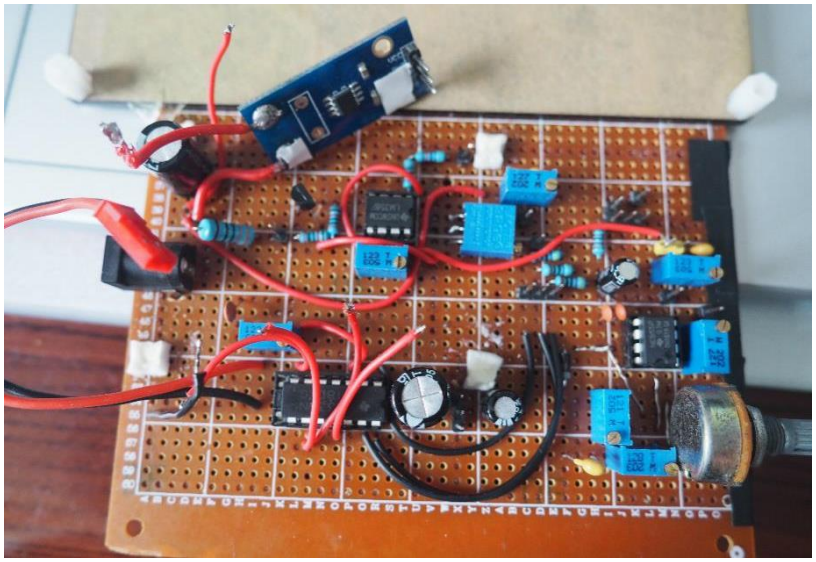


开关电源，过零检测电路（电阻分压网络），电机驱动电路硬件实物图：

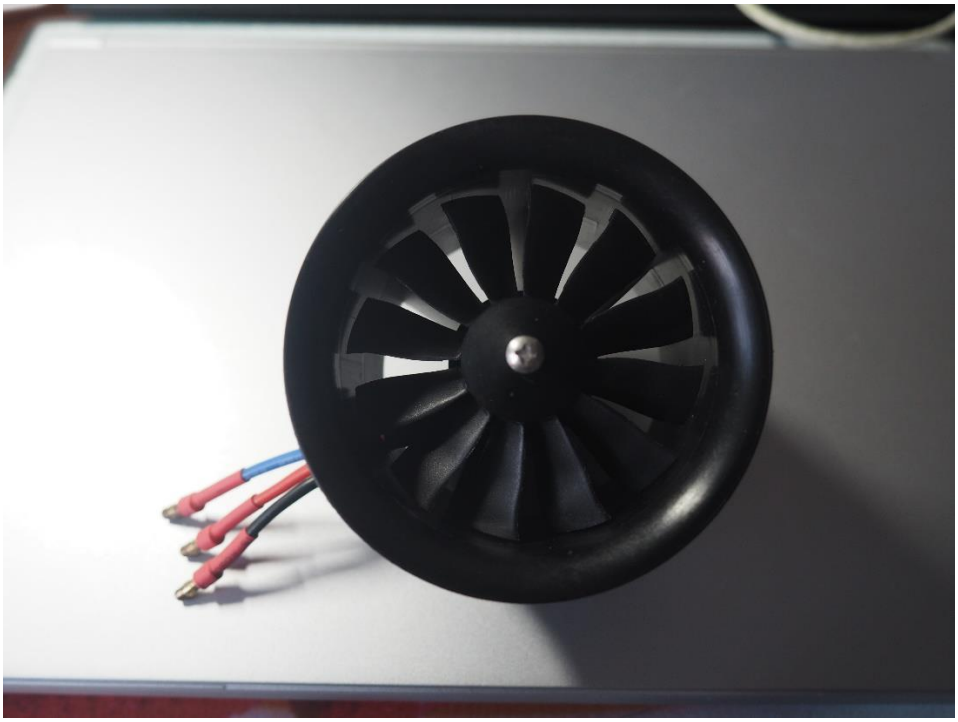
555 时基电路实物图：



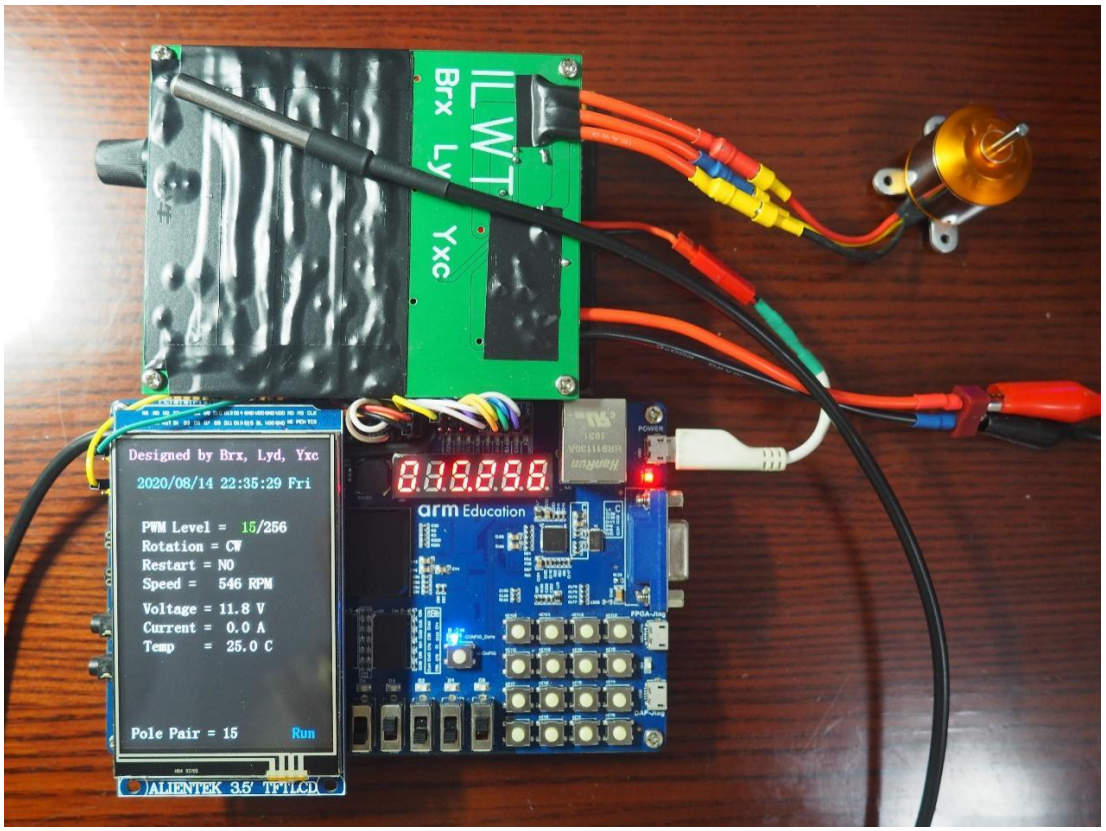
电流采样模块实物图（早期验证系统）：



BLDC 驱动的涵道航空发动机：

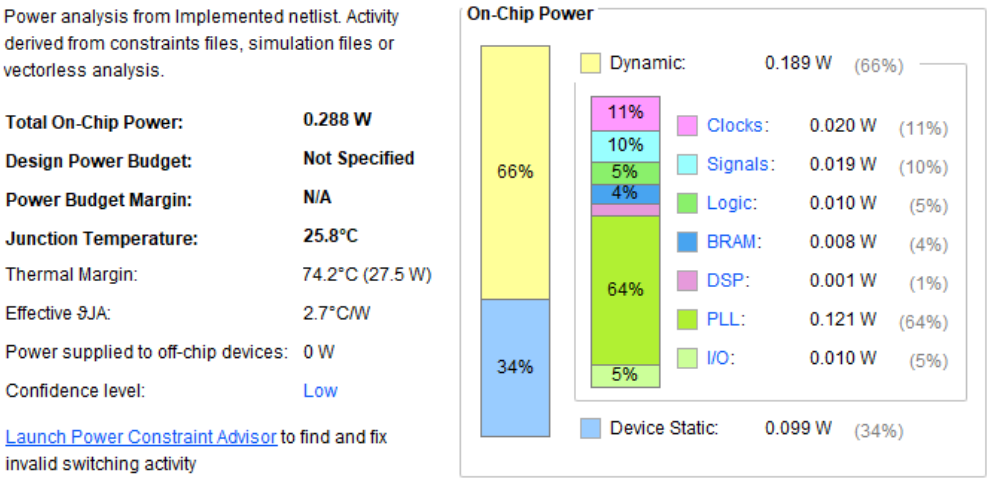


9 成果展示



智能 BLDC 驱动器整体展示

我们对高速外设和低速外设划分了不同的时钟域。CPU 与 AHB 高速外设运行在 48MHz 时钟下，而 APB 低速外设运行在 24MHz 时钟下，达到了节能的效果。为保证系统稳定运行，进行了严格的时序约束，功耗分析如下图所示：



