- 嵌入式系统有多种表现形式，包括计算机 MCU、SOC 片上系统、SOPC 片上系统、GPU 和 FPGA 等。
  - MCU(微控制器): 是最基本也是最常见的嵌入式系统形式,是**集成了 CPU、ROM、RAM、IO 口、定时器、中断控制器等组件的单一芯片**。MCU 广泛用于电器电子产品的控制。
  - SoC(系统片上芯片): 集成了多个功能模块,可实现复杂计算任务。与 MCU 相比,SoC 集成度更高,性能更强,但也更昂贵。典型的 SoC 如手机 SoC、车用 SoC 等。
  - FPGA(现场可编程门阵列): 具有高度可编程性和可配置性的半定制芯片。FPGA 可在上电后通过配置逻辑电路来实现特定功能，具有非常高的灵活性和可重构性，但成本较高。
  - ASIC(专用集成电路): 专门为某个应用设计的集成电路，性能高、功耗低、成本高，适合大规模生产。
  - GPP(通用处理器): 通用处理器是一种能够执行通用计算任务的处理器，比如 Intel 的 x86、ARM 的 Cortex-A 等。通用处理器不仅可以运行各种操作系统，还可以运行各种应用程序，因此具有非常广泛的适用范围。
  - GPU(图形处理器): 专门为图形处理而设计的处理器，能够高速地进行并行计算，在游戏、视频处理、科学计算等领域有广泛应用。
  - SoPC(System on Programmable Chip): 是**构建在 FPGA 基础上**的单芯片系统,具有高自由度和可配置性。SoPC 通过用户自定义逻辑将 IP 模块集成在一起，可以在 FPGA 上实现各种数字电路和嵌入式系统。

- 不同的嵌入式系统表现形式适用于不同的应用场景，具有各自的**优缺点**。
  - MCU（微控制器）：集成度低，但成本低廉，适合于对资源要求不高的低功耗应用，例如家电、智能家居、玩具等。
  - SoC（系统片上芯片）：集成度高，性能强，适合于高性能、多功能、复杂计算的应用场景，例如智能手机、车载电子、智能摄像头等。成本较高，不适合于低成本、低功耗的应用场景。
  - FPGA（现场可编程门阵列）：具有高度可编程性和可重构性，适合于需要快速定制、快速原型设计的应用场景，例如数字信号处理、通信、高速计算等。
  - ASIC（专用集成电路）：适合于需要高性能、低功耗、大批量生产的应用场景，例如芯片卡、网络交换机、路由器等。需要长时间的设计和生产周期，不适合于快速原型设计和快速定制的应用场景。
  - GPP（通用处理器）：适合于需要高性能、通用性、可编程性的应用场景，例如桌面计算机、服务器、移动设备等。功耗较高，不适合于低功耗应用场景。
  - GPU（图形处理器）：适合于需要高性能图形处理的应用场景，例如游戏、虚拟现实、深度学习等。专门用于图形处理，不适合于通用计算任务。
  - SoPC（基于 FPGA 的系统芯片）：具有高自由度和可配置性，适合于需要快速原型设计、快速定制的应用场景，例如数字信号处理、嵌入式系统设计等。需要设计和配置的时间和成本较高，不适合于低成本应用场景。

- 计算机是软件和硬件的结合体制，软件和硬件可以相互转化。历史上第一台计算机全由硬件构成，现在绝大部分场景下，计算机都是软硬件综合起来的。

- **对于实时性要求很强的末端控制系统等特定场景，也可以完全用 FPGA 来实现，全硬件实现软件的功能。**

- **FPGA 具有可编程性**，可以根据应用场景的需要进行**定制化开发**，从而达到更高的性能和可靠性。

- ARM 芯片是一种广泛应用于嵌入式系统中的处理器架构

  - 国内有很多公司在做 ARM 芯片，包括**全志科技、瑞芯微**等。
  - 联发科技、展讯通信、华为海思也在做

- GPU 也是嵌入式系统中常用的处理器

  - 国内的**景嘉微**是一个比较好的 GPU 芯片厂商。

- FPGA 是一种可编程硬件

  - 国外的主要厂商包括**Xilinx、Altera、Lattice、Microsemi**
  - 国内的**复旦微**是一家 FPGA 芯片厂商。

- 单片机 51 系列是一种经典的微控制器芯片

  - 单片机 51 系列是一种经典的微控制器芯片，是由**英特尔**推出的一款 8 位微控制器，应用广泛，具有低功耗、易于编程、成本低等特点。
  - 国内有很多厂家在做，其中**宏晶电子和 STC**是比较有名的厂商。

- 国内的嵌入式系统产业已经比较成熟，有很多能力强、产品质量好的厂家能够提供各种类型的芯片产品，与国外的产品相当甚至更好。

- ARM 是一种处理器架构，也是一种技术的总称，同时也是一家公司的名字。

- ARM 公司本身并不生产 SOC 或 MCU，而是向全球 100 多家半导体制造商出售 IP CORE 或 ISA（指令集架构）。

- ARM 芯片最开始的命名规则为 ARM、ARM7、ARM9、ARM11，后来发展出了基于 Cortex 架构的三个系列：Cortex-A、Cortex-R 和 Cortex-M。

  - Cortex-A 系列适用于**高性能**场景，如手机等；
  - Cortex-R 系列适用于**实时性**要求较高的场景；
  - Cortex-M 系列适用于**低功耗**、成本较低的场景。

