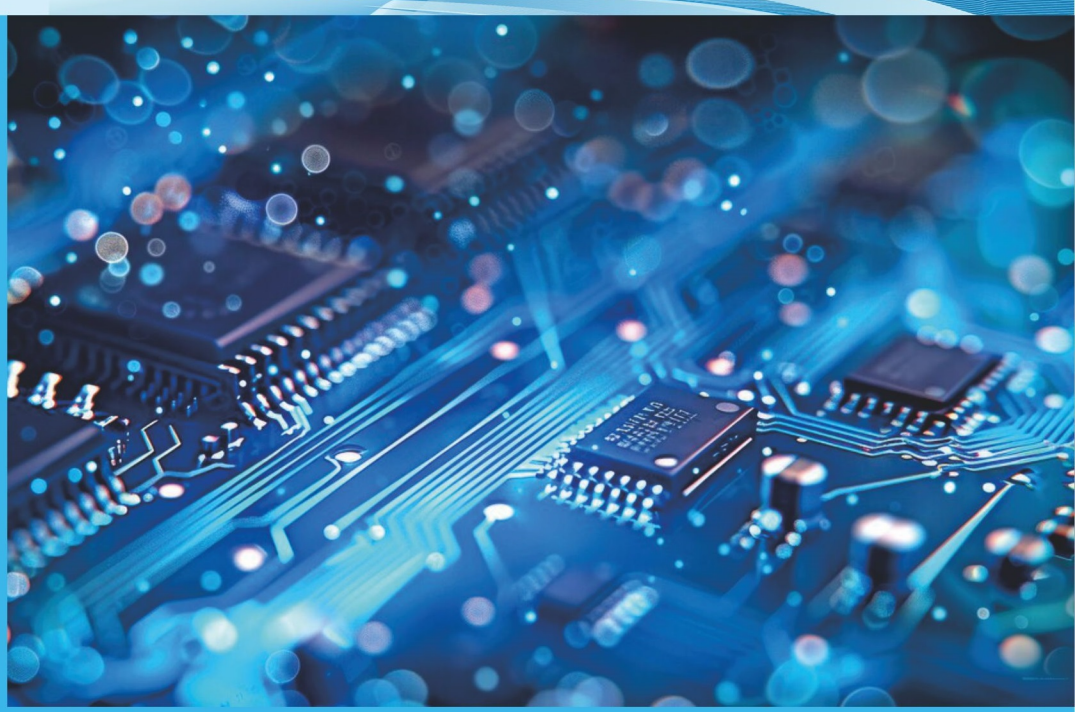
高职本科“十五五”系列教材 机电类专业

嵌入式技术应用开发

主编 张晓阳 赵 蕾

移动终端
微课资源
双色印刷
即扫即看



南京大学出版社

高职本科“十五五”系列教材 机电类专业

嵌入式技术应用开发

主　编　张晓阳　赵　蕾
副主编　肖　建　李从宏　何　龙

微信扫码关注免费获取学习资源

南京大学出版社

内容简介

《嵌入式技术应用开发》是一本以实践为导向的项目化教程，旨在系统性培养读者的嵌入式系统设计与开发能力。全书摒弃传统理论罗列，通过八个由浅入深、贴近实际应用的完整项目，引导读者在实践中掌握核心知识与技能。

本书以广泛使用的 STM32 系列微控制器为硬件平台，内容覆盖 GPIO、串口、定时器、中断、ADC、I^2C、SPI、显示屏、电机控制等关键技术。从最简单的 LED 控制到复杂的测距仪、环境监测、智能窗帘及电动车调速系统，每个项目均以产品原型为目标，详细阐述需求分析、电路设计、代码编写到功能调试的全过程。

通过“做中学”的方式，读者不仅能牢固掌握嵌入式 C 语言编程、硬件接口与系统架构，更能培养解决复杂工程问题的综合思维与动手能力，为从事物联网、智能硬件等领域的开发工作奠定坚实基础。

本书适用于职业本科电子与信息大类相关专业的教材，也可作为相关技术开发人员的参考用书。

图书在版编目(CIP)数据

嵌入式技术应用开发 / 张晓阳，赵蕾主编. -- 南京 ：南京大学出版社，2026.2. -- ISBN 978-7-305-29944-5

Ⅰ. TP332.021

中国国家版本馆 CIP 数据核字第 20263EM531 号

出版发行　南京大学出版社
社　　址　南京市汉口路 22 号　　邮　　编　210093

书　　名　嵌入式技术应用开发
QIANRUSHI JISHU YINGYONG KAIFA
主　　编　张晓阳　赵　蕾
责任编辑　吴　华　　编辑热线　025-83596997

照　　排　南京开卷文化传媒有限公司
印　　刷　南京百花彩色印刷广告制作有限责任公司
开　　本　787 mm×1092 mm　1/16　印张 16.5　字数 422 千
版　　次　2026 年 2 月第 1 版　2026 年 2 月第 1 次印刷
ISBN　978-7-305-29944-5
定　　价　55.00 元

网　　址：http://www.njupco.com
官方微博：http://weibo.com/njupco
微信公众号：NJUYUNSHU
销售咨询热线：(025)83594756

前 言

进入 21 世纪以来，嵌入式技术渗透进了现代生活的每一个角落。对于许多初学者乃至相关专业的学生而言，嵌入式技术领域常被视为一座由晦涩概念、复杂电路与抽象代码构筑的险峰。传统的教学模式往往偏重理论知识的单向灌输与碎片化实验，导致学习者虽掌握孤立的知识点，却难以融会贯通，更缺乏解决真实世界复杂工程问题的系统能力。

本书彻底摒弃了按知识体系平铺直叙的传统结构，创新性地采用“全项目化”的驱动模式。全书精心设计了八个循序渐进、贴近实际应用的综合性项目，构成一条清晰的能力进阶路线：从项目一“玩转 LED”带你轻松叩开嵌入式世界的大门，掌握最基本的 I/O 控制；到项目二“多路抢答器设计”引入中断与按键扫描，理解人机交互；再到项目三“激光测距仪设计”与项目四“温室大棚环境检测仪设计”，深入传感器数据采集、处理与通信；继而通过项目五“电子万年历设计”掌握实时时钟与复杂状态管理；在项目六“音乐播放器”中探索频率与声音关系；最后，在项目七“百叶窗帘控制系统设计”与项目八“电动自行车调速系统设计”中，综合运用所学，直面电机控制、PWM 调速、闭环反馈等更为复杂的系统集成与控制问题。

每一个项目均是一个完整的工作过程，涵盖从需求分析、方案设计、硬件搭建、软件编程到系统调试、性能优化的全部环节。力求在有限的篇幅内，不仅阐明“怎么做”，更深入探讨“为何这样做”，引导读者举一反三。书中代码规范、注释详尽，电路原理清晰，并穿插了关键的技术要点与故障排查思路，旨在帮助读者构建起扎实且可迁移的嵌入式系统开发知识体系与工程素养。

本书的成稿，得益于南京工业职业技术大学浓厚的产学研氛围与对高素质技能型人才培养的高度重视。张晓阳编写绪论、项目三和项目七，并负责全书的统稿定稿工作；赵蕾编写项目一、项目二和项目四，并负责全书整体架构设计；李从宏编写项目五和项目六；南京邮电大学肖建教授和百科荣创（北京）科技发展有限公司何龙高级工程师编写项目八，并对全书的项目架构和设计方案提出了许多宝贵的意见。

我们深信，无论是高职本科院校电子信息、物联网应用、自动化、计算机等相关专业的学生，还是从事嵌入式相关工作的工程师，抑或是一位热情的电子技术爱好者，只要跟随本书的项目步伐，亲手实践，不断思考，就一定能够跨越理论与实践的鸿沟，获得操控硬件、实现创意的乐趣与成就感，最终在嵌入式技术的广阔天地中自由翱翔。

由于时间紧迫，编者水平有限，书中难免存在疏漏与不足之处，恳请广大读者与专家批评指正。

编　者

2025 年冬于松山湖畔

目 录

绪 论

学习目标

知识目标

1. 嵌入式系统概念。
2. 嵌入式系统发展历程。
3. MCU 运行原理。
4. STM32 单片机的特点与选型。

能力目标

1. 掌握嵌入式系统、51 单片机和 STM32 单片机的关系。
2. 了解微处理的运行方式。

素质目标

1. 嵌入式系统作为信息技术的核心载体，其发展形势呈现出全球竞争与合作并存的格局。国际上，嵌入式技术正朝着高性能、低功耗、智能化和高度集成方向快速演进，物联网、人工智能、汽车电子等领域的融合创新不断加速。在国内，随着“中国制造 2025”等国家战略的深入推进，嵌入式系统在工业控制、智能终端、航空航天等关键行业得到广泛应用，自主技术积累和产业生态建设日益加强，整体发展势头迅猛。

2. 嵌入式芯片的国产化具有至关重要的战略意义。芯片作为嵌入式系统的“大脑”，其自主可控直接关系到关键基础设施、国防装备和信息系统的安全可靠。推动嵌入式芯片的国产化替代，能够减少对国外技术的依赖，降低供应链中断和潜在后门带来的风险，从根本上增强国家在科技竞争中的主动权和抗风险能力，是维护国家网络安全、经济安全和长远发展的重要保障。

任务描述

阅读章节内容，建立嵌入式系统的理念，了解嵌入式系统发展过程，掌握单片机运行原理，根据项目要求合理选择 STM32 单片机的类型，掌握实验平台的电路原理。

0.1 嵌入式系统概念

一、什么是嵌入式系统呢?

根据 IEEE(国际电气和电子工程师协会)的定义:嵌入式系统是“用于控制、监视或者辅助操作机器和设备的装置”(原文为 Devices Used to Control, Monitor, or Assist the Operation of Equipment, Machinery or Plants)。

如果你觉得“嵌入式系统”听起来很专业,先来一个形象的比喻:你家扫地机器人、冰箱、电视机,甚至你手里那台手机,里面其实都藏着一颗“小电脑”。它不需要像 PC 机那样跑 Windows,也不会装 Office,但它可以精准地干一件事情:比如扫地、显示画面、接打电话。这颗“小电脑”,就是嵌入式系统。

嵌入式系统的特点是:

(1) 系统内核小

由于嵌入式系统一般是应用于小型电子装置,系统资源相对有限,所以内核较之传统的操作系统要小得多。比如 ENEA 公司的 OSE 分布式系统,内核只有 5 KB,而 Windows 的内核则要大得多。

(2) 专用性强

嵌入式系统的个性化很强,其中的软件系统和硬件的结合非常紧密,一般要针对硬件进行系统的移植,即使在同一品牌、同一系列的产品中也需要根据系统硬件的变化和增减不断进行修改。同时,针对不同的任务,往往需要对系统进行较大更改;程序的编译下载要和系统相结合,这种修改和通用软件的“升级”是完全不同的概念。

(3) 系统精简

嵌入式系统一般没有系统软件和应用软件的明显区分,不要求其功能的设计及实现过于复杂,这样一方面利于控制系统成本,同时也利于实现系统安全。

(4) 高实时性

高实时性的操作系统软件是嵌入式软件的基本要求,而且软件要求固化存储,以提高速度。软件代码要求高质量和高可靠性。

(5) 多任务的操作系统

嵌入式软件开发要想走向标准化,就必须使用多任务的操作系统。嵌入式系统的应用程序可以没有操作系统而直接在芯片上运行;但是为了合理地调度多任务,利用系统资源、系统函数以及专家库函数接口,用户必须自行选配 RTOS(Real Time Operating System)开发平台,这样才能保证程序执行的实时性、可靠性,并减少开发时间,保障软件质量。

(6) 专门的开发工具和环境

嵌入式系统开发需要专门的开发工具和环境。由于嵌入式系统本身不具备自主开发能

力，即使设计完成以后，用户通常也不能对其中的程序功能进行修改，因此必须有一套开发工具和环境才能进行开发，这些工具和环境一般是基于通用计算机上的软硬件设备以及各种逻辑分析仪、混合信号示波器等。开发时往往有主机和目标机的概念，主机用于程序的开发，目标机作为最后的执行机，开发时需要交替结合进行。

简单地讲，嵌入式系统就是嵌入到各种设备里的专用计算机系统。嵌入式系统是一种为特定应用定制、深度集成到宿主设备中、在资源受限条件下运行的专用计算机系统。它是现代智能设备和工业系统的“大脑”和“神经系统”，让这些设备变得智能、自动化和互联互通。

二、MCU 的概念

微控制单元（MCU，Microcontroller Unit），又称单片微型计算机（SCM，Single Chip Microcomputer）或者单片机，是把中央处理器（CPU，Central Process Unit）的频率与规格做适当缩减，并将内存（Memory）、计数器（Timer）、USB、A/D 转换、UART、PLC、DMA 等周边接口，甚至 LCD 驱动电路都整合在单一芯片上，形成芯片级的计算机，为不同的应用场合做不同组合控制。诸如手机、PC 外围、遥控器，至汽车电子、工业上的步进马达、机器手臂的控制等，都可见到 MCU 的身影。

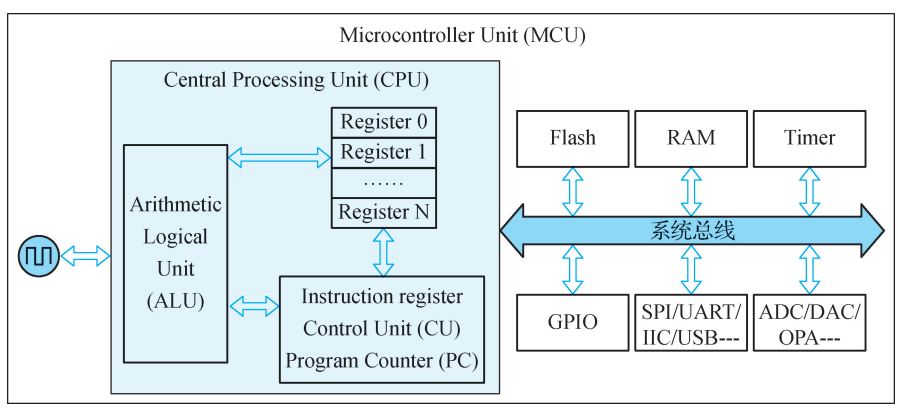


图 0.1　微控制单元结构框图

图中显示 MCU 的结构框图。中央处理器单元（CPU）在时钟节拍下完成取指、执行和存储等工作，并通过系统总线与各种外设实现互联。

三、我们常说的单片机与嵌入式系统有什么关联呢？

单片机是嵌入式系统的硬件大脑，嵌入式系统必须靠单片机才能跑起来。单片机可以理解为一颗小型的计算机芯片，它把 CPU、存储器、I/O 口、定时器、中断系统、通信接口等集成在一块硅片里。它不像 PC 机一样庞大，但能干很多“小而精”的活。比如，按键控制灯的亮和灭；使用传感器采集温度、湿度等数据；实现电机驱动，让风扇转起来，让窗帘自动开合动作等。

我们常说的 51 单片机，就是经典的入门款。它便宜、资料多、学习门槛低，所以在学生和低端产品里还很常见。但 51 处理能力有限，速度慢、资源少，已经逐渐被淘汰。

说到这里，你可能会问：那 STM32 又是什么？

简单说，STM32 也是单片机，但它属于进阶版。它基于 ARM Cortex-M 内核，是 32 位 MCU。相比 51 单片机的“老慢”，STM32 有这些优势：

性能更强：主频高，跑得快；资源更丰富：有 ADC、DAC、定时器、通信接口，应有尽有；编程更方便：ST 官方提供函数库，很多功能一行函数就能搞定。所以，现在大多数智能硬件、物联网设备，基本都会首选 STM32，而不是 51。

四、嵌入式、单片机和 STM32 这三者之间有什么关联呢？

嵌入式是一个大概念，指“专用小电脑系统”；单片机是实现嵌入式的硬件核心；STM32 是单片机的明星代表，比 51 更强大。

可以这样理解：嵌入式系统包含了基于单片机的硬件电路和以单片机为载体的程序驱动软件这两部分。其中硬件电路又可以根据 MCU 的类型，分为 51 系列的、STM32 系列的以及其他系列的单片机电路。

嵌入式系统
├—— 单片机(硬件)
|　　├—— 51 系列(经典老兵)
|　　└—— STM32(性能强者)
└—— 软件(程序驱动)

五、为什么要学习嵌入式技术呢？

原因很简单：嵌入式无处不在。未来十年，物联网、智能家居、智能汽车、工业自动化国防、航空等重要领域统统离不开嵌入式。

学会嵌入式，你能做的事情包括自己写个智能家居控制器、开发一台小机器人、给无人机写飞控，甚至参与汽车 ECU 开发等。这就是为什么很多电子、计算机专业的学生，都会绕不开嵌入式的原因。

0.2 嵌入式系统发展历程

单片机出现的历史并不长，但发展十分迅猛。它的产生与发展和微处理器的产生与发展大体同步，自 1971 年美国 Intel 公司首先推出 4 位微处理器以来，它的发展到目前为止大致可分为 5 个阶段。下面以 Intel 公司的单片机发展为例进行介绍。

第 1 阶段　单片机发展的初级阶段(1971—1976 年)。1971 年 11 月 Intel 公司首先设计出集成度为 2 000 多只晶体管/片的 4 位微处理器 Intel 4004，并配有 RAM、ROM 和移位寄存器，构成了第一台 MCS—4 微处理器，而后又推出了 8 位微处理器 Intel 8008，其他各公司也相继推出 8 位微处理器。

第 2 阶段　低性能单片机阶段(1976—1980 年)。以 1976 年 Intel 公司推出的 MCS—48 系列为代表，采用将 8 位 CPU、8 位并行 I/O 接口、8 位定时/计数器、RAM 和 ROM 等集成于一块半导体芯片上的单片结构，虽然其寻址范围有限(不大于 4 KB)，也没有串行 I/O，RAM、ROM 容量小，中断系统也较简单，但其功能可满足一般工业控制和智能化仪器、仪表等的需要。

第 3 阶段　高性能单片机阶段(1980—1983 年)。这一阶段推出的高性能 8 位单片机普

遍带有串行口，有多级中断处理系统，多个 16 位定时器/计数器。片内 RAM、ROM 的容量加大，且寻址范围可达 64 KB，个别片内还带有 A/D 转换接口。

第 4 阶段　16 位单片机阶段(1983 年—80 年代末)。1983 年 Intel 公司又推出了高性能的 16 位单片机 MCS—96 系列，由于其采用了最新的制造工艺，使芯片集成度高达 12 万只晶体管/片。

第 5 阶段　高水平发展阶段(1990 年—现在)单片机在集成度、功能、速度、可靠性、应用领域等全方位向更高水平发展。

按照单片机的特点，单片机的应用分为单机应用与多机应用。在一个应用系统中，只使用一片单片机称为单机应用。单片机的单机应用范围包括：

(1) 测控系统。用单片机可以构成各种不太复杂的工业控制系统、自适应控制系统、数据采集系统等，达到测量与控制的目的。

(2) 智能仪表。用单片机改造原有的测量、控制仪表，促进仪表向数字化、智能化、多功能化、综合化、柔性化方向发展。

(3) 机电一体化产品。单片机与传统的机械产品相结合，使传统机械产品结构简化，控制智能化。

(4) 智能接口。在计算机控制系统，特别是在较大型的工业测控系统中，用单片机进行接口的控制与管理，加之单片机与主机的并行工作，大大提高了系统的运行速度。

(5) 智能民用产品。如在家用电器、玩具、游戏机、声像设备、电子秤、办公设备、厨房设备等许多产品中，单片机控制器的引入不仅使产品的功能大大增强，性能得到提高，而且获得了良好的使用效果。

单片机的多机应用系统可分为功能集散系统、并行多机处理及局部网络系统。

(1) 功能集散系统。多功能集散系统是为了满足工程系统多种外围功能的要求而设置的多机系统。

(2) 并行多机控制系统。并行多机控制系统主要解决工程应用系统的快速性问题，以便构成大型实时工程应用系统。

(3) 局部网络系统。单片机按应用范围又可分成通用型和专用型。专用型是针对某种特定产品而设计的，例如用于体温计的单片机、用于洗衣机的单片机等等。在通用型的单片机中，又可按字长分为 4 位、8 位、16 位、32/64 位。

4 位 MCU 大部分应用在计算器、车用仪表、车用防盗装置、呼叫器、无线电话、CD 播放器、LCD 驱动控制器、LCD 游戏机、儿童玩具、磅秤、充电器、胎压计、温湿度计、遥控器及傻瓜相机等；8 位 MCU 大部分应用在电表、马达控制器、电动玩具机、变频式冷气机、呼叫器、传真机、来电辨识器(CallerID)、电话录音机、CRT 显示器、键盘及 USB 等；8 位、16 位单片机主要用于一般的控制领域，一般不使用操作系统，16 位 MCU 大部分应用在行动电话、数字相机及摄录放影机等；32 位 MCU 大部分应用在 Modem、GPS、PDA、HPC、STB、Hub、Bridge、Router、工作站、ISDN 电话、激光打印机与彩色传真机，用于网络操作、多媒体处理等复杂处理的场合，一般要使用嵌入式操作系统；64 位 MCU 大部分应用在高阶工作站、多媒体互动系统、高级电视游乐器(如 SEGA 的 Dreamcast 及 Nintendo 的 GameBoy)及高级终端机等。

到目前为止，中国的单片机应用和嵌入式系统开发走过了二十余年的历程，随着嵌入式系统逐渐深入社会生活各个方面，单片机课程的教学也有从传统的 8 位处理器平台向 32 位高级 RISC 处理器平台转变的趋势。国民经济建设、军事及家用电器等各个领域，尤其是手

机、汽车自动导航设备、PDA、智能玩具、智能家电、医疗设备等行业都是国内急需单片机人才的行业。行业高端目前有超过 10 余万名从事单片机开发应用的工程师，但面对嵌入式系统工业化的潮流和我国大力推动建设“嵌入式软件工厂”的机遇，我国的嵌入式产品要融入国际市场，形成产业，则必将急需大批单片机应用型人才，这为高职类学生从事这类高技术行业提供了巨大机会。

0.3 模拟 MCU 运行

单片机是如何按照设计者的要求运行的呢？

设计者需要将编写好的程序经过编译后生成的可执行文件，下载到 Flash ROM 中，单片机才能运行程序。

下图是 Cortex-M 内核的地址映射，注意到图中标红的 Code 和 SRAM 两个区域。这两个区域是单片机下载代码和数据后存放的位置。

Code 以 0x00000000 开始的一片 ROM(FLASH)区域，SRAM 是以 0x20000000 开始的一片 RAM 区域，SRAM 的读取速度相比于 Code 更快，用来保留堆栈和数据，保证 CPU 的执行效率。

这里要注意，Code 是从 0x00000000 开始，但是我们的代码是从 0x08000000 开始，因为前面由厂家配置 boot 程序等。

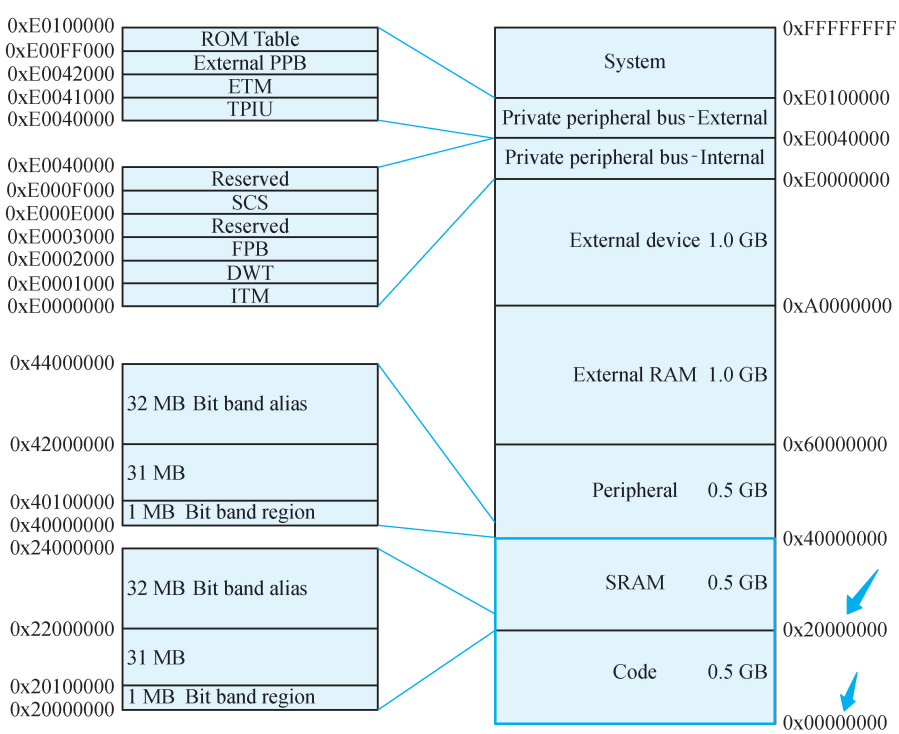


图 0.2 Cortex-M 内核的地址映射

单片机能够直接执行机器码，也就是可执行文件。C 程序经过编译链接之后获得可执行文件。C 代码经过编译、链接之后成为可执行文件。之后装入单片机的 FLASH 中，由单片机执行可执行文件。这里的 FLASH 指的就是 Code 区域，FLASH 是 ROM 的最新技术，结构简单，功耗低，很适合保存程序。

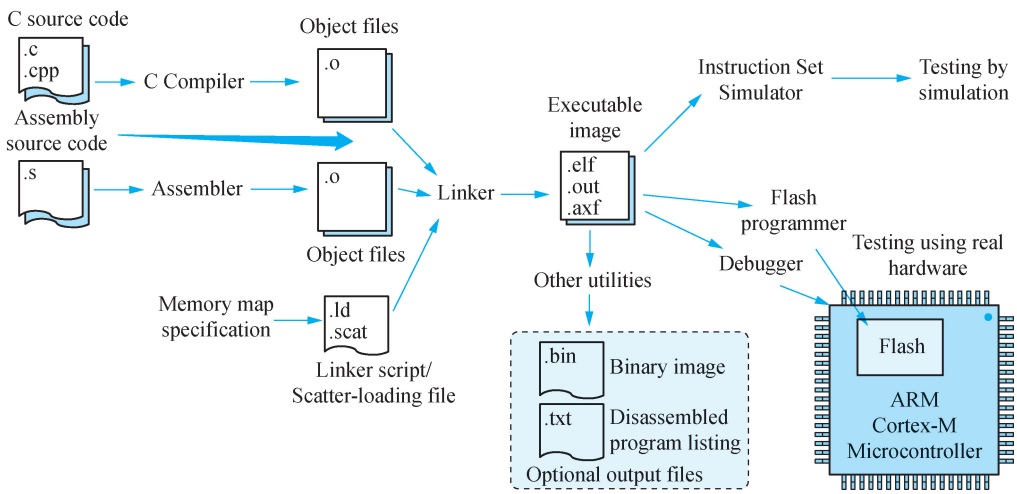


图 0.3　程序编译后处理流程

下面来模拟 MCU 的运行过程。MCU 中 CPU 的 Control Unit 控制单元，负责取指(Fetch)、译码(Decode)、执行(Execute)这样的循环运行。CPU 从 memory 取得指令，进行译码，识别指令的需要，并执行对应的操作，为了便于理解，我们将系统简化，并进行如下假设。

(1) MCU 只有 2 个通用寄存器：R0 和 R1；

(2) 有一个简单的数据内存，地址从 0x00 开始；

(3) 支持三类指令：LOAD、STORE、ADD；

(4) 每条指令为 8 位，格式固定。

如表 0-1 所示，假定有 3 种指令："01"表示装载，"10"表示存储，"11"表示加法操作。指令 aaaa rr 01 的作用是将 aaaa 地址中的数据装载到 rr 寄存器中。指令 aaaa rr 10 的作用是将 rr 寄存器中的数据存到 aaaa 地址的内容中。指令 rr1 rr2 11 的作用是将寄存器 rr1 和 rr2 中的数据相加，并将结果放到 rr1 寄存器中去。

表 0-1　指令代码结构

指令编码结构 Instruction	描述 Description	操作码 Opcode	操作数 Operand
aaaa rr 01	从 Address 中加载到寄存器	LOAD：01	rr：00 => R0，01 => R1
aaaa rr 10	从寄存器存到 Address 中	STORE：10	rr：寄存器编号
rr1 rr2 11	ADD：R[rr1] = R[rr1] + R[rr2]	ADD：11	两个寄存器编号

下面我们来分析一下表 0－2 中的 6 条指令的执行情况。

表 0－2　待执行的 6 条指令

Address	Data
0	01000001
1	01010101
2	00010011
3	01100010
4	00000010
5	00000011
……	xxxxxxx

第一步　PC 指针指向 Address 0 处，取得指令 01000001，译码执行将 Address 4 中的数据 00000010 存放到 Register 0 中。当前 PC 指针指向 00000000。

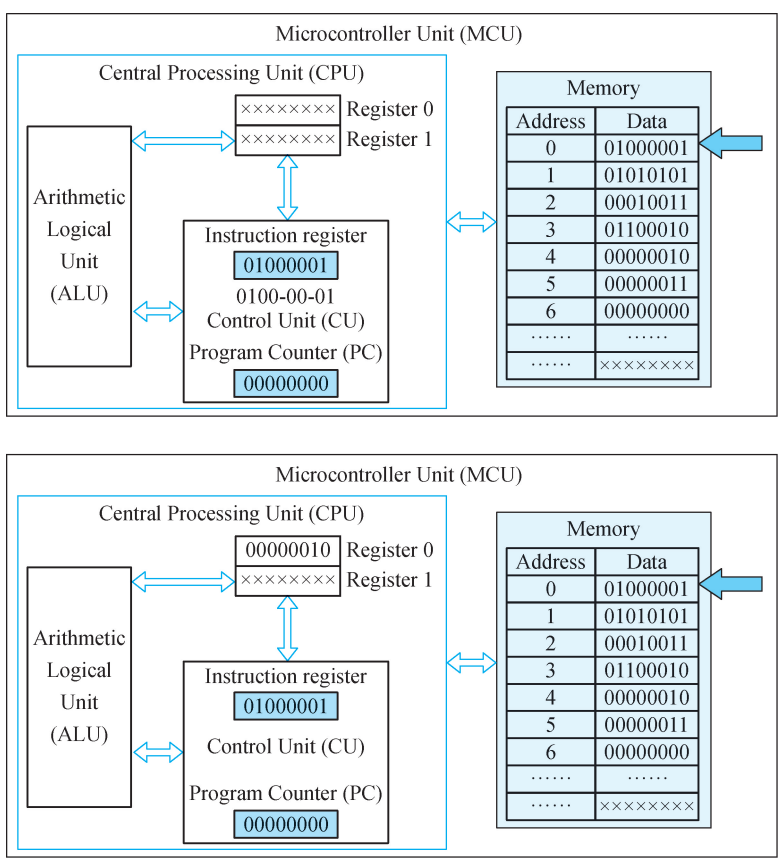


图 0.4　步骤 1 执行

第二步　PC 指向 Address 1，取得指令 01010101，译码执行，将 Address 5 中的数据存放到 Register 1 中。当前 PC 指针指向 00000001。

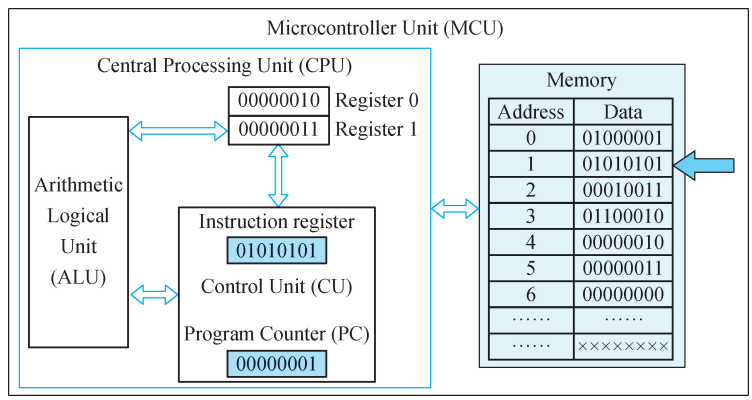


图 0.5　步骤 2 执行

第三步　PC 指向 Address 2，取得指令 00010011，译码执行，将 Register 0 和 Register 1 中的数据相加，结果存放到 Register 0 中。当前 PC 指针指向 00000002。

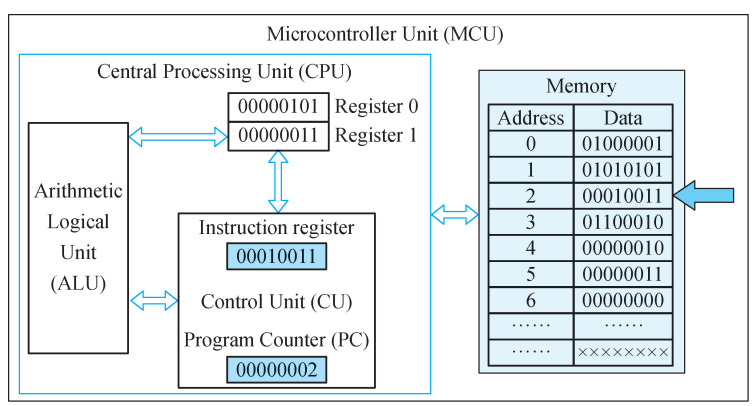


图 0.6　步骤 3 执行

第四步　PC 指向 Address 3，取得指令 01100010，译码执行，将 Register 0 中的数据存放到 Address 6 中。当前 PC 指针指向 00000003。

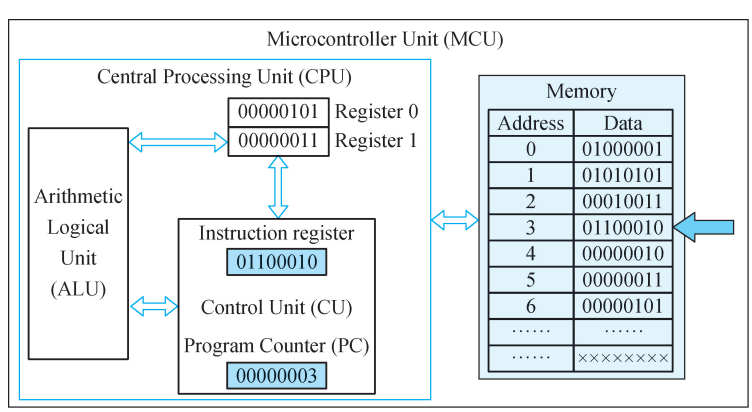


图 0.7　步骤 4 执行

这样，MCU 就完成了一段程序的执行，计算了 3+2，得到的结果 5 保存在 Memory 中。

需要注意的是程序中的指令是我们自己定义的，而目前实际 MCU 产品中运行的指令都是非常复杂的。程序中都是 0 和 1 这样的数据，实际编程开发中，设计者则是使用 C 语言编写程序，经过编译后生成可执行的文件，转换成二进制的 0、1 这样的数据下载到 Flash 中供单片机执行指令。