SoC 功耗分析

为了减少逻辑资源占用，我们未使用 AXI 总线的 Xilinx 软核，而是使用了 arm Design Start Eval 提供的源码，配合自制外设以减少逻辑资源占用。除去 CPU 内核，仲裁器和 RAM interface 以外，所有的外设全部由我们自主设计并验证。

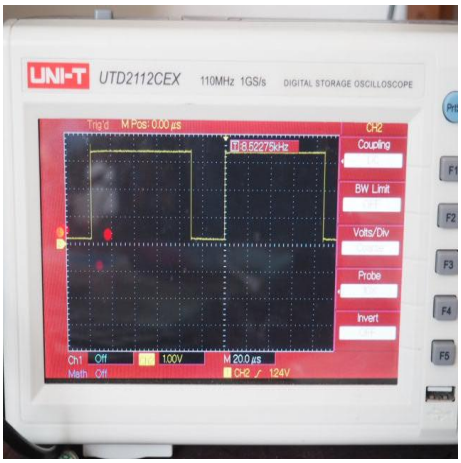
我们还对管脚进行了合理分配，通过 GPIO 复用的方式将外设对应的接口引出。如智能电机驱动加速器，LCD 驱动器，通用定时器等，既保证了系统功能的完备性，也减少了对 IO 口的需求。实际应用时，可以将 FPGA 从 xc7a75t 降级到 xc7a35t 以降低成本。资源占用如下表所示：