- STM32 是**意法半导体公司**购买了 ARM 的 IP CORE 生产的 MCU，属于**Cortex-M**系列。

- 对于实时性要求较高的系统，我们需要对硬件有基本的了解。

  - 快速处理器：选择高性能的处理器可以提高系统的响应速度和处理能力。例如，基于**ARM Cortex-R**系列的微控制器，能够满足严格的实时性要求，具有硬实时（Hard Real-Time）设计，提供高性能、可靠性和可扩展性，被广泛应用于汽车、工业控制、医疗设备等领域。

  实时操作系统（RTOS）：采用实时操作系统可以提高系统的响应速度和稳定性。实时操作系统可以提供任务调度、中断处理、内存管理等功能，以确保系统的实时性和可靠性。例如，**FreeRTOS 和 uC/OS**是一款轻量级的实时操作系统，适用于各种嵌入式系统，能够提供快速、稳定、可靠的实时性能。

快速存储器：选择快速的存储器可以提高系统的数据处理速度和数据传输速度。例如，使用 SRAM 可以比使用 DRAM 更快地处理数据，SRAM 的读写速度比 DRAM 快得多，而且不需要刷新操作。使用高速的闪存可以提高数据传输速度，例如，**UFS（Universal Flash Storage）**是一种高速闪存，具有更高的传输速度和更低的延迟，适用于高速数据传输的场景。

快速总线：选择高速的总线可以提高系统的数据传输速度和响应速度。例如，使用 USB 3.0 可以比使用 USB 2.0 更快地传输数据，USB 3.0 的传输速度可以达到 5Gbps，是 USB 2.0 的 10 倍以上。使用**PCI Express**可以提供更高的带宽和更快的数据传输速度，PCI Express 的传输速度可以达到 32Gbps，是 PCI 的 4 倍以上。

实时监测和控制电路：使用实时监测和控制电路可以提高系统的实时性和稳定性。例如，使用实时监测电路可以实时监测电压、温度、湿度等参数，以保证系统的稳定性和可靠性。使用实时控制电路可以实时控制电流、电压、功率等参数，以确保系统的安全性和可靠性。例如，**电源管理芯片（PMIC）**可以实时监测和控制电源输出，以保证电源的稳定性和可靠性。

- **ARM 和 x86 是两种不同的处理器架构，硬件架构上有很多不同之处。**

  - 架构：ARM 架构由英国公司**ARM Holdings**开发，而 x86 架构由**英特尔（Intel）**和**AMD（Advanced Micro Devices）**等公司开发。
  - 指令集：ARM 和 x86 采用不同的指令集。**ARM 处理器使用 RISC**（Reduced Instruction Set Computing）指令集，而**x86 处理器使用 CISC**（Complex Instruction Set Computing）指令集。与 CISC 指令集相比，RISC 指令集更加简单，指令长度一般为固定长度，执行速度更快。
  - 思想：**ARM 架构的设计思想是低功耗、高效率和可移植性**，因此它广泛应用于**移动设备和嵌入式系统**。而**x86 架构的设计思想则是高性能、多功能和兼容性**，因此它主要用于**个人电脑和服务器**等领域。
  - 寄存器：**ARM 处理器拥有 16 个通用寄存器**，而**x86 处理器则拥有 8 个通用寄存器**。此外，ARM 处理器还有一些专用寄存器，如程序计数器（PC）和状态寄存器（CPSR）等。
  - 栈：**自由选择：Arm（但个别指令仅支持向低）；而在 x86 处理器中，栈则是从高向低地址增长的。**

- ![image-20230622165156514](C:/Users/29185/AppData/Roaming/Typora/typora-user-images/image-20230622165156514.png)

- ARM 有七种工作模式，分别是**用户模式、系统模式、监管模式、服务模式、中断模式、异常模式和未定义模式**。每种模式有不同的特点和限制，需要清楚了解。

  - 用户模式（User Mode）：用户模式是处理器最常使用的模式，**它只能访问受保护的指令和数据，不能直接访问操作系统的资源**。这种模式适用于普通应用程序的运行。
  - 系统模式（System Mode）：系统模式是处理器最高的特权模式，它可以**执行所有指令**，包括特权指令和访问操作系统的资源。这种模式适用于操作系统内核的运行和系统级别的任务。
  - 监管模式（Supervisor Mode）：监管模式是类似于系统模式的特权模式，但权限更低。它可以执行一些特权指令和访问一些受保护的资源。这种模式适**用于操作系统内核的子系统或驱动程序的运行**。
  - 服务模式（Service Mode）：服务模式是一种特殊的模式，**用于支持调试和性能分析工具的使用**。它可以执行一些特殊的指令，以便进行调试和性能分析。
  - 中断模式（Interrupt Mode）：当处理器接收到中断信号时，会进入中断模式。在该模式下，

处理器**会执行中断处理程序，以响应中断事件**。
  - 异常模式（Abort Mode）：当处理器发生异常事件时，会进入异常模式。在该模式下，处理器会执行异常处理程序，以处理异常事件。异常事件包括指令或数据存取错误、未定义指令、软件中断等。
  - 未定义模式（Undefined Mode）：当处理器执行未定义的指令或指令序列时，会进入未定义模式。在该模式下，处理器会停止执行，并触发异常事件。

- 嵌入式系统中的操作系统可以分为实时系统和非实时系统，带操作系统和不带操作系统等不同类型。

  - 强实时操作系统一般用于对实时性要求较高的场景，软实时操作系统则对实时性要求相对较低。
  - μCOSII 是一个简单的实时操作系统，有休眠、就绪、运行、中断、挂起等状态。不同状态之间可以相互转换，需要清楚了解转换规则。