0.4 STM32 单片机的发展与选型

当前，嵌入式开发平台大多以主流的 STM32 系列单片机为 MCU。它属于意法半导体（ST，STMicroelectronics）的 32 位微控制器（MCU）家族，基于 ARM Cortex-M 内核。STM32 第一款单片机 2007 年在国内公布，诞生于北京。

Cortex-M3 处理器内核是单片机的中央处理单元（CPU）。完整的基于 CM3 的 MCU 还需要很多其他组件。在芯片制造商得到 CM3 处理器内核的使用授权后，它们就可以把 CM3 内核用在自己的硅片设计中，添加存储器、外设、I/O 以及其他功能块。不同厂家设计出的单片机会有不同的配置，包括存储器容量、类型、外设等都各具特色。如图 0.8 所示，Cortex-M3 处理器内核和调试系统是 ARM 设计的，不同类型的 ARM 芯片是由芯片制造商在 ARM 内核基础上集成了内部总线、外设、存储器、时钟复位电路和 IO 口设计出来的。

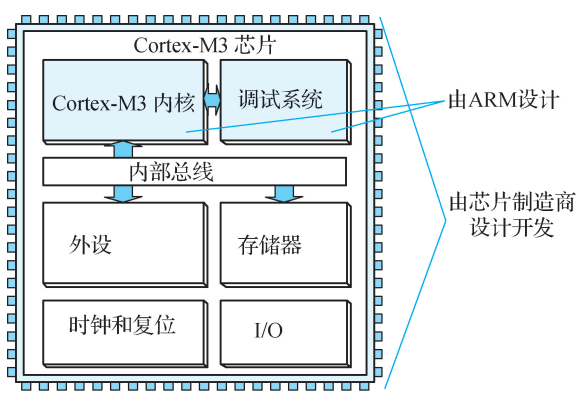


图 0.8　ARM 芯片内部结构示意

STM32 系列单片机的开发环境大多采用 keil C 软件，其开发方式经历了从寄存器到标准库，再到 HAL 库（硬件抽象层）的演变。它们的特点如下：

➢ 寄存器

（1）寄存器众多，需要经常翻阅芯片手册，费时费力；

（2）更大灵活性，可以随心所欲达到自己的目的；

（3）深入理解单片机的运行原理，知其然更知其所以然。

➢ 标准库

（1）将寄存器底层操作都封装起来，提供一整套接口（API）供开发者调用；

（2）每款芯片都编写了一份库文件，也就是工程文件里 stm32F1xx…之类的；

（3）配置结构体变量成员就可以修改外设的配置寄存器，从而选择不同的功能；

（4）大大降低单片机开发难度，但是在不同芯片间不方便移植。

➢ HAL 库

（1）ST 公司目前主力推的开发方式，新的芯片已经不再提供标准库；

（2）为了实现在不同芯片之间移植代码；

（3）为了兼容所有芯片，导致代码量庞大，执行效率低下。

寄存器方式需要开发者对微控制器的底层硬件结构有深入的了解，编写代码时需要直接操作寄存器地址，技术门槛较高。随着技术的进步和 ST 公司对 STM32 软件生态系统的投入，标准库（如 STM32 Standard Peripheral Libraries）逐渐成熟并流行起来。标准库通过封装底层硬件操作的函数，为开发者提供了更加简洁、易用的接口，降低了开发难度，提高了开发效率。从 2010 年代初开始，标准库开发方式逐渐取代寄存器开发，成为主流的开发方式。2014 年，ST 公司推出了 HAL（硬件抽象层）驱动库和 MCU 图形化配置软件 STM32CubeMX，为开发者提供了更加高级、通用的接口。作为标准库开发的进一步抽象，HAL 开发方式提供了更加高级、通用的接口。这使得开发者可以更加专注于应用层的设计和实现，而不必关心底层硬件的具体实现。HAL 库还具备跨平台的能力，为开发者提供了更多的灵活性。

本书内容以标准库为基础，介绍 STM32 单片机的开发技术。

意法半导体微控制器和微处理器拥有广泛的产品线，包含低成本的 8 位单片机和基于 ARM® Cortex®-M0、M0＋、M3、M4、M33、M7 及 A7 内核并具备丰富外设选择的 32 位微控制器及微处理器。STM32 致力于 ARM® Cortex®内核单片机和微处理器市场和技术方面，目前提供 24 大产品线（C0，F0，G0，F1，F2，F3，G4，F4，F7，H7，H5，MP1，MP2，L0，L1，L4，L4＋，L5，U0，U5，WB，WBA，WB0，WL），超过 1 000 个型号。STM32 产品广泛应用于工业控制、消费电子、物联网、通信设备、医疗服务、安防监控等应用领域，其优异的性能进一步推动了生活和产业智能化的发展。

STM32 产品家族分为微处理器 MPU、高性能 MCU、主流级 MCU、超低功耗 MCU 和无线 MCU 几类。

主流级 MCU 包括

- STM32G4 系列—Arm® Cortex®-M4 高性能模数混合型 MCU
- STM32G0 系列—Arm® Cortex®-M0＋入门级 MCU
- STM32C0 系列—Arm Cortex-M0＋超值入门型 MCU
- STM32F3 系列—Arm® Cortex®-M4 模数混合型 MCU
- STM32F1 系列—Arm® Cortex®-M3 基础型 MCU
- STM32F0 系列—Arm® Cortex®-M0 入门级 MCU

高性能 MCU 包括

- STM32N6 系列—Arm® Cortex®-M55 超高性能旗舰 AI MCU

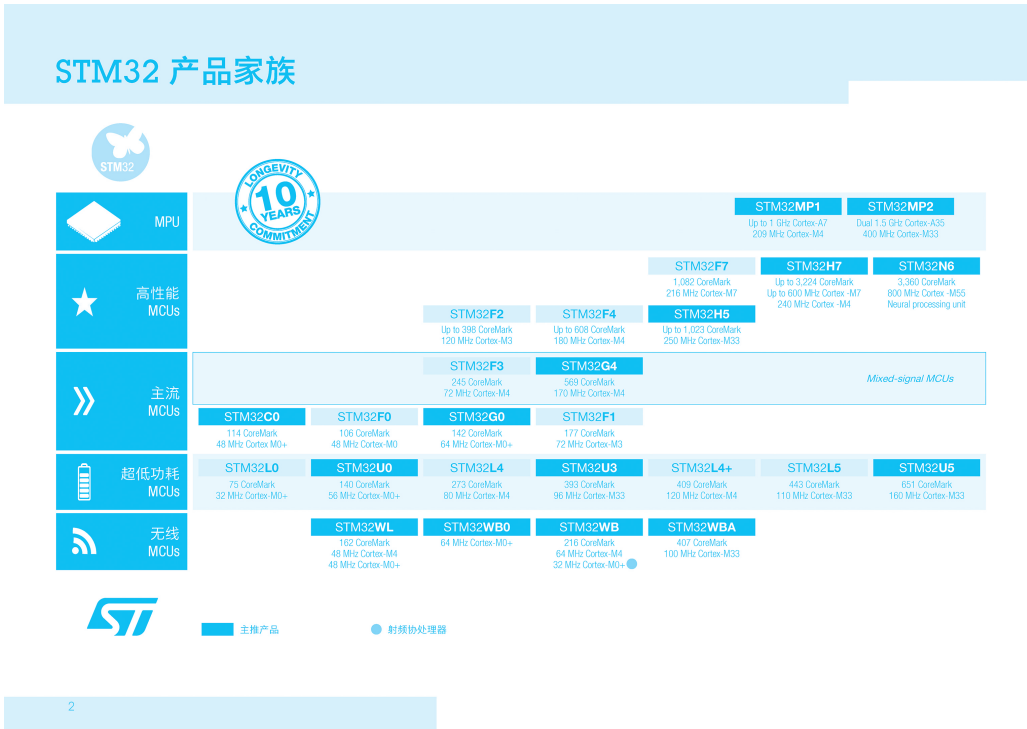


图 0.9　STM32 产品家族

- STM32H7 系列—Arm® Cortex®-M7/Cortex®-M7 + M4 超高性能/Cortex®-M7 Boot Flash 高性能 MCU
- STM32F7 系列—Arm® Cortex®-M7 高性能 MCU
- STM32H5 系列—Arm® Cortex®-M33 高性能 MCU
- STM32F4 系列—Arm® Cortex®-M4 高性能 MCU
- STM32F2 系列—Arm® Cortex®-M3 高性能 MCU

超低功耗 MCU 包括

- STM32U0 系列—Arm® Cortex®-M0+新一代超低功耗入门级 MCU
- STM32U3 系列—Arm® Cortex®-M33 新一代超低功耗近阈值设计 MCU
- STM32U5 系列—Arm® Cortex®-M33 新一代超低功耗旗舰系列 MCU
- STM32L5 系列—Arm® Cortex®-M33 超低功耗高性能高安全 MCU
- STM32L4+系列—Arm® Cortex®-M4 超低功耗高性能 MCU
- STM32L4 系列—Arm® Cortex®-M4 超低功耗 MCU
- STM32L1 系列—Arm® Cortex®-M3 超低功耗 MCU
- STM32L0 系列—Arm® Cortex®-M0+超低功耗 MCU

无线 MCU 包括

- STM32WB0 系列—Arm® Cortex®-M0+紧凑、节能设计的 2.4 GHz BLE 5.4 无线 MCU
- STM32WB 系列—Arm® Cortex®-M4 和 M0+双核 2.4G 多协议无线 MCU

- STM32WBA 系列—Arm® Cortex®-M33 超低功耗高性能安全 2.4 GHz BLE 5.4 等多协议无线 MCU
- STM32WL3 系列—Arm® Cortex®-M0＋Sub1G 长距离无线 MCU
- STM32WL5 系列—Arm® Cortex®-M4 和 M0＋支持 LoRa 等多协议的 Sub1G 长距离无线 MCU
- 射频收发器
- SPIRIT 系列—Sub1G Hz 射频收发器

微处理器 MPU

- STM32MP1 系列—Arm® Cortex®-A7/Cortex®-A7＋M4 高性价比工业级 MPU
- STM32MP2 系列—Arm® Cortex®-A35＋Cortex®-M33＋NPU 64 位工业级边缘 AI

表 0-3 STM32 的分类

内核	系列	描述
Cortex-M0	STM32-F0	入门级
	STM32-L0	低功耗
Cortex-M3	STM32-F1	基础型，主频 72 MHz
	STM32-F2	高性能
	STM32-L1	低功耗
Cortex-M4	STM32-F3	混合信号
	STM32-F4	基础型，主频 180 MHz
	STM32-L4	低功耗
Cortex-M7	STM32-F7	高性能

STM32 的命名方式如图 0.10 所示。以 STM32F103VET6 为例进行说明。STM32 代表意法半导体生产的 ARM Cortex-M 内核的 32 位微控制器。F 代表芯片子系列是基础型的，如果是 L 就是超低功耗型的；103 代表增强型系列；V 代表的是引脚数目为 100，常见的 C 表示 48 脚，R 表示 64 脚，Z 表示 144 脚；E 这一项代表内嵌 Flash 容量，其中 6 代表 32 K 字节 Flash，8 代表 64 K 字节 Flash，B 代表 128 K 字节 Flash，C 代表 256 K 字节 Flash，D 代表 384 K 字节 Flash，E 代表 512 K 字节 Flash，G 代表 1 M 字节 Flash；T 这一项代表封装，其中 H 代表 BGA 封装，T 代表 LQFP 封装，U 代表 VFQFPN 封装；6 这一项代表工作温度范围，其中 6 代表－40 ℃～85 ℃，7 代表－40 ℃～105 ℃。

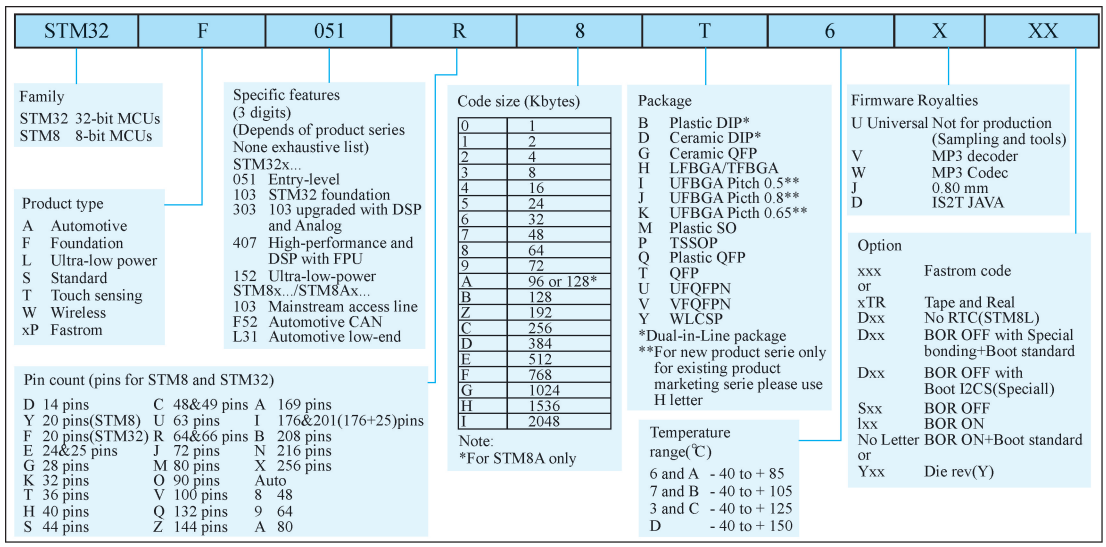


图 0.10　STM32 单片机的命名规则

了解了 STM32 的分类和命名方法之后，就可以根据项目的具体需求先大概选择哪类内核的 MCU。STM32 有很多系列，可以满足市场的各种需求，从内核上分有 Cortex-M0、M3、M4 和 M7 这几种，每个内核又大概分为主流、高性能和低功耗。对于普通应用，不需要接大屏幕的一般选择 Cortex-M3 内核的 F1 系列，如果要追求高性能，需要大量的数据运算，且需要外接 RGB 大屏幕的，则选择 Cortex-M4 内核的 F4 系列。我们如何选用合适 STM32 单片机呢？这里有一个原则：花最少的钱，做最多的事。

在确定项目需求的情况下，一般按照下面的顺序来选择合适的 MCU：

1. 选择哪种内核的芯片，内核越高意味着功耗也越高；
2. 选择多少引脚的芯片，引脚多少决定了资源的多少，也影响价格；
3. 选择多少 RAM 和 FLASH 的芯片，FLASH 越大，价格越贵；
4. 还要考虑所选型号采购是否容易，供货是否稳定。

0.5　嵌入式开发平台介绍

当前，市场上的嵌入式开发平台品种繁多，其中以 STM32F103 系列的为最常见、价格便宜的是最小系统板，上面仅包含使芯片能够正常运行的简单外围电路和指示灯。学习某一功能时，需要使用杜邦线连接相应的功能模块，连接和整理都不方便，并且时常会出现连接错误或通信不可靠的情况。也有学习者直接购买功能齐全的单片机开发板，不像使用杜邦线那样麻烦，也不会出现连接不可靠的问题，唯一的缺点就是价格昂贵。

基于项目化教学的需求，我们设计了一款性价比极高的嵌入式开发平台。采用的主控制器是 STM32F103VET6 型单片机，根据“玩转 LED”“多路抢答器”“激光测距仪”“环境检测仪”“电子万年历”“MP3 播放器”“百叶窗帘控制”和“电动车控制系统”这 8 个项目，进行

模块化设计，设备功能齐全、接口丰富，能最大化地满足嵌入式学习者对学习资源的要求。

图 0.11 是嵌入式系统硬件实验平台，集成了本书项目所需的多种传感器和控制电路，能够有效满足嵌入式学习者的资源需求。

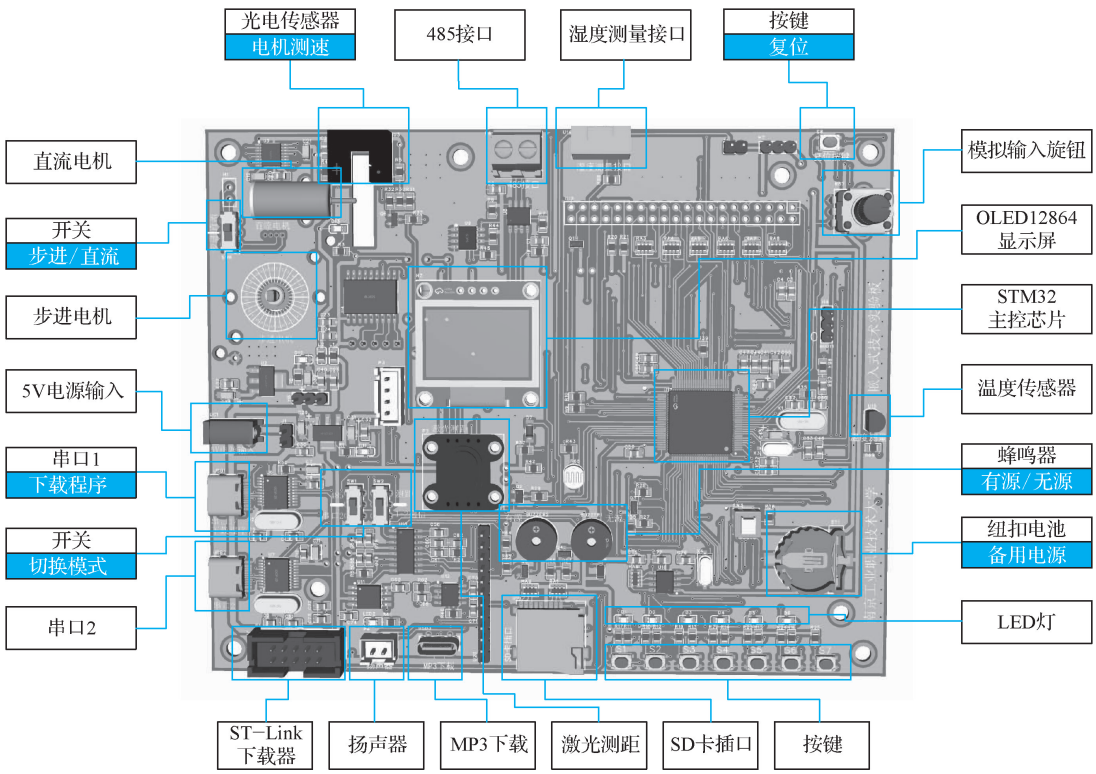


图 0.11　嵌入式硬件实验平台

项目一 玩转 LED

学习目标

扫码可见本项目微课

知识目标

1. GPIO 引脚结构和相关寄存器。
2. C 语言基础语法。
3. Keil C 工程建立、调试、下载。
4. 系统嘀嗒定时器延时。
5. 系统文件函数。

能力目标

1. 寄存器配置能力：能熟练查阅 STM32 参考手册，理解 GPIO 相关寄存器（模式寄存器、输出数据寄存器等）的功能和位定义，并运用 C 语言位操作独立完成 GPIO 引脚的初始化配置（推挽输出模式）。

2. 基础编程实现能力：能运用 C 语言基础语法（变量、数据类型、运算符、控制流 if/for/while）编写程序，实现单个 LED 的点亮、熄灭、闪烁等基本控制逻辑。

3. 工程管理能力：能独立在 Keil MDK 环境下完成新工程的创建、源文件添加、基本编译选项配置、代码编译、下载到开发板以及利用调试器（如 ST-Link）进行单步调试、设置断点观察寄存器/变量值。

4. 模块化编程与接口运用能力：能理解 STM32 标准外设库中 GPIO 模块系统文件函数（如 GPIO_Init，GPIO_SetBits，GPIO_ResetBits，GPIO_WriteBit 等）的功能和接口定义，并运用这些函数重构 LED 控制程序，提升代码可读性和可移植性。

5. 理解位带控制知识，能运用位带方式控制 GPIO 口的输出电平状态，判断 GPIO 口的输入电平状态。

素质目标

1. 严谨细致与规范操作：养成在操作硬件（连接电路）和编写软件（配置寄存器、调用库函数）时一丝不苟、遵守规范的习惯，深刻理解“差之毫厘，谬以千里”在嵌入式开发中的体现，培养工程实践中的工匠精神。

2. 实践求真与科学探索：通过亲手搭建电路、编写代码、观察现象、调试问题，体验“实

践是检验真理的唯一标准”，培养从现象出发、通过实验验证假设、探究问题本质的科学思维方法和实证精神。

1. 利用开发板上一组 LED 灯，实现多种动态流水灯效果。

效果 1：顺序流水灯。LED 依次从左到右（或指定顺序）逐个点亮并熄灭，形成单向流动效果（如：LED1 亮-> 灭 & LED2 亮 -> … -> LEDn 亮 -> 灭 & LED1 亮…）。

效果 2：往返流水灯。LED 依次从左到右点亮至最右端，再从右到左点亮至最左端，如此循环往复。

2. 利用 GPIO 端口输出 BCD 码，配合 CD4511 芯片实现数码管的静态显示。

3. 实现多位一体的数码管动态显示。

选用两位一体的共阴极数码管，设计数码管动态驱动电路，实现两位数字在数码管上的动态稳定显示。

1.1 项目背景

嵌入式系统的显示模块是实现人机交互的关键。常用的显示方式有：LED 指示灯、数码管、LED 点阵屏和液晶显示等。

LED 指示灯最为基础，通过亮灭状态直观反映系统运行、报警等信息，驱动简单，成本极低。数码管由八段 LED 组成，能显示数字和部分字母，分为共阴与共阳两种结构，通常采用动态扫描驱动，以较少 I/O 口控制多位显示，在工业仪表中应用广泛。LED 点阵屏由多个 LED 按矩阵排列，通过行列扫描可显示自定义字符、简单图形甚至动画，常见于信息发布和零售价签。而 LCD 液晶显示屏是当前主流的显示方案，缺点是成本较高。其中字符型 LCD 用于显示文本，图形点阵 LCD（如 TFT）则能呈现丰富的彩色图像和用户界面，其驱动通常需要专用的控制器和显存。

1.1.1 LED 简介

LED（发光二极管）广泛应用于工业设备指示灯（如设备运行/故障状态指示）、消费电子数码管显示（如家电面板数字显示）、智能照明（如氛围灯、物联网控制灯具）等核心领域。而数码管作为由多个 LED 集成封装的显示器件，成为嵌入式数值显示场景（如计时器、抢答器组别显示）的关键组件。要深入理解二者在嵌入式系统中的应用逻辑，需先掌握其核心原理：

LED 本质是半导体发光器件，核心结构为 PN 结（由 P 型与 N 型半导体构成），外部封装包含正极引脚、负极引脚及透明/有色灯罩（聚焦光线），当 PN 结正向偏置（正极接高、负

极接低)时,外部电场打破载流子平衡,P区空穴与N区自由电子在PN结附近复合,能量差以光子形式释放形成可见光。光的颜色由半导体材料决定,如GaAs发红光、GaP发绿光、GaN发蓝光。图1.1给出了LED内部结构,正向导通时载流子复合发光原理如图1.2所示。

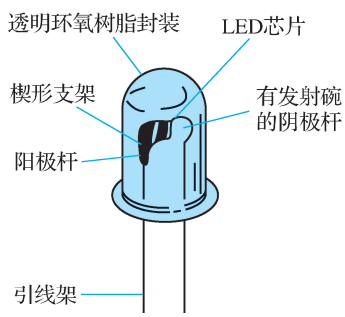


图1.1　LED内部结构示意图

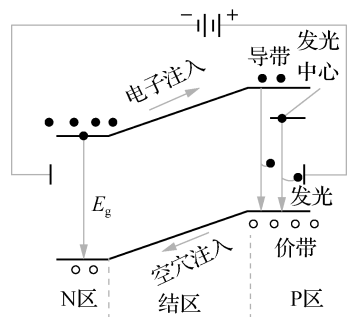


图1.2　LED正向导通时载流子复合发光原理

1.1.2　数码管简介

数码管结构组成(以两位共阴极为例):两位共阴极数码管本质是将16个独立LED(发光二极管)集成封装为一个模块化器件,具体结构可分为"内部LED阵列""引脚接口""封装外壳"三部分。

内部LED阵列(核心功能单元):如图1.3所示,数码管内部包含16个LED,按功能分为"段LED"和"位LED"两类,形成两组独立的显示单元(对应十位、个位)。

当某一位LED的阴极接地(共阴极特性)、阳极接高电平时,该位LED导通,对应的段LED阵列被激活,可通过段控制引脚控制段LED亮灭;若位LED阳极接低电平,则该位显示单元关闭,无论段控制引脚状态如何,均不显示。

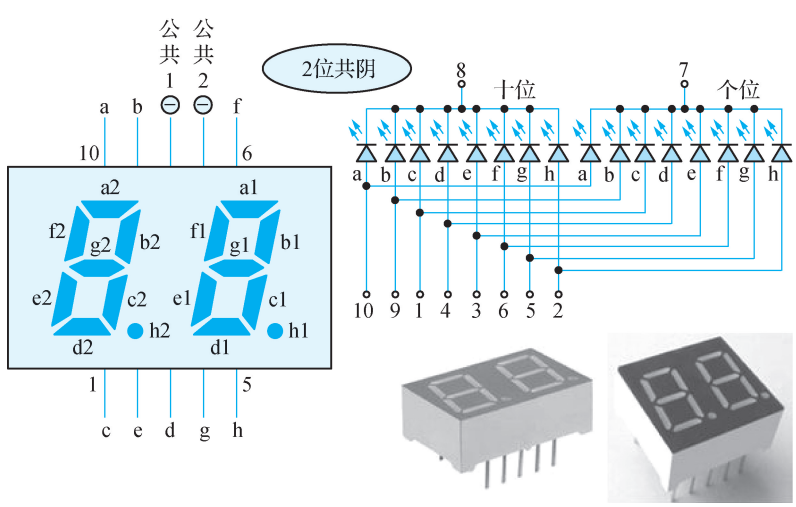


图1.3　两位共阴极数码管内部结构

引脚接口(外部控制端):标准2位共阴极数码管采用10脚直插式封装,引脚功能定义固定,便于硬件接线。段控制引脚(a~g、dp)为两组显示单元共用,即控制a段的引脚同时

连接十位和个位的 a 段 LED，因此，无法通过段引脚单独控制某一位的段 LED，需结合位选引脚实现“分时显示”。

数码管工作机制(动态刷新技术)

由于两位数码管的段控制引脚共用，若直接同时驱动两组显示单元，会导致两位显示相同数字(如控制 a 段亮时，十位和个位的 a 段均亮)，无法实现“十位显示、个位显示”的差异化效果。因此，需采用“动态刷新技术”，利用人眼视觉暂留效应(视觉残留时间约 100 ms)，通过快速切换位选引脚和段控制引脚，实现“两位数字同时显示”的视觉效果。

1.2 STM32 的 GPIO

1.2.1 STM32F10x 系列引脚分类

GPIO(General Purpose Input Output)是通用输入输出端口的简称，简单来说就是 STM32 可控制的引脚，STM32 芯片的 GPIO 引脚与外部设备连接起来，从而实现与外部通信、控制以及数据采集的功能。

STM32 芯片的 GPIO 被分成很多组，每组有 16 个引脚，如型号为 STM32F103VET6 型号的芯片有 GPIOA 至 GPIOE 共 5 组 GPIO，芯片一共 100 个引脚，其中 GPIO 就占了一大部分，所有的 GPIO 引脚都有基本的输入输出功能。除了 GPIO 引脚外，其他的还有电源、晶振、下载、BOOT 和复位引脚，以及 NC 引脚。STM32 芯片的引脚分类如表 1-1 所示。

表 1-1　STM32F10x 系列引脚分类

引脚分类	引脚说明
电源	(VBAT)、(VDD VSS)、(VDDA VSSA)、(VREF+ VREF−)等
晶振 IO	主晶振 IO，RTC 晶振 IO
下载 IO	用于 JTAG 下载的 IO：JTMS、JTCK、JTDI、JTDO、NJTRST
BOOT IO	BOOT0、BOOT1，用于设置系统的启动方式
复位 IO	NRST，用于外部复位
上面 5 部分 IO 组成的系统我们也叫作最小系统	
GPIO	专用器件接到专用的总线，比如 12C，SPI，SDIO，FSMC，DCMI 这些总线的器件需要接到专用的 IO
	普通的元器件接到 GPIO，比如蜂鸣器、LED、按键等元器件用普通的 GPIO
	如果还有剩下的 IO，可根据项目需要引出或者不引出

1.2.2 GPIO 引脚结构和输出寄存器

GPIO 是微控制器中最基本且最常用的外设接口之一，其核心功能在于可通过软件灵活配置为输入或输出模式，实现与外部电路的信号交互。

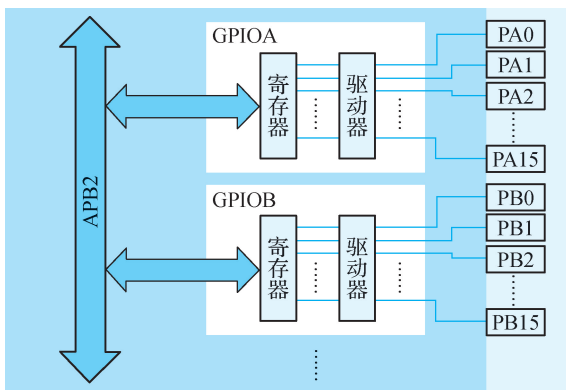


图 1.4 GPIO 基本结构

GPIO 基本结构如图 1.4 所示，在 STM32 中所有的 GPIO 都是挂载在 APB2 外设总线上，GPIO 外设按端口分组，命名为 GPIOA、GPIOB、GPIOC 等（具体数量取决于芯片型号）。每个 GPIO 外设，总共有 16 的引脚，编号是从 0 到 15，引脚采用“端口名＋引脚号”的方式标识，例如 GPIO 的第 0 号引脚，通常把它称作 PA0，接着第 1 号就是 PA1，然后 PA2，以此类推，一直到 PA15。

在每个 GPIO 模块内部，其核心组件主要包括寄存器组和输出驱动器。寄存器就是一段特殊的存储器，内核可以通过 APB2 总线对寄存器进行读写，内核通过读写这些寄存器来配置 GPIO 的工作模式、控制输出电平、读取输入电平状态。寄存器的每一位对应一个引脚，由于 STM32 是 32 位的单片机，所以 STM32 内部的寄存器都是 32 位的，但相应的端口只有 16 位，所以寄存器只有低 16 位对应的有端口，高 16 位与端口引脚无直接对应关系。寄存器操作逻辑分为输出控制和输入读取，其中输出寄存器写 1，对应的引脚就会输出高电平，写 0 就对应输出低电平，输入寄存器读取为 1，就证明对应的端口目前是高电平，读取为 0，就是低电平。驱动器主要负责增大驱动能力，确保能够可靠驱动外部负载（如 LED）。

GPIO 位结构的基本结构图如图 1.5 所示。其中左边这三个就是寄存器，中间这一部分就是驱动器，右边这个就是某一个 IO 口的引脚了。整个结构可以分为两个部分，上面是输入部分，下面是输出部分。

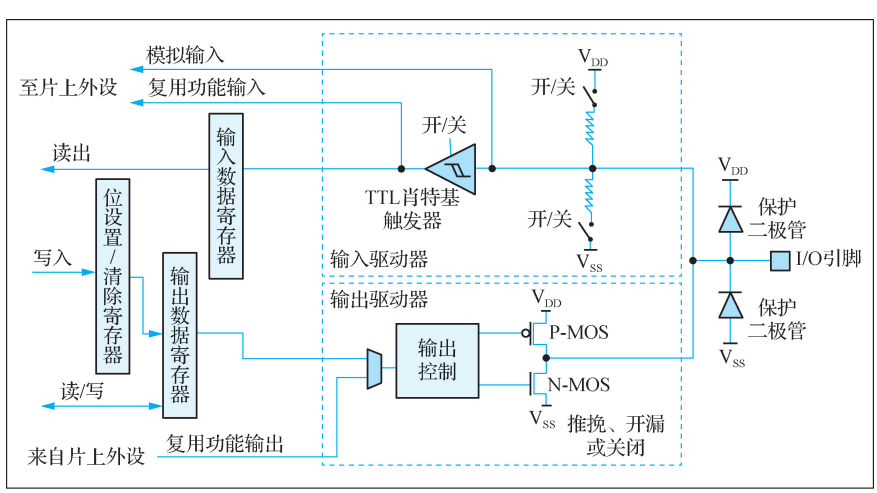


图 1.5 GPIO 位结构框图

输入部分：I/O 引脚内侧接了两个保护二极管，利用二极管的钳位作用对输入电压进行限幅。

输入通道连接了一组上拉电阻和下拉电阻，开关的通断可以通过程序进行配置。如果上面导通、下面断开，就是上拉输入模式；如果下面导通、上面断开，就是下拉输入模式；如果两个都断开，就是浮空输入模式。

输入通道中有个肖特基施密特触发器，这个施密特触发器的作用就是对输入电压进行整形，如果输入电压大于某一阈值，输出就会瞬间升为高电平；如果输入电压小于某一阈值，输出就会瞬间降为低电平。经过施密特触发器整形的波形可以直接写入输入数据寄存器，再用程序读取输入数据寄存器中对应某一位的数据，就可以知道端口的输入电平了。最上面还有两路线路，它们是连接到片上外设的一些端口，其中有模拟输入，用于 ADC 模块接收模拟量，由于是连接到 ADC 上的，所以这根线是接到施密特触发器前面的。另一个是复用功能输入，这个是连接到其他需要读取端口的外设上的，比如串口的输入引脚等，这根线接收的是数字量，所以要接到施密特触发器后面。