Resource	Utilization	Available	Utilization %
LUT	18655	47200	39.52
LUTRAM	12	19000	0.06
FF	8365	94400	8.86
BRAM	65	105	61.90
DSP	3	180	1.67
IO	100	285	35.09
BUFG	6	32	18.75
PLL	1	6	16.67

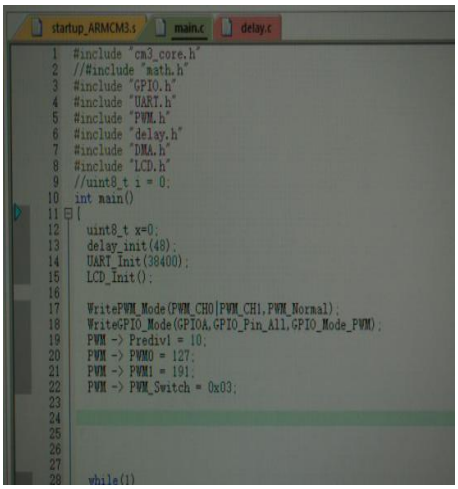
xc7a75t 资源消耗

未来，我们将对智能 BLDC 驱动系统进行扩展，加入智能 FOC 功能从而实现转矩控制和零速启动，为用户提供更多的选择。此外，进一步增强系统稳定性，使用 ECC 算法防止内存错误导致系统被迫重启。