优先级：

- μCOSII 中有 64 个任务，其中有两个缺省任务，一个是最低优先级的空闲任务，一个是统计任务，可以删除。
  - μC/OS—II 中可以定义的最大任务数是**64**个,最高优先级定为 0; 最低优先级的任务的名称为**空闲任务,**其优先级为**63**。

- **任务优先级**的高低可以通过优先级任务组和就绪表来判断，不同优先级任务的就绪情况可能不同。
- 操作系统管理每个任务的任务控制块，通过任务控制块的指针列表和优先级来找到任务本身的堆栈地址和代码段地址。
- **优先级反转**：指高优先级任务等待低优先级任务所占用的资源，导致低优先级任务优先级变高，从而影响高优先级任务的执行。可以通过使用信号量或者禁止抢占的方式来消除优先级反转，保证高优先级任务不会被低优先级任务所阻塞。

异常处理和中断处理：

* 异常是指在指令执行的过程中发生的中断，需要在 CPU 内部执行一些微操作来响应异常。

- **ARM 处理器有 7 个异常类型**，分别是复位异常、未定义指令异常、软件中断异常、抢占中断异常、数据中止异常、指令中止异常和外部中断异常。
- 每个异常类型都有一个中断向量地址，中断向量地址是一个字，放的是对应异常的中断服务程序的入口地址。当某个异常发生时，处理器会根据对应的向量地址跳转到中断服务程序的入口地址。

- 当指令执行完毕时，CPU 会自动查询外部设备是否有中断请求，如果有且 CPU 允许中断，那么 CPU 会响应中断并执行中断服务程序。此时指令执行过程中可能会增加一个中断周期。

- 当操作系统主程序被中断时，会进入中断状态，任务恢复后可能会发生状态转换。之前的高优先级任务可能会由挂起状态变为就绪状态，需要注意优先级的调度。

- 对于同一台计算机上的同一段程序，在不同的时刻执行同一条指令时，可能会出现中断周期

的差异。中断周期是由中断请求引起的，可能会影响指令的执行。

- 在 x86 汇编中，中断周期需要保存断点、关中断、找到中断服务程序入口地址，通常使用 push 指令将 CS、IP 和 PSW 等信息保存到内存中。由于 push 指令涉及到内存的操作，因此中断在 x86 中的速度较慢。

- 在 ARM 架构中，**每种异常模式都有自己私有的寄存器**，如链接计数器**LR**和状态保存寄存器**SPSR**等。可以使用 LR 寄存器保存被中断的那条指令的下一条指令地址，使用 SPSR 保存中断前的程序状态。由于状态的断点是从寄存器到寄存器，因此**ARM 在实现中断引起的过程中不需要访问内存，速度较快**。

- 在 ARM 中，每条指令都包括取指周期、译码周期和执行周期。当发生异常时，可能还会包括中断周期。指令存储在内存中，处理器需要从内存中取指令并进行译码和执行操作，其中取指令的过程需要花费时间。

- ARM 架构大量使用寄存器，操作数可以来自于指令本身（立即数寻址）或者寄存器（寄存器寻址）。在指令执行过程中，ARM 不需要访问内存，因此处理速度较快。

- 在中断周期中，需要进行一系列操作来保存断点。具体步骤如下：
  1. 使用当前异常模式下的 LR 寄存器保存被中断程序的下一条指令地址。
  2. 使用当前模式下的 SPSR 寄存器保存被中断程序工作段的 CPSR。
  3. 将 CPSR 的模式位修改为当前模式（如普通中断模式、快速中断模式、软中断模式等）。
  4. 将状态设置为 ARM 状态。
  5. 关闭普通中断，如果当前中断是快速中断，则需要关闭快速中断。
  6. 将 PC 指向中断服务程序对应的中断向量地址，在中断向量地址中取中断服务程序的入口地址。

- 这些操作需要使用汇编语言进行描述，可以使用伪指令等方式来实现。

- 在硬件设计方面，需要对所使用的 ARM 芯片的架构有比较清晰的了解。

  - 所学的 ARM 芯片为 ARM 7TDMI S3C44B0X，具有**32 位地址线，可以接 4 GB 内存空间**。
  - 由于嵌入式系统一般用不了 4GB 的内存空间，因此实际上只能够引出 256M 空间，**分为八个块**，**每个块大小为 32M 字节**。
  - 每个块可以分为**SRAM、DRAM、FLASH**等类型，总线的宽度和速度可以通过存储器控制器进行调试和配置。

- ARM 架构是基于 RISC 的，它的 IO 指令和内存是共享空间的。所有 IO 的配置、存储器控制器、中断控制器以及 IO 设备的配置都是通过第一块里面最高的 4M 的寄存器的地址进行配置。

- **存储器控制器和中断控制器是嵌入式系统中重要的组成部分。**

  - 存储器控制器是一个硬件模块，用于管理嵌入式系统中的存储器，包括内部 RAM 和外部存储器。它可以控制存储器的读写、时序等操作。
  - 中断控制器也是一个硬件模块，用于管理嵌入式系统中的中断。中断控制器通常包括一个主模块和多个从模块，可以控制中断的优先级、向量中断或非向量中断等。

- 对于普通中断，可以通过中断控制器将其扩展到多个优先级进行配置，以便更好地管理中断。此外，还可以将中断设置为**向量中断或非向量中断**，具体取决于嵌入式系统的需求。