输出部分：数字部分可以由输出数据寄存器或片上外设控制，两种控制方式通过这个数据选择器接到了输出控制部分，如果选择通过输出数据寄存器进行控制，就是普通的 I/O 口输出，写这个数据寄存器的某一位就可以操作对应的某个端口了。最左边还有个位设置/清除寄存器，可以用来单独操作输出数据寄存器的某一位。第一种方式是先读出这个寄存器，然后用按位与和按位或的方式更改某一位，最后再将更改后的数据写回去。在 C 语言中就是&=和|=的操作，这种方法比较麻烦，效率不高，对于 IO 口的操作而言不太合适；第二种方式是设置位设置和位清除寄存器，如果我们要对某一位进行置 1 的操作，在位设置寄存器的对应位写 1 即可，剩下不需要操作的位写 0，这样它内部就会有电路，自动将输出数据寄存器中对应位置为 1，而剩下写 0 的位则保持不变，这样就保证了只操作其中某一位而不影响其他位，如果想对某一位进行清零的操作，就在位清除寄存器的对应位写 1 即可，这样内部电路就会把这一位清零了，也就是位设置和位清除寄存器的作用。

输出控制之后的接到两个 MOS 管，上面是 P-MOS，下面是 N-MOS，这个 MOS 管就是一种电子开关，输出的信号控制开关的导通和关闭，开关负责将 IO 口接到 VDD 或者 VSS，可以选择推挽、开漏或关闭三种输出方式。在推挽输出模式下，P-MOS 和 N-MOS 均有效，数据寄存器为 1 时，上管导通，下管断开，输出直接接到 VDD，就是输出高电平；数据寄存器为 0 时，上管断开，下管导通，输出直接接到 VSS，就是输出低电平。这种模式下，高低电平都有较强的驱动能力。在开漏输出模式下，没有上面的 P-MOS 开关，只有 N-MOS 在工作，数据寄存器为 1 时，下管断开，这时输出相当于断开，也就是高阻模式；数据寄存器为 0 时，下管导通，输出直接接到 VSS，也就是输出低电平，这种模式下，只有低电平有驱动能力。那这个模式有什么用呢？这个开漏模式可以作为通信协议的驱动方式，比如 I^2C 通信的引脚，就是使用的开漏模式。在多机通信的情况下，这个模式可以避免各个设备的相互干扰。第三种状态就是关闭，当引脚配置为输入模式的时候，这两个 MOS 管都无效，也就是输出关闭，端口的电平由外部信号来控制。

通过配置 GPIO 的端口配置寄存器，端口可以配置成 8 种模式，如表 1－2 所示。

表 1-2　GPIO 端口模式

模式名称	性质	特征
模拟输入	模拟输入	GPIO 无效，引脚直接接入内部 ADC
浮空输入	数字输入	可读取引脚电平，若引脚悬空，则电平不确定
下拉输入	数字输入	可读取引脚电平，内部连接下拉电阻，悬空时默认低电平
上拉输入	数字输入	可读取引脚电平，内部连接上拉电阻，悬空时默认高电平
推挽输出	数字输出	可输出引脚电平，高电平接 VDD，低电平接 VSS
开漏输出	数字输出	可输出引脚电平，高电平为高阻态，低电平接 VSS
复用推挽输出	数字输出	由片上外设控制，高电平接 VDD，低电平接 VSS
复用开漏输出	数字输出	由片上外设控制，高电平为高阻态，低电平接 VSS

1.2.3　GPIO 的寄存器

每个 GPIO 端口都有 7 个 32 位的寄存器，配置寄存器(GPIOx_CRL，GPIOx_CRH)，端口输入数据寄存器 GPIOx_IDR，端口输出数据寄存器 GPIOx_ODR，置位/复位寄存器(GPIOx_BSRR)，复位寄存器(GPIOx_BRR)，配置锁定寄存器(GPIOx_LCKR)。

```
typedef struct
{
  _IO uint32_t CRL;
  _IO uint32_t CRH;
  _IO uint32_t IDR;
  _IO uint32_t ODR;
  _IO uint32_t BSRR;
  _IO uint32_t BRR;
  _IO uint32_t LCKR;
} GPIO_TypeDef;
#define GPIOA              ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB              ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC              ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD              ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE              ((GPIO_TypeDef *) GPIOE_BASE)
```

一、端口配置低寄存器(GPIOx_CRL)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

二、端口配置高寄存器(GPIOx_CRH)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

端口配置低寄存器可以配置各组端口的 0～7 位引脚工作模式。端口配置高寄存器可以配置各组端口的 8～15 位引脚工作模式。MODEx[1：0]和 CNFx[1：0]的配置方式如表 1－3 所示。

表 1－3　GPIO 端口的配置方式

配置模式		CNFx[1：0]	MODEx[1：0]	PxODR 寄存器
通用输出	推挽输出	00	01:最大输出速度 10 MHz 10:最大输出速度 2 MHz 11:最大输出速度 50 MHz	0 或 1
	开漏输出	01		0 或 1
复用功能输出	推挽输出	10		不使用
	开漏输出	11		不使用
输入	模拟输入	00	00	不使用
	浮空输入	01		不使用
	下拉输入	10		0
	上拉输入			1

举例:以 PA0 引脚配置为例,如何将它配置成通用推挽输出,最大输出速度为 10 MHz 呢? PA0 处于引脚的低 8 位,应使用端口配置低寄存器 GPIOA_CRL 的 bit0～bit3 位。

```
1  // 清空控制 PA0 的端口位
2 GPIOA -> CRL &= ~( 0x0F << (4*0));
3  // 配置 PA0 为通用推挽输出,速度为 10M
4 GPIOA -> CRL |= (1 << 4 *0);
```

在以上的程序代码中,先将 PA0 的端口位控制寄存器的 CNF0 和 MODE0 清零,然后再将它赋值“0001b”,从而使 PA0 引脚设置成输出模式,最大输出速度为 10 MHz。代码中使用了&=～、|=这种操作方法是为了避免影响到寄存器中的其他位。

假设 PA0 引脚接到一个 LED 灯的阴极,使用低电平方式驱动 LED 灯。如何让设置好的 PA0 引脚输出低电平,以点亮 LED 灯,或输出高电平以熄灭 LED 灯呢? 我们再来学习几个重要的寄存器。

三、端口输入数据寄存器(GPIOx_IDR)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

端口输入数据寄存器的值反映了一组端口上所有输入引脚的电平高低状态。IDRy[15:0]:端口输入数据(y=0,…,15)。这些位为只读,并只能以字(16 位)的形式读出。读出的值为对应 I/O 口的状态。

四、端口输出数据寄存器(GPIOx_ODR)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

端口输出数据寄存器可以控制端口上对应输出引脚的输出高电平和低电平。ODRy[15:0]:端口输出数据(y=0,…,15)。这些位可读可写并只能以字(16 位)的形式操作。

五、端口位设置/清除寄存器(GPIOx_BSRR)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

端口位设置/清除寄存器的高 16 位用来清除输出数据寄存器的对应位为 0。BRy:清除端口 x 的位 y(y=0,…,15)。这些位只能写入并只能以字(16 位)的形式操作。0:对对应的 ODRy 位不产生影响;1:清除对应的 ODRy 位为 0。

注意:如果同时设置了 BSy 和 BRy 的对应位,BSy 位起作用。

端口位设置/清除寄存器的低 16 位用来设置输出数据寄存器的对应位为 1。BSy:设置端口 x 的位 y(y=0,…,15)。这些位只能写入并只能以字(16 位)的形式操作。0:对对应的 ODRy 位不产生影响;1:设置对应的 ODRy 位为 1。

六、端口位清除寄存器(GPIOx_BRR)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

端口位清除寄存器的高 16 位保留，低 16 位用来清除输出数据寄存器的对应位为 0。BRy:清除端口 x 的位 y(y=0,…,15)。这些位只能写入并只能以字(16 位)的形式操作。0:对对应的 ODRy 位不产生影响;1:清除对应的 ODRy 位为 0。

七、端口配置锁定寄存器(GPIOx_LCKR)(x=A,…,E)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															LCKK
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

端口配置锁定寄存器用来锁定端口位的配置。LCKK 是锁键。该位可随时读出，它只可通过锁键写入序列修改。锁键的写入序列为：

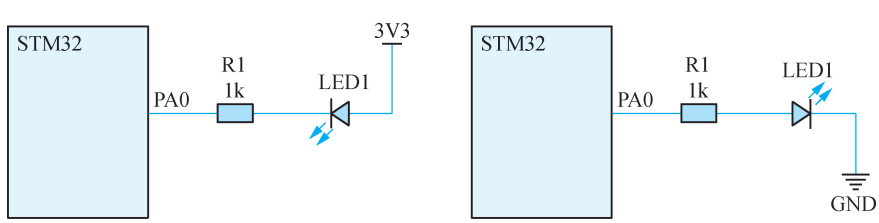
写 1 -> 写 0 -> 写 1 -> 读 0 -> 读 1。

最后一个读操作可省略，但可以用来确认锁键已被激活。

注意:在操作锁键的写入序列时，不能改变 LCK[15：0]的值。操作锁键写入序列中的任何错误将不能激活锁键。

LCKy 是端口 x 的锁位 y(y=0,…,15)。这些位可读可写但只能在 LCKK 位为 0 时写入。为 1 时锁定端口的配置，为 0 时结束锁定操作。

LED 灯的驱动方式有两种，如图 1.6 所示。一种是低电平驱动方式，另一种是高电平驱动方式。



(a) 低电平驱动　　(b) 高电平驱动

图 1.6　LED 的驱动电路

如何使用寄存器点亮 LED 灯呢？学习了以上的寄存器，我们就可以控制 PA0 引脚输出低电平和高电平，实现 LED 灯点亮和熄灭了。

```
// 使用输出数据寄存器控制 PA0 输出低电平
GPIOA -> ODR &= ~(1 << 0);
// 使用输出数据寄存器控制 PA0 输出高电平
GPIOA -> ODR |= (1 <<*0);
// 使用端口位设置/清除寄存器控制 PA0 输出低电平
GPIOA -> BSRR &= (1 <<*16);
// 使用端口位设置/清除寄存器控制 PA0 输出高电平
GPIOA -> BSRR |= (1 << 0);
// 使用端口位清除寄存器控制 PA0 输出低电平
GPIOA -> BRR &= (1 << 0);
```

可以看出，ODR 和 BSRR 寄存器能实现引脚输出高电平；ODR、BSRR 和 BRR 寄存器能实现引脚输出低电平。

其实，以上操作暂时还不能控制 PA0 引脚的输出状态，还需要打开 A 端口的时钟。

由于 STM32 的外设很多，为了降低功耗，每个外设都对应着一个时钟，在芯片刚上电的时候这些时钟都是被关闭的，如果想要外设工作，必须把相应的时钟打开。STM32 的所有外设的时钟由一个专门的外设来管理，叫 RCC（Reset and Clock Control），所有的 GPIO 都挂载到 APB2 总线上，具体的时钟由 APB2 外设时钟使能寄存器（RCC_APB2ENR）来控制。

八、APB2 外设时钟使能寄存器（RCC_APB2ENR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADC3 EN	USART 1 EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	保留	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

如果我们使用 PA0 引脚，就需要打开 A 端口的时钟。

```
// 开启 GPIOA 端口时钟
RCC -> APB2ENR |= (1 << 2);
```

这样就可以控制 PA0 引脚输出高、低电平了。

1.2.4 GPIO 的库函数

我们已经掌握使用寄存器配置 GPIO 口的工作方式和控制引脚输出高低不同的状态电平。STM32 的寄存器数量众多，采用控制寄存器方式编写程序，需要花费大量时间查阅寄存器的信息，对设计程序来说很不方便。STM32 官方提供了所有外设的库函数，为编程者

提供一种便利的设计方式。

表 1－4 中列举和描述了固件函数库使用的所有文件。每一个外设都有一个对应的源文件：stm32f10x_ppp. c 和一个对应的头文件：stm32f10x_ppp. h。文件 stm32f10x_ppp. c 包含了使用外设 PPP 所需的所有固件函数。提供所有外设一个存储器映像文件 stm32f10x_map. h。它包含了所有寄存器的声明，既可以用于 debug 模式，也可以用于 release 模式。头文件 stm32f10x_lib. h 包含了所有外设头文件的头文件。它是唯一一个用户需要包括在自己应用中的文件，起到应用和库之间界面的作用。文件 stm32f10x_conf. h 是唯一一个需要由用户修改的文件。它作为应用和库之间的界面，指定了一系列参数。

表 1－4　固件函数库文件描述

文件名	描述
stm32f10x_conf. h	参数设置文件，起到应用和库之间界面的作用。用户必须在运行自己的程序前修改该文件。用户可以利用模板使能或者失能外设。也可以修改外部晶振的参数。也可以是用该文件在编译前使能 debug 或者 release 模式。
stm32f10x_it. h	头文件，包含所有中断处理函数原型。
stm32f10x_it. c	外设中断函数文件。用户可以加入自己的中断程序代码。对于指向同一个中断向量的多个不同中断请求，可以利用函数通过判断外设的中断标志位来确定准确的中断源。固件函数库提供了这些函数的名称。
stm32f10x_lib. h	包含了所有外设的头文件的头文件。它是唯一一个用户需要包括在自己应用中的文件，起到应用和库之间界面的作用。
stm32f10x_lib. c	debug 模式初始化文件。它包括多个指针的定义，每个指针指向特定外设的首地址，以及在 debug 模式被使能时，被调用的函数的定义。
stm32f10x_map. h	该文件包含了存储器映像和所有寄存器物理地址的声明，既可以用于 debug 模式，也可以用于 release 模式。所有外设都使用该文件。
stm32f10x_type. h	通用声明文件。包含所有外设驱动使用的通用类型和常数。
stm32f10x_ppp. c	由 C 语言编写的外设 PPP 的驱动源程序文件。
stm32f10x_ppp. h	外设 PPP 的头文件。包含外设 PPP 函数的定义和这些函数使用的变量。
cortexm3_macro. h	文件 cortexm3_macro. s 的头文件。
cortexm3_macro. s	Cortex-M3 内核特殊指令的指令包装。

STM32 系列 GPIO 的相关库函数如表 1－5 所示。

表 1－5　GPIO 的库函数

函数名	功能
void GPIO_DeInit();	将外设 GPIOx 寄存器重设为缺省值
void GPIO_AFIODeInit();	将复用功能(重映射事件控制和 EXTI 设置)重设为缺省值

续 表

函数名	功能
void GPIO_Init();	根据 GPIO_InitStruct 指定的参数初始化外设 GPIOx 寄存器
void GPIO_StructInit();	把 GPIO_InitStruct 中的每一个参数按缺省值填入
uint8_t GPIO_ReadInputDataBit();	读取指定端口管脚的输入
uint16_t GPIO_ReadInputData();	读取指定的 GPIO 端口输入
uint8_t GPIO_ReadOutputDataBit();	读取指定端口管脚的输出
uint16_t GPIO_ReadOutputData();	读取指定的 GPIO 端口输出
void GPIO_SetBits();	设置指定的数据端口位
void GPIO_ResetBits();	清除指定的数据端口位
void GPIO_WriteBit();	设置或者清除指定的数据端口位
void GPIO_Write();	向指定 GPIO 数据端口写入数据
void GPIO_PinLockConfig();	锁定 GPIO 管脚设置寄存器
void GPIO_EventOutputConfig();	选择 GPIO 管脚用作事件输出
void GPIO_EventOutputCmd();	使能或者失能事件输出
void GPIO_PinRemapConfig();	改变指定管脚的映射
void GPIO_EXTILineConfig();	选择 GPIO 管脚用作外部中断线路

下面详细说明几个常用的 GPIO 库函数。

一、GPIO_Init()

函数名	void GPIO_Init(GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct)
功能描述	根据 GPIO_InitStruct 中指定的参数初始化外设 GPIOx 寄存器
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	GPIO_InitStruct:指向结构 GPIO_InitTypeDef 的指针,包含了外设 GPIO 的配置信息
输出参数	无
返回值	无

GPIO_InitTypeDef 定义于文件“stm32f10x_gpio.h”中,包含了 3 个成员:

```
typedef struct
{
   u16 GPIO_Pin;
   GPIOSpeed_TypeDef GPIO_Speed;
   GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

GPIO_Pin 是选择引脚的参数，定义于文件“stm32f10x_gpio. h”中，定义情况为：

```
#define GPIO_Pin_0                 ((uint16_t)0x0001)  /*! < Pin 0 selected */
#define GPIO_Pin_1                 ((uint16_t)0x0002)  /*! < Pin 1 selected */
#define GPIO_Pin_2                 ((uint16_t)0x0004)  /*! < Pin 2 selected */
#define GPIO_Pin_3                 ((uint16_t)0x0008)  /*! < Pin 3 selected */
#define GPIO_Pin_4                 ((uint16_t)0x0010)  /*! < Pin 4 selected */
#define GPIO_Pin_5                 ((uint16_t)0x0020)  /*! < Pin 5 selected */
#define GPIO_Pin_6                 ((uint16_t)0x0040)  /*! < Pin 6 selected */
#define GPIO_Pin_7                 ((uint16_t)0x0080)  /*! < Pin 7 selected */
#define GPIO_Pin_8                 ((uint16_t)0x0100)  /*! < Pin 8 selected */
#define GPIO_Pin_9                 ((uint16_t)0x0200)  /*! < Pin 9 selected */
#define GPIO_Pin_10                ((uint16_t)0x0400)  /*! < Pin 10 selected */
#define GPIO_Pin_11                ((uint16_t)0x0800)  /*! < Pin 11 selected */
#define GPIO_Pin_12                ((uint16_t)0x1000)  /*! < Pin 12 selected */
#define GPIO_Pin_13                ((uint16_t)0x2000)  /*! < Pin 13 selected */
#define GPIO_Pin_14                ((uint16_t)0x4000)  /*! < Pin 14 selected */
#define GPIO_Pin_15                ((uint16_t)0x8000)  /*! < Pin 15 selected */
#define GPIO_Pin_All               ((uint16_t)0xFFFF)  /*! < All pins selected */
```

这样就可以方便地选择对应的引脚了。如果同时使用多个引脚，可以使用“|”符号将多个引脚连接起来。比如，当使用 PA0、PA1 和 PA2 引脚时，可以直接写成

```
GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2
```

比直接操作寄存器方便多了。

GPIO_Speed 用于设置选中引脚的速率，定义于文件“stm32f10x_gpio. h”中，有三种速率挡位可选。

```
typedef enum
{
  GPIO_Speed_10MHz = 1;
  GPIO_Speed_2MHz;
  GPIO_Speed_50MHz;
}GPIOSpeed_TypeDef;
```

GPIO_Mode 用于设置选中引脚的工作状态，定义于文件“stm32f10x_gpio. h”中，有 4

种输入模式和 4 种输出模式。

```
typedef enum
{
  GPIO_Mode_AIN = 0x0;
  GPIO_Mode_IN_FLOATING = 0x04;
  GPIO_Mode_IPD = 0x28;
  GPIO_Mode_IPU = 0x48;
  GPIO_Mode_Out_OD = 0x14;
  GPIO_Mode_Out_PP = 0x10;
  GPIO_Mode_AF_OD = 0x1C;
  GPIO_Mode_AF_PP = 0x18;
}GPIOMode_TypeDef;
```

二、GPIO_ReadOutputDataBit( )

函数名	u8 GPIO_ReadOutputDataBit(GPIO_TypeDef * GPIOx, u16 GPIO_Pin)
功能描述	读取指定端口管脚的输出
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	GPIO_Pin:待读取的端口位
输出参数	输出端口管脚值
返回值	无

三、GPIO_ReadOutputData( )

函数名	u16 GPIO_ReadOutputData(GPIO_TypeDef * GPIOx)
功能描述	读取指定的 GPIO 端口输出
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	无
输出参数	GPIO 输出数据端口值
返回值	无

四、GPIO_SetBits( )

函数名	void GPIO_SetBits(GPIO_TypeDef * GPIOx, u16 GPIO_Pin)
功能描述	设置指定的数据端口位
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	GPIO_Pin:待设置的端口位
输出参数	无
返回值	无

五、GPIO_ResetBits( )

函数名	void GPIO_ResetBits(GPIO_TypeDef * GPIOx, u16 GPIO_Pin)
功能描述	清除指定的数据端口位
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	GPIO_Pin:待清除的端口位
输出参数	无
返回值	无

六、GPIO_WriteBit( )

函数名	void GPIO_WriteBit(GPIO_TypeDef * GPIOx, u16 GPIO_Pin, BitAction BitVal)
功能描述	设置或者清除指定的数据端口位
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	GPIO_Pin:待设置的端口位
输入参数 3	BitVal:该参数指定了待写入的值:Bit_RESET:清除数据端口位;Bit_SET:设置数据端口位
输出参数	无
返回值	无

其中 BitAction 的定义如下：

```
typedef enum
{
   Bit_RESET = 0;
   Bit_SET;
}BitAction;
```

七、GPIO_Write( )

函数名	void GPIO_Write(GPIO_TypeDef * GPIOx, u16 PortVal)
功能描述	向指定 GPIO 数据端口写入数据
输入参数 1	GPIOx:x 可以是 A,B,C,D 或者 E,来选择 GPIO 外设
输入参数 2	PortVal:待写入端口数据寄存器的值
输出参数	无
返回值	无

1.2.5 GPIO 的位带操作

除了使用寄存器和库函数方式控制 GPIO 的输出引脚输出高低电平外，STM32 单片机还提供了 GPIO 的位带操作方式。

支持了位带操作后，可以使用普通的加载/存储指令来对单一的比特进行读写。在 CM3 中，有两个区中实现了位带。其中一个是 SRAM 区的最低 1MB 范围（0x2000_0000-0x200F_FFFF），第二个则是片内外设区的最低 1MB 范围（0x4000_0000-0x400F_FFFF）。这两个区中的地址除了可以像普通的 RAM 一样使用外，它们还都有自己的“位带别名区”，位带别名区把每个比特膨胀成一个 32 位的字。通过位带别名区访问这些字时，就可以达到访问原始比特的目的。

外设位带区覆盖了所有片上外设寄存器，理论上可通过宏为每个寄存器位定义别名地址。下面以输出数据寄存器 ODR、输入数据寄存器 IDR 说明 STM32 的 GPIO 口的位操作方式。

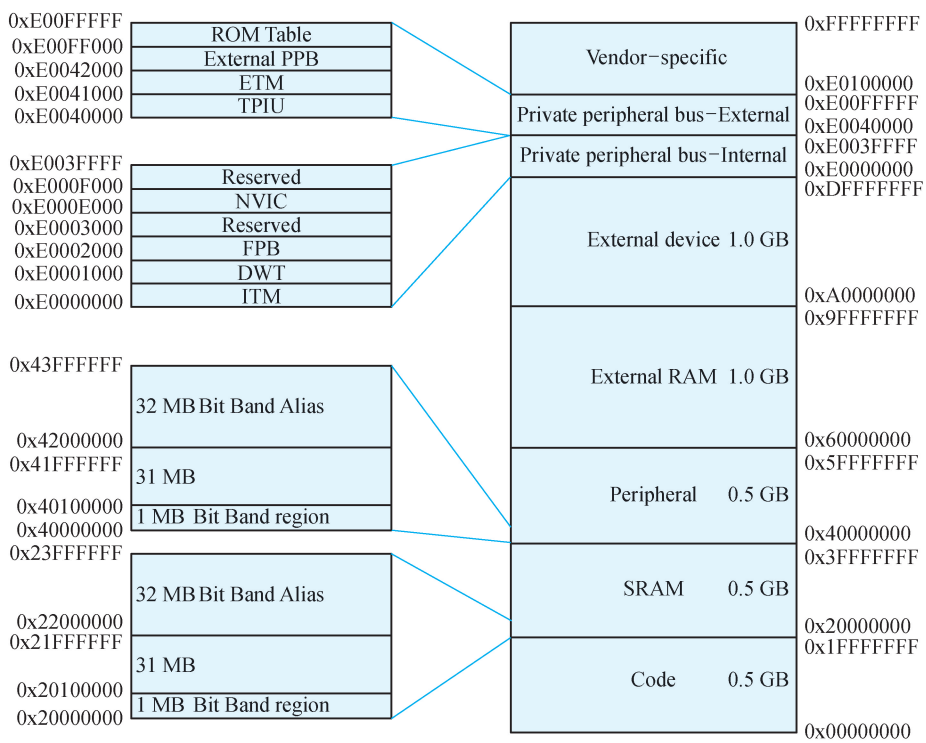


图 1.7 Cortex M3 存储器的映射

表 1-6 GPIO 端口寄存器的映射

寄存器起始地址	端口号
0x4001 1800-0x4001 1BFF	GPIO 端口 E
0x4001 1400-0x4001 17FF	GPIO 端口 D
0x4001 1000-0x4001 13FF	GPIO 端口 C
0X4001 0C00-0x4001 0FFF	GPIO 端口 B
0x4001 0800-0x4001 0BFF	GPIO 端口 A

```
// IO 口操作宏定义
#define BITBAND(addr, bitnum) ((addr & 0xF0000000) + 0x2000000 + ((addr & 0xFFFFF) << 5)
+ (bitnum << 2))
#define MEM_ADDR(addr)  *((volatile unsigned long  *) (addr))
#define BIT_ADDR(addr, bitnum)  MEM_ADDR(BITBAND(addr, bitnum))
// GPIOx_ODR 寄存器地址映射
#define GPIOA_ODR_Addr    (GPIOA_BASE + 12) // 0x4001080C
#define GPIOB_ODR_Addr    (GPIOB_BASE + 12) // 0x40010C0C
#define GPIOC_ODR_Addr    (GPIOC_BASE + 12) // 0x4001100C
#define GPIOD_ODR_Addr    (GPIOD_BASE + 12) // 0x4001140C
#define GPIOE_ODR_Addr    (GPIOE_BASE + 12) // 0x4001180C
#define GPIOF_ODR_Addr    (GPIOF_BASE + 12) // 0x40011A0C
#define GPIOG_ODR_Addr    (GPIOG_BASE + 12) // 0x40011E0C
// GPIOx_IDR 寄存器地址映射
#define GPIOA_IDR_Addr    (GPIOA_BASE + 8)  // 0x40010808
#define GPIOB_IDR_Addr    (GPIOB_BASE + 8)  // 0x40010C08
#define GPIOC_IDR_Addr    (GPIOC_BASE + 8)  // 0x40011008
#define GPIOD_IDR_Addr    (GPIOD_BASE + 8)  // 0x40011408
#define GPIOE_IDR_Addr    (GPIOE_BASE + 8)  // 0x40011808
#define GPIOF_IDR_Addr    (GPIOF_BASE + 8)  // 0x40011A08
#define GPIOG_IDR_Addr    (GPIOG_BASE + 8)  // 0x40011E08
// IO 口操作,只对单一的 IO 口!
//确保 n 的值小于 16!
#define Pxout(n)   BIT_ADDR(GPIOx_ODR_Addr,n)  //输出
#define Pxin(n)    BIT_ADDR(GPIOx_IDR_Addr,n)  //输入
```

这样就可以使用位操作的方式控制 IO 口输出高低电平了。

例:使用位带操作控制 PA0 引脚输出高电平。

```
PAout(0) = 1;
```

三种操作方式的特点:使用位段功能与直接操作寄存器速度差不多,使用库函数速度明显慢得多,可以使用软件仿真调试分别查看三种方式的执行时间。

1.2.6　特殊的 GPIO 引脚

在 STM32 的学习和项目实践中,我们可能会遇到代码逻辑完全正确,但某个引脚就是没有按照程序的指令输出相应的电平。这时,很可能是我们误用了 STM32 上的一些功能特殊的引脚。“特殊”的原因是这些引脚的默认功能并不是 GPIO 口。由于 STM32 的引脚数量有限,但需要实现各种丰富的功能。为了能在有限的引脚上实现更多功能,芯片设计者采用了“引脚复用”技术,即一个物理引脚同时对应多种不同的功能。其中默认与调试接口和 RTC 时钟复用的引脚就是常见的特殊引脚。

一、调试接口

以 STM32F103VET 为例，所用的调试接口如表 1－7 所示：

表 1－7　STM32 的 JTAG/SWD 调试引脚

引脚序号	引脚名称	默认功能	重映射功能
72	PA13	JTMS/SWDIO	PA13
76	PA14	JTCK/SWCLK	PA14
77	PA15	JTDI	TIM2_CH1_ETR/PA15/SPI1_NSS
89	PB3	JTDO	TIM2_CH2/PB3/TRACESWO/SPI1_SCK
90	PB4	JNTRST	TIM3_CH1/PB4/SPI1_MISO

JTAG(Joint Test Action Group)：传统的调试接口，需要用到全部 5 个引脚。具有芯片测试、系统调试、边界扫描技术功能。

SWD(Serial Wire Debug)：现代最常用的调试方式，仅需 PA13(SWDIO)和 PA14(SWCLK)两根线。相比传统 JTAG 接口减少了硬件连接复杂度，适用于系统级调试、生产编程等场景。

假如将 PA13 设置为低电平，点亮 LED 程序如下：

```
#include "stm32f10x.h"
int main (void)
{
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);
  GPIO_InitTypeDef GPIO_InitStructure;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  GPIO_ResetBits(GPIOA, GPIO_Pin_13);
  while(1)  {  };
}
```

以上程序并不能使 PA13 输出低电平。PA13 默认是 JTMS/SWDIO 功能，而不是 PA13 的 GPIO 普通功能。为了使用 PA13 的 GPIO 功能，可以使用引脚重映射方式。即在初始化代码的最开始，使能 AFIO 时钟并进行“重映射”。