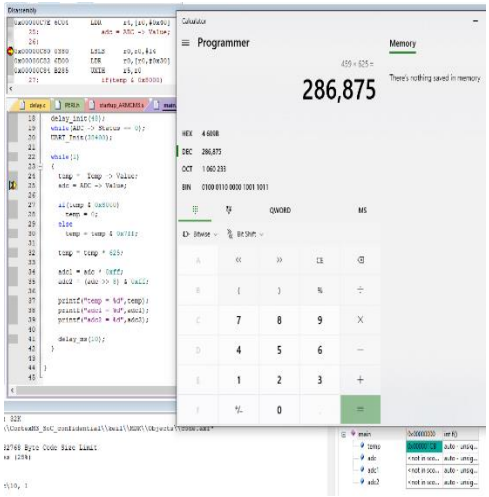
LCD 上板验证结果



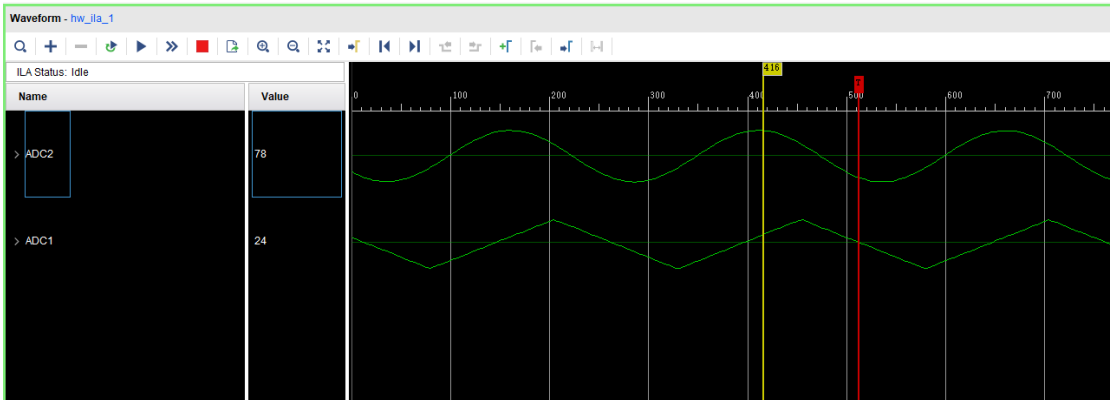
通用定时器上板验证结果



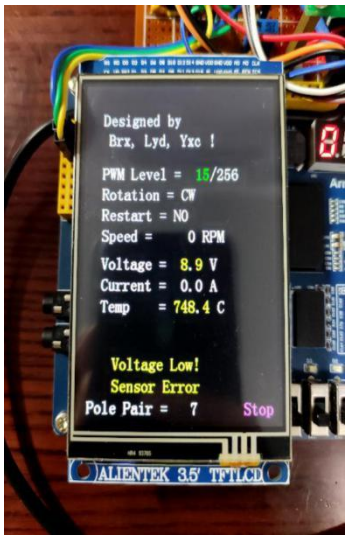
通用定时器上板验证程序



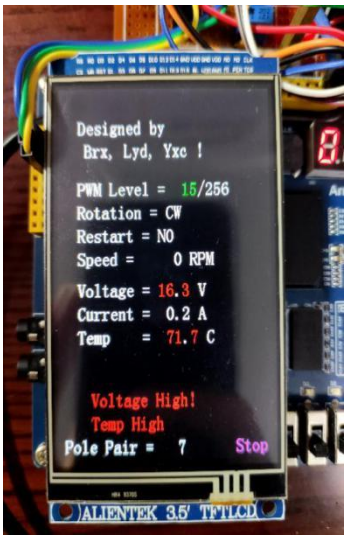
温度传感器上板验证结果



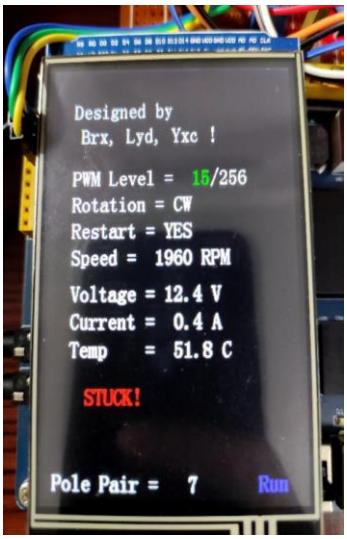
双通道 ADC 的验证结果



欠压与温度传感器错误警告

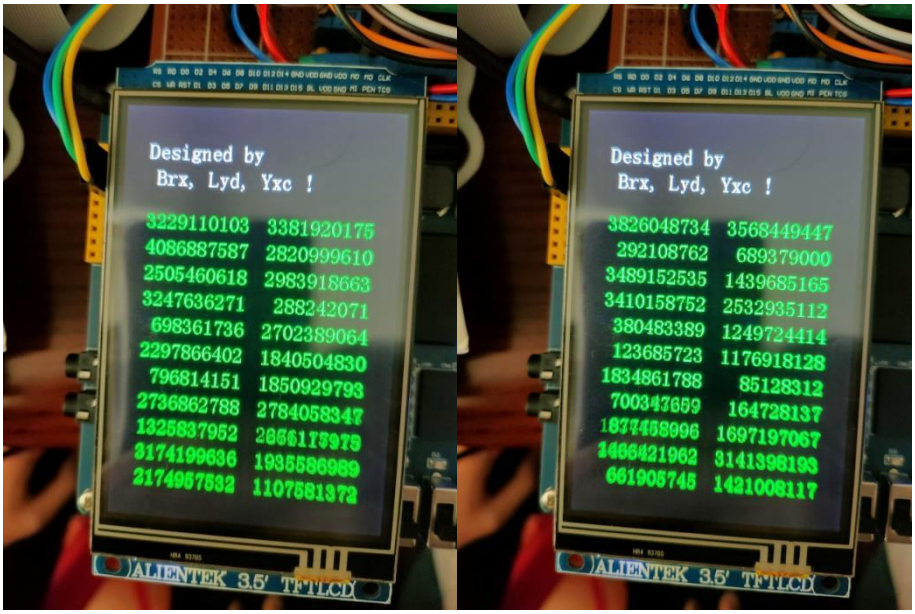


过电压警告与高温警告

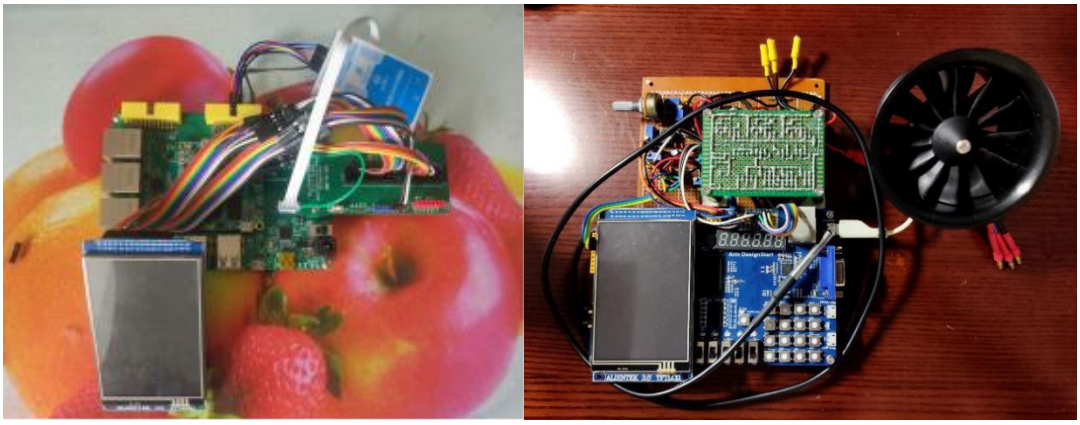


堵转警告

电机运行状态图



随机数生成器上板验证结果



前期的测试系统

10 参考文献

[1] Matthew Scarpino [MOTORS for MAKERS :A Guide to Steppers,Servos,and Other Electrical Machines]. America:Que Publishing,2015-12-10

[2] Joseph Yiu [System-on-Chip Design with Arm Cortex-M Processors].ISBN: 978-1-911531-19-7

[3] Xilinx [Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2]

[4] arm [ARM® Cortex®-M3 DesignStart™ Eval Customization Guide]

[5] arm [ARMv7-M Architecture Reference Manual]

[6] arm [AMBA specification]