- 中断可以被设置为向量中断或非向量中断，具体取决于嵌入式系统的需求。向量中断是指每个中断源都分配了唯一的中断向量号，并且中断向量号被存储在中断向量表中，处理器可以直接跳转到中断服务例程来处理中断。非向量中断是指中断源没有分配唯一的中断向量号，处理器需要通过中断控制器来确定中断源的类型和优先级，并选择相应的中断服务例程来处理中断。

- I/O 方面的基本概念包括发送和接收。**RS 232C 是全双工的**。发送和接收都可以引发中断。**串口通常使用中断方式处理单字节数据**。

- I2C 是用于系统间信息交互的通信协议。它只有两根线：SDA 和 SCL。多个主设备可以通过这两根线实现半双工通信。在同一时刻，只能进行发送或接收操作。在使用 I2C 总线进行通信时，需要确保总线空闲，并避免竞争。

- 当主设备要访问从设备时，需要先给出从设备的地址。从设备地址是七位。第 8 位决定读或写。在使用 I2C 总线时，需要明确七位地址高位在前还是低位在前。I2C 总线的工作原理及其如何知道总线是否有竞争，如何知道自己是否与其他设备存在竞争，以及如何获取和放弃总线使用权都需要清楚了解。

- 在使用 I2C 总线时，从设备的地址是 7 位二进制数，其中最高位是读写位，用于指示主设备是否要向从设备读取数据或写入数据。当读写位为 0 时，表示主设备要写入数据到从设备，当读写位为 1 时，表示主设备要从从设备读取数据。

在 I2C 总线中，从设备地址的高位可以是在前或者在后，这取决于具体的 I2C 设备。通常情况下，在 I2C 总线中，地址高位在前，低位在后。

I2C 总线的工作原理是基于两根线路：SDA（串行数据线）和 SCL（串行时钟线）。主设备通过 SCL 线发送时钟信号来控制通信时序，通过 SDA 线发送和接收数据。从设备通过 SCL 和 SDA 线接收和发送数据，从而与主设备进行通信。

当多个设备同时访问 I2C 总线时，可能会出现竞争情况。I2C 总线的硬件设计中包含了冲突检测机制，可以检测到总线是否有竞争。当检测到总线有竞争时，总线会进入冲突检测模式，此时所有设备都会停止发送数据，等待下一次通信时序的到来。

在 I2C 总线中，每个设备都有一个地址，当主设备要访问从设备时，它会首先发送从设备的地址，然后等待从设备的响应。如果在超时时间内没有收到响应，主设备会认为总线上有竞争发生。如果多个设备同时发送地址，将会出现冲突。在这种情况下，总线将会进入冲突检测模式，等待所有设备重新发送地址。

为了获取和放弃总线使用权，每个 I2C 设备都有一个控制寄存器，用于控制设备的发送和接收操作。当设备要发送数据时，它会首先检查总线是否被占用，如果总线空闲，设备将会获取总线使用权，并发送数据。如果总线被其他设备占用，设备将会等待总线空闲，并重新尝试获取总线使用权。当设备发送完成后，它会放弃总线使用权，以便其他设备可以使用总线。

- 看门狗是一种监控系统正常运行的机制。可以通过中断或复位来恢复正常。通常将看门狗放

置在**优先级最低**的任务中，以确保其正常工作，并衡量任务调度是否达到最优状态。

- ARM 有七种模式，其中包括用户模式、系统模式、监管模式、中断模式、快速中断模式、抢占式模式和未定义模式。

- ARM 还有两种状态：ARM 状态和 THUMB 状态，可以用不同的编程方式进行编程。其中 THUMB 编程相对于 ARM 编程来说更加简单，适用于一些资源有限的系统。

- 程序阅读题：
  - 阅读给定的程序代码，并在代码中加入注释，解释程序的功能和结果。
  - 分析程序中使用的指令，以确定程序的作用和功能。
  - 总结就是语句语法含义和整体模块含义
  - **涉及到 C 调汇编、汇编调 C 的问题，因此需要清楚参数传递和返回的规则。**

- 系统设计题目：
  - 根据给定的功能要求，设计出一个满足要求的嵌入式系统。
  - 考虑硬件和软件两个方面
    - 对于硬件方面，需要**画出硬件方块图，以展示系统的结构和组成部件**；
    - 对于软件方面，需要**画出流程图或者使用伪指令进行描述，以展示系统的运行流程和功能实现方式**。
    - 在设计过程中，还需要考虑如何**利用所学知识去实现一些改变的功能**。

选择题

电子设计自动化工具（EDA）

* 工业软件芯片之母国外有几个典型的公司，三家企业分别为新思科技(Synopsys)、铿腾电子(Cadence)和 Siemens EDA（领导国际），前两者均为美国企业，第三家是德国西门子旗下企业。
* 国内比较标识叫华大九天，他是一个上市公司。

MCU 处理器

* MIPS 系列，X86 系列，ARM 系列，RISC-V
  * X86: 基于 PC 的这样台式机
  * 51 系列的话，低端的应用大量的市场，单片机也是先锋系统的一种，大概占有 70%的市场
  * ARM 系列和 MIPS 系列，在移动终端，小区应用是一个主流
  * 开放性最好的是 RISC-V，在版权方面没有什么限制，但是生态不好
* 嵌入式系统的表现形式
  * 计算机 MCU，SOC 片上系统，SOPC 片上系统可以编程的，还有 GPU，还有 FPGA
  * 对于实时性要求很强的末端控制系统的话，也可以完全用 FPGA 来实现，全硬件
  * 国内做 ARM 芯片：
    * 通用的 ARM 芯片珠海的全志科技
    * 福州的瑞芯微；