```
/*使能 AFIO 时钟，重映射必备。*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
/*禁用 JTAG，释放 PA15, PB3, PB4；但保留 SWD 下载功能。*/
GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable,  ENABLE);
```

二、RTC 时钟引脚

以 STM32F103VET 为例，RTC 时钟引脚如表 1－8 所示：

表 1－8　STM32 的 RTC 时钟引脚

引脚序号	引脚名称	默认功能	重映射功能
7	PC13	TAMPER-RTC	—
8	PC14	OSC32_IN	—
9	PC15	OSC32_OUT	—

STM32 的 PC13、PC14 和 PC15 引脚属于备份电源域，具有严格的电气限制。这三个引脚共用一个驱动能力极弱的电源开关，总输出电流被限制在 3 mA 以内，它们的输出速度不得超过 2 MHz，且不能同时作为输出使用。

1.3　基于 Keil C 的程序设计基础

1.3.1　Keil C 工程的创建

目前 STM32 的开发平台多采用 Keil C，下面介绍 Keil 工程创建、调试、下载的步骤。

（1）首先在 STM32 社区下载 STM32 官方标准库提供的固件库文件，下载解压之后得到如下文件：

图 1.8　固件库文件夹内容

第一个文件夹里是两个图片；

第二个 Libraries 里面就是库函数的文件了，之后建工程时会用到；

第三个 Project 是官方提供的工程示例和模板，以后使用库函数的时候可以参考一下；

第四个 Utilities 是 STM32 官方评估板的相关例程，这个评估板就是官方用 STM32 做的一个小电路板，用来测评 STM32，存放的是测评程序。

（2）准备建工程所需要的文件夹，打开 Keil，点击 new uVision project，新建工程

STM32(原来有工程需要关闭),保存在刚刚新建的 project 文件夹里面。

第一步　可以更改成你想要的工程名称,可以起一个通用名称,比如 Project,然后点击保存。

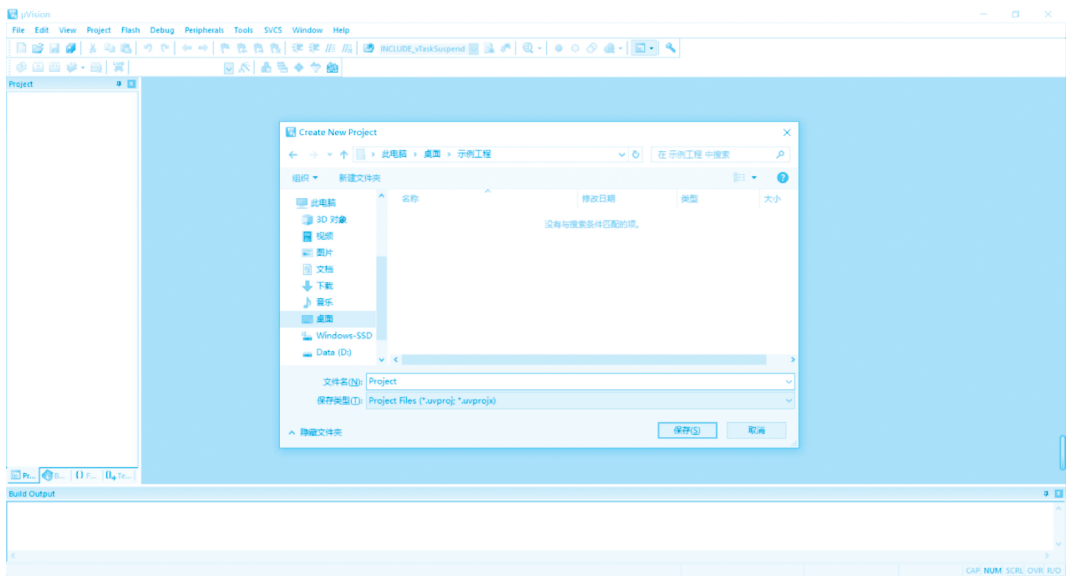


图 1.9　新建工程界面

第二步　选择器件型号,这里以 STM32F103C8T6 为例。

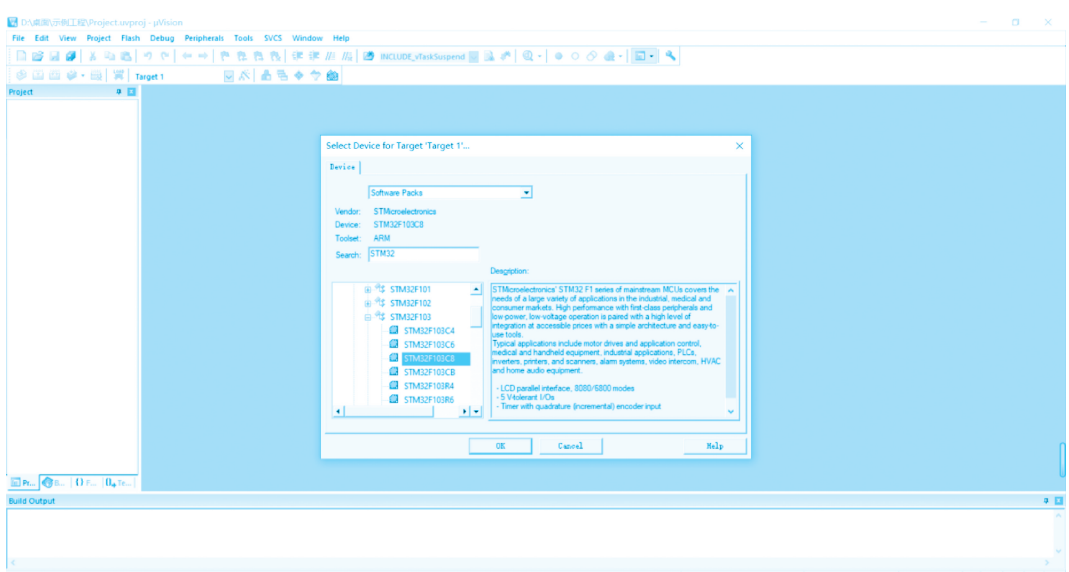


图 1.10　新建工程型号选择

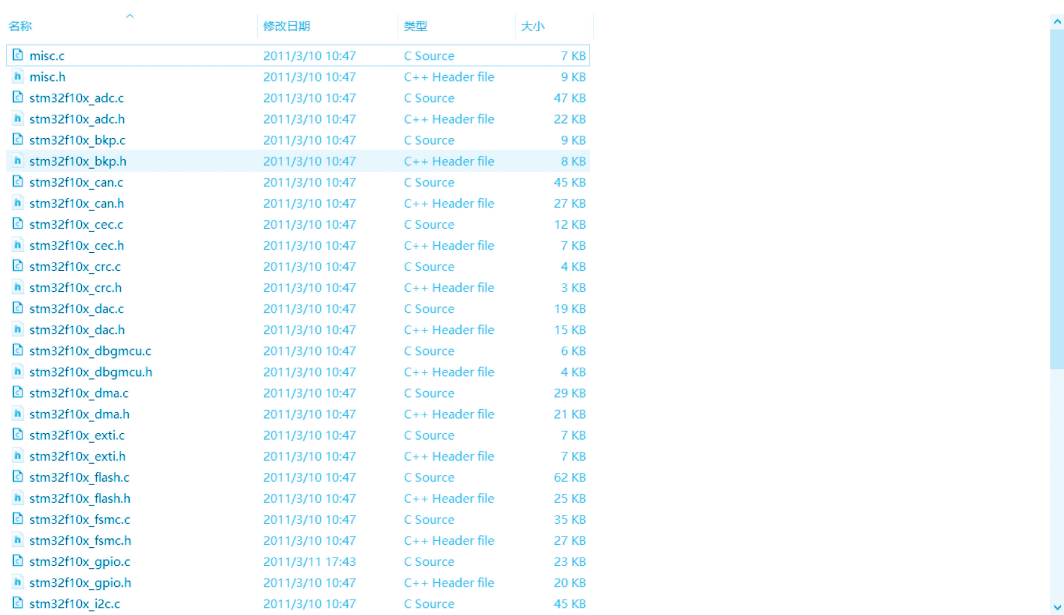
第三步　添加工程文件,在刚才创建的文件夹中新建三个文件夹 Start(存放内核函数及启动引导文件),User(存放用户自己的函数),Library(存放库函数),打开刚才下载的官方标准库,将 Libraries\CMSIS\CM3\CoreSupport 中的文件和 Libraries\CMSIS\CM3\

DeviceSupport\ST\STM32F10x 中除 startup 文件夹以及 Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm 中的文件全部复制到刚才新建的 Start 中，如图 1.11 所示。

名称	修改日期	类型	大小
core_cm3.c	2010/6/7 10:25	C Source	17 KB
core_cm3.h	2011/2/9 14:59	C++ Header file	84 KB
startup_stm32f10x_cl.s	2011/3/10 10:52	Assembler Source	16 KB
startup_stm32f10x_hd.s	2011/3/10 10:52	Assembler Source	16 KB
startup_stm32f10x_hd_vl.s	2011/3/10 10:52	Assembler Source	16 KB
startup_stm32f10x_ld.s	2011/3/10 10:52	Assembler Source	13 KB
startup_stm32f10x_ld_vl.s	2011/3/10 10:52	Assembler Source	14 KB
startup_stm32f10x_md.s	2011/3/10 10:52	Assembler Source	13 KB
startup_stm32f10x_md_vl.s	2011/3/10 10:51	Assembler Source	14 KB
startup_stm32f10x_xl.s	2011/3/10 10:51	Assembler Source	16 KB
stm32f10x.h	2011/3/10 10:51	C++ Header file	620 KB
system_stm32f10x.c	2011/3/10 10:51	C Source	36 KB
system_stm32f10x.h	2011/3/10 10:51	C++ Header file	3 KB

图 1.11　内核函数及启动引导文件

接着将官方库中 Libraries\STM32F10x_StdPeriph_Driver 中的 inc 和 src 文件夹中的所有内容复制到刚才新建的 Library 文件夹中，如图 1.12 所示。



名称	修改日期	类型	大小
misc.c	2011/3/10 10:47	C Source	7 KB
misc.h	2011/3/10 10:47	C++ Header file	9 KB
stm32f10x_adc.c	2011/3/10 10:47	C Source	47 KB
stm32f10x_adc.h	2011/3/10 10:47	C++ Header file	22 KB
stm32f10x_bkp.c	2011/3/10 10:47	C Source	9 KB
stm32f10x_bkp.h	2011/3/10 10:47	C++ Header file	8 KB
stm32f10x_can.c	2011/3/10 10:47	C Source	45 KB
stm32f10x_can.h	2011/3/10 10:47	C++ Header file	27 KB
stm32f10x_cec.c	2011/3/10 10:47	C Source	12 KB
stm32f10x_cec.h	2011/3/10 10:47	C++ Header file	7 KB
stm32f10x_crc.c	2011/3/10 10:47	C Source	4 KB
stm32f10x_crc.h	2011/3/10 10:47	C++ Header file	3 KB
stm32f10x_dac.c	2011/3/10 10:47	C Source	19 KB
stm32f10x_dac.h	2011/3/10 10:47	C++ Header file	15 KB
stm32f10x_dbgmcu.c	2011/3/10 10:47	C Source	6 KB
stm32f10x_dbgmcu.h	2011/3/10 10:47	C++ Header file	4 KB
stm32f10x_dma.c	2011/3/10 10:47	C Source	29 KB
stm32f10x_dma.h	2011/3/10 10:47	C++ Header file	21 KB
stm32f10x_exti.c	2011/3/10 10:47	C Source	7 KB
stm32f10x_exti.h	2011/3/10 10:47	C++ Header file	7 KB
stm32f10x_flash.c	2011/3/10 10:47	C Source	62 KB
stm32f10x_flash.h	2011/3/10 10:47	C++ Header file	25 KB
stm32f10x_fsmc.c	2011/3/10 10:47	C Source	35 KB
stm32f10x_fsmc.h	2011/3/10 10:47	C++ Header file	27 KB
stm32f10x_gpio.c	2011/3/11 17:43	C Source	23 KB
stm32f10x_gpio.h	2011/3/10 10:47	C++ Header file	20 KB
stm32f10x_i2c.c	2011/3/10 10:47	C Source	45 KB

图 1.12　库函数文件

然后将官方库中的 Project\STM32F10x_StdPeriph_Template 中的这几个文件复制到新建的 User 中。

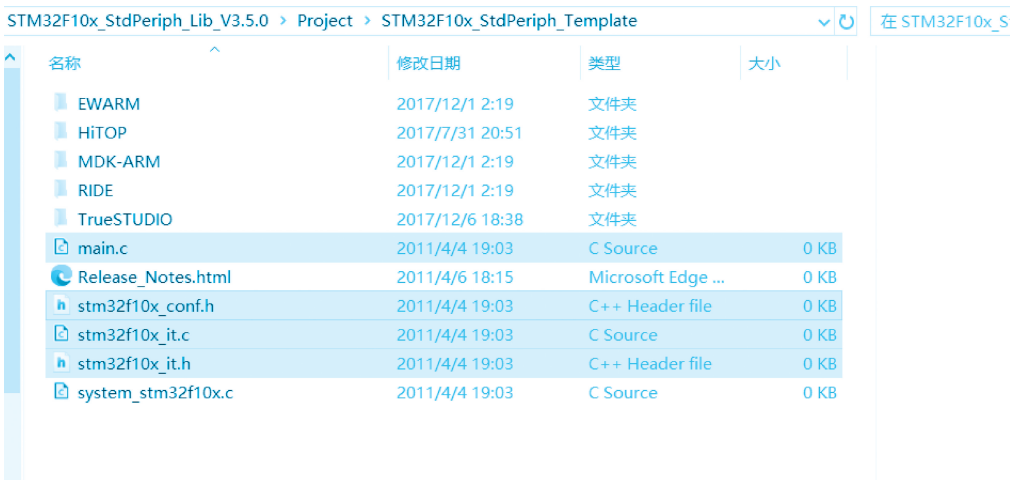


图 1.13　用户函数文件

最后打开 Keil，把我们刚才复制的那些文件添加到工程里，点红线圈出的图表，然后删除掉 SourceGroup1，在 Groups 中依次添加 Start、Library、User，把刚才复制到文件夹中的文件都添加到对应文件夹中，其中 Start 只需添加 startup_stm32f10x_md. s 启动文件，后续会补充说明。

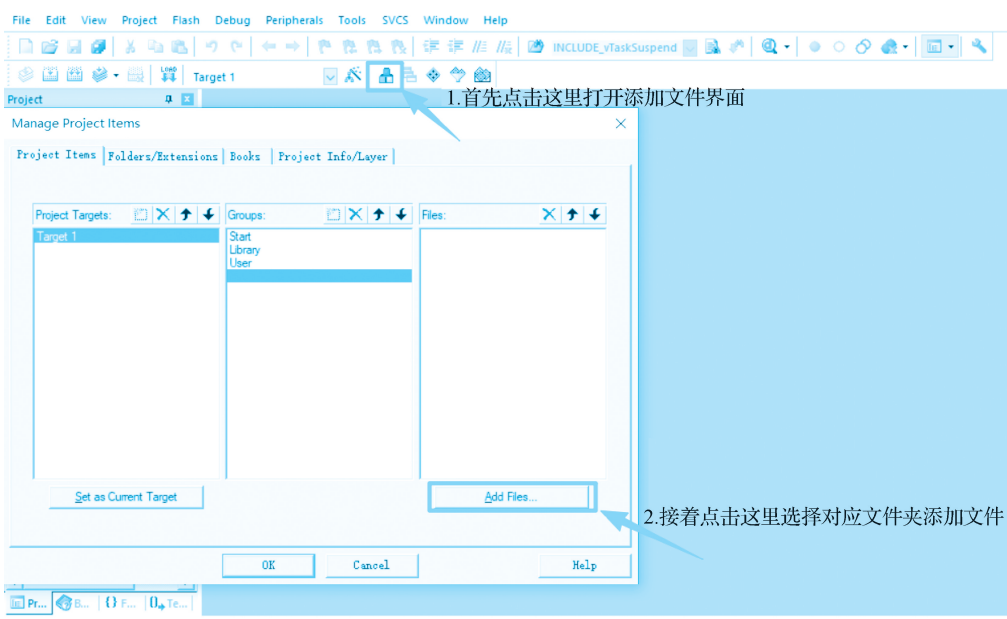


图 1.14　添加工程文件

第四步　添加完毕点击 OK。然后点击魔法棒，进入 C/C++设置界面，在 Define 一栏输入 USE_STDPERIPH_DRIVER，STM32F10X_HD，并在 include path 栏加入头文件路径 .\Start;.\Library;.\User，这样，整体基于库函数的工程就建好了，使用编译，最后无报错无警告就代表工程建立成功。

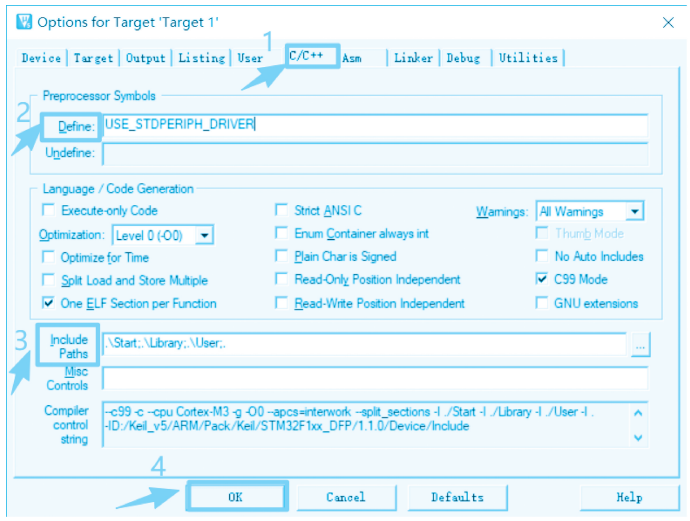


图 1.15　魔法棒设置

最后工程建立完成，总结一下整体步骤：

第一步　建立工程文件夹，Keil 中新建工程，选择对应型号；

第二步　工程文件夹里建立 Start、Library、User 等文件夹，复制固件库里面的文件到工程文件夹；

第三步　工程里对应建立 Start、Library、User 等同名称的分组，然后将文件夹内的文件添加到工程分组里；

第四步　工程选项，C/C++，Include Paths 内声明所有包含头文件的文件夹；

第五步　工程选项，C/C++，Define 内定义 USE_STDPERIPH_DRIVER；

第六步　工程选项，Debug，下拉列表选择对应调试器，Settings，Flash Download 里勾选 Reset and Run；

第七步　工程调试器设置完成，点击下载按钮 Download，下载完毕整体调试成功。

补充说明启动文件的类型选择：

表 1-9　启动文件型号分类

缩写	名称	Flash 容量	型号
LD_VL	小容量产品超值系列	16～32 K	STM32F100
MD_VL	中容量产品超值系列	64～128 K	STM32F100
HD_VL	大容量产品超值系列	256～512 K	STM32F100
LD	小容量产品	16～32 K	STM32F101/102/103
MD	中容量产品	64～128 K	STM32F101/102/103
HD	大容量产品	256～512 K	STM32F101/102/103
XL	加大容量产品	大于 512 K	STM32F101/102/103
CL	互联型产品	—	STM32F105/107

根据表 1－9 启动文件型号分类，如果用户使用 STM32F100 的型号，就选择带 VL 的启动文件，然后再根据 Flash 的大小选择 LD、MD，还是 HD；如果用户使用 STM32F101/102/103 的型号，就选择不带 VL 的，然后根据 Flash 的大小选择 LD、MD、HD，还是 XL；如果用户使用 STM32F105/107 的型号，直接选择 CL 的启动文件即可。

1.3.2 C 语言基本语法

一、C 语言数据类型

C 语言数据类型如表 1－10 所示。左边这三列就是 C 语言数据类型的关键字、所占位数和数的表示范围了，比如 char 是有符号字符型，占 8 位，可以表示－128～127 的整数等，需要注意的是在 51 单片机中 int 类型占 16 位，而在 STM32 单片机中占 32 位，不要混淆引用了，然后接着这个 long 和 unsigned long 占用的也是 32 位，跟 int 是一样的，如果想用 64 位的，需要用到 long long 和 unsigned long long 这个数据类型，最后是 float 和 double，这些是用来存小数的。接着看一下表格右边，在这里写的是 C 语言 stdint. h 文件和 ST 对这些变量的重命名，由于左边关键字这个名字比较长，而且这个 int 的位数根据系统的不同还有可能不一样，这个名字有时候会名不对题，比如这个 char 本意是字符型数据的意思，按名字来说它就应该存放字符类型数据，但在单片机使用中通常用它来存放整数而不是字符。所以综上各种原因，C 语言和 ST 就给这些变量换了个名字，C 语言提供的有 stdint 这个头文件，使用了新的名字，比如 int8_t 就是 char 的新名字，8 位整型数据右边加个_t，表示这是用 typedef 重新命名的变量类型。另外表格中还列举出了这个 ST 定义的名字，像 u8、u16 这些，这是 ST 库函数以前用的名字，那在新版库函数这里，已经换成了 uint8_t 和 uint16_t，但依然都可以使用，为了程序规范化，尽量使用 stdint 关键字的数据类型。

表 1－10　C 语言数据类型

关键字	位数	表示范围	stdint 关键字	ST 关键字
char	8	－128～127	int8_t	s8
unsigned char	8	0～255	uint8_t	u8
short	16	－32768～32767	int16_t	s16
unsigned short	16	0～65535	uint16_t	u16
int	32	－2147483648～2147483647	int32_t	s32
unsigned int	32	0～4294967295	uint32_t	u32
long	32	－2147483648～2147483647		
unsigned long	32	0～4294967295		
long long	64	－(2^64)/2～(2^64)/2－1	int64_t	
unsigned long long	64	0～(2^64)－1	uint64_t	
float	32	－3. 4e38～3. 4e38		
double	64	－1. 7e308～1. 7e308		

二、C 语言关键字

（1）关键字：#define

这是宏定义，它的用途是用一个字符串代替一个数字，便于理解，防止出错；提取程序中经常出现的参数，便于快速修改。

定义宏定义：

```
#define ABC 12345
```

引用宏定义：

```
Int a = ABC,这里使用的宏定义等效于 int  a = 12345。
```

（2）关键字：typedef

用途：将一个比较长的变量类型名换个名字，便于使用。

定义 typedef：

```
typedef unsigned char uint8_t;
```

引用 typedef：

```
uint8_t a,等效于 unsigned char a。
```

typedef 和宏定义有哪些区别呢？首先，宏定义的新名字在左边，typedef 的新名字在右边；其次，宏定义不需要分号，typedef 后面必须加分号；还有就是宏定义任何名字都可以换，而 typedef 只能专门给变量类型换名字。对于变量类型重命名而言，使用 tpedef 更加安全。因为宏定义只是改名时不做合法性检查，而 typedef 会对命名进行检查，如果不是变量类型的名字，那是不行的，所以给变量类型重命名时我们一般用 typedef。

（3）关键字：struct

结构体的用途是数据打包，是不同类型变量的集合。

定义结构体变量：

```
struct{char x; int y; float z;} StructName;
```

上述方式只能定义一个结构体变量，可以使用 typedef 定义一个结构体，再用该结构体定义结构体变量。如：

```
typedef struct{char x; int y; float z;} StructName1;
structName1 a;
```

通过结构体变量访问结构体变量：

```
a.x = 'a';
a.y = 100;
a.z = 12.5f;
```

也可以通过结构体指针访问结构体变量，如：

pStructName -> x = 'A'；结构体是一种组合数据类型，在函数之间的数据传递中，通常用的是地址传递而不是值传递，所以在这里 pStructName 为结构体的首地址。

```
StructName -> y = 66;
pStructName -> z = 1.23;
```

结构体的作用就是组合不同的数据类型。在一个复杂的程序里，用结构体将一些数据打包起来，将有利于我们管理或者传递这些数据，并且有利于程序员的理解。

（4）关键字：enum

枚举的用途是定义一个取值受限制的整型变量，用于限制变量取值范围。

定义枚举变量：

```
enum{FALSE = 0, TRUE = 1} EnumName;
```

因为枚举变量类型较长，所以通常用 typedef 更改变量类型名。

引用枚举成员：

```
EnumName = FALSE;
EnumName = TRUE;
```

1.4 GPIO 开发实例

1.4.1 LED 流水灯

STM32VET6 单片机的 PA0～PA7 端口引脚驱动 LED0～LED7（红、橙、黄、绿、蓝、靛、紫、白），输出低电平点亮对应的 LED 灯，输出高电平熄灭对应的 LED 灯。编写程序实现 LED 灯的正向流水灯的循环显示，一个周期内按“红 -> 橙 -> 黄 -> 绿 -> 蓝 -> 靛 -> 紫 -> 白”的顺序依次点亮，每路 LED 点亮时长 500 ms，循环执行。

```
#include "stm32f10x.h"
#include "sys.h"
#include <stdio.h>
typedef enum{red = 0, orange, yellow, green, blue, indigo, violet, white,all}LedType;
//定义 LED 类型
void All_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;   //定义 GPIO 结构体
    /* 允许总线 CLOCK，在使用 GPIO 之前必须允许相应端的时钟，从 STM32 的设计角度上说，没被允许的端将不接入时钟，也就不会耗能，这是 STM32 节能的一种技巧 */
```

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);//使能 GPIOA 口时钟
    GPIO_InitStructure.GPIO_Pin = 0x00FF; // PA0～PA7 配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;        //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure); //根据以上参数初始化结构体
}
/*************************************************************
** 函数名: LedOn
** 功能描述: 点亮指定 LED 灯
** 输入参数: LedType _color
** 输出参数: 无
** 说明:下面介绍三种实现方式
    1. 直接操作寄存器
    2. 使用 STM32 的位段功能
    3. 使用库函数操作
*************************************************************/
void LedOn(LedType _color)
{
    switch(_color)
    {
        //寄存器控制方式
        case red:GPIOA -> ODR &= ～(1 << 0);break;
        case orange:GPIOA -> BSRR |= (1 << 17);break;
        case yellow:GPIOA -> BRR |= 0x0004;break;
        case green:GPIOA -> BRR |= GPIO_Pin_3;break;
        //位带操作方式
        case blue:PAout(4) = 0;break;
        case indigo:PAout(5) = 0;break;
        //库函数方式
        case violet:GPIO_ResetBits(GPIOA,GPIO_Pin_6);break;
        case white:GPIO_WriteBit(GPIOA,GPIO_Pin_7,Bit_RESET);break;
        case all:GPIO_Write(GPIOA,0);break;
        default:break;
    }
}
void LedOff(LedType _color)
{
    switch(_color)
    {
        //寄存器控制方式
        case red:GPIOA -> ODR |= (1 << 0);break;
        case orange:GPIOA -> ODR |= GPIO_Pin_1;break;
```

```
        case yellow:GPIOA -> BSRR |= 0x0004;break;
        case green:GPIOA -> BSRR |= GPIO_Pin_3;break;
        //位带操作方式
        case blue:PAout(4) = 1;break;
        case indigo:PAout(5) = 1;break;
        //库函数方式
        case violet:GPIO_SetBits(GPIOA,GPIO_Pin_6);break;
        case white:GPIO_WriteBit(GPIOA,GPIO_Pin_7,Bit_SET);break;
        case all:GPIO_Write(GPIOA,GPIO_Pin_All);break;
        default:break;
    }
}
//定义 8 种颜色的 LED 灯
LedType colors[8]={red,orange,yellow,green,blue,indigo,violet,white};
void delayms(u16 _cnt)
{
    u8 i,j;
    for(i = 0;i < 10;i ++)
       for(j = 0;j <_cnt;j ++);
}
/************************************************************
** 函数名: main
** 功能描述: 实现 LED 流水灯
** 输入参数: 无
** 输出参数: 无
************************************************************/
int main(void)
{
    uint8_t i,size;
    SystemInit(); //系统时钟初始化,72MHz
    delay_init(72);//系统 SysTick 初始化
    All_GPIO_Config();
    size = sizeof(colors) / sizeof(colors[0]); // 计算枚举值的数量
    for(;;)//循环
    {
        for (i = 0; i < size; i ++)
        {
           LedOff(all);//关闭所有灯
           LedOn(colors[i]);//点亮当前灯
           delayms(500);//延迟 500ms
        }
    }
}
```

图 1.16 是软件仿真波形图，低电平点亮 LED 灯，高电平熄灭 LED 灯。从图上可以看出，流水灯以 500 ms 依次向右循环点亮。

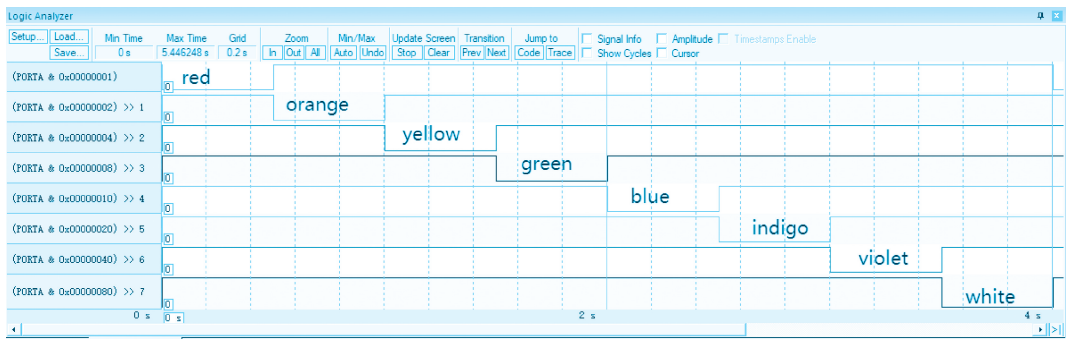


图 1.16　流水灯仿真波形

1.4.2　数码管的静态显示

静态方式显示数码管具有 LED 亮度高和程序工作量小的特点，可用在室外显示场合。

如图 1.17 所示的电路中，STM32VET6 单片机的 PA0～PA3 端口引脚驱动 CD4511 的数据输入端，CD4511 的输出端驱动共阴极数码管的段码，编写程序实现数码管 0～9 的递增计数。

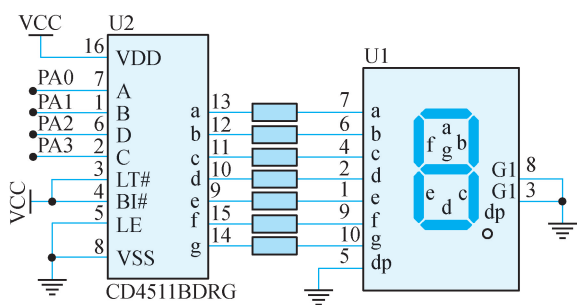


图 1.17　数码管静态显示驱动电路

CD4511 是一片 CMOS BCD—锁存/7 段译码/驱动器，用于驱动共阴极 LED(数码管)显示器的 BCD 码-七段码译码器。具有 BCD 转换、消隐和锁存控制、七段译码及驱动功能的 CMOS 电路能提供较大的拉电流，可直接驱动共阴 LED 数码管。

CD4511 的真值表如表 1－11 所示。其中 ABCD 为 BCD 码输入，A 为最低位。LT 为灯测试端，加高电平时，显示器正常显示，加低电平时，显示器一直显示数码“8”，各笔段都被点亮，以检查显示器是否有故障。BI 为消隐功能端，低电平时使所有笔段均消隐，正常显示时，B1 端应加高电平。另外 CD4511 有拒绝伪码的特点，当输入数据越过十进制数 9(1001)时，显示字形也自行消隐。LE 是锁存控制端，高电平时锁存，低电平时传输数据。a～g 是 7 段输出，可驱动共阴 LED 数码管。

表 1－11　CD4511 真值表

Inputs							Outputs							
LE	$\overline{BI}$	$\overline{LT}$	D	C	B	A	a	b	c	d	e	f	g	Display
X	X	0	X	X	X	X	1	1	1	1	1	1	1	B
X	0	1	X	X	X	X	0	0	0	0	0	0	0	
0	1	1	0	0	0	0	1	1	1	1	1	1	0	0
0	1	1	0	0	0	1	0	1	1	0	0	0	0	1
0	1	1	0	0	1	0	1	1	0	1	1	0	1	2
0	1	1	0	0	1	1	1	1	1	1	0	0	1	3
0	1	1	0	1	0	0	0	1	1	0	0	1	1	4
0	1	1	0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	1	0	0	0	1	1	1	1	1	6
0	1	1	0	1	1	1	1	1	1	0	0	0	0	7
0	1	1	1	0	0	0	1	1	1	1	1	1	1	8
0	1	1	1	0	0	1	1	1	1	0	0	1	1	9
0	1	1	1	0	1	0	0	0	0	0	0	0	0	
0	1	1	1	0	1	1	0	0	0	0	0	0	0	
0	1	1	1	1	0	0	0	0	0	0	0	0	0	
0	1	1	1	1	0	1	0	0	0	0	0	0	0	
0	1	1	1	1	1	0	0	0	0	0	0	0	0	
0	1	1	1	1	1	1	0	0	0	0	0	0	0	
1	1	1	X	X	X	X				*				*

```
#include "stm32f10x.h"
#include "delay.h"
#include "sys.h"
#define CD4511_A PAout(0)
#define CD4511_B PAout(1)
#define CD4511_C PAout(2)
#define CD4511_D PAout(3)
//软件延时编程:通过循环执行空操作来实现延时,适用于对精度要求不高的场景。
void delayus(unsigned int us)
{
    unsigned char n;
    while(us --)
        for(n = 0;n < 15;n ++);
}
void delayms(unsigned int ms)
{
    while(ms --)
        delayus(1000);
}
/**********************************************************
** 函数名: void All_GPIO_Config
```

```
**功能描述：所有的 GPIO 口配置
**输入参数：无
**输出参数：无
*************************************************************/
void All_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;   //定义 GPIO 结构体
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);//使能 A 口时钟
    GPIO_InitStructure.GPIO_Pin = 0x000F; // PA0,PA1,PA2,PA3 配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;// 2M 时钟速度
    GPIO_Init(GPIOA, &GPIO_InitStructure); //根据以上参数初始化结构体
}
void LED_Dis_num(u8 _num)
{
    switch(_num)
    {
        case 0:CD4511_D = 0;CD4511_C = 0;CD4511_B = 0;CD4511_A = 0;break;
        case 1:CD4511_D = 0;CD4511_C = 0;CD4511_B = 0;CD4511_A = 1;break;
        case 2:CD4511_D = 0;CD4511_C = 0;CD4511_B = 1;CD4511_A = 0;break;
        case 3:CD4511_D = 0;CD4511_C = 0;CD4511_B = 1;CD4511_A = 1;break;
        case 4:CD4511_D = 0;CD4511_C = 1;CD4511_B = 0;CD4511_A = 0;break;
        case 5:CD4511_D = 0;CD4511_C = 1;CD4511_B = 0;CD4511_A = 1;break;
        case 6:CD4511_D = 0;CD4511_C = 1;CD4511_B = 1;CD4511_A = 0;break;
        case 7:CD4511_D = 0;CD4511_C = 1;CD4511_B = 1;CD4511_A = 1;break;
        case 8:CD4511_D = 1;CD4511_C = 0;CD4511_B = 0;CD4511_A = 0;break;
        case 9:CD4511_D = 1;CD4511_C = 0;CD4511_B = 0;CD4511_A = 1;break;
        default:break;
    }
}
/*************************************************************
**函数名：main
**功能描述：使用系统文件使 PA1,PA2,PB1 口引脚输出方波
**输入参数：无
**输出参数：无
*************************************************************/
int main(void)
{
    u8 i;
    SystemInit(); //系统时钟初始化
    All_GPIO_Config();//所有 GPIO 配置
    for(;;)//循环
```

```
    {
        if(i < 10)
            LED_Dis_num(i ++);//递增显示
        else
            LED_Dis_num(0);//清零显示
        delayms(1000);
    }
}
```

1.4.3 数码管的动态显示

静态显示方式编程简单，但是静态显示数码管相应笔段一直处于点亮状态，因此功耗大，而且占用硬件资源多，只适合于显示器位数较少的场合。

动态显示的硬件特点是将所有数码管的同名段选线并联在一起，通过控制公共端的位选信号来控制数码管的点亮。数码管采用动态扫描显示就是逐位轮流点亮每位数码管，即每个数码管的位选被轮流选中，多个数码管共用一组段选，字段码仅对位选被选中的数码管有效。

必须注意：扫描周期必须控制在视觉停顿时间内，一般在 20 ms 以内，否则会出现闪烁或跳动现象。

应用实例：使用 STM32F103VET6 单片机，将学号的后两位显示在两位共阴极数码管上。数码管的段选信号由 GPIOB 的 PB0～PB7 控制，位选信号由 GPIOA 的 PA0(十位)和 PA1(个位)控制。系统要求通过动态扫描方式实现两位数字的稳定显示，扫描频率约为 1 kHz。

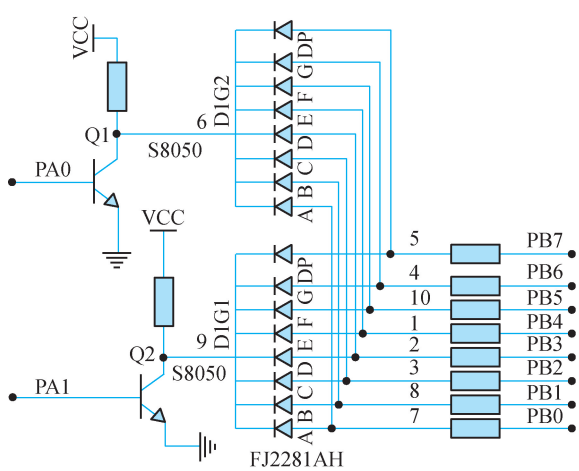


图 1.18 两位一体数码管驱动电路

程序设计如下：

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
// 数码管段选码表 (0- 9 共阴极)
const uint8_t SEGMENT_CODE[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F,
0x6F};
// 学号后两位 (请根据实际学号修改)
#define STUDENT_ID_TENS 2     // 十位数字
#define STUDENT_ID_UNITS 5    // 个位数字
// 函数声明
void GPIO_Configuration(void);
void TIM_Configuration(void);
int main(void)
{
    // 系统时钟配置
    SystemInit();
    // GPIO 配置
    GPIO_Configuration();
    // 定时器配置
    TIM_Configuration();
    // 启动定时器
    TIM_Cmd(TIM2, ENABLE);
    while(1)
    {
        // 主循环不执行具体任务,显示由定时器中断处理
    }
}
void GPIO_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 使能 GPIOA 和 GPIOB 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);
    // 配置 GPIOB PB0 - PB7 为推挽输出 (段选信号)
    GPIO_InitStructure.GPIO_Pin = 0x00FF;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    // 配置 GPIOA PA0 - PA1 为推挽输出 (位选信号)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
}
void TIM_Configuration(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    // 使能 TIM2 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
    // 定时器基础配置
    // 系统时钟 72MHz,预分频 72 - 1,计数器频率为 1MHz
    // 自动重装载值 500 - 1,产生 2kHz 中断频率
    // 每个数码管显示时间约 0.5ms,扫描频率约 1kHz
    TIM_TimeBaseStructure.TIM_Period = 500 - 1;
    TIM_TimeBaseStructure.TIM_Prescaler = 72 - 1;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
    // 使能 TIM2 更新中断
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
    // 配置 NVIC
    NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
// 定时器 2 中断服务函数
void TIM2_IRQHandler(void)
{
    static uint8_t digit_select = 0; // 位选控制变量
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        // 关闭所有位选(消隐)
        GPIO_SetBits(GPIOA, GPIO_Pin_0 | GPIO_Pin_1);
        if (digit_select == 0)
        {
            // 显示十位数
            GPIO_ResetBits(GPIOA, GPIO_Pin_0); // 选中十位数码管
            GPIO_Write(GPIOB, SEGMENT_CODE[STUDENT_ID_TENS]); // 输出十位段码
        }
        else
        {
```