  * 做 GPU 的：就是景嘉微；
  * 做 FPGA
    * 国外公司的话是 Xilinx（赛灵思）、Altera（阿尔特拉）、Lattice（莱迪思）、Microsemi(美高森美)
    * 国内上海的复旦微。
  * 单片机 51
    * 国内：宏晶电子，STC，又好又便宜。
*

填空简答题 6*5=30 分

1、ARM 有 2 种工作状态，分别是 ARM 态 ， Thumb 态 ，当 ARM 工作于 ARM 状态时，其指令长度为 32 位，当 ARM 工作于 Thumb 状态时，其指令长度为 16 位；

* 第一种为 ARM 状态,此时处理器执行 32 位的字对齐的 ARM 指令;
* 第二种为 Thumb 状态,此时处理器执行 16 位的、半字对齐的 Thumb 指令。

2：ARM 有 7 种工作模式，分别是：用户模式、系统模式、快速中断模式、中断模式、管理模式、终止模式、未定义指令模式；

3：ARM 处理器共有 37 个寄存器，其中有 31 个通用寄存器，6 个状态寄存器；R13（SP）常用作堆栈寄存器，R14（LR）为链接寄存器，R15（PC）为程序计数器。

一个最简单的操作系统 μCOSII：

* 休眠，就绪，运行，中断，挂起
* 转换图：p337

简答题（30 分，每题 10 分）

1**：什么是嵌入式系统？单片机与嵌入式系统有何关系？**

嵌入式系统：以应用为中心,以计算机为基础,软件硬件可剪裁,适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

单片机与嵌入式系统的区别：单片机只是嵌入式系统中的一个组成部分，而嵌入式系统则是一个更加综合的概念；单片机本身就是嵌入式芯片或嵌入式微控器，单片机系统的实际应用就是单片嵌入式系统的典型实例。

2：**什么是 ARM？**ARM3C44BOX 芯片常用的通信接口有哪些？

ARM 是一种处理器架构，由 ARM Holdings 公司开发。ARM 架构被广泛应用于嵌入式系统和移动设备等领域，因其低功耗、高性能、低成本和可定制性等特点而备受青睐。ARM 架构的处理器有很多种，例如 Cortex-A、Cortex-R、Cortex-M 等系列，它们针对不同的应用场景和性能要求进行了优化。

3：完整的 ARM 嵌入式开发环境包括哪几部分？请图示各部分的相互关系。

嵌入式开发环境组成

- 交叉开发环境
- 软件模拟环境
- 目标板与评估板

4： 指令 MOVS PC，LR 执行时同时做了哪两件事？

1)pc=lr,结果实现一个返回跳转

2)cpsr=spsr,结果实现一个状态的恢复

5： 简述 ARM 异常的分类及其异常中断响应的过程 ** <u>p65+p66+p68**</u>

ARM 包括以下几种异常类型：

* 指令执行引起的直接异常
  * 软件中断
  * 未定义指令
  * 预期指终止 88
* 指令执行引起的间接异常
  * 数据中止
* 外部产生的与指令流无关的异常
  * 复位
  * IRQ
  * FIQ

1、异常中断响应过程：
（1）在相应的链接寄存器 LR （r14）中保存下一条指令的地址；
（2）保存处理器当前状态，中断屏蔽以及各条件标志位，即将当前 CPSR 复制到新的异常模式的 SPSR；
（3）强制使 CPSR 模式位置成对应异常类型的值：根据异常类型，重新设置 CPSR 的运行模式位 CPSR[4：0]，使微处理器进入相应的工作模式；
（4）强制给 PC 赋值，将相应的向量地址赋给 PC，从而跳转到相应的异常处理程序处执行程序；
（5）设置 CPSR 中的中断禁止位，以禁止中断发生。
2、异常返回流程：
（1）由链接寄存器 LR 的值恢复 PC，返回到发生异常中断的指令的下一条指令处执行程序；
（2）将 SPSR 复制回 CPSR 中；

（3）若在进入异常处理时设置了中断禁止位，要在此清除。

使用伪代码描述普通中断：p67

**I2C 总线通信机制怎么知道总线没有存在竞争 ** p276

简述任务切换过程

任务控制块（Task Control Block，TCB）是操作系统中用于维护进程或任务信息的数据结构。每个任务都有一个对应的 TCB，用于描述任务的状态、上下文等信息。在操作系统中，TCB 通常由操作系统内核来管理和维护。

在操作系统中，任务控制块通常是通过指针列表来进行管理的。每个任务都有一个唯一的任务 ID，对应着一个在指针列表中的位置。指针列表中的每个元素都是一个指向任务控制块的指针，根据任务 ID 可以快速找到对应的指针。根据任务优先级的高低，可以在指针列表中找到优先级最高的任务控制块指针。

任务控制块中保存了任务的堆栈地址。在任务运行时，CPU 会将当前的上下文信息（如寄存器值等）保存在任务堆栈中。当任务被调度出去时，CPU 会将任务的上下文信息从堆栈中恢复出来，并将任务的堆栈地址赋值给 SP 寄存器。这样，下次调度到该任务时，CPU 会从任务的堆栈中恢复上下文信息，并将 PC 寄存器设置为任务代码段的入口地址，从而继续执行该任务的代码。通过这种方式，操作系统可以实现多任务的并发执行。

为什么 ARM 比 X86 要快？

1. 使用寄存器：ARM 架构大量使用寄存器，指令执行时操作数要么来自于指令本身（立即数寻址），要么来自于寄存器（寄存器寻址）。由于 ARM 在指令执行过程中不需要访问内存，因此基于 RISC 的 ARM 速度比较快。

2. 中断处理：ARM 处理中断时，每种异常模式都有自己的私有寄存器，如链接计数器（LR）和保存程序状态寄存器（SPSR）。在中断处理过程中，状态的断点是从寄存器到寄存器，因此 ARM 在实现中断引起的过程中不需要访问内存，速度较快。

3. 指令执行：ARM 指令集的精简和规范使得指令执行速度更快，此外 RISC 保证每条指令都是 1 个周期，也有利于流水线并行。相比之下，x86 是 CISC 指令集更加复杂，执行时需要更多的时钟周期。

程序阅读题 2*10 分

p85

p99

p103

```ARM
ADD R3,R2,R1,LSR,#2 ; R3 <- R2 + R1/4

LDR R0,[R1] ; R0 <- MEM[R1]
STR R0,[R1] ; MEM[R1] <- R0

前变址
LDR R0,[R1,#4] ; R0 <- MEM[R1+4]
LDR R0,[R1,R2,LSL,#2] ; R0 <- MEM[R1+R*4]

自动编址
LDR R0,[R1,#4]! ; R0 <- MEM[R1+4]; R1 = R1 + 4

后编址
LDR R0,[R1],#4 ; R0 <- MEM[R1]; R1 = R1 + 4

指针（地址指向）
ADR R1,TABLE1

STMFD SP!, {R1-R7,LR} ; 把 R1-R7 和 LR 入栈
LDMFD SP!, {R1-R7,LR} ; 对应的出栈指令

STMDB R0!, {R2-R9} ; R0=R0-4；将数据存储到 R0 指向的空间；
LDMIA R0!, {R2-R9} ; 将 R0 指向空间的数据载入会寄存器；R0=R0+4

ADDS R4,R0,R2 ; 修改标志位的加
ADC R5,R1,R3 ; 带进位加

RSB

MOV R0,R1 ; R0 <- R1
MOVS R0,R1 ; R0 <- R1；更新 N 和 Z 标志

CMP R1, #0      ; 比较 R1 和 0 的值
LDRNE R0, [R2]  ; 如果 R1 不等于 0，则从[R2]地址处加载一个字（32 位）的数据到 R0 寄存器中

```


块拷贝

```asm
```

```
AREA  Word,  CODE,  READONLY      ; 定义代码段 Word，只读
num    EQU     20                ; 定义符号常量 num 为 20
       ENTRY                     ; 定义程序的入口点为 start

start                            ; 程序开始
       LDR     r0, =src          ; 将 src 的地址装入寄存器 r0
       LDR     r1, =dst          ; 将 dst 的地址装入寄存器 r1
       MOV     r2, #num          ; 将 num 的值装入寄存器 r2

wordcopy                         ; 标签，用于循环跳转
       LDR     r3, [r0], #4      ; 从 r0 指向的内存中读取一个 4 字节的数据，存入寄存器 r3，并将 r0 加上 4
       STR     r3, [r1], #4      ; 将寄存器 r3 中的数据写入 r1 指向的内存中，并将 r1 加上 4
       SUBS    r2, r2, #1        ; 将寄存器 r2 减去 1，并设置标志位
       BNE     wordcopy          ; 当标志位不为 0 时，跳转到 wordcopy 标签处继续循环执行

Stop    MOV    r0, #0x18         ; 将 0x18（24）装入寄存器 r0
       LDR     r1, =0x20026      ; 将 0x20026 的地址装入寄存器 r1
       SWI     0x123456          ; 调用 SWI 指令，将 r0 和 r1 的值传递给操作系统处理

       AREA  BlockData,  DATA, READWRITE  ; 定义数据段 BlockData，可读写
src    DCD     1,2,3,4,5,6,7,8,8,7,6,5,4,3,2,1,1,2,3,4   ; 定义一个长度为 20 的整型数组 src，赋初值
dst    DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0   ; 定义一个长度为 20 的整型数组 dst，初值为 0
       END                       ; 程序结束
```

该程序的功能是将一个长度为 20 的整型数组 src 中的数据复制到另一个长度为 20 的整型数组 dst 中。具体实现方式是通过循环来逐个复制，每次复制 4 字节。最后，程序将返回值 0x18 和地址 0x20026 传递给操作系统处理。

冒泡排序

```
; asm file: bubble_sort.s

       AREA asmfile, CODE, READONLY
       EXPORT bubble_sort

bubble_sort
       STMFD sp!, {r4-r11, lr}  ; 保存寄存器
```

```
        LDR r4, =arr              ; 加载数组的地址到 r4 中
        LDR r5, =len              ; 加载数组的长度到 r5 中

        MOV r6, #0                ; 初始化外层循环计数器
outer_loop
        CMP r6, r5                ; 检查是否已经比较了所有的元素
        BGE done                  ; 如果是，则直接退出
        MOV r7, #0                ; 初始化内层循环计数器
inner_loop
        CMP r7, r5-1              ; 检查是否已经比较了所有的元素
        BGE end_inner_loop        ; 如果是，则跳出循环
        LDR r8, [r4, r7, LSL #2] ; 加载当前位置的元素到 r8 中
        LDR r9, [r4, r7+1, LSL #2] ; 加载下一个位置的元素到 r9 中
        CMP r8, r9                ; 比较 r8 和 r9
        BLE end_inner_loop        ; 如果 r8 小于等于 r9，则跳出循环
        STR r8, [r4, r7+1, LSL #2] ; 将 r8 存储到下一个位置
        STR r9, [r4, r7, LSL #2] ; 将 r9 存储到当前位置
end_inner_loop
        ADD r7, r7, #1            ; 增加内层循环计数器
        B inner_loop              ; 继续比较下一个元素
done
        LDMFD sp!, {r4-r11, pc}  ; 恢复寄存器并返回

        ; 定义要排序的数组和数组长度
        DATA
        arr DCD 4, 2, 1, 3, 5     ; 数组元素
        len DCD 5                 ; 数组长度

        END
```