```
            // 显示个位数
            GPIO_ResetBits(GPIOA, GPIO_Pin_1); // 选中个位数码管
            GPIO_Write(GPIOB, SEGMENT_CODE[STUDENT_ID_UNITS]); // 输出个位段码
        }
        // 切换位选
        digit_select = ! digit_select;
    }
}
```

注意：动态数码管显示拖尾、重影如何解决？

数码管动态显示可能会出现拖尾、重影的现象，解决的方法是：① 如果是先送位选信号，再送段选信号，则在换位时，将段选信号清除；② 如果是先送段选信号，后送位选信号，则在换段时，将位选信号清除。

1.5 项目小结

本项目以掌握 STM32 GPIO 控制与基础时序逻辑为目标，系统学习了 STM32 单片机的 GPIO 内部结构、工作模式以及多种的输出引脚控制方式。分为 3 个任务：

任务 1　实现 GPIOA 端口的 PA0～PA7 驱动 8 路 LED 单向流水功能，每路点亮 500 ms，采用低电平驱动，GPIO 配置为 50 MHz 推挽输出，软件方式延时控制状态切换。

任务 2　采用 BCD 码的方式实现二进制数值的显示，结合 CD45 芯片驱动 1 位数码管，将二进制数直观地在数码管上显示出来。

任务 3　从实际应用需求出发，实现多位一体的数码管的动态显示方式，动态扫描显示就是逐位轮流点亮每位数码管。使用 GPIO 端口分时复用方式驱动数码管的段码，以较少的端口资源实现数码管的多位显示。

构建自顶向下的模块化编程理念，为后续复杂程序编写时的团队合作打好坚实基础。

实战练习

基础任务

1. LED 的亮度主要由什么决定？（　　）。

A. 电压大小　　B. 电流大小　　C. 电阻大小　　D. 颜色

2. STM32 控制 LED 常用的 GPIO 模式是（　　）。

A. 浮空输入　　B. 推挽输出　　C. 上拉输入　　D. 模拟输入

3. 共阴极数码管中，位选引脚为高电平时（　　）。

A. 对应位的段 LED 全部点亮　　B. 对应位的段 LED 全部熄灭

C. 对应位被选中，可通过段选控制显示数字　　D. 小数点一定点亮

4. 数码管动态显示的原理是利用人眼的_____效应。

5. 若共阴极数码管要显示数字“3”，段选信号应为二进制 01011011，其十六进制值为_____。

6. 动态显示中，位选切换频率过低会导致什么问题？

7. 比较 LED 低电平驱动与高电平驱动的优缺点。

8. 简述 LED 的发光原理。

9. 若 STM32 输出 3.3 V，LED 正向压降为 2 V，额定电流为 10 mA，应选择多大限流电阻？

10. 若系统时钟为 72 MHz，使用 SysTick 产生 1 ms 延时，重装载寄存器的值应为多少？

11. 如何在数码管上同时显示“十位＝1、个位＝8”？写出关键代码思路。

进阶挑战

1. LED 与数码管综合控制：设计电路并编写程序，实现 8 路 LED 流水灯与 2 位数码管联动：数码管显示流水灯当前流动方向（“0”表示正向 A0→A7，“1”表示反向 A7→A0）。要求：流水灯切换间隔 500 ms；数码管无闪烁。

2. 学号滚动显示：用 2 位数码管循环显示学号的每两位数字（例如学号 20240318，依次显示 20 -> 24 -> 03 -> 18，循环）。

每次显示停留 1 秒，并用 LED 指示当前显示的是第几组（如第一组亮 LED1，第二组亮 LED2，以此类推）。

项目二 多路抢答器设计

学习目标

扫码可见本项目微课

知识目标

1. 独立按键的两种识别方法(扫描法和中断法)。
2. 数码管的动态显示延时程序。
3. I^2C 总线通信协议与应用。

能力目标

1. 多信号输入处理能力:能够设计程序,实时、准确、无冲突地检测多路抢答按键的信号,并正确判断首个抢答者。

2. 人机交互界面设计能力:能够编程驱动多位数码管,稳定、无闪烁地显示抢答者编号、倒计时、得分等动态信息。

3. 系统设计与模块化编程能力:能够独立设计实现抢答器的基本功能模块,包括按键输入、OLED 显示与数据存储,能够综合运用 STM32 的 GPIO、I^2C 等外设资源,完成多模块协同设计。

素质目标

1. 工匠精神与工程理论:强调在连接 I^2C 总线、数码管等较多连线的电路时,必须规范布线、做好标记,培养严谨的工程习惯。代码编写中,要求对中断服务程序、通信函数等关键部分添加详尽注释,体现代码规范性。

2. 创新意识与全局观念:在完成基本功能后,进行创新性设计,如增加不同难度的题目分值、设置多种比赛模式、设计更炫酷的显示效果等,激发创新思维。培养从系统整体角度而非单个模块角度去思考、设计和调试的系统工程思维。

任务描述

本任务旨在利用 STM32 单片机,设计并实现一套多路抢答器系统。该系统可应用实现多名参赛者同时抢答、抢答成功者的及时显示、得分统计以及数据的掉电保存等功能。

1. 按键输入与识别,系统有若干独立按键,分别对应不同选手。能够通过扫描法和中

断法两种方式识别按键输入，实现对抢答操作的准确检测与响应。

2. OLED12864 液晶屏显示抢答编号、得分、排名等信息，要求显示内容清晰、切换流畅。

3. I^2C 通信与数据存储，利用 I^2C 总线协议，将选手得分、抢答顺序、历史数据等信息存入串行 E2PROM 芯片 AT24C02，实现数据的掉电保存和随时读取。

2.1 项目背景

抢答器作为知识竞赛、课堂互动、技能比拼等场景的核心交互设备，其核心价值在于快速响应与结果公正——需在毫秒级时间内捕捉参与者的抢答信号，并通过直观的声光提示（如 LED 指示灯、蜂鸣器报警）反馈抢答结果，同时避免“信号冲突”“误触发”等影响公正性的问题。传统简易抢答器多依赖分立元器件搭建，功能单一（仅支持 2—4 路抢答）且扩展性差，难以满足多组别竞赛、计分统计、历史数据回溯等复杂需求，而基于 STM32 单片机的抢答器系统则可通过软件编程与外设扩展，实现功能的灵活升级与性能优化。

本项目以 STM32F103C8T6 单片机为核心控制器，聚焦“多路抢答＋数据存储”的综合需求，引导学生从“单一外设控制”（如前文 LED 项目的 GPIO 操作）迈向“多模块协同集成”的嵌入式系统设计。项目需综合运用 STM32 的核心技术模块：通过 GPIO 输入功能采集多路抢答按键信号（需设计防抖处理，避免机械按键抖动导致的误判）；通过外部中断实现抢答信号的快速捕获（中断响应时间≤10 μs，确保“先按先得”的公正性）；通过定时器（如 TIM2）实现“抢答倒计时”功能（如 30 秒倒计时，时间到自动锁定抢答通道）；通过 I^2C 通信连接 EEPROM 存储芯片（如 AT24C02），实现抢答历史数据（如组别、抢答时间、得分）的掉电保存，支持后续数据查询与统计；同时结合 GPIO 输出功能驱动 LED 指示灯（如每路抢答对应独立 LED，点亮表示该路抢答成功）与蜂鸣器（如抢答成功时短鸣、超时未抢答时长鸣），构建完整的人机交互体系。

本项目具备极强的可拓展性：基础版实现“多路抢答、倒计时、声光提示和数据存储”核心功能后，可进一步扩展串口通信功能（如与上位机软件连接，实现抢答数据的实时显示与报表生成）、LCD 显示功能（如通过 OLED12864 液晶屏显示倒计时、当前抢答组别、得分情况），为学生后续探索更复杂的嵌入式系统（如智能家居控制、工业数据采集终端）积累关键技术经验，真正实现从“入门级实验”到“应用型产品开发”的能力跨越。

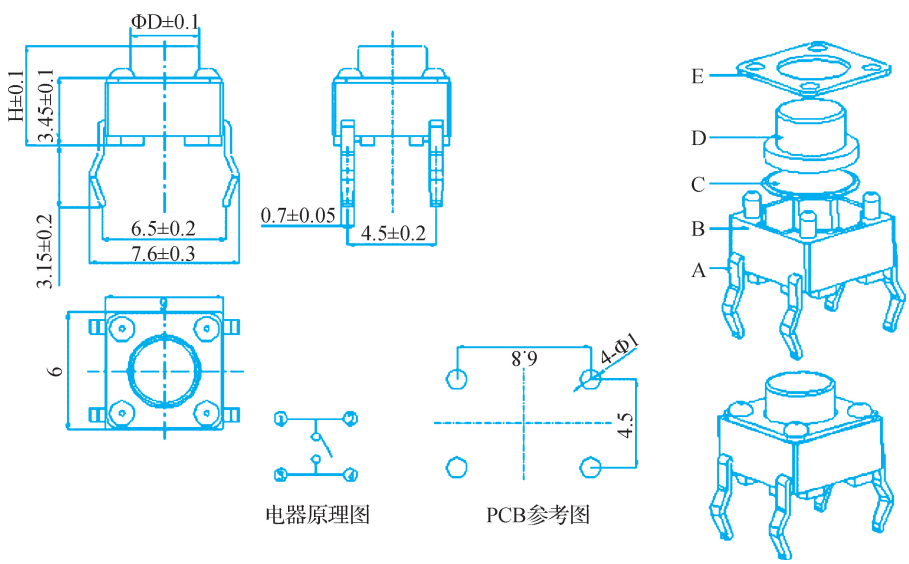
2.2 按键检测

2.2.1 按键分类与结构

按键是嵌入式系统中最基础的输入设备，核心功能是通过机械触点的通断实现“按下输入”，常见类型包括机械轻触按键、薄膜按键、拨码按键等，机械轻触按键具有两个触点分别为：固定触点（静触点）和活动触点（动触点）。

机械轻触按键如图 2.1 所示。

机械轻触按键的核心功能是通过内部金属弹片的通断实现信号输入，未按下时弹片分离，电路断开；按下时弹片接触，电路导通。



A：插脚　B：基座　C：弹片　D：按钮　E：盖板

图 2.1　机械轻触按键

2.2.2 按键驱动电路设计

按键驱动电路设计需结合 STM32 GPIO 的输入模式，常见按键的四种电路方式如下。

➢ 第一种：内部上拉输入：按键一端接 STM32 GPIO 引脚（如 PA0），另一端接地，GPIO 配置为“上拉输入模式”（GPIO_Mode_IPU）。未按下时，GPIO 引脚通过内部上拉电阻接 3.3 V，输入高电平；按下时，引脚经按键接地，输入低电平（如图 2.2(a)内部上拉输入）。此方案无需外部电阻，简化硬件布线。

➢ 第二种：外部上拉输入：按键一端接 GPIO 引脚，另一端接地，同时引脚经 10 kΩ 外部上拉电阻接 3.3 V。未按下时，引脚由外部电阻拉为高电平；按下时，接地变低电平（如图

2.2(b)外部上拉输入)。此方案抗干扰性更强,适合电磁环境复杂的场景。

➢第三种:内部下拉输入:按键一端接 GPIO 引脚,另一端接 3.3 V,GPIO 配置为“下拉输入模式”(GPIO_Mode_IPD)。未按下时引脚拉为低电平,按下时接 3.3 V 变高电平(如图 2.2(c)内部下拉输入),适用于需“高电平触发”的场景。

➢第四种:外部下拉输入:按键一端接 GPIO 引脚,另一端接 3.3 V,同时引脚经 10 kΩ 外部下拉电阻接地(如图 2.2(d)外部下拉输入),功能与方案 3 类似,适合对电平稳定性要求较高的场景。

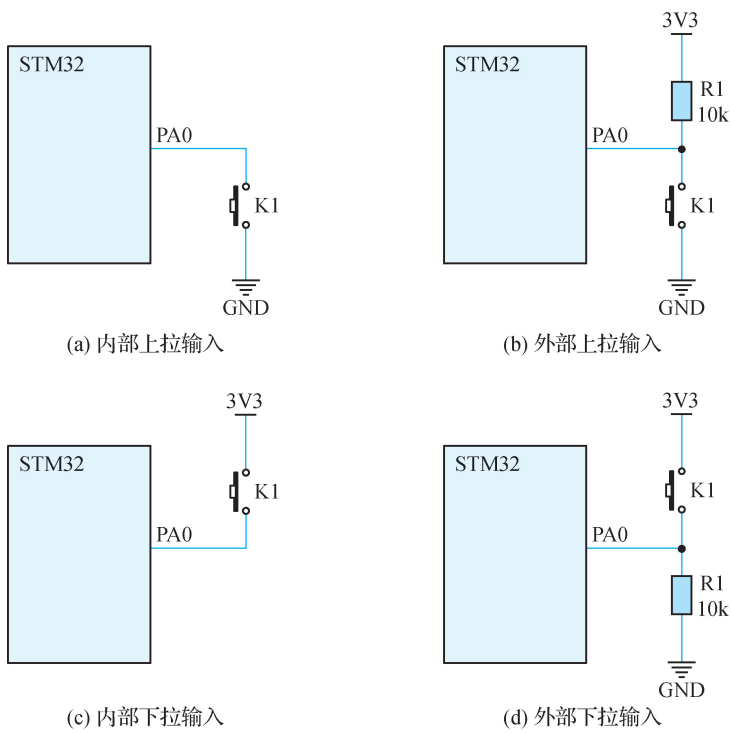


图 2.2　按键驱动电路的四种常见方案

2.2.3　按键识别

按键状态的识别一般使用查询法和中断法。在图 2.3 中,按键检测节点 KEY1 接 PD3 引脚。按键按下时,PD3 为低电平;按键释放时,PD3 为高电平。使用查询法识别按键 1 按下状态控制 LED 灯主要有以下几种方式:

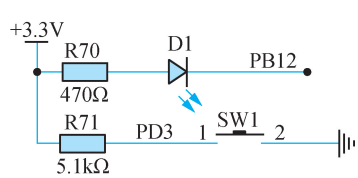


图 2.3　按键与显示电路图

（1）寄存器法

```
if((GPIOD -> IDR & GPIO_Pin_3) == 0)   GPIOB -> BRR |=  GPIO_Pin_12;
```

（2）位带操作法

```
if(PDin(3) == 0)   PBout(12) = 0;
```

（3）库函数法

```
if(GPIO_ReadInputDataBit(GPIOD,GPIO_Pin_3) == Bit_RESET)
    GPIO_WriteBit(GPIOB, GPIO_Pin_12, Bit_RESET);
if((GPIO_ReadInputData(GPIOD) & GPIO_Pin_3) == 0)
    GPIO_ResetBits(GPIOB, GPIO_Pin_12);
```

查询法需要在程序运行时定时查询输入引脚的状态，会占用一定的程序资源。按键识别也可以采用外部中断法，按键有动作时产生外部中断，进入中断服务程序后完成指定任务；完成任务后返回主程序继续执行指令。

2.2.4　外部中断/事件控制器

一、中断基础知识

中断是计算机系统中由硬件和软件协同实现的响应机制，用于协调 CPU 处理多任务事件。当中断源发出请求时，CPU 暂停当前程序执行中断服务程序，完成后恢复原流程。该机制广泛应用于实时控制、设备通信及故障处理，是现代计算机多任务调度的基础。

引起中断的事件称为中断源。中断源向 CPU 提出处理的请求称为中断请求。发生中断时被打断程序的暂停点称为断点。CPU 暂停现行程序而转为响应中断请求的过程称为中断响应。处理中断源的程序称为中断处理程序。CPU 执行有关的中断处理程序称为中断处理。而返回断点的过程称为中断返回。

二、嵌套向量中断控制器

STM32 单片机的嵌套向量中断控制器（NVIC，Nested Vectored Interrupt Controller）和处理器核的接口紧密相连，可以实现低延迟的中断处理和高效地处理晚到的中断。

表 2－1　STM32F10 系列的向量表

位置	优先级	优先级类型	名称	说明	地址
	—	—	—	保留	0x0000_0000
	－3	固定	Reset	复位	0x0000_0004

续 表

位置	优先级	优先级类型	名称	说明	地址
	−2	固定	NMI	不可屏蔽中断 RCC 时钟安全系统(CSS)联接到 NMI 向量	0x0000_0008
	−1	固定	硬件失效	所有类型的失效	0x0000_000C
	0	可设置	存储管理	存储器管理	0x0000_0010
	1	可设置	总线错误	预取指失败,存储器访问失败	0x0000_0014
	2	可设置	错误应用	未定义的指令或非法状态	0x0000_0018
	—	—	—	保留	0x0000_001C ～0x0000_002B
	3	可设置	SVCall	通过 SWI 指令的系统服务调用	0x0000_002C
	4	可设置	调试监控	调试监控器	0x0000_0030
	—	—	—	保留	0x0000_0034
	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C
0	7	可设置	WWDG	窗口定时器中断	0x0000_0040
1	8	可设置	PVD	连到 EXTI 的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050
5	12	可设置	RCC	复位和时钟控制(RCC)中断	0x0000_0054
6	13	可设置	EXTI0	EXTI 线 0 中断	0x0000_0058
7	14	可设置	EXTI1	EXTI 线 1 中断	0x0000_005C
8	15	可设置	EXTI2	EXTI 线 2 中断	0x0000_0060
9	16	可设置	EXTI3	EXTI 线 3 中断	0x0000_0064
10	17	可设置	EXTI4	EXTI 线 4 中断	0x0000_0068
11	18	可设置	DMA1 通道 1	DMA1 通道 1 全局中断	0x0000_006C
12	19	可设置	DMA1 通道 2	DMA1 通道 2 全局中断	0x0000_0070
13	20	可设置	DMA1 通道 3	DMA1 通道 3 全局中断	0x0000_0074
14	21	可设置	DMA1 通道 4	DMA1 通道 4 全局中断	0x0000_0078
15	22	可设置	DMA1 通道 5	DMA1 通道 5 全局中断	0x0000_007C

续　表

位置	优先级	优先级类型	名称	说明	地址
16	23	可设置	DMA1 通道 6	DMA1 通道 6 全局中断	0x0000_0080
17	24	可设置	DMA1 通道 7	DMA1 通道 7 全局中断	0x0000_0084
18	25	可设置	ADC	ADC 全局中断	0x0000_0088
19	26	可设置	USB_HP_CAN_TX	USB 高优先级或 CAN 发送中断	0x0000_008C
20	27	可设置	USB_LP_CAN_RX0	USB 低优先级或 CAN 接收 0 中断	0x0000_0090
21	28	可设置	CAN_RX1	CAN 接收 1 中断	0x0000_0094
22	29	可设置	CAN_SCE	CAN SCE 中断	0x0000_0098
23	30	可设置	EXTI9_5	EXTI 线[9：5]中断	0x0000_009C
24	31	可设置	TIM1_BRK	TIM1 断开中断	0x0000_00A0
25	32	可设置	TIM1_UP	TIM1 更新中断	0x0000_00A4
26	33	可设置	TIM1_TRG_COM	TIM1 触发和通信中断	0x0000_00A8
27	34	可设置	TIM1_CC	TIM1 捕获比较中断	0x0000_00AC
28	35	可设置	TIM2	TIM2 全局中断	0x0000_00B0
29	36	可设置	TIM3	TIM3 全局中断	0x0000_00B4
30	37	可设置	TIM4	TIM4 全局中断	0x0000_00B8
31	38	可设置	I2C1_EV	I^2C1 事件中断	0x0000_00BC
32	39	可设置	I2C1_ER	I^2C1 错误中断	0x0000_00C0
33	40	可设置	I2C2_EV	I^2C2 事件中断	0x0000_00C4
34	41	可设置	I2C2_ER	I^2C2 错误中断	0x0000_00C8
35	42	可设置	SPI1	SPI1 全局中断	0x0000_00CC
36	43	可设置	SPI2	SPI2 全局中断	0x0000_00D0
37	44	可设置	USART1	USART1 全局中断	0x0000_00D4
38	45	可设置	USART2	USART2 全局中断	0x0000_00D8
39	46	可设置	USART3	USART3 全局中断	0x0000_00DC
40	47	可设置	EXTI15_10	EXTI 线[15：10]中断	0x0000_00E0
41	48	可设置	RTCAIarm	连到 EXTI 的 RTC 闹钟中断	0x0000_00E4
42	49	可设置	USB 唤醒	连到 EXTI 的从 USB 待机唤醒中断	0x0000_00E8
43	50	可设置	TIMB_BRK	TIM8 断开中断	0x0000_00EC
44	51	可设置	TIMB_UP	TIM8 更新中断	0x0000_00F0

续 表

位置	优先级	优先级类型	名称	说明	地址
45	52	可设置	TIM8_TRG_COM	TIM8 触发和通信中断	0x0000_00F4
46	53	可设置	TIM8_CC	TIM8 捕获比较中断	0x0000_00F8
47	54	可设置	ADC3	ADC3 全局中断	0x0000_00FC
48	55	可设置	FSMC	FSMC 全局中断	0x0000_0100
49	56	可设置	SDIO	SDIO 全局中断	0x0000_0104
50	57	可设置	TIM5	TIM5 全局中断	0x0000_0108
51	58	可设置	SPI3	SPI3 全局中断	0x0000_010C
52	59	可设置	UART4	UART4 全局中断	0x0000_0110
53	60	可设置	UART5	UART5 全局中断	0x0000_0114
54	61	可设置	TIM6	TIM6 全局中断	0x0000_0118
55	62	可设置	TIM7	TIM7 全局中断	0x0000_011C
56	63	可设置	DMA2 通道 1	DMA2 通道 1 全局中断	0x0000_0120
57	64	可设置	DMA2 通道 2	DMA2 通道 2 全局中断	0x0000_0124
58	65	可设置	DMA2 通道 3	DMA2 通道 3 全局中断	0x0000_0128
59	66	可设置	DMA2 通道 4_5	DMA2 通道 4 和 DMA2 通道 5 全局中断	0x0000_012C

NVIC 寄存器包括以下几种功能：

- 中断优先级：NVIC 允许针对每个可能的中断源设置优先级，通过设置优先级来确定中断的响应顺序。通常，较低的数值表示更高的优先级。
- 中断使能：可以通过 NVIC 寄存器来使能或禁用特定的中断源，以控制中断请求的触发。
- 中断向量表偏移寄存器：用于指定中断服务程序（ISR）的地址，当特定中断触发时，处理器会跳转到相应的中断服务程序开始执行。

三、中断优先级

常用的寄存器有 ISER、ICER 和 IP，ISER 用来使能中断，ICER 用来失能中断，IP 用来设置中断优先级。

中断优先级由内核外设 SCB 的应用程序中断及复位控制寄存器 AIRCR 的 PRIGROUP[10:8]位决定。Cotex-M 内核的优先级包括主优先级（抢占式优先级）和子优先级（副优先级）两级优先级。表 2-2 给出了不同优先级分组时，主优先级和子优先级的取值范围。

表 2－2　Cotex-M 优先级

优先级分组	主优先级	子优先级	描述
NVIC_PriorityGroup_0	0	0～15	主-0bit，子-4bit
NVIC_PriorityGroup_1	0～1	0～7	主-1bit，子-3bit
NVIC_PriorityGroup_2	0～3	0～3	主-2bit，子-2bit
NVIC_PriorityGroup_3	0～7	0～1	主-3bit，子-1bit
NVIC_PriorityGroup_4	0～15	0	主-4bit，子-0bit

外部中断/事件控制器由 20 个产生事件/中断请求的边沿检测器组成，每个输入线可以独立地配置输入类型（脉冲或挂起）和对应的触发事件（上升沿或下降沿或双沿触发方式）。每个输入线都可以独立地被屏蔽。挂起寄存器保持着状态线的中断请求。

斜杠 20，表示在控制器内部类似的信号线路有 20 个，EXTI 总共有 20 个中断/事件线斜杠 20，表示在控制器内部类似的信号线路有 20 个，EXTI 总共有 20 个中断/事件线。

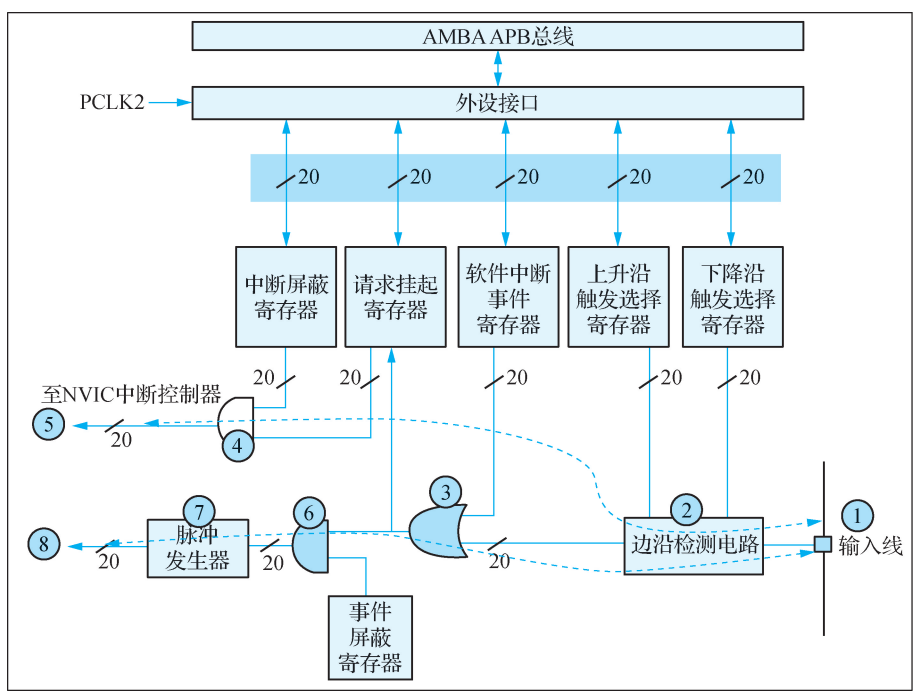


图 2.4　外部中断功能框图

产生中断线路的目的是把输入信号输入到 NVIC，进一步运行会中断服务函数，实现功能，这样是软件级的。而产生事件线路的目的就是传输一个脉冲信号给其他外设使用，并且是电路级别的信号传输，属于硬件级的。

四、中断事件线

EXTI 有 20 个中断/事件线，每个 GPIO 都可以被设置为输入线，占用 EXTI0 至 EXTI15，还有另外 4 根用于特定的外设事件，见表 2－3EXTI 中断/事件线。

4 根特定外设中断/事件线由外设触发，具体用法参考《STM32F10X-中文参考手册》中对外设的具体说明。

表 2－3　EXTI 中断/事件线

中断/事件线	输入源
EXTI0～EXTI15	Px0～Px15（x 取 A、B、C、D、E）
EXTI16	PVD 输出
EXTI17	RTC 闹钟事件
EXTI18	USB 唤醒事件
EXTI19	以太网唤醒事件（只适用于互联型）

EXTI0 至 EXTI15 用于 GPIO，通过编程控制可以实现任意一个 GPIO 作为 EXTI 的输入源。由表 2－3EXTI 中断/事件线可知，EXTI0 可以通过 AFIO 的外部中断配置寄存器 1（AFIO_EXTICR1）的 EXTI0[3:0]位选择配置为 PA0、PB0、PC0、PD0、PE0。

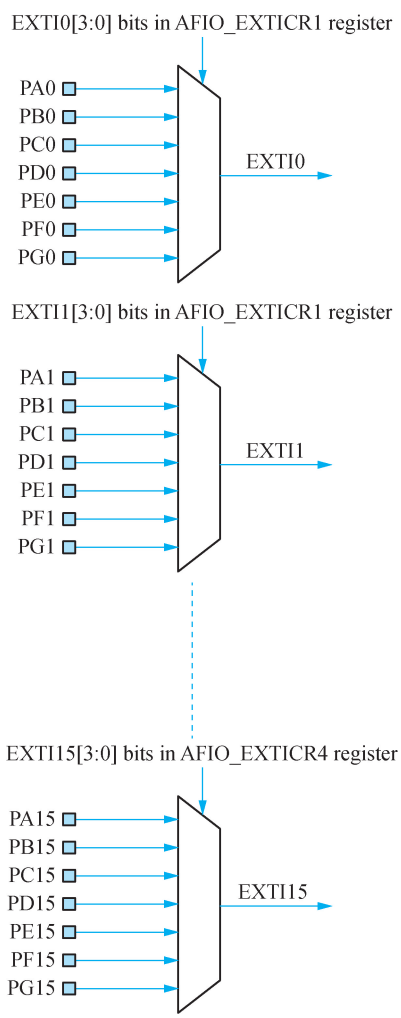


图 2.5　外部中断 GPIO 映像

五、EXTI 初始化结构体

```
typedef struct
{
  uint32_t EXTI_Line;                    /*中断/事件线 */
  EXTIMode_TypeDef EXTI_Mode;          /*EXTI 模式 */
  EXTITrigger_TypeDef EXTI_Trigger;      /*触发类型*/
  FunctionalState EXTI_LineCmd;           /*EXTI 使能*/
}EXTI_InitTypeDef;
```

EXTI 中断/事件线选择，可选 EXTI0 至 EXTI19。

EXTI 模式选择，可选为产生中断（EXTI_Mode_Interrupt）或者产生事件（EXTI_Mode_Event）。

EXTI 边沿触发事件，可选上升沿触发（EXTI_Trigger_Rising）、下降沿触发（EXTI_Trigger_Falling）或者上升沿和下降沿都触发（EXTI_Trigger_Rising_Falling）。

EXTI 使能位控制是否使能 EXTI 线，可选使能 EXTI 线（ENABLE）或禁用（DISABLE）。

2.2.5　按键中断程序设计

编写按键中断程序，使图 2.4 中的按键切换 LED 灯的亮灭状态。程序结构主要按以下步骤设计：

第一步　初始化用来产生中断的 GPIO；

第二步　初始化 EXTI；

第三步　配置 NVIC；

第四步　编写中断服务函数。

```
void GPIO_LED_Config(void)   // IO 配置函数
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
}
// EXIT 中断配置
void EXTI_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;//定义 GPIO 结构体变量
    EXTI_InitTypeDef EXTI_InitStructure;//定义 EXTI 结构体变量
```

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD,ENABLE);//使能按键端口时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD,&GPIO_InitStructure);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD,GPIO_PinSource3);// EXTI 的信号源
    EXTI_ClearITPendingBit(EXTI_Line3);//清除 line3 线标志位
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;//下降沿触发
    EXTI_InitStructure.EXTI_Line = EXTI_Line3;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);
}
// NVIC 配置
void NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//配置 NVIC 为优先级组 2

    NVIC_InitStructure.NVIC_IRQChannel = EXTI3_IRQn;//配置中断源为 line3
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//配置主优先级为 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;//配置子优先级为 2
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能中断通道
    NVIC_Init(&NVIC_InitStructure);
}
//中断服务函数
void EXTI3_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line3)!= RESET)//确认产生了 line3 的外部中断
    {
          EXTI_ClearITPendingBit(EXTI_Line3);//清除中断标志
          GPIOB -> ODR ^= GPIO_Pin_12;// PB12 取反
    }
}
//主函数
int main(void)
{
    SystemInit();
    delay_init(72);
    GPIO_LED_Config();
```

```
    EXTI_Config();
    NVIC_Config();
    while(1);
}
```

2.2.6　按键的消抖

前面我们编程都是按照按键按下是低电平、释放是高电平识别的。实际应用中(如图2.6所示),由于金属弹片的弹性振动,按下与松手瞬间会产生5～10 ms的“抖动区间”;按下时电平从高到低的过程中,会出现3～5次短暂的“低→高”波动;松手时电平从低到高的过程中,同样存在多次“高→低”波动,并非理论上的平滑切换。

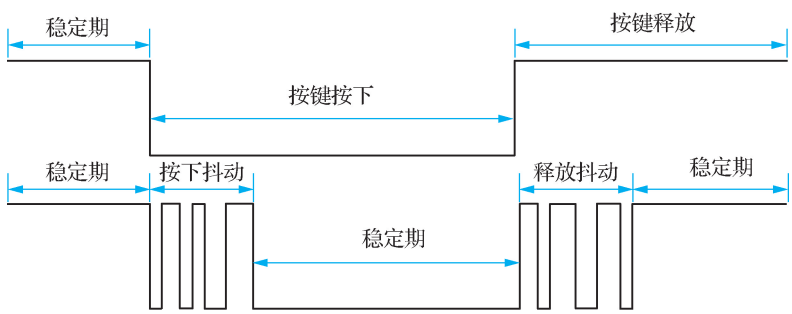


图2.6　按键按下/松手时的抖动波形(上端为理论波形,下端为实际波形)

STM32单片机主频高达72 MHz,单次指令执行时间仅数十纳秒,若不处理抖动,会将这5～10 ms内的波动误识别为多次按键输入(如一次按下可能被判定为3～5次触发),导致抢答误判、功能紊乱等问题。因此,必须采用软硬件结合的消抖方案:软件层面可通过10 ms延时消抖(检测到电平变化后延时,再确认稳定电平),硬件层面可串联*RC*滤波电路(抑制高频波动),确保按键信号准确识别。

2.3　SysTick 定时器

2.3.1　SysTick 定时器的概念

SysTick是一个24位减计数定时器,属于Cortex-M处理器的内核资源,所有的Cortex-M内核处理器都具有相同的SysTick,方便程序的移植。它集成于嵌套向量中断控制器(NVIC)中,可用于周期性中断产生、延时函数实现、定时时间测量等,也用于任务调度、时间管理、上下文切换等操作系统核心功能。

SysTick常用于产生周期为1 ms的基准时钟。它有4个寄存器。

(1) 控制及状态寄存器(CTRL):对系统嘀嗒定时器做控制,以及读取对应的状态。

表 2-4　SysTick 控制及状态寄存器(地址:0xE000_E010)

位断	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后,SysTick 已经数到了 0,则该位为 1。如果读取该位,该位自动清零
2	CLKSOURCE	R/W	0	置 0=外部时钟源(STCLK) 置 1=内核时钟(FCLK)
1	TICKINT	R/W	0	置 1:使能中断 置 0:失能中断
0	ENABLE	R/W	0	置 1:使能计数器 一直重复工作 置 0:失能计数器

(2) 重装载数值寄存器(LOAD):提供计数器的最大值。

表 2-5　SysTick 重装载数值寄存器(地址:0xE000E014)

位断	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时,将被重装载的值

(3) 当前数值寄存器(VAL):反映计数器的当前值。

表 2-6　SysTick 当前数值寄存器(地址:0xE000 E018)

位断	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取这个寄存器:能够获取到计数器的当前值 写入这个寄存器:任意值都能清除计数标志位

(4) 校准数值寄存器(CALIB):一般不使用。

表 2-7　SysTick 校准数值寄存器(地址:0xE000_E01C)