汇编和 c 的相互调用

实现字符串拷贝：

```
; asm file: strcpy.s

        AREA asmfile, CODE, READONLY
        EXPORT asm_strcpy

asm_strcpy
        loop
            LDRB r4, [r0], #1       ; 从源地址读取一个字节，然后递增源地址
            CMP r4, #0              ; 判断是否读到了字符串的结尾
```

```
            BEQ over                ; 如果是，则跳转到结束标签
            STRB r4, [r1], #1       ; 将读到的字节写入目标地址，然后递增目标地址
            B loop                  ; 继续读取下一个字节
        over
        MOV pc, lr                  ; 返回调用者

        END
```

```
#include <stdio.h>

extern void asm_strcpy(const char *src, char *dest);

int main() {
    const char *s = "seasons in the sun";
    char d[32];

    asm_strcpy(s, d);
    printf("source: %s\n", s);
    printf("destination: %s\n", d);
    return 0;
}
```

使用汇编语言完成一个随机数产生函数，通过 c 语言调用该函数产生一些列随机数存放到数组中

```
; asm file: random.s

        AREA asmfile, CODE, READONLY
        EXPORT random_numbers

random_numbers
        LDR r0, =0x12345678         ; 设置随机数种子，可以改变这个值来得到不同的随机数序列
        MOV r1, #0                  ; 设置计数器
loop
        BL random_byte              ; 调用 random_byte 函数，得到一个随机字节
        STRB r0, [r2, r1]           ; 将寄存器 r0 中的最低字节（也就是最后 8 位）的值存储到地址为 r2 + r1 的内存中
        ADD r1, r1, #1              ; 增加计数器
        CMP r1, r3                  ; 判断计数器是否到达数组长度
        BLT loop                    ; 如果还没达到，则继续生成随机数
        MOV pc, lr                  ; 返回调用者
```

```
; 生成一个随机字节
random_byte
        MOV r3, #1103515245       ; 乘数
        MOV r4, #12345            ; 增量
        MUL r0, r0, r3            ; 种子乘以乘数
        ADD r0, r0, r4            ; 加上增量
        AND r0, r0, #0xFF         ; 只保留最低 8 位（因为返回值是 char 类型）
        MOV pc, lr                ; 返回调用者
        END
```

```
#include <stdio.h>

extern void random_numbers(unsigned int seed, unsigned char *buf, unsigned int len);

int main() {
    unsigned char buf[10];
    random_numbers(0x12345678, buf, 10);

    printf("Random numbers:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d\n", buf[i]);
    }

    return 0;
}
```

编写一完整的 ARM 汇编程序，实现把一个正整数有序地插入到一个已排好序的（正整数）内存序列中，并画出程序流程图。

```
; asm file: insert_sort.s

        AREA asmfile, CODE, READONLY
        EXPORT insert_sort

insert_sort
        STMFD sp!, {r4-r9, lr}    ; 保存寄存器

        LDR r4, [r0]              ; 将第一个元素加载到 r4 中
        CMP r1, #0                ; 检查序列是否为空
        BEQ done                  ; 如果是，则直接退出
        ADD r1, r1, #1            ; 增加序列长度计数器
        MOV r3, #1                ; 将计数器初始化为 1，因为第一个元素已经被加载到 r4 中了
```

```
loop
        LDR r5, [r0, r3, LSL #2] ; 加载下一个元素到 r5 中
        CMP r5, r4               ; 比较 r5 和 r4
        MOVGT r4, r5             ; 如果 r5 大于 r4，则将 r5 赋值给 r4
        SUBS r3, r3, #1          ; 减少计数器
        BGE loop                 ; 如果计数器大于等于 0，则继续循环

        ADD r3, r3, #1           ; 增加计数器，使其指向新元素的插入位置
        LDR r5, [r0, r3, LSL #2] ; 加载当前位置的元素到 r5 中
        MOV r6, r4               ; 将新元素的值保存到 r6 中
        STR r4, [r0, r3, LSL #2] ; 将新元素存储到当前位置
        ADD r3, r3, #1           ; 增加计数器
        MOV r7, r3               ; 将计数器的值保存到 r7 中
        MOV r8, r5               ; 将当前位置的元素值保存到 r8 中
        SUBS r1, r1, #1          ; 减少序列长度计数器
inner_loop
        CMP r7, r1               ; 检查是否已经比较完所有元素
        BGE done                 ; 如果是，则退出循环
        LDR r9, [r0, r7, LSL #2] ; 加载下一个元素到 r9 中
        CMP r6, r9               ; 比较 r6 和 r9
        BLE end_inner_loop       ; 如果 r6 小于等于 r9，则跳转到 end_inner_loop 标签
        STR r8, [r0, r7, LSL #2] ; 将 r8 存储到当前位置
        MOV r8, r9               ; 将 r9 赋值给 r8
        ADD r7, r7, #1           ; 增加计数器
        B inner_loop             ; 继续比较下一个元素
end_inner_loop
        STR r6, [r0, r7, LSL #2] ; 将 r6 存储到当前位置
done
        LDMFD sp!, {r4-r9, pc}   ; 恢复寄存器并返回

        END
```

```
#include <stdio.h>

extern void insert_sort(int *arr, int len);

int main() {
    int arr[] = {1, 2, 4, 6, 8, 9};
    int len = sizeof(arr) / sizeof(arr[0]);

    printf("Before sorting:\n");
    for (int i = 0; i < len; i++) {
        printf("%d ", arr[i]);
    }
```

```
    insert_sort(arr, len);
    printf("\nAfter sorting:\n");
    for (int i = 0; i < len; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

设计题-打卡系统

- 主控制器：例如 STM32 微控制器，用于处理各个模块间的数据通信和整个系统的控制逻辑。
- 读卡器（RFID）：用于读取学生卡的信息。例如 RC522 RFID 模块。
- 网络模块：用于将签到信息上传至服务器。例如 ESP32。
- OLED 显示屏：用于显示签到状态。例如 128x64 SSD1306。
- 电源管理模块：用于管理系统的电源，例如 AXP192。
- 电池供电模块：用于提供系统的电源，例如 3.7V 锂电池模块。
- 充电管理模块：用于管理电池的充电，例如 TP4056。
- 定时重启程序：用于定时重启系统，以确保系统的稳定性和可靠性。
- 人体传感器：用于检测教室内是否有人，以便自动唤醒系统。

主控制程序