位断	名称	类型	复位值	描述
31	NOREF	R	—	置 1=没有外部参考时钟(STCLK 不可用) 置 0=外部参考时钟可用
30	SKEW	R	—	置 1=校准值不是准确的 10 ms 置 0=校准值是准确的 10 ms
23:0	TENMS	R/W	0	10 ms 的时间内倒计数的格数。芯片设计者应该通过 Cortex-M3 的输入信号提供该数值。若该值读回零,则表示无法使用校准功能

在使用 SysTick 时,通常只需配置前三个关键寄存器即可实现精确延时,无需手动操作 CALIB 寄存器。其校准值由系统自动计算并写入该寄存器,用户无需干预。

2.3.2　SysTick 定时器的工作原理

下面我们来看系统嘀嗒定时器的工作原理。

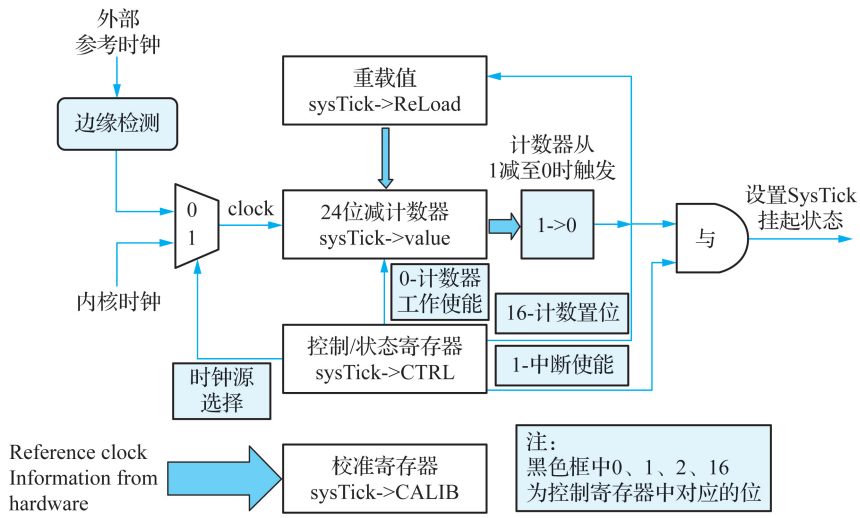


图 2.7　SysTick 结构框图

Cortex-M3 为 SysTick 提供了两个时钟源：外部时钟源和内核时钟源。控制寄存器的 clksource 位为 0 时选择外部时钟源，为 1 时选择内核时钟源。在时钟信号节拍下，24 位减计数器自动递减，递减到 0 时，溢出并触发重装载寄存器将重装载值装入计数器中。控制及状态寄存器可使能计数器工作。中断使能时，可触发溢出中断。校准寄存器可以实现时间校准。

SysTick 时间长度是如何控制的呢？

SysTick 为 24 位计数器，其最大计数值 $n=2^{24}-1=16\ 777\ 215$。

由计数值 n 可确定对应的时间 $t=n\times T=n/f$

由预定的时间 t 可确定相应的计数值 $n=t\times f$

当时钟源为参考时钟：9 MHz 时，最大计时时长：

$$t_{max}=\frac{16\ 777\ 215}{9\ 000\ 000}\approx 1\ 864\ \text{ms}$$

2.3.3　SysTick 定时器的程序设计

delay_init()函数用来初始化 2 个重要参数：fac_us 以及 fac_ms；同时把 SysTick 的时钟源选择外部时钟。具体代码如下：

```
void delay_init(u8 SYSCLK)
{
    SysTick -> CTRL & = 0xfffffffb;    // bit2 清空，选择外部时钟   HCLK / 8
    fac_us = SYSCLK / 8;
    fac_ms = (u16) fac_us * 1000;
}
// delay_us(uint32_t xus )用来延时指定的微秒时间。
void delay_us(uint32_t xus)
{
```

```
    SysTick -> LOAD = 72 * xus;              //设置定时器重装值
    SysTick -> VAL = 0x00;                   //清空当前计数值
    SysTick -> CTRL = 0x00000005;            //设置时钟源为 HCLK,启动定时器
    while(!(SysTick -> CTRL & 0x00010000));    //等待计数到 0
    SysTick -> CTRL = 0x00000004;            //关闭定时器
}
// delay_ms(uint32_t xms )用来延时指定的毫秒时间。
void delay_ms(uint32_t xms)
{
    while(xms --)
    {
      Delay_us(1000);
    }
}
// delay_s(uint32_t xs )用来延时指定的秒时间。
void delay_s(uint32_t xs)
{
    while(xs --)
    {
      Delay_ms(1000);
    }
}
```

以下是主函数,初始化部分对时钟、SysTick 和输出引脚进行配置,循环体中每隔 1 000 ms 改变依次输出引脚的电平状态。使用示波器测得延时效果如图 2.8 所示,可以精确延时 1 000 ms。

```
int main(void)
{
      SystemInit(); //系统时钟初始化,72MHz
      delay_init(72);//系统 SysTick 初始化
      All_GPIO_Config();//所有 GPIO 配置
      for(;;)//循环
      {
            delay_ms(1000);
            PCout(13) = 0;
            delay_ms(1000);
            PCout(13) = 1;
      }
```

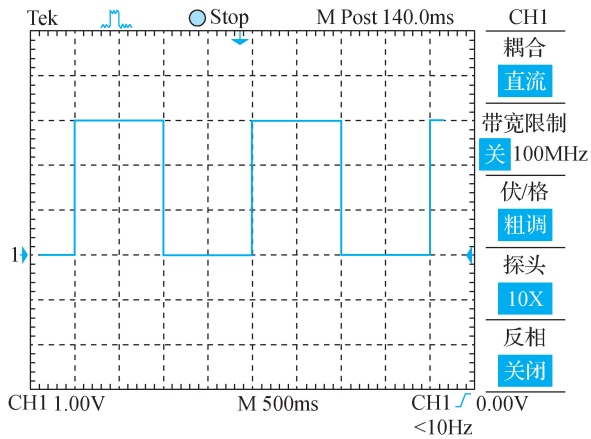


图 2.8　SysTick 的延时效果

SysTick 为 ARM 微处理器中最基本的定时器，其功能简单易于使用，STM32 中其他基本定时器和更复杂的高级定时器均是在其基础上进行功能扩充而设计的，或者是基于定时器集成某些专用功能电路，比如定时器中实现 PWM 输出电路、脉冲测量等等。所以掌握好 SysTick 的基本工作原理、深刻理解其定时和延时的本质对于学习好单片机具有非常重要的意义。

2.4　I^2C 通信

2.4.1　I^2C 通讯基础

I^2C(Inter—Integrated Circuit)是由 Philips 公司开发的一种通用数据总线，硬件结构简单的同步、半双工、串行通信总线。它只需要两根线(SDA：串行数据线，SCL：串行时钟线)即可实现多个设备间的数据通信，非常适合与各种外围器件(如 E^2PROM、传感器、RTC 等)进行短距离通信，带数据应答，支持总线挂载多设备(一主多从、多主多从)。

硬件电路模型如图 2.9 所示，这是一个一主多从电路模型，左边 CPU 就是主控 STM32 单片机，作为总线的主机。主线的权力最大，包括对 SCL 线的完全控制，另外在空闲状态下，主机可以主动发起对 SDA 的控制，只有在从机发送数据和从机应答的时候，主机才会转交 SDA 的控制权给从机。最下面都是挂载在 I^2C 总线上的从机，可以是姿态传感器、OLED、存储器等。从机的权力比较小，对于 SCL 时钟线，在任何时刻都只能被动地读取，对于 SDA 数据线，从机不允许主动发起对 SDA 的控制，只有在主机发送读取从机的命令后，或者从机应答的时候，从机才能短暂地取得 SDA 的控制权，这就是一主多从模型中协议的规定。硬件电路设计接线所有 I^2C 设备的 SCL 连在一起，SDA 连在一起，设备的 SCL 和 SDA 均要配置成开漏输出模式，SCL 和 SDA 需要各添加一个上拉电阻，阻值一般为 4.7 kΩ 左右。

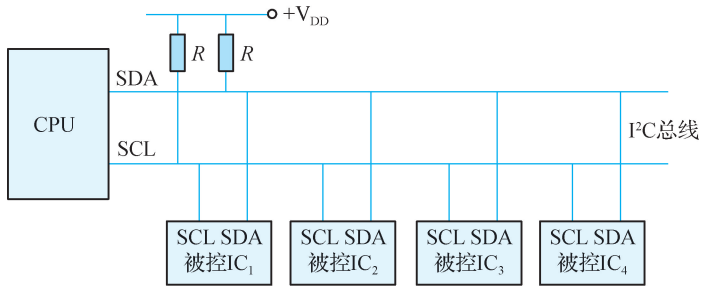


图 2.9　I²C 电路模型

I²C 主设备与从设备的一般通信过程：

一、主设备给从设备发送/写入数据

第一步　主设备发送起始(START)信号。

第二步　主设备发送设备地址到从设备。

第三步　等待从设备响应(ACK)。

第四步　主设备发送数据到从设备，一般发送的每个字节数据后会跟着等待接收来自从设备的响应(ACK)。

第五步　数据发送完毕，主设备发送停止(STOP)信号终止传输。

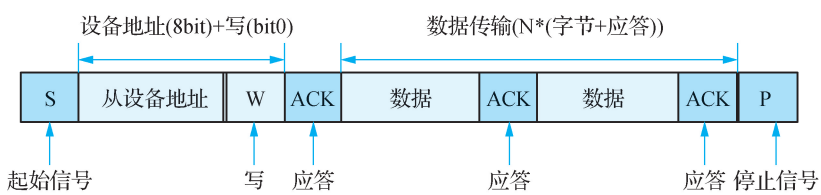


图 2.10　主->从控制信号

二、主设备从从设备接收/读取数据

第一步　设备发送起始(START)信号。

第二步　主设备发送设备地址到从设备。

第三步　等待从设备响应(ACK)。

第四步　主设备接收来自从设备的数据，一般接收的每个字节数据后会跟着向从设备发送一个响应(ACK)。

第五步　一般接收到最后一个数据后会发送一个无效响应(NACK)，然后主设备发送停止(STOP)信号终止传输。

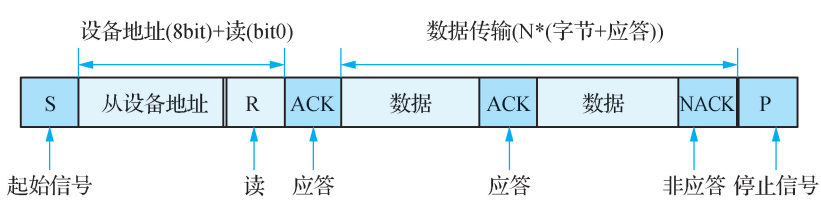


图 2.11　主<-从接收机信号

I²C 软件时序的基本单元，在使用 I²C 通信时，需要触发 I²C 的起始条件为：SCL 高电平期间，SDA 从高电平切换到低电平。终止条件为：SCL 高电平期间，SDA 从低电平切换到高电平。

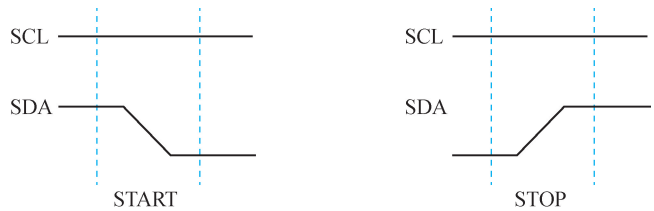


图 2.12　I²C 开始信号和停止信号

I²C 发送一个字节时序框图如图 2.13 所示。SCL 低电平期间，主机将数据位依次放到 SDA 线上（高位先行即从 B7 开始，串口通信是低位先行），然后释放 SCL，从机将在 SCL 高电平期间读取数据位，所以 SCL 高电平期间 SDA 不允许有数据变化，依次循环上述过程 8 次，即可发送一个字节。

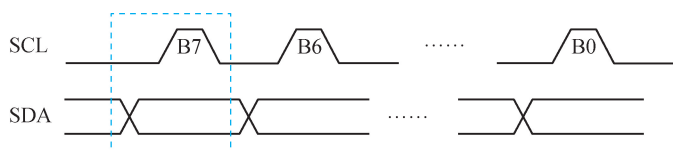


图 2.13　I²C 发送字节时序框图

I²C 接收一个字节时序框图如图 2.14 所示。SCL 低电平期间，从机将数据位依次放到 SDA 线上（高位先行），然后释放 SCL，主机将在 SCL 高电平期间读取数据位，所以 SCL 高电平期间 SDA 不允许有数据变化，依次循环上述过程 8 次，即可接收一个字节（主机在接收之前，需要释放 SDA）。

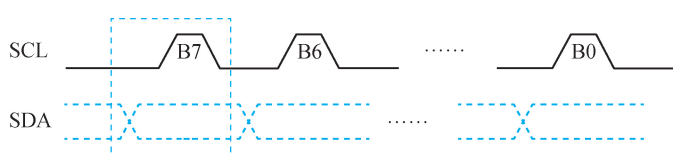


图 2.14　I²C 接收字节时序框图

I²C 发送应答和接收应答时序框图如图 2.15 所示。接着是 I²C 最后两个基本单元就是应答机制的设计，分别为发送应答和接收应答，它们的时序，分别和发送一个字节、接收一个字节的其中一位是相同的。发送应答：主机在接收完一个字节之后，在下一个时钟发送一位数据，数据 0 表示应答，数据 1 表示非应答。接收应答：主机在发送完一个字节之后，在下一个时钟接收一位数据，判断从机是否应答，数据 0 表示应答，数据 1 表示非应答（主机在接收之前，需要释放 SDA）。

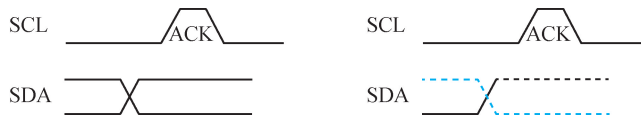


图 2.15　I²C 发送应答和接收应答时序框图

I²C 的完整时序，主要有指定地址写、当前地址读和指定地址读这 3 种。在同一条 I²C 总线里，挂载的每个设备地址必须不一样，否则，主机写入一个地址，有多个设备都响应，不符合设计需求。从机设备地址，在 I²C 协议标准里分为 7 位地址和 10 位地址，从机设备地址，在 I²C 协议标准里分为 7 位地址和 10 位地址，7 位地址应用范围最广，在每个 I²C 设备出厂时，厂商都会为它分配一个 7 位地址。第一个时序是指定地址写，对于指定设备(Slave Address)，在指定地址(Reg Address)下，写入指定数据(Data)；第二个时序是当前地址读，对于指定设备(Slave Address)，在当前地址指针指示的地址下，读取从机数据(Data)；第三个时序是指定地址读，对于指定设备(Slave Address)，在指定地址(Reg Address)下，读取从机数据(Data)。

2.4.2　I²C 通信的实现

I²C 通信可以使用硬件 I²C 控制器和 GPIO 软件模拟两种方法。

硬件 I²C 控制器就是使用芯片上的 I²C 外设，它有相应的 I²C 驱动电路和专用的 I²C 引脚，效率更高，写代码会相对简单，只要调用 I²C 的控制函数即可，不需要用代码去控制 SCL、SDA 的各种高低电平变化来实现 I²C 协议，只需要将 I²C 协议中的可变部分(如：从设备地址、传输数据等等)通过函数传参给控制器，控制器自动按照 I²C 协议实现传输。

常用的 I²C 通信方式是 GPIO 通过软件模拟实现。通过使用任意 IO 口去模拟实现 I²C 通信协议，手动写代码去控制 IO 口的电平变化，模拟 I²C 协议的时序，实现 I²C 的信号和数据传输，下面会讲到根据通信协议如何用软件去模拟。

一、OLED 显示屏的驱动

OLED12864 是一种高性能的图形点阵液晶显示模块，采用 128×64 点阵分辨率，能够清晰展示文字、图形和简单图像。通常支持多种接口方式，包括 I²C、SPI 等串行通信协议。广泛应用在工业仪表、医疗设备、消费电子产品(如智能手环、便携式仪器等)。

下面以 I²C 接口的 OLED12864 来说明驱动程序设计。

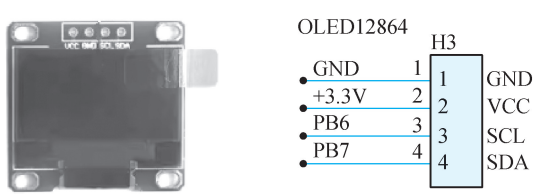


图 2.16　OLED12864 接线图

表 2－8　OLED12864 指令表

指令	功能	指令	功能
0xAE	关闭显示	0x20	设置内存地址模式(页/列/行)
0xAF	开启显示	0x21	设置列地址(低四位)
0xA8	设置驱动路数(16～64)	0x22	设置页地址(低三位)
0xD3	设置显示偏移	0xB0－0xB7	页地址设置(对应页码 0～7)
0x81	设置对比度(亮度调节)	0x10/0x1F	列地址高四位/低四位
0xD9	设置预充电周期	0xE0/0xEE	设置 Gamma 曲线(亮度校正)
0xDA	设置 COM 引脚配置	0xA1/0xA0	设置列扫描方向(正向/反向)
0x8D	开启电荷泵(供电模式)	0xC0/0xC8	设置 COM 扫描方向(正常/重定义)

```
//----------------------- oled.h ----------------------------
#ifndef _OLED_H
#define _OLED_H
#include "stm32f10x.h"
// OLED 控制定义
#define OLED_ADDRESS 0x78   // OLED I²C 地址
// 引脚定义
#define OLED_SCL_PIN GPIO_Pin_6
#define OLED_SDA_PIN GPIO_Pin_7
#define OLED_PORT GPIOB
#define OLED_RCC RCC_APB2Periph_GPIOB
// 函数声明
void OLED_Init(void);
void OLED_Clear(void);
void OLED_ShowString(uint8_t x, uint8_t y, char *str);
void OLED_ShowNum(uint8_t x, uint8_t y, uint32_t num, uint8_t len);
void OLED_Refresh(void);
#endif
//----------------------- oled.c ----------------------------
#include "oled.h"
#include "delay.h"
#include "font.h"   // 字库文件
static uint8_t OLED_GRAM[128][8];   // 显示缓存
// I²C 起始信号
void OLED_I²C_Start(void)
{
    GPIO_SetBits(OLED_PORT, OLED_SDA_PIN);// SDA = 1
    GPIO_SetBits(OLED_PORT, OLED_SCL_PIN);// SCL = 1
```

```
    Delay_us(4);
    GPIO_ResetBits(OLED_PORT, OLED_SDA_PIN);// SDA = 0
    Delay_us(4);
    GPIO_ResetBits(OLED_PORT, OLED_SCL_PIN);// SCL = 0
}
// I²C 停止信号
void OLED_I²C_Stop(void)
{
    GPIO_ResetBits(OLED_PORT, OLED_SCL_PIN); // SCL = 0
    GPIO_ResetBits(OLED_PORT, OLED_SDA_PIN); // SDA = 0
    Delay_us(4);
    GPIO_SetBits(OLED_PORT, OLED_SCL_PIN); // SCL = 1
    Delay_us(4);
    GPIO_SetBits(OLED_PORT, OLED_SDA_PIN); // SDA = 1
}

// I²C 等待应答
uint8_t OLED_I²C_WaitAck(void)
{
    uint8_t ack;

    GPIO_SetBits(OLED_PORT, OLED_SDA_PIN);
    Delay_us(1);
    GPIO_SetBits(OLED_PORT, OLED_SCL_PIN);
    Delay_us(1);
    if(GPIO_ReadInputDataBit(OLED_PORT, OLED_SDA_PIN) == 0)
        ack = 0;
    else
        ack = 1;
    GPIO_ResetBits(OLED_PORT, OLED_SCL_PIN);
    return ack;
}
// I²C 写一个字节
void OLED_I²C_WriteByte(uint8_t data)
{
    uint8_t i;
    for(i = 0; i < 8; i ++)
    {
        GPIO_ResetBits(OLED_PORT, OLED_SCL_PIN);
        if(data & 0x80)
            GPIO_SetBits(OLED_PORT, OLED_SDA_PIN);
        else
```

```
            GPIO_ResetBits(OLED_PORT, OLED_SDA_PIN);
        Delay_us(1);
        GPIO_SetBits(OLED_PORT, OLED_SCL_PIN);
        Delay_us(1);
        data <<= 1;
    }
}
// 写命令
void OLED_WriteCmd(uint8_t cmd)
{
    OLED_I²C_Start();
    OLED_I²C_WriteByte(OLED_ADDRESS);
    OLED_I²C_WaitAck();
    OLED_I²C_WriteByte(0x00);  // 写命令
    OLED_I²C_WaitAck();
    OLED_I²C_WriteByte(cmd);
    OLED_I²C_WaitAck();
    OLED_I²C_Stop();
}
// 写数据
void OLED_WriteData(uint8_t data)
{
    OLED_I²C_Start();
    OLED_I²C_WriteByte(OLED_ADDRESS);
    OLED_I²C_WaitAck();
    OLED_I²C_WriteByte(0x40);  // 写数据
    OLED_I²C_WaitAck();
    OLED_I²C_WriteByte(data);
    OLED_I²C_WaitAck();
    OLED_I²C_Stop();
}
// OLED 初始化
void OLED_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 使能 GPIO 时钟
    RCC_APB2PeriphClockCmd(OLED_RCC, ENABLE);
    // 配置 GPIO
    GPIO_InitStructure.GPIO_Pin = OLED_SCL_PIN | OLED_SDA_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(OLED_PORT, &GPIO_InitStructure);
```

```
    Delay_ms(100);
    // OLED 初始化序列
    OLED_WriteCmd(0xAE); // 关闭显示
    OLED_WriteCmd(0x20); // 设置内存地址模式
    OLED_WriteCmd(0x10); // 页地址模式
    OLED_WriteCmd(0xB0); // 设置页起始地址
    OLED_WriteCmd(0xC8); // 设置扫描方向
    OLED_WriteCmd(0x00); // 设置低列地址
    OLED_WriteCmd(0x10); // 设置高列地址
    OLED_WriteCmd(0x40); // 设置起始行
    OLED_WriteCmd(0x81); // 设置对比度
    OLED_WriteCmd(0xFF);
    OLED_WriteCmd(0xA1); // 设置段重映射
    OLED_WriteCmd(0xA6); // 设置正常显示
    OLED_WriteCmd(0xA8); // 设置多路复用率
    OLED_WriteCmd(0x3F);
    OLED_WriteCmd(0xA4); // 全部像素点打开
    OLED_WriteCmd(0xD3); // 设置显示偏移
    OLED_WriteCmd(0x00);
    OLED_WriteCmd(0xD5); // 设置振荡器频率
    OLED_WriteCmd(0xF0);
    OLED_WriteCmd(0xD9); // 设置预充电周期
    OLED_WriteCmd(0x22);
    OLED_WriteCmd(0xDA); // 设置 COM 硬件引脚配置
    OLED_WriteCmd(0x12);
    OLED_WriteCmd(0xDB); // 设置 VCOMH
    OLED_WriteCmd(0x20);
    OLED_WriteCmd(0x8D); // 设置电荷泵
    OLED_WriteCmd(0x14);
    OLED_WriteCmd(0xAF); // 开启 OLED 显示
    OLED_Clear();
    OLED_Refresh();
}
// 清屏
void OLED_Clear(void)
{
    uint8_t i, j;
    for(i = 0; i < 8; i ++)
    {
        for(j = 0; j < 128; j ++)
        {
            OLED_GRAM[j][i] = 0x00;
```

```
        }
    }
}
// 刷新显示
void OLED_Refresh(void)
{
    uint8_t i, j;
    for(i = 0; i < 8; i ++)
    {
        OLED_WriteCmd(0xB0 + i); // 设置页地址
        OLED_WriteCmd(0x00);     // 设置低列地址
        OLED_WriteCmd(0x10);     // 设置高列地址

        for(j = 0; j < 128; j ++)
        {
            OLED_WriteData(OLED_GRAM[j][i]);
        }
    }
}
// 显示字符
void OLED_ShowChar(uint8_t x, uint8_t y, char chr)
{
    uint8_t c = 0, i = 0;
    c = chr - ';  // 得到偏移后的值
    if(x > 128 - 1) {x = 0; y ++;}
    for(i = 0; i < 8; i ++)
    {
        OLED_GRAM[x][y] = F8X16[c *16 + i];
        x ++;
    }
    for(i = 0; i < 8; i ++)
    {
        OLED_GRAM[x][y] = F8X16[c *16 + i + 8];
        x ++;
    }
}
// 显示字符串
void OLED_ShowString(uint8_t x, uint8_t y, char *str)
{
    while(*str != '\0')
    {
        OLED_ShowChar(x, y, *str);
```

```
        x += 8;
        if(x > 120)
        {
            x = 0;
            y ++;
        }
        str ++;
    }
}

// 显示数字
void OLED_ShowNum(uint8_t x, uint8_t y, uint32_t num, uint8_t len)
{
    char str[10];
    uint8_t i;
    for(i = 0; i < len; i ++)
    {
        str[len - i - 1] = num % 10 + '0';
        num /= 10;
    }
    str[len] = '\0';
    OLED_ShowString(x, y, str);
}
//------------------------主程序 main.c ----------------------------
#include "stm32f10x.h"
#include "oled.h"
#include "delay.h"
int main(void)
{
    Delay_Init();     // 初始化延时函数
    OLED_Init();     // 初始化 OLED
    OLED_Clear();     // 清屏
    // 显示第一行文字:"党史知识竞赛"
    OLED_ShowString(0, 0, "党史知识竞赛");
    // 显示第二行数字:5
    OLED_ShowNum(0, 2, 5, 1);
    OLED_Refresh();     // 刷新显示
    while(1)
    {
        // 主循环
    }
}
```

二、AT24C02 数据读写

AT24C02 是 Atmel 公司推出的一款经典 EEPROM 存储器芯片，采用 I^2C 总线接口，在嵌入式系统中广泛应用于数据存储领域。

该芯片存储容量为 2Kbit(256×8bit)，支持字节读写和页写入操作，页写缓冲器为 8 字节。具有写保护功能，当 WP 引脚接高电平时禁止写入操作，有效防止数据误改写。

AT24C02 通过硬件地址引脚可配置多种器件地址，支持同一总线上挂载多片芯片。且性能稳定、接口简单，是单片机系统中存储配置参数、用户设置、运行日志等非易失性数据的理想选择，在工业控制、智能家居、消费电子等领域应用广泛。

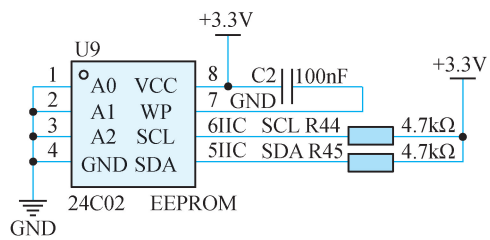


图 2.17　24C02 接线图

AT24C02 的 SCL 接 PB10 引脚，SDA 接 PB11 引脚，编写程序实现 AT24C02 的数据存储与读取。

```
//----------------------- 24C02.h ----------------------------
#ifndef _AT24C02_H
#define _AT24C02_H
#include "stm32f10x.h"
#include "delay.h"
// AT24C02 定义
#define AT24C02_ADDRESS 0xA0   // 器件地址(7 位地址)
// 引脚定义
#define AT24C02_SCL_PIN GPIO_Pin_10
#define AT24C02_SDA_PIN GPIO_Pin_11
#define AT24C02_PORT GPIOB
#define AT24C02_RCC RCC_APB2Periph_GPIOB
// 函数声明
void AT24C02_Init(void);
uint8_t AT24C02_ReadByte(uint16_t ReadAddr);
void AT24C02_WriteByte(uint16_t WriteAddr, uint8_t DataToWrite);
void AT24C02_WritePage(uint16_t WriteAddr, uint8_t *Data, uint8_t Len);
void AT24C02_ReadBuffer(uint16_t ReadAddr, uint8_t *pBuffer, uint16_t Len);
void AT24C02_WriteBuffer(uint16_t WriteAddr, uint8_t *pBuffer, uint16_t Len);
// I²C 底层函数
void AT24C02_I²C_Start(void);
```

```
void AT24C02_I²C_Stop(void);
uint8_t AT24C02_I²C_WaitAck(void);
void AT24C02_I²C_Ack(void);
void AT24C02_I²C_NAck(void);
void AT24C02_I²C_SendByte(uint8_t txd);
uint8_t AT24C02_I²C_ReadByte(unsigned char ack);
#endif
//------------------------ 24C02.c ----------------------------
#include "at24c02.h"
// I²C 初始化
void AT24C02_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 使能 GPIO 时钟
    RCC_APB2PeriphClockCmd(AT24C02_RCC, ENABLE);
    // 配置 GPIO 为开漏输出
    GPIO_InitStructure.GPIO_Pin = AT24C02_SCL_PIN | AT24C02_SDA_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(AT24C02_PORT, &GPIO_InitStructure);
    // 初始化为高电平
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
}
// I²C 起始信号
void AT24C02_I²C_Start(void)
{
    GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    Delay_us(5);
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    Delay_us(5);
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
}
// I²C 停止信号
void AT24C02_I²C_Stop(void)
{
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    Delay_us(5);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    Delay_us(5);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
```

```
    Delay_us(5);
}
// 等待应答
uint8_t AT24C02_I²C_WaitAck(void)
{
    uint8_t ucErrTime = 0;
    GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    Delay_us(1);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    Delay_us(1);
    while(GPIO_ReadInputDataBit(AT24C02_PORT, AT24C02_SDA_PIN))
    {
        ucErrTime ++;
        if(ucErrTime > 250)
        {
            AT24C02_I²C_Stop();
            return 1;
        }
    }
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    return 0;
}
// 产生 ACK 应答
void AT24C02_I²C_Ack(void)
{
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    Delay_us(2);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    Delay_us(2);
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
}
// 不产生 ACK 应答
void AT24C02_I²C_NAck(void)
{
    GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
    Delay_us(2);
    GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
    Delay_us(2);
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
}
// I²C 发送一个字节
void AT24C02_I²C_SendByte(uint8_t txd)
```

```
{
    uint8_t t;
    for(t = 0; t < 8; t ++)
    {
        GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
        if(txd & 0x80)
            GPIO_SetBits(AT24C02_PORT, AT24C02_SDA_PIN);
        else
            GPIO_ResetBits(AT24C02_PORT, AT24C02_SDA_PIN);
        txd <<= 1;
        Delay_us(2);
        GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
        Delay_us(2);
    }
    GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
}
// I²C 读取一个字节
uint8_t AT24C02_I²C_ReadByte(unsigned char ack)
{
    unsigned char i, receive = 0;
    for(i = 0; i < 8; i ++)
    {
        GPIO_ResetBits(AT24C02_PORT, AT24C02_SCL_PIN);
        Delay_us(2);
        GPIO_SetBits(AT24C02_PORT, AT24C02_SCL_PIN);
        receive <<= 1;
       if(GPIO_ReadInputDataBit(AT24C02_PORT, AT24C02_SDA_PIN))
           receive ++;
       Delay_us(1);
    }
    if(! ack)
       AT24C02_I²C_NAck();
    else
       AT24C02_I²C_Ack();
    return receive;
}
// 在 AT24C02 指定地址读取一个字节
uint8_t AT24C02_ReadByte(uint16_t ReadAddr)
{
    uint8_t data;
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS);   // 发送器件地址+写命令
```

```
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_SendByte(ReadAddr);          // 发送要读取的地址
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS | 0x01); // 发送器件地址+读命令
    AT24C02_I²C_WaitAck();
    data = AT24C02_I²C_ReadByte(0);          // 读取数据,不发送 ACK
    AT24C02_I²C_Stop();
    return data;
}
// 在 AT24C02 指定地址写入一个字节
void AT24C02_WriteByte(uint16_t WriteAddr, uint8_t DataToWrite)
{
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS);   // 发送器件地址+写命令
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_SendByte(WriteAddr);         // 发送要写入的地址
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_SendByte(DataToWrite);       // 发送要写入的数据
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_Stop();
    Delay_ms(10);   // 等待写入完成
}
// 页写入(最多 8 字节)
void AT24C02_WritePage(uint16_t WriteAddr, uint8_t *Data, uint8_t Len)
{
    uint8_t i;
    if(Len > 8) Len = 8;   // AT24C02 页大小为 8 字节
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS);   // 发送器件地址+写命令
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_SendByte(WriteAddr);         // 发送起始地址
    AT24C02_I²C_WaitAck();
    for(i = 0; i < Len; i ++)
    {
        AT24C02_I²C_SendByte(Data[i]);       // 发送数据
        AT24C02_I²C_WaitAck();
    }
    AT24C02_I²C_Stop();
    Delay_ms(10);   // 等待写入完成
}
```

```
// 读取多个字节
void AT24C02_ReadBuffer(uint16_t ReadAddr, uint8_t *pBuffer, uint16_t Len)
{
    uint16_t i;
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS);  // 发送器件地址+写命令
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_SendByte(ReadAddr);         // 发送要读取的起始地址
    AT24C02_I²C_WaitAck();
    AT24C02_I²C_Start();
    AT24C02_I²C_SendByte(AT24C02_ADDRESS | 0x01); // 发送器件地址+读命令
    AT24C02_I²C_WaitAck();
    for(i = 0; i < Len; i ++)
    {
        if(i == (Len - 1))
            pBuffer[i] = AT24C02_I²C_ReadByte(0); // 最后一个字节不发送 ACK
        else
            pBuffer[i] = AT24C02_I²C_ReadByte(1); // 发送 ACK
    }
    AT24C02_I²C_Stop();
}
// 写入多个字节(自动分页)
void AT24C02_WriteBuffer(uint16_t WriteAddr, uint8_t *pBuffer, uint16_t Len)
{
    uint16_t i;
    for(i = 0; i < Len; i ++)
    {
        AT24C02_WriteByte(WriteAddr + i, pBuffer[i]);
    }
}
//-----------------------主程序 main.c ----------------------------
#include "stm32f10x.h"
#include "at24c02.h"
#include "delay.h"
#include "stdio.h"
int main(void)
{
    uint8_t write_data[] = {0x12, 0x34, 0x56, 0x78, 0xAB, 0xCD, 0xEF};
    uint8_t read_data[7];
    uint8_t i;

    // 初始化延时函数
```

```
    Delay_Init();
    // 初始化串口(用于调试输出)
    USART1_Init();
    // 初始化 AT24C02
    AT24C02_Init();
    printf("AT24C02 Test Start...\r\n");
    // 测试单字节写入和读取
    AT24C02_WriteByte(0x00, 0xAA);
    printf("Write Data: 0xAA\r\n");
    printf("Read Data: 0x%02X\r\n", AT24C02_ReadByte(0x00));
    // 测试多字节写入和读取
    AT24C02_WriteBuffer(0x10, write_data, sizeof(write_data));
    printf("Write Buffer: ");
    for(i = 0; i < sizeof(write_data); i++)
    {
        printf("0x%02X ", write_data[i]);
    }
    printf("\r\n");
    AT24C02_ReadBuffer(0x10, read_data, sizeof(read_data));
    printf("Read Buffer: ");
    for(i = 0; i < sizeof(read_data); i++)
    {
        printf("0x%02X ", read_data[i]);
    }
    printf("\r\n");

    // 测试页写入
    uint8_t page_data[] = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88};
    AT24C02_WritePage(0x20, page_data, 8);
    printf("Page Write Complete\r\n");

    while(1)
    {
        // 循环读取并显示数据
        printf("Current Data at 0x00: 0x%02X\r\n", AT24C02_ReadByte(0x00));
        Delay_ms(1000);
    }
}
```

2.5 抢答器程序设计

学校团委举办的党史知识竞赛，经过初赛决出了入围决赛的四支队伍。现委托电子创新协会制作一套决赛竞赛系统。要求实现以下功能：

（1）主持人念完题目，喊“开始抢答”的同时按下开始键。

（2）主机检测四支队伍的面板上最先按下的按键，并在 OLED 屏第 1 行上显示抢答队伍号，下方显示四支队伍的得分情况。

（3）主持人根据回答情况打分，正确加 10 分，错误扣 10 分，同时 OLED 屏第 1 行的抢答队伍号清零。

（4）比赛结束后，主持人长按开始键 3 秒以上，显示最终比赛成绩，并将成绩存入 AT24C02 中。

表 2-9　外设与芯片引脚连接情况

类型	外设引脚	MCU 引脚	类型	外设引脚	MCU 引脚
OLED12864	OLED_SCL	PB6	功能键	开始键	PB4
	OLED_SDA	PB7		正确键	PA6
AT24C02	24C02_SCL	PB10		错误键	PD1
	24C02_SDA	PB11	抢答键	1～4	PA4，PA5，PD3，PD4

```
#include "stm32f10x.h"
#include "stm32f10x.h"
#include "oled.h"
#include "at24c02.h"
#include "key.h"
#include "delay.h"
#include <stdio.h>

// 队伍定义
typedef enum {
    TEAM_NONE = 0,
    TEAM_1 = 1,
    TEAM_2 = 2,
    TEAM_3 = 3,
    TEAM_4 = 4
} Team_t;

// 系统状态
```

```
typedef enum {
    STATE_IDLE = 0,          // 空闲状态
    STATE_ANSWERING = 1,     // 抢答状态
    STATE_SCORING = 2,       // 评分状态
    STATE_GAME_OVER = 3      // 比赛结束
} SystemState_t;

// 全局变量
volatile uint8_t g_team_scores[4] = {0, 0, 0, 0};  // 四支队伍得分
volatile Team_t g_current_team = TEAM_NONE;         // 当前抢答队伍
volatile SystemState_t g_system_state = STATE_IDLE; // 系统状态
volatile uint32_t g_start_press_time = 0;           // 开始按键按下时间

// 函数声明
void System_Init(void);
void Display_Main_Screen(void);
void Display_Final_Score(void);
void Save_Scores_To_EEPROM(void);
void Read_Scores_From_EEPROM(void);
void Process_Answer_Key(void);
void Process_Start_Key(void);
void Process_Correct_Wrong_Keys(void);
Team_t Check_First_Team_Pressed(void);

int main(void)
{
    System_Init();
    while(1)
    {
        Process_Start_Key();
        Process_Answer_Key();
        Process_Correct_Wrong_Keys();
        Delay_ms(10);
    }
}
// 系统初始化
void System_Init(void)
{
    Delay_Init();           // 延时初始化
    KEY_Init();             // 按键初始化
    OLED_Init();            // OLED 初始化
    AT24C02_Init();         // EEPROM 初始化
```

```
    // 从 EEPROM 读取历史成绩(可选)
    // Read_Scores_From_EEPROM();
    // 初始显示
    OLED_Clear();
    Display_Main_Screen();
}
// 处理开始按键
void Process_Start_Key(void)
{
    static uint8_t last_start_state = 1;
    uint8_t current_start_state = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_4);
    if (last_start_state == 1 && current_start_state == 0) // 按下
    {
        g_start_press_time = Get_SystemTick();
    }
    else if (last_start_state == 0 && current_start_state == 1) // 释放
    {
        uint32_t press_duration = Get_SystemTick() - g_start_press_time;
        if (press_duration > 3000) // 长按 3 秒以上
        {
           if (g_system_state != STATE_GAME_OVER)
           {
               g_system_state = STATE_GAME_OVER;
               Display_Final_Score();
               Save_Scores_To_EEPROM();
           }
           else
           {
               g_system_state = STATE_IDLE;
               Display_Main_Screen();
           }
        }
        else // 短按
        {
           if (g_system_state == STATE_IDLE)
           {
               g_system_state = STATE_ANSWERING;
               g_current_team = TEAM_NONE;
               OLED_Clear();
               OLED_ShowString(1, 1, "Ready...", 16);
           }
        }
```

```
    }
    last_start_state = current_start_state;
}
// 处理抢答按键
void Process_Answer_Key(void)
{
    if (g_system_state == STATE_ANSWERING && g_current_team == TEAM_NONE)
    {
        Team_t first_team = Check_First_Team_Pressed();
        if (first_team != TEAM_NONE)
        {
            g_current_team = first_team;
            g_system_state = STATE_SCORING;
            Display_Main_Screen();
        }
    }
}
// 处理正确/错误按键
void Process_Correct_Wrong_Keys(void)
{
    static uint8_t last_correct_state = 1;
    static uint8_t last_wrong_state = 1;
    uint8_t current_correct_state = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_6);
    uint8_t current_wrong_state = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_1);
    // 处理正确按键
    if (last_correct_state == 1 && current_correct_state == 0)
    {
        if (g_system_state == STATE_SCORING && g_current_team != TEAM_NONE)
        {
            g_team_scores[g_current_team - 1] += 10;
            g_current_team = TEAM_NONE;
            g_system_state = STATE_IDLE;
            Display_Main_Screen();
        }
    }
    // 处理错误按键
    if (last_wrong_state == 1 && current_wrong_state == 0)
    {
        if (g_system_state == STATE_SCORING && g_current_team != TEAM_NONE)
        {
            if (g_team_scores[g_current_team - 1] >= 10)
            {
```

```
                g_team_scores[g_current_team - 1] -= 10;
            }
            g_current_team = TEAM_NONE;
            g_system_state = STATE_IDLE;
            Display_Main_Screen();
        }
    }
    last_correct_state = current_correct_state;
    last_wrong_state = current_wrong_state;
}
// 检查最先按下的队伍
Team_t Check_First_Team_Pressed(void)
{
    uint8_t team_keys[4] = {
        GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_4),
        GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_5),
        GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3),
        GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_4)
    };
    for (int i = 0; i < 4; i ++)
    {
        if (team_keys[i] == 0) // 按键按下
        {
            return (Team_t)(i + 1);
        }
    }
    return TEAM_NONE;
}
// 显示主界面
void Display_Main_Screen(void)
{
    OLED_Clear();
    // 显示抢答队伍
    if (g_current_team != TEAM_NONE)
    {
        char team_str[16];
        sprintf(team_str, "Team: %d", g_current_team);
        OLED_ShowString(1, 1, team_str, 16);
    }
    else
    {
        OLED_ShowString(1, 1, "Team: --", 16);
```

```
    }
    // 显示各队得分
    for (int i = 0; i < 4; i ++)
    {
        char score_str[16];
        sprintf(score_str, "Team %d: %3d", i + 1, g_team_scores[i]);
        OLED_ShowString(1, 2 + i, score_str, 16);
    }
}
// 显示最终成绩
void Display_Final_Score(void)
{
    OLED_Clear();
    OLED_ShowString(1, 1, "Final Score:", 16);
    for (int i = 0; i < 4; i ++)
    {
        char final_str[16];
        sprintf(final_str, "Team %d: %3d", i + 1, g_team_scores[i]);
        OLED_ShowString(1, 2 + i, final_str, 16);
    }
}
// 保存成绩到 EEPROM
void Save_Scores_To_EEPROM(void)
{
    for (int i = 0; i < 4; i ++)
    {
        AT24C02_WriteOneByte(i, g_team_scores[i]);
        Delay_ms(5);
    }
}
// 从 EEPROM 读取成绩
void Read_Scores_From_EEPROM(void)
{
    for (int i = 0; i < 4; i ++)
    {
        g_team_scores[i] = AT24C02_ReadOneByte(i);
    }
}
```

按键初始化文件

```
#include "key.h"
#include "stm32f10x.h"
void KEY_Init(void)
```

```
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 开启时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
RCC_APB2Periph_GPIOD, ENABLE);
    // 开始键 PB4
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    // 正确键 PA6
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    // 错误键 PD1
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    // 抢答键 PA4, PA5, PD3, PD4
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

2.6 项目小结

本项目作为嵌入式系统进阶实践，是从“单一模块控制”到“多模块协同系统”的关键跨越。通过开发多路抢答器，不仅深化了 STM32 中断、I^2C 等外设的技术理解，更建立了“需求分析→硬件搭建→软件编程→调试优化”的完整嵌入式开发流程思维。总体而言，圆满达成项目二“多路抢答＋得分存储”的核心目标，为后续复杂嵌入式项目（如物联网环境监测终端、工业控制设备）奠定了模块整合与系统设计的基础，是嵌入式学习从“入门”到“进阶”的重要里程碑。

实战练习

基础任务

1. STM32 外部中断触发方式不包括以下哪种？（　　）。

A. 上升沿触发　　B. 下降沿触发　　C. 高低电平触发　　D. 随机触发

2. I^2C 通信中，SDA 线的控制权在什么情况下会从主机转交从机？

3. 4 路抢答器中，若 2 号选手和 3 号选手同时按下按键，如何通过软件确保仅识别“先按者”？

4. 简述 AT24C02 芯片的作用，以及“掉电保存”的原理。

5. 编写按键扫描函数，实现“按下按键后 LED 点亮，松手后 LED 熄灭”，要求包含软件消抖（延时 10 ms），按键接 PA0（内部上拉输入），LED 接 PB0（低电平点亮）。

6. 简述 I^2C 通信的“起始条件”和“终止条件”，并写出软件实现的核心代码片段（基于 GPIO 模拟 I^2C）。

7. 4 路抢答器中，数码管初始显示“0000”，1 号选手抢答成功后显示“0001”，2 秒后切换显示该选手当前得分（初始得分为 0），编写数码管显示切换的核心逻辑代码。

8. 为什么 I^2C 总线的 SCL 和 SDA 引脚需要配置为“开漏输出”模式？

进阶挑战

1. 4 路抢答器功能升级：增加倒计时与犯规判断

需求：在原有抢答功能基础上，增加“30 秒倒计时”（用 TIM2 定时器实现）。

要求：

① 倒计时时间到后，蜂鸣器长鸣 2 秒，锁定所有抢答按键；

② 主持人按“重置按键”PA5 后，重置倒计时与抢答锁定状态；

③ 犯规记录需存入 AT24C02 的 0x04—0x07 地址（对应 4 路选手的犯规次数）。

2. 8 路抢答器与上位机通信：串口数据上报

需求：4 路扩展为 8 路（按键 PA0—PA7、LED 对应），通过 USART1 与上位机双向交互。

要求：

① 8 路按键触发后 100 ms 内，上报“抢答组别、抢答时间、是否犯规”；

② 接收上位机指令（如“RESET”重置、“SCORE＋1＃1”给 1 组加分），执行后返回结果。

项目三 激光测距仪设计

学习目标

扫码可见本项目微课

知识目标

1. 理解激光测距的基本原理。
2. 掌握 STM32 串口接收功能的配置与使用。
3. 学会使用 STM32 标准库进行程序开发。
4. 实现完整的测距数据采集与显示系统。

能力目标

1. USART 的查询收发模式及阻塞情况分析。
2. USART 的中断收发程序设计。
3. USART 的 DMA 通信程序设计。
4. USART 的多通道协同通信程序设计。

素质目标

本项目旨在通过激光测距仪的开发，达成以下三方面素质目标：

1. 技术溯源与系统认知

了解激光测距技术的发展历程。引导学生建立技术迭代的系统观念，培养其从历史视角分析工程问题的能力。

2. 核心接口与工程实践

掌握 USART（通用同步异步收发器）在微控制器与测距模块间进行指令发送与数据接收的关键应用。通过实际配置波特率、数据帧格式，并实现稳定可靠的双工通信，将理论协议转化为解决设备间“对话”的实际工程能力。

3. 创新思维与问题解决

鼓励在现有方案基础上进行思考与改进，例如优化测量流程以提升效率、探索数据滤波算法或拓展其应用于特定场景（如简易三维扫描）。旨在激发主动探索精神，培养在约束条件下定义问题、创造性地设计方案并验证的创新能力。

任务描述

本项目基于 STM32 单片机，设计并实现一款激光测距仪。该系统可发射激光测距模块的测量距离信号并实时显示距离信息等功能。

任务实施

3.1 项目背景

常用的测距仪主要有声波测距仪和光电式测距仪两类产品。

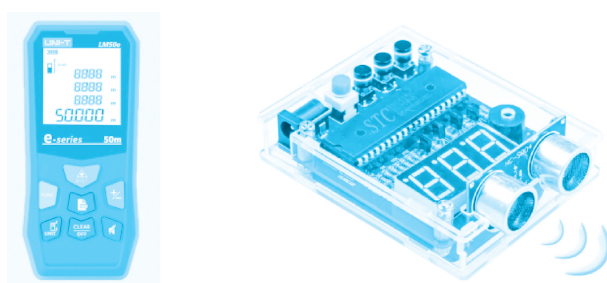


图 3.1　激光测距仪与超声波测距仪

声波测距是利用声波的反射特性而进行测量的一种仪器，一般采用超声波作为调制对象。超声波发射器向某一方向发射超声波，在发射同时开始计时，超声波在空气中传播，途中碰到障碍物就立即返回来，超声波接收器收到反射波就立即中断停止计时。通过不断检测产生波发射后遇到障碍物所反射的回波，从而测出发射超声波和接收到回波的时间差 T，然后求出距离 L。

由于声波具有扩散传播特性，如果超波扩散范围内有比被测物体更近的障碍物，超声波就会提前被反射，测得的距离就是模块到障碍物的距离。超声波这一特性被广泛应用到倒车雷达上。

图 3.2　超声波倒车雷达

如果要测距被测物体的分布点距离信息，就需要采用具有定向发光的激光测距仪。

激光测距仪一般采用两种方式来测量距离：脉冲法和相位法。脉冲法测距时，测距仪发射出的激光经被测量物体的反射后又被测距仪接收，测距仪同时记录激光往返的时间。光速和往返时间的乘积的一半，就是测距仪和被测量物体之间的距离。脉冲法测量距离的精度是一般是在±1 米左右。另外，此类测距仪的测量盲区一般是 15 米左右。

激光测距是光波测距中的一种测距方式，如果光以速度 c 在空气中传播，在 A、B 两点间往返一次所需时间为 t，则 A、B 两点间距离 D 可用下列表示。

$$D=\frac{c\times t}{2} \tag{3-1}$$

式中：c 为光在大气中传播的速度，等于 299 792 458 m/s；t 为光往返 A、B 一次所需的时间。

由上式可知，要测量 A、B 距离实际上是要测量光传播的时间 t，激光测距仪通常采用相位式方式测量时间。

相位式激光测距仪是用无线电波段的频率，对激光束进行幅度调制并测定调制光往返测线一次所产生的相位延迟，再根据调制光的波长，换算此相位延迟所代表的距离。即用间接方法测定出光经往返测线所需的时间。

相位式激光测距仪一般应用在精密测距中，其精度高，可达毫米级。为了有效地反射信号，并使测定的目标限制在与仪器精度相称的某一特定点上，对这种测距仪都配置了被称为合作目标的反射镜。

若调制光角频率为 ω，在待测量距离 D 上往返一次产生的相位延迟为 Φ，则对应时间 t 可表示为：

$$t=\frac{\Phi}{\omega} \tag{3-2}$$

代入(3-1)式，距离 D 可表示为

$$D=\frac{c\times t}{2}=\frac{c\times\Phi}{2\omega}=\frac{c\times(N\pi+\Delta\Phi)}{4\pi f}=\frac{c}{4f(N+\Delta N)}$$

式中：

Φ——信号往返测线一次产生的总的相位延迟。

ω——调制信号的角频率，$\omega=2\pi f$。

U——单位长度，数值等于 1/4 调制波长。

N——测线所包含调制半波长个数。

$\Delta\Phi$——信号往返测线一次产生相位延迟不足 π 部分。

ΔN——测线所包含调制波不足半波长的小数部分。

$\Delta N=\Phi/\omega$

在给定调制和标准大气条件下，频率 $c/(4\pi f)$ 是一个常数，此时测量的距离变成了测线所包含半波长个数的测量和不足半波长的小数部分的测量，即测 N 或 Φ，由于近代精密机械加工技术和无线电测相技术的发展，已使 Φ 的测量达到很高的精度。

为了测得不足 π 的相角 Φ，可以通过不同的方法来进行测量，通常应用最多的是延迟测

相和数字测相，短程激光测距仪均采用数字测相原理来求得 Φ。

表 3－1　常用测距方案的特性对比

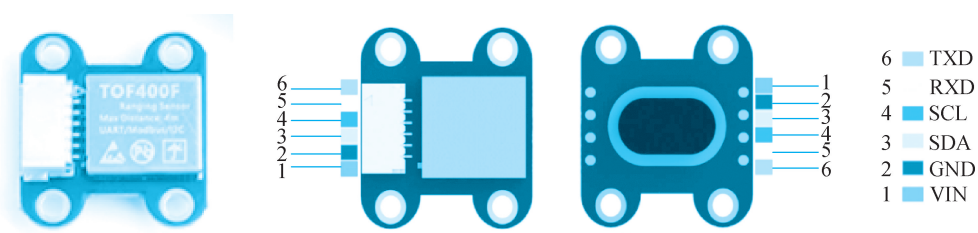
特性	VL53L1X（ToF 激光）	红外测距	超声波
精度	毫米级	厘米级	厘米级
抗干扰能力	强（不受温湿度影响）	弱（易受光干扰）	弱（受温湿度影响）
响应速度	快（毫秒级）	中等	慢（受声速限制）

3.2　TOF 激光测距模块介绍

3.2.1　TOF400F 传感器简介

TOF400F 测距传感器是基于 ST 的 VL53L1X 设计制造的可提供精确和可重复的远距离测量功能的一款激光测距模块。它通过测量光脉冲从发射到被目标反射后返回传感器所需的时间来计算距离。

TOF400F 测距范围最高可达 4 m，并可根据需求选择高精度或者远距测试模式。高精度模式的量程为 1.3 m，测量周期为 30 ms。长距离模式的量程为 4 m，测量周期为 200 ms，测量误差在±5%以内，测量的范围与目标物体的反射率无关，可工作在高红外光的环境下。



(a) 实物图　　(b) 引脚图

图 3.3　TOF400F 激光测距传感器

TOF400F 同时支持串口模式，串口模拟 Modbus 模式以及 IIC 模式，很好适应各种不同的应用场景。其引脚描述如表 3－2 所示。

表 3－2　TOF400F 引脚描述

引脚序号	引脚名称	属性	功能描述
1	VIN	/	电源正极 3～5 V
2	GND	/	电源地
3	SDA	输入/输出	IIC 数据口

续 表

引脚序号	引脚名称	属性	功能描述
4	SCL	输入	IIC 时钟口
5	RX	INPUT	串口输入,TTL 电平
6	TX	OUTPUT	串口输出,TTL 电平

3.2.2 TOF400F 的应用

TOF400F 传感器的工作模式如表 3-3 所示,使用最多的是串口模式。

表 3-3 TOF400F 的工作模式

工作模式	切换方式	详解
串口模式(默认)	无需切换	单片机串口数据收发,实际遵循 Modbus_RTU 协议,配套串口助手可方便设置与调试
Modbus 协议模式	无需切换	可以用标准的 Modbus_RTU 访问寄存器,方便与工业设备交互。可设置单独地址、公用广播地址,方便实现多模块协同工作
IIC 模式	指令切换	可直接使用 IIC 访问传感器芯片

TOF400F 传感器与 MCU 的串口通信方式连接方式如图 3.4 所示。

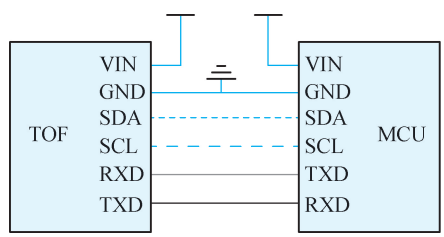


图 3.4 TOF400F 的串行通信连接

串口+modbus 模式

表 3-4 串口通信协议说明

默认波特率 Bits per Second:	115200
数据位 Data Bits:	8
校验位 Parity:	无 None
停止位 Stop bits:	1
流控制 Stop bits:	无 None

表 3-5　modbus 格式说明

读取命令(MCU→TOF400,以 0x01 号从机为例)							
从机地址	功能号	寄存器地址高位	寄存器地址低位	数据高位	数据低位	CRC 校验低位	CRC 校验高位
DR	RW	RegH	RegL	DH	DL	CL	CH
0x01	0x03	RegH	RegL	DH	DL	CL	CH
传感器返回(TOF400F→MCU)							
从机地址	功能号	数据字节个数	数据字节 1 高位	数据字节 1 低位	…	CRC 校验低位	CRC 校验高位
DR	RW	D	DATA1H	DATA1L	…	CL	CH
0x01	0x03	D	DATA1H	DATA1L	…	CL	CH

举例子:主机发送:01 03 00 10 00 01 85 CF 读取 1 从机的测距值
模块回复:01 03 02 00 65 78 6F 测距值为 0x0065(101 mm)

表 3-6　设置返回指令

写命令(以 0x01 号从机为例)							
从机地址	功能号	寄存器地址高位	寄存器地址低位	数据高位	数据低位	CRC 校验低位	CRC 校验高位
DR	RW	RegH	RegL	DH	DL	CL	CH
0x01	0x06	RegH	RegL	DH	DL	CL	CH
传感器返回							
从机地址	功能号	寄存器地址高位	寄存器地址低位	数据高位	数据低位	CRC 校验低位	CRC 校验高位
DR	RW	RegH	RegL	DH	DL	CL	CH
0x01	0x06	RegH	RegL	DH	DL	CL	CH

举例子:主机发送:01 06 00 04 00 01 09 CB 设置 1 从机的测距模式为高精度
模块回复:01 06 00 04 00 01 09 CB 设置成功响应
特别说明:CRC 校验规则为 CRC-16/MODBUS X16+X15+X2+1

表 3-7　寄存器列表

类别	数据地址	数据	功能	W/R
特殊寄存器	0x0001	0xAA55	恢复默认参数	只写
		0x1000	重启	
		0x0000	测试通信	
设备地址寄存器	0x0002	0xXXXX	0 为广播地址	可读可写

续 表

类别	数据地址	数据	功能	W/R
波特率寄存器	0x0003	0x0001	1:38400	可读可写
		0x0002	2:9600	
		0x0003/else	其他:115200	
量程寄存器	0x0004	0x0000	0:默认,30 ms,1.3 m	可读可写
		0x0001	1:长距离,200 ms,4 m	
连续输出控制寄存器	0x0005	0x0000	0:不自输出	可读可写
		0xXXXX	XX:XXms	
加载校准寄存器	0x0006	0xXXXX	0:不加载;1:加载	可读可写
偏移修正值寄存器	0x0007	0xXXXX	偏移修正值	可读可写
xtalk 修正值寄存器	0x0008	0xXXXX	xtalk 修正值	可读可写
禁止 iic 使能寄存器	0x0009	0x0000	0:不禁止(默认)	可读可写
		0x0001	1:禁止(MCU 释放 io)	
测量结果	0x0010	0x0001	距离值:mm	只读
offset 校准寄存器	0x0020	0xXXXX	xx:实际值为 xx,推荐 14 cm	只写
xtalk 校准寄存器	0x0021	0xXXXX	xx:实际值为 xx	只写

3.3 USART 通信基础

串口分多种类别,广义的串口有常见的 RS232、RS485 串口,RJ45 网线接口,USB(Universal Serial BUS)通用串行总线等。这里我们特指单片机中的 TTL 串口,如图 3.5 所示,属于全双工通信方式。设计和开发产品过程中,程序调试、数据传输和版本升级等,都离不开串口接口。

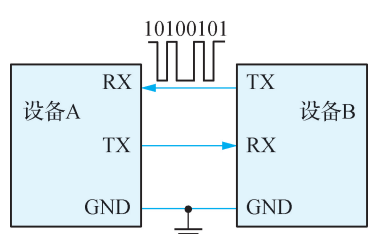


图 3.5 串口通信接线图

3.3.1　USART 介绍

通用同步异步收发器(USART)提供了一种灵活的方法与使用工业标准 NRZ 异步串行数据格式的外部设备之间进行全双工数据交换。USART 利用分数波特率发生器提供宽范围的波特率选择。它支持同步单向通信和半双工单线通信,也支持 LIN(局部互连网),智能卡协议和 IrDA(红外数据组织)SIR ENDEC 规范,以及调制解调器(CTS/RTS)操作。它还允许多处理器通信。使用多缓冲器配置的 DMA 方式,可以实现高速数据通信。

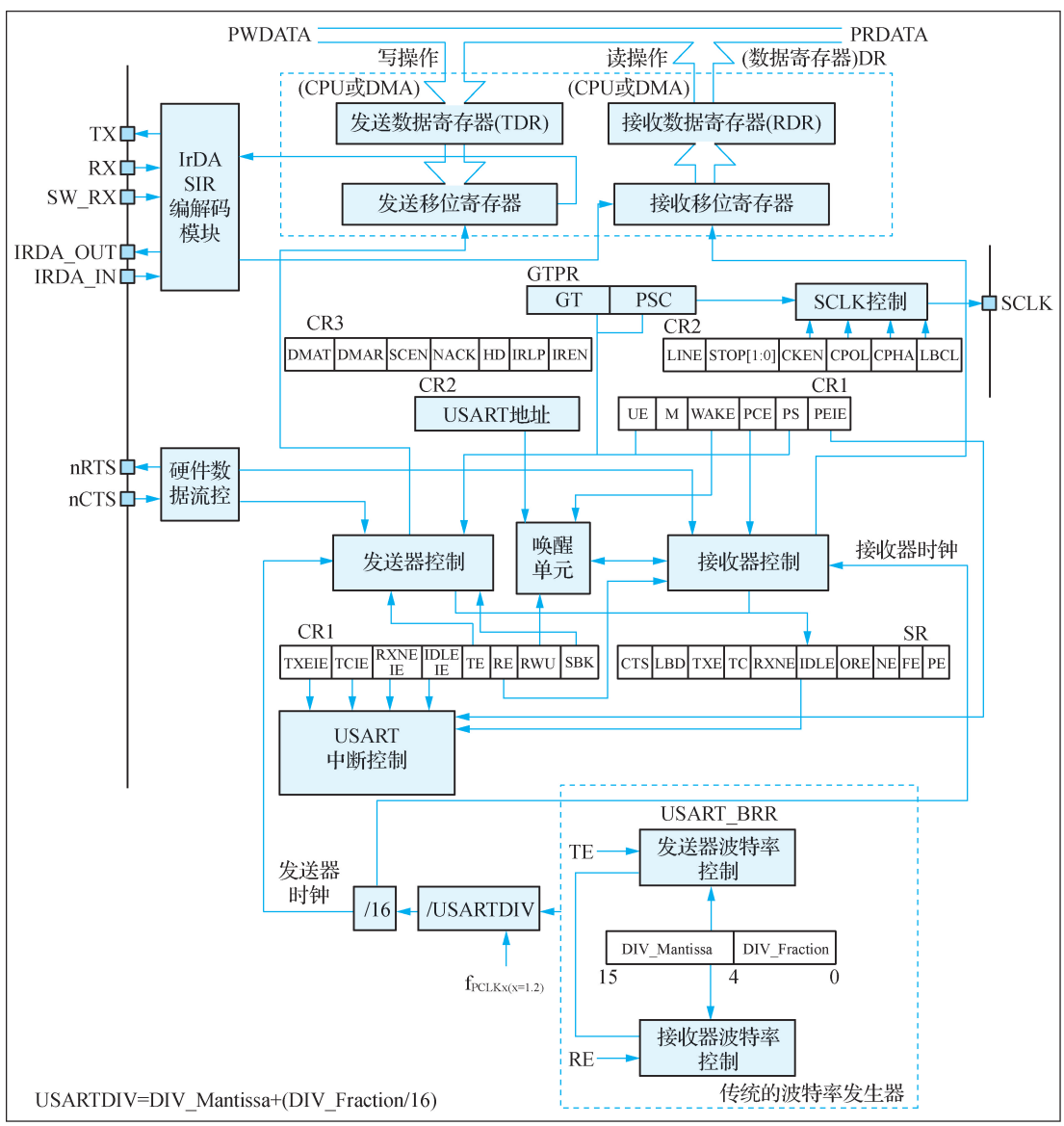


图 3.6　USART 框图

USART 模块框图可以概括为以下几个核心部分,它们协同工作以实现串行通信。

一、发送器(Transmitter)

发送数据寄存器(TDR):CPU 或 DMA 将待发送数据写入此寄存器。

发送移位寄存器:从 TDR 加载数据,并按位通过 TX 引脚串行输出(LSB 或 MSB 优先)。

控制逻辑:检查状态寄存器(SR)中的 TXE(发送数据寄存器空)和 TC(发送完成)标志。

二、接收器(Receiver)

接收移位寄存器:从 RX 引脚按位采样并接收串行数据。

接收数据寄存器(RDR):将接收到的完整字节从移位寄存器传输到 RDR,供 CPU 或 DMA 读取。

控制逻辑:检测 RXNE(接收数据寄存器非空)标志,支持过载错误检测。

三、波特率发生器(Baud Rate Generator)

$$\text{BaudRate}=\frac{f_{ck}}{16\times \text{USARTDIV}}$$

BRR 寄存器:16 位寄存器(高 4 位为小数部分,低 12 位为整数部分),存储分频值 USARTDIV。

四、控制与状态寄存器

CR1/CR2/CR3:配置工作模式(数据位、停止位、校验、中断使能等)。

SR:状态标志位(TXE、TC、RXNE、ORE、FE、PE 等)。

GTPR:用于智能卡模式。

五、中断与 DMA 接口

中断控制器(NVIC):支持 TXE、TC、RXNE、ORE、PE 等中断请求。

发送 DMA 请求:TXE 事件触发 DMA 传输。

接收 DMA 请求:RXNE 事件触发 DMA 传输。

六、引脚接口

TX:发送引脚(推挽输出)。

RX:接收引脚(浮空输入或上拉)。

可选硬件流控制:RTS 请求发送(输出);CTS 清除发送(输入)。

发送流程:CPU/DMA→TDR→发送移位寄存器→TX 引脚。

接收流程:RX 引脚→接收移位寄存器→RDR→CPU/DMA。

字长可以通过编程 USART_CR1 寄存器中的 M 位,选择成 8 或 9 位。在起始位期间,TX 脚处于低电平,在停止位期间处于高电平。

空闲符号被视为完全由‘1’组成的一个完整的数据帧,后面跟着包含了数据的下一帧的

开始位(‘1’的位数也包括了停止位的位数)。

断开符号被视为在一个帧周期内全部收到‘0’(包括停止位期间,也是‘0’)。在断开帧结束时,发送器再插入 1 或 2 个停止位(‘1’)来应答起始位。

发送和接收由一共用的波特率发生器驱动,当发送器和接收器的使能位分别置位时,分别为其产生时钟。

3.3.2　USART 发送器

TDR 是用户写入待发送数据的缓存寄存器。在 8 位数据模式下,TDR 的低 8 位有效;在 9 位模式下,全部 9 位均有效。写入 TDR 的数据会自动加载到发送移位寄存器中,当检测到 TDR 为空时,硬件会设置状态标志,允许用户写入下一字节数据。

发送移位寄存器从 TDR 获取数据,并按比特位从高位到低位通过 TX 引脚依次移出。移位操作由波特率发生器提供的时钟驱动,确保数据以精确的时序发送。

发送器根据 M 位的状态发送 8 位或 9 位的数据字。当发送使能位(TE)被设置时,发送移位寄存器中的数据在 TX 脚上输出,相应的时钟脉冲在 CK 脚上输出。

在 USART 发送期间,在 TX 引脚上首先移出数据的最低有效位。USART_DR 寄存器包含了一个内部总线和发送移位寄存器之间的缓冲器。每个字符之前都有一个低电平的起始位;之后跟着的停止位,可配置为 0.5、1、1.5 和 2 个停止位。

3.3.3　USART 接收器

接收器是一个高度可配置的串行数据接收模块,负责将串行数据流转换为并行数据供 CPU 处理。

其核心工作流程始于引脚检测:当使能接收器后,RX 引脚开始监视起始位(一个由高到低的跳变)。随后,接收控制器按照预先设定的波特率对数据位进行采样,通常采用 16 倍过采样技术以提高抗噪能力。数据经串并转换后,被送入接收数据寄存器(RDR)。

接收过程支持多种数据格式,包括数据位长度(8/9 位)、停止位数量(0.5/1/1.5/2 位)和奇偶校验位,可通过控制寄存器灵活配置。接收器还具备噪声检测、帧错误检测和溢出错误检测等硬件错误管理功能,能通过状态寄存器或中断/DMA 请求及时通知 CPU。

USART 可以根据 USART_CR1 的 M 位接收 8 位或 9 位的数据字。

起始位侦测:在 USART 中,如果辨认出一个特殊的采样序列,那么就认为侦测到一个起始位。

该序列为:1 1 1 0 X 0 X 0 X 0 0 0 0

在 USART 接收期间,数据的最低有效位首先从 RX 脚移进。在此模式里,USART_DR 寄存器包含的缓冲器位于内部总线和接收移位寄存器之间。

配置步骤:

第一步　将 USART_CR1 寄存器的 UE 置 1 来激活 USART。

第二步　编程 USART_CR1 的 M 位定义字长。

第三步　在 USART_CR2 中编写停止位的个数。

第四步　如果需多缓冲器通信,选择 USART_CR3 中的 DMA 使能位(DMAR)。按多缓冲器通信所要求的配置 DMA 寄存器。

第五步　利用波特率寄存器 USART_BRR 选择希望的波特率。

第六步　设置 USART_CR1 的 RE 位。激活接收器，使它开始寻找起始位。

当一个字符被接收到时，RXNE 位被置位。它表明移位寄存器的内容被转移到 RDR。换句话说，数据已经被接收并且可以被读出（包括与之有关的错误标志）。如果 RXNEIE 位被设置，产生中断。在接收期间如果检测到帧错误，噪音或溢出错误，错误标志将被置起。在多缓冲器通信时，RXNE 在每个字节接收后被置起，并由 DMA 对数据寄存器的读操作而清零。在单缓冲器模式里，由软件读 USART_DR 寄存器完成对 RXNE 位清除。RXNE 标志也可以通过对它写 0 来清除。RXNE 位必须在下一字符接收结束前被清零，以避免溢出错误。

注意：在接收数据时，RE 位不应该被复位。如果 RE 位在接收时被清零，当前字节的接收被丢失。

3.4 USART 程序设计

3.4.1 串口配置

USART 的配置需要考虑映射到 MCU 的哪个引脚。USART1、USATR2 和 USART3 均有多个通道，分别称为没有重映像、部分重映像和完全重映像。如表 3－8～3－10 所示。

表 3－8　USART1 重映像

复用功能	USART1_REMAP=0	USART1_REMAP=1
USART1_TX	PA9	PB6
USART1_RX	PA10	PB7

表 3－9　USART2 重映像

复用功能	USART2_REMAP=0	USART2_REMAP=1(1)
USART2_CTS	PA0	PD3
USART2_RTS	PA1	PD4
USART2_TX	PA2	PD5
USART2_RX	PA3	PD6
USART2_CK	PA4	PD7

表 3－10　USART3 重映像

复用功能	USART3_REMAP[1：0]=00(没有重映像)	USART3_REMAP[1：0]=01（部分重映像）(1)	USART3_REMAP[1：0]=11（完全重映像）(2)
USART3_TX	PB10	PC10	PD8
USART3_RX	PB11	PC11	PD9
USART3_CK	PB12	PC12	PD10
USART3_CTS	PB13		PD11
USART3_RTS	PB14		PD12

例如，USART2 的默认的端口在 PA 口，现需通过重印像到 PD 口，可以使用下面的语句。