```
while True:
    # 等待读卡器检测到学生卡
    if card_detected():
        # 读取学生卡信息
        card_id = read_card()
        # 获取当前时间
        current_time = get_current_time()
        # 将学生卡信息和签到时间上传至服务器
        upload_data(card_id, current_time)
        # 提示签到成功
        beep_success()
        # 显示签到状态
        show_status("签到成功")
    # 如果检测到人体红外信号
    if motion_detected():
        # 检测当前温度
        current_temp = get_current_temp()
        # 显示当前温度
        show_temp(current_temp)
```

```
    # 等待下一次检测
    sleep(1)

# 红外传感器检测程序

def motion_detected():
    # 读取人体红外传感器状态
    motion = read_motion_sensor()
    # 如果检测到人体红外信号，返回 True
    if motion:
        return True
    else:
        return False

# 温度感知程序

def get_current_temp():
    # 读取红外热释电传感器数据
    data = read_thermal_sensor()
    # 解析数据，计算出温度
    temp = parse_data(data)
    # 返回当前温度
    return temp

# 显示程序

def show_temp(temp):
    # 在 OLED 显示屏上显示当前温度
    oled_display(temp)

def show_status(status):
    # 在 OLED 显示屏上显示签到状态
    oled_display(status)

# 蜂鸣器提示程序

def beep_success():
    # 发出签到成功的提示音
    beep()

# 电源管理程序

def power_management():
    # 检测电池电量
    battery_level = get_battery_level()
    # 如果电量低于一定值，提示更换电池
    if battery_level < 10:
```

```
        show_status("电量低，请更换电池")
        beep_low_battery()

# 电池管理程序

def get_battery_level():
    # 读取电池电量
    battery_level = read_battery_level()
    # 返回当前电池电量
    return battery_level

# 充电管理程序

def charging():
    # 如果检测到充电电源，开始充电
    if charging_detected():
        start_charging()
```

以下是一些可能的方案，可以利用所学知识实现一些改变的功能：

1. 增加人脸识别功能：可以利用深度学习技术，实现人脸识别功能，以提高签到的准确性和安全性。可以使用开源的深度学习框架，如 TensorFlow、PyTorch 等，对已有的人脸数据进行训练，以实现人脸识别功能。同时，还需要考虑如何集成人脸识别模块到现有的硬件系统中，以及如何管理和存储人脸数据等问题。
2. 实现语音识别功能：可以利用语音识别技术，实现语音签到功能，以提高签到的便利性和效率。可以使用开源的语音识别库，如 CMU Sphinx、Kaldi 等，对语音数据进行训练，以实现语音识别功能。同时，还需要考虑如何集成语音识别模块到现有的硬件系统中，以及如何处理语音数据和命令等问题。
3. 添加 GPS 定位功能：可以利用 GPS 模块，实现签到时的位置定位功能，以提高签到的精确度和实时性。可以使用现有的 GPS 模块或者开源的 GPS 库，如 TinyGPS++等，对位置数据进行获取和处理，以实现 GPS 定位功能。同时，还需要考虑如何将位置数据与学生信息进行关联，如何存储和管理位置数据等问题。
4. 实现远程监控功能：可以利用网络模块，实现远程监控功能，以便管理员能够实时监控签到情况，及时处理异常情况。可以使用现有的网络模块或者开源的网络库，如 Socket、MQTT 等，实现数据的传输和通信。同时，还需要考虑如何实现安全的数据传输和存储，如何处理网络延迟和故障等问题。

设计题-基于嵌入式的自助售货机设计

1.使用 STC89C52RC 单片机做主控制器件。

2.使用 LCD12864 做显示屏，12864 可显示汉字、数字、字母等。显示屏显示欢迎界面、货物信

息等。

3.使用按键做货物选择、投币、确实、返回等选择。

4.使用步进电机做出货口的控制，出货完成后，步进电机自动归位。

5.使用蜂鸣器做报警提示，当付款完成后，蜂鸣器会自动响一下，伴随电机转动，自动出货。

设计题-门禁卡系统设计