```
//使能串口 21,GPIOD,AFIO 总线
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO,ENABLE);
RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);
GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);//重映射到 PD 端口
```

串口 1 是最常用的通信接口，还是串行下载程序的专用接口。以下是 USART1 的配置函数。

```
void Usart1_Configuration(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure; // GPIO 库函数结构体
    USART_InitTypeDef  USART_InitStructure;// USART 库函数结构体
    USART_ClockInitTypeDef USART_ClockInitStructure;// USART 时钟函数结构体
    //使能 APB2 外设时钟
     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO | RCC_
APB2Periph_USART1, ENABLE);
    /*Configure USART1 Tx (PA9) as alternate function push-pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;// PA9 时钟速度 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);// 将 GPIO A 端口设置为特定的工作模式和参数
    /*Configure USART1 Rx (PA10) as input floating */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;//上拉输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    USART_InitStructure.USART_BaudRate = 9600; //波特率 9600
    USART_InitStructure.USART_WordLength = USART_WordLength_8b; // 8 位数据
    USART_InitStructure.USART_StopBits = USART_StopBits_1; // 1 个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No; //奇偶失能
```

```
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_
None; //硬件流控制失能
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //发送、接收使能

    USART_ClockInitStructure.USART_Clock = USART_Clock_Disable;//禁止时钟输出
    USART_ClockInitStructure.USART_CPOL = USART_CPOL_Low;//空闲时钟为低电平
    USART_ClockInitStructure.USART_CPHA = USART_CPHA_2Edge;//在第二个时钟边沿采样
    USART_ClockInitStructure.USART_LastBit = USART_LastBit_Disable;//最后一位数据的
时钟脉冲不在 SCLK 引脚输出脉冲

    USART_ClockInit(USART1, &USART_ClockInitStructure);
    USART_Init(USART1, &USART_InitStructure);//初始化结构体
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //使能串口 1 接收及发送中断
    USART_ClearFlag(USART1, USART_FLAG_TC);
    USART_Cmd(USART1, ENABLE); //使能串口 1
}
```

嵌套向量中断优先级配置函数。

```
void NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);  //采用组别 2
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;//配置串口中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//占先式优先级设置为 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //副优先级设置为 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//中断使能
    NVIC_Init(&NVIC_InitStructure);//中断初始化
}
```

3.4.2 串口查询程序设计

调用一次发送字符函数，能够实现单个字符的输出，发送完成推出函数。

```
void USART_Putc(USART_TypeDef *USARTx,unsigned char c)
{
  USARTx -> DR = (u8)c; //要发送的字符赋给串口数据寄存器
  while((USARTx -> SR&0X40)== 0); //等待发送完成
}
```

发送字符串函数能实现一组字符串的连续输出。

```
void USART_Puts(USART_TypeDef * USARTx,char * str)
{
    while(*str)
    {
      USARTx -> DR = *str ++;
      while((USARTx -> SR& 0X40) == 0);//等待发送完成
    }
}
```

执行发送换行符，串行口界面重新换到下一行输出。

```
void UART_Send_Enter(USART_TypeDef * USARTx,char)
{
    USART_Putc(USARTx,0x0d);
    USART_Putc(USARTx,0x0a);
}
```

每调用一次发送数据函数，执行单个字节数据的发送，不等待发送完成就退出函数。

```
void USART_SendData(USART_TypeDef * USARTx, uint16_t Data)
{
  /* Check the parameters */
  assert_param(IS_USART_ALL_PERIPH(USARTx));
  assert_param(IS_USART_DATA(Data));
  /* Transmit Data */
  USARTx -> DR = (Data & (uint16_t)0x01FF);
}
```

每调用一次查询法接收数据，返回当前串口接收寄存器中的数值。

```
uint16_t USART_ReceiveData(USART_TypeDef * USARTx)
{
  /* Check the parameters */
  assert_param(IS_USART_ALL_PERIPH(USARTx));
  /* Receive Data */
  return (uint16_t)(USARTx -> DR & (uint16_t)0x01FF);
}
```

3.4.3　串口中断程序设计

前面我们学习了 TX 引脚对外输出数据，以及 RX 引脚接收来自其他设备或传感器模块的数据。这样的轮询模式存在一些弊端，比如程序必须等待发送或者接收结束才能接着执行，只能接收确定长度的数据。可以使用中断方式解决程序等待问题。

我们当调用发送函数 USART_SendData( )发送一段数据时，STM32 的 CPU 会依次将

数据移到寄存器中，发送移位寄存器中的数据会按照设定的波特率转化成高低电平，从 TX 引脚输出。发送数据寄存器中的数据会在发送移位寄存器发送完成后，被移到发送移位寄存器进行下一次发送。而在此过程中，CPU 需要不断地去查询发送数据寄存器中的数据是否已经移送到发送移位寄存器。移过了，就立即将下一个数据塞进来；如果还没有移，就再接着不停地查询，直到将要发送的数据全部发完，或者用时超过所设定的超时时间。

查询模式下的串口接收也是类似，调用接收函数 USART_ReceiveData( )后，从 RX 引脚接收的高低电平信号依次转换后存入接收移位寄存器。接收移位寄存器每接收完一帧，就将数据移到接收数据寄存器。CPU 会一直查询接收数据寄存器中是否有新数据可以读，一旦检测到，就立即把数据从寄存器移到用来接收数据的变量中。直到接收完希望接收的字节数，或者接收时间超时。

显然，在轮询模式下，不管是发送还是接收，CPU 一直处于忙碌状态，一轮一轮地去查询寄存器是否可用，无暇顾及其他任务。一般称这种一直等待使程序暂时无法向下执行的状态为“堵塞”。

如何解决这种长期占用 CPU 的堵塞问题呢？中断模式就是很好的解决方案。使用中断模式发送数据时，CPU 将数据塞入寄存器后就可以继续去进行其他任务了。当发送移位寄存器中的数据发送出去之后，则会触发“发送数据寄存器空”中断把 CPU 叫回来。CPU 在中断处理函数中将数据塞入发送数据寄存器后，就又可以去处理其他任务了。如此反复，指导全部发送完成。这一过程并不需要我们自己去实现。

中断接收时也非常类似。每当接收移位寄存器将一帧数据移入接收数据寄存器，就会触发一次“接收数据寄存器非空”中断，把正在处理其他事情的 CPU 叫回来，将数据读入变量中，然后 CPU 再去处理其他事情。直到接收到了设定的数据长度，完成全部接收。

表 3-11　USART 中断请求

中断事件	事件标志	使能位
发送数据寄存器空	TXE	TXEIE
CTS 标志	CTS	CTSIE
发送完成	TC	TCIE
接收数据就绪可读	TXNE	TXNEIE
检测到数据溢出	ORE	
检测到空闲线路	IDLE	IDLEIE
奇偶检验错	PE	PEIE
断开标志	LBD	LBDIE
噪声标志，多缓冲通信中的溢出错误和帧错误	NE 或 ORT 或 FE	EIE(1)

仅当使用 DMA 接收数据时，才使用这个标志位。USART 的各种中断事件被连接到同一个中断向量，有以下各种中断事件：

- 发送期间：发送完成、清除发送、发送数据寄存器空。
- 接收期间：空闲总线检测、溢出错误、接收数据寄存器非空、校验错误、LIN 断开符号

检测、噪音标志(仅在多缓冲器通信)和帧错误(仅在多缓冲器通信)。

如果设置了对应的使能控制位,这些事件就可以产生各自的中断。

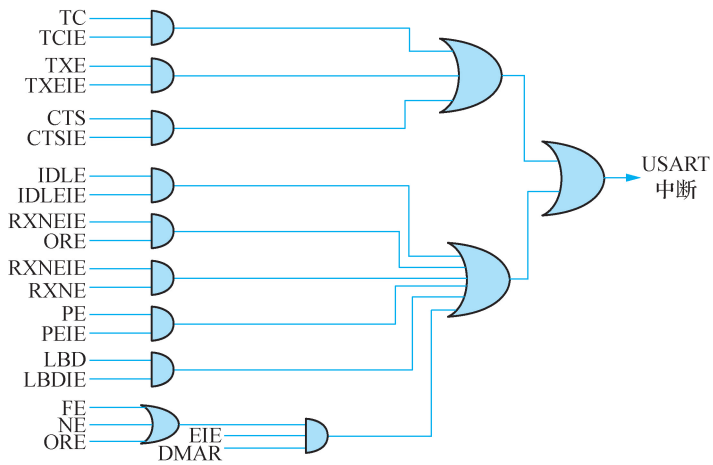


图 3.7　USART 中断映像图

每个 USART 只有一个中断向量,如图 3.7 所示。除了要用的“接收数据寄存器非空”中断,还有“发送数据寄存器空”中断、“线路空闲”中断等好几个中断,也共用了此中断处理函数,所以就需要一点点简单的判断才能确定当前是因何原因触发的中断。

```
void USART1_IRQHandler(void)
{
    // 处理发送中断
    if (USART_GetITStatus(USART1, USART_IT_TXE) != RESET)
    {
        if (usart1_tx_buffer.is_sending)
        {
            if (usart1_tx_buffer.index < usart1_tx_buffer.size)
            {
                // 发送下一个字节
                USART_SendData(USART1, usart1_tx_buffer.buffer[usart1_tx_buffer.
index]);
                usart1_tx_buffer.index ++;
            }
            else
            {
                // 发送完成,禁用发送中断
                USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
                usart1_tx_buffer.is_sending = 0;
                usart1_tx_buffer.size = 0;
                usart1_tx_buffer.index = 0;
```

```
                // 可以在这里添加发送完成回调函数
            if (usart1_tx_complete_callback != NULL)
                usart1_tx_complete_callback();
            }
        }
        else
        {
            // 没有数据要发送,禁用中断
            USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
        }
        USART_ClearITPendingBit(USART1, USART_IT_TXE);
    }
    //处理接收中断
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        // 接收数据处理
        uint8_t received_data = USART_ReceiveData(USART1);
        // 处理接收到的数据...

        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    }
}
// 定义回调函数类型
typedef void (*USART_TxCompleteCallback_t)(void);
// 设置回调函数指针
USART_TxCompleteCallback_t usart1_tx_complete_callback = NULL;
//回调函数
void USART1_SetTxCompleteCallback(USART_TxCompleteCallback_t callback)
{
    usart1_tx_complete_callback = callback;
}
```

3.4.4 串口的 DMA 控制程序设计

一、DMA 的概念

虽然使用中断方式的串口收发程序可以有效解决长期占用 CPU 的堵塞问题。但对于 CPU 本身来说却是屡屡被打断，疲于在中断搬运数据与处理正常任务代码间辗转。那有没有一种可能给 CPU 找个小助手，让它来帮着在寄存器与内存间搬运数据呢？

另一方面，CPU 无时无刻地在处理着大量的事务，但有些事情却没有那么重要，比方说数据的复制和存储数据，如果我们把这部分的 CPU 资源拿出来，让 CPU 去处理其他的复杂计算事务，是不是能够更好地利用 CPU 的资源呢？

因此，转移数据（尤其是转移大量数据）可以不需要 CPU 参与。比如希望读取外设的数据内存，只要给外设和内存之间提供一条数据通路，直接让数据由外设拷贝到内存，而不经过 CPU 的处理。

直接内存访问（DMA，Direct Memory Access）用来提供在外设和存储器之间或者存储器和存储器之间的高速数据传输。无须 CPU 干预，数据可以通过 DMA 快速地移动，这就节省了 CPU 的资源来做其他操作。两个 DMA 控制器有 12 个通道（DMA1 有 7 个通道，DMA2 有 5 个通道），每个通道专门用来管理来自一个或多个外设对存储器访问的请求。还有一个仲裁器来协调各个 DMA 请求的优先权。

DMA 的作用非常简单，只要我们创建一条 DMA 通道，告诉 DMA 将数据从哪里搬到哪里，DMA 就会在合适的时机帮我们将数据从源地址向目标地址进行内存搬运。等全部搬运完成，再通过中断提醒我们。例如，我们只需要为串口的发送和接收创建两条 DMA 通道，就可以让 DMA 帮着在串口的寄存器与内存之间搬运数据了。

二、DMA 的主要特性

- 12 个独立的可配置的通道（请求）：DMA1 有 7 个通道，DMA2 有 5 个通道。
- 每个通道都直接连接专用的硬件 DMA 请求，每个通道都同样支持软件触发。这些功能通过软件来配置。
- 在同一个 DMA 模块上，多个请求间的优先权可以通过软件编程设置（共有四级：很高、高、中等和低），优先权设置相等时由硬件决定（请求 0 优先于请求 1，以此类推）。
- 独立数据源和目标数据区的传输宽度（字节、半字、全字），模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。
- 支持循环的缓冲器管理。
- 每个通道都有 3 个事件标志（DMA 半传输、DMA 传输完成和 DMA 传输出错），这 3 个事件标志逻辑或成为一个单独的中断请求。
- 存储器和存储器间的传输。
- 外设和存储器、存储器和外设之间的传输。
- 闪存、SRAM、外设的 SRAM、APB1、APB2 和 AHB 外设均可作为访问的源和目标。
- 可编程的数据传输数目：最大为 65535。

（1）DMA2 仅存在于大容量产品和互联型产品。

（2）SPI/I2S3、UART4、TIM5、TIM6、TIM7 和 DAC 的 DMA 请求仅存在于大容量产品和互联型产品。

（3）ADC3、SDIO 和 TIM8 的 DMA 请求仅存在于大容量产品。

在发生一个事件后，外设向 DMA 控制器发送一个请求信号。DMA 控制器根据通道的优先权处理请求。当 DMA 控制器开始访问发出请求的外设时，DMA 控制器立即发送给它一个应答信号。当从 DMA 控制器得到应答信号时，外设立即释放它的请求。一旦外设释放了这个请求，DMA 控制器同时撤销应答信号。如果有更多的请求时，外设可以启动下一个周期。

每次 DMA 传送由 3 个操作组成：

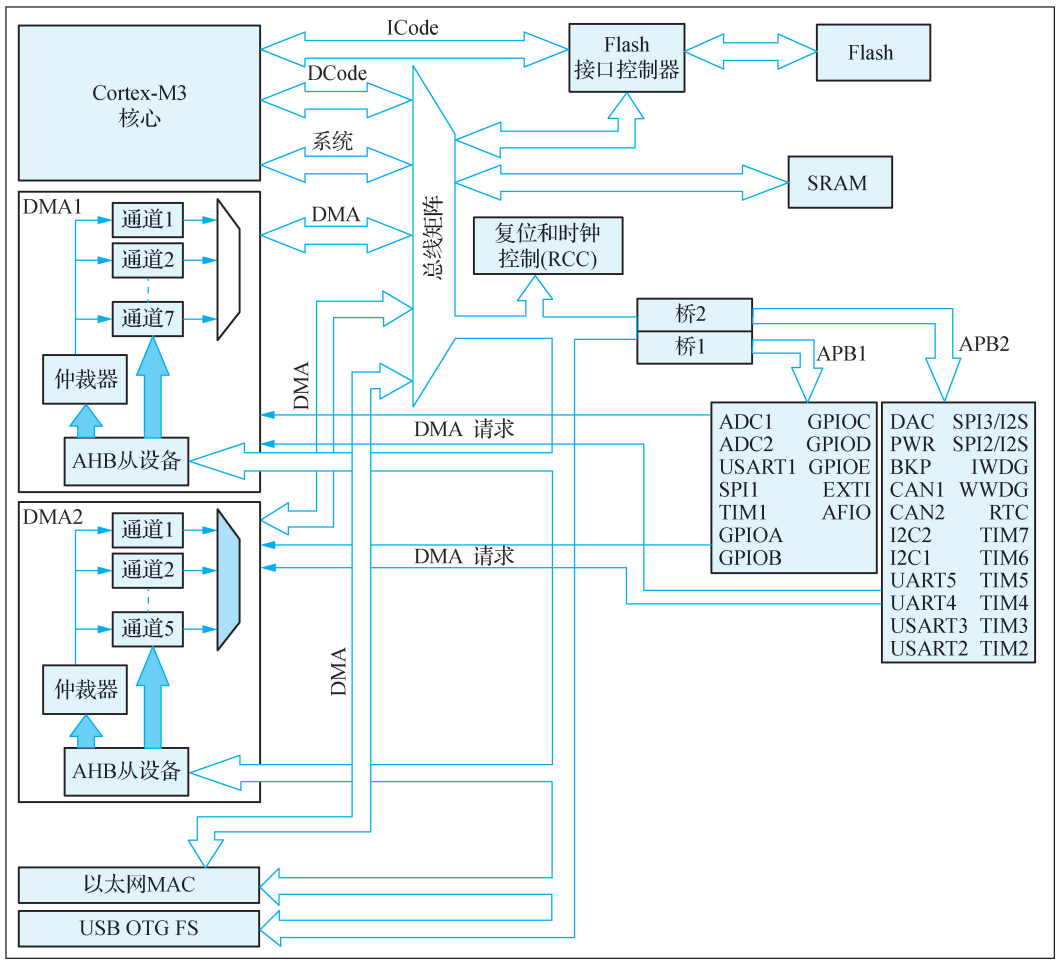


图 3.8　DMA 功能框图

- 从外设数据寄存器或者从当前外设/存储器地址寄存器指示的存储器地址取数据，第一次传输时的开始地址是 DMA_CPARx 或 DMA_CMARx 寄存器指定的外设基地址或存储器单元。
- 存数据到外设数据寄存器或者当前外设/存储器地址寄存器指示的存储器地址，第一次传输时的开始地址是 DMA_CPARx 或 DMA_CMARx 寄存器指定的外设基地址或存储器单元。
- 执行一次 DMA_CNDTRx 寄存器的递减操作，该寄存器包含未完成的操作数目。

三、DMA 的优先级

每个通道的优先权可以在 DMA_CCRx 寄存器中设置，有 4 个等级：

- 最高优先级
- 高优先级
- 中等优先级
- 低优先级

如果 2 个请求有相同的软件优先级，则较低编号的通道比较高编号的通道有较高的优先权。在大容量产品和互联型产品中，DMA1 控制器拥有高于 DMA2 控制器的优先级。

下面是配置 DMA 通道 x 的过程（x 代表通道号）：

（1）在 DMA_CPARx 寄存器中设置外设寄存器的地址。发生外设数据传输请求时，这个地址将是数据传输的源或目标。

（2）在 DMA_CMARx 寄存器中设置数据存储器的地址。发生外设数据传输请求时，传输的数据将从这个地址读出或写入这个地址。

（3）在 DMA_CNDTRx 寄存器中设置要传输的数据量。在每个数据传输后，这个数值递减。

（4）在 DMA_CCRx 寄存器的 PL[1:0]位中设置通道的优先级。

（5）在 DMA_CCRx 寄存器中设置数据传输的方向、循环模式、外设和存储器的增量模式、外设和存储器的数据宽度、传输一半产生中断或传输完成产生中断。

（6）设置 DMA_CCRx 寄存器的 ENABLE 位，启动该通道。一旦启动了 DMA 通道，它即可响应连到该通道上的外设的 DMA 请求。当传输一半的数据后，半传输标志（HTIF）被置 1，当设置了允许半传输中断位（HTIE）时，将产生一个中断请求。在数据传输结束后，传输完成标志（TCIF）被置 1，当设置了允许传输完成中断位（TCIE）时，将产生一个中断请求。

四、循环模式

循环模式用于处理循环缓冲区和连续的数据传输（如 ADC 的扫描模式）。在 DMA_CCRx 寄存器中的 CIRC 位用于开启这一功能。当启动了循环模式，数据传输的数目变为 0 时，将会自动地被恢复成配置通道时设置的初值，DMA 操作将会继续进行。

五、存储器到存储器模式

DMA 通道的操作可以在没有外设请求的情况下进行，这种操作就是存储器到存储器模式。

当设置了 DMA_CCRx 寄存器中的 MEM2MEM 位之后，在软件设置了 DMA_CCRx 寄存器中的 EN 位启动 DMA 通道时，DMA 传输将马上开始。当 DMA_CNDTRx 寄存器变为 0 时，DMA 传输结束。存储器到存储器模式不能与循环模式同时使用。

表 3－12　DMA 的中断请求

中断事件	事件标志位	使能控制位
传输过半	HTIF	HTIE
传输完成	TCIF	TCIE
传输错误	TEIF	TEIE

在大容量产品中，DMA2 通道 4 和 DMA2 通道 5 的中断被映射在同一个中断向量上。在互联型产品中，DMA2 通道 4 和 DMA2 通道 5 的中断分别有独立的中断向量。所有其他的 DMA 通道都有自己的中断向量。

六、DMA 的请求映像

DMA1 控制器从外设(TIMx[x=1、2、3、4]、ADC1、SPI1、SPI/I^2S2、I^2Cx[x=1、2]和 USARTx[x=1、2、3])产生的 7 个请求,通过逻辑或输入到 DMA1 控制器,这意味着同时只能有一个请求有效。参见图 3.9 的 DMA1 请求映像。外设的 DMA 请求,可以通过设置相应外设寄存器中的控制位,被独立地开启或关闭。

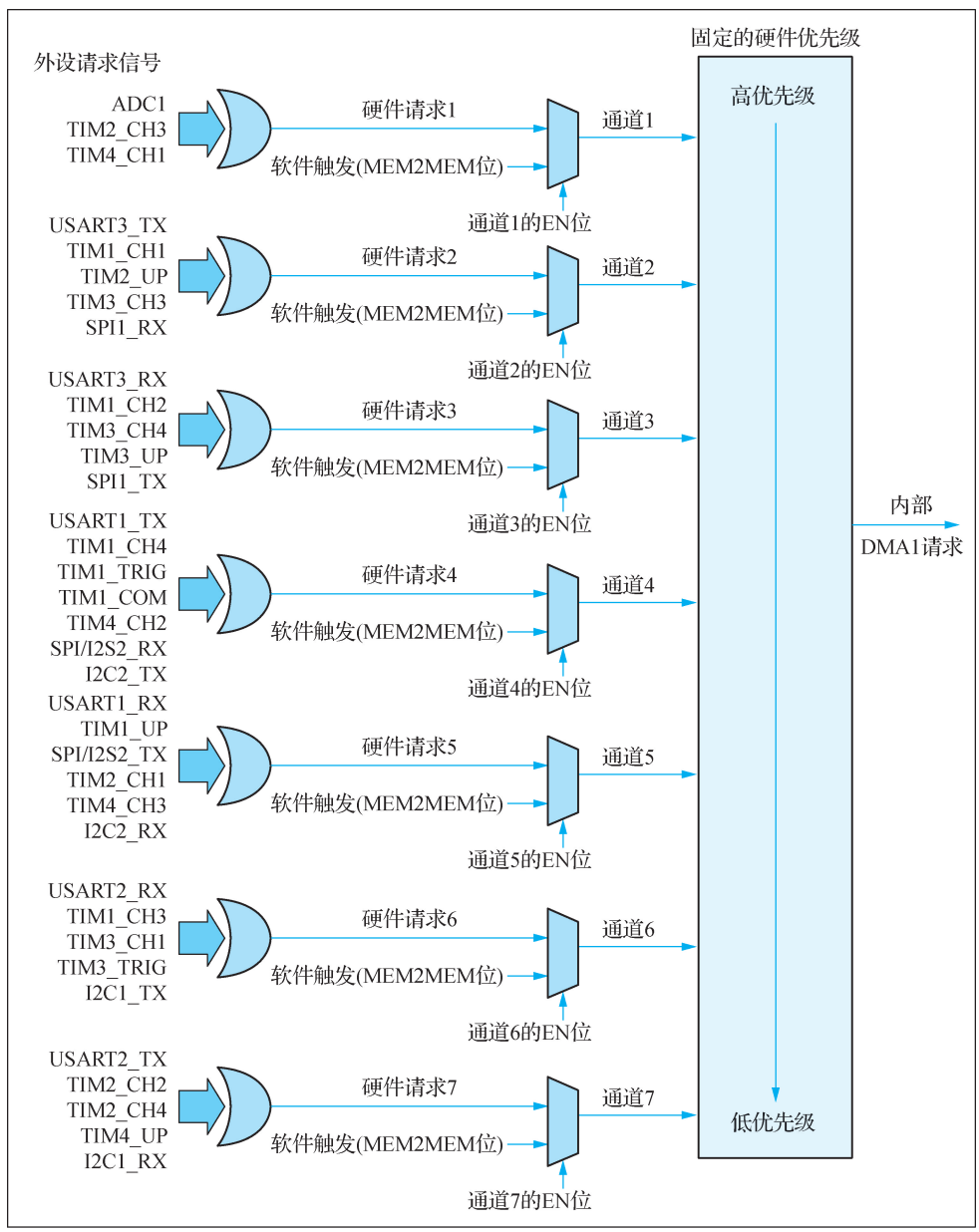


图 3.9　DMA1 请求通道

DMA2 控制器从外设(TIMx[5、6、7、8]、ADC3、SPI/I2S3、UART4、DAC 通道 1、2

和 SDIO)产生的 5 个请求,经逻辑或输入到 DMA2 控制器,这意味着同时只能有一个请求有效。参见图 3.10 的 DMA2 请求映像。外设的 DMA 请求,可以通过设置相应外设寄存器中的 DMA 控制位,被独立地开启或关闭。

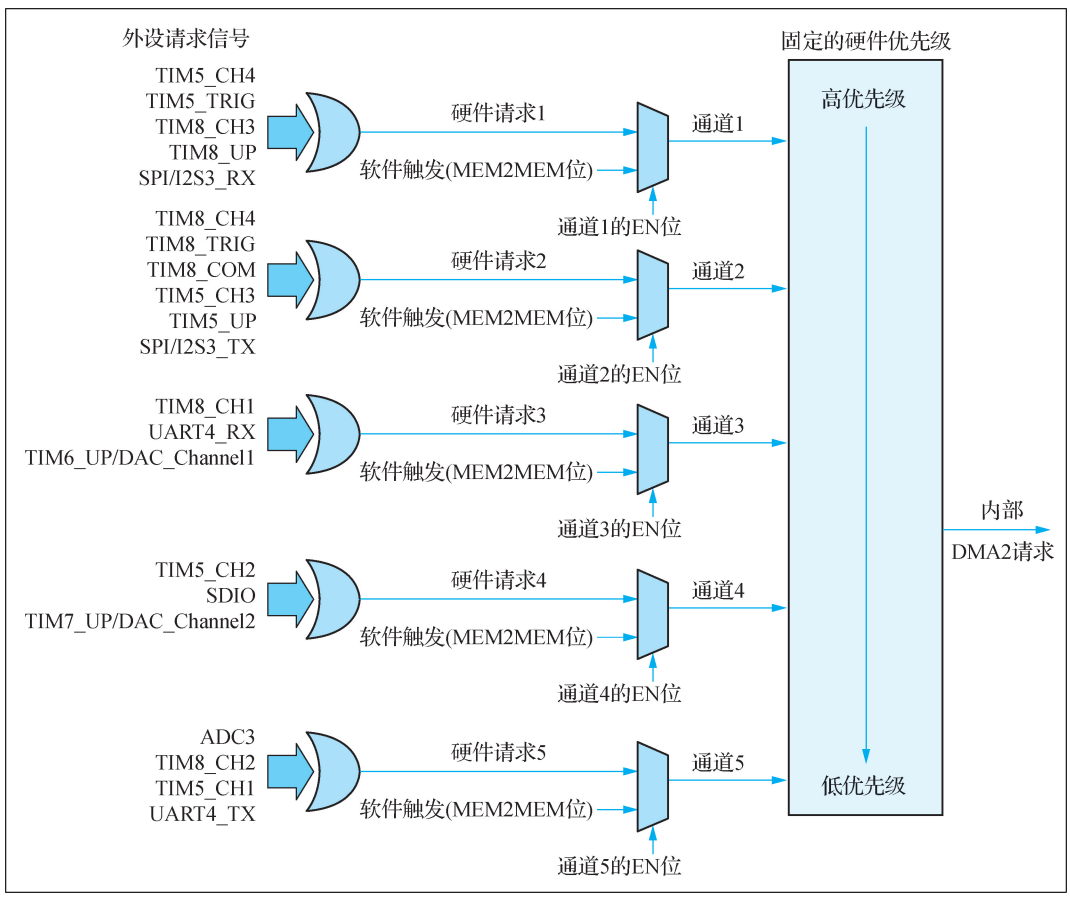


图 3.10　DMA2 请求通道

```
/************DMA 方式传输 ****************************/
#define SRC_USART1_DR(&(USART1 -> DR))          //串口接收寄存器作为源头
// DMA 目标缓冲,这里使用双缓冲
u8 DMA_Buf1[dma_Lenth];
u8 DMA_Buf2[dma_Lenth];
bool Buf_Ok;// buf 可用标志
BUF_NO Free_Buf;//空闲的 BUF 号
DMA_InitTypeDef DMA_InitStructure;//定义 DMA 结构体
void USART_DMAToBuf1(void)
{
      RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能 DMA 时钟
```

```
    DMA_DeInit(DMA1_Channel5);
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)SRC_USART1_DR;//源头 BUF
    DMA_InitStructure.DMA_MemoryBaseAddr = (u32)DMA_Buf1;//目标 BUF
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;//外设作源头
    DMA_InitStructure.DMA_BufferSize = dma_Lenth;     // BUF 大小
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设地址寄存器不递增
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//内存地址递增
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;//外设字节为单位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_PeripheralDataSize_Byte;//内存字节为单位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;//循环模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;// 4 优先级之一的(高优先)
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;//非内存到内存
    DMA_Init(DMA1_Channel5, &DMA_InitStructure);
    DMA_ITConfig(DMA1_Channel5, DMA_IT_TC, ENABLE);// DMA5 传输完成中断
    USART_DMACmd(USART1,USART_DMAReq_Rx,ENABLE);//使能串口接收 DMA
    //初始化 BUF 标志
    Free_Buf = buf2;
    Buf_Ok = FALSE;
    DMA_Cmd(DMA1_Channel5, ENABLE);// 使能 DMA
}
void NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//采用组 2
    NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel5_IRQn;// DMA1 通道 5 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //先占式优先级设为 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;//副优先级设为 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//中断向量使能
    NVIC_Init(&NVIC_InitStructure);//初始化结构体
}
void DMA1_Channel5_IRQHandler(void)
{
    if(DMA_GetITStatus(DMA1_IT_TC5))
    {
        DMA_ClearITPendingBit(DMA1_IT_GL5);//清除全部中断标志

        //转换可操作 buf
        if(Free_Buf == buf1) //如果 buf1 空闲,将 DMA 接收数据赋值给 buf1
        {
```

```
                DMA_InitStructure.DMA_MemoryBaseAddr = (u32)DMA_Buf1;
                DMA_Init(DMA1_Channel5, &DMA_InitStructure);
                Free_Buf = buf2;
            }
            else   //如果 buf2 空闲,将 DMA 接收数据赋值给 buf2
            {
                DMA_InitStructure.DMA_MemoryBaseAddr = (u32)DMA_Buf2;
                DMA_Init(DMA1_Channel5, &DMA_InitStructure);
                Free_Buf = buf1;
            }
            Buf_Ok = TRUE;
        }
    }
```

3.5　激光测距仪程序设计

3.5.1　设计要求

基于 STM32F103VET6 单片机和 TOF400F 激光测距模块，设计一款激光测距仪。要求实现以下功能：

（1）开始按键按下，MCU 发送命令启动激光测距模块；

（2）测量完成，对收到的距离数据进行解析；

（3）蜂鸣器响 1 声，并在 OLED 屏上显示测量结果；

（4）模式按键可以实现高精度和远距离的切换。

单片机设置 TOF400F 的模式命令为 01 06 00 04 00 01 CB 0B（高精度模式），01 06 00 04 00 01 09 CB（长距离模式）。

单片机查询 1 号地址 TOF400F 的命令为 01 03 00 10 00 01 85 CF，TOF400F 模块回传给单片机的距离数据命令 01 03 02 00 65 78 6F（其中 00 65 表示距离值为 0x0065，即 101 mm），78 6F 是 CRC-16/MODBUSX16＋X15＋X2＋1 的校验值。

表 3－13　外设与芯片引脚连接情况

外设引脚		MCU 引脚	外设引脚	MCU 引脚
OLED12864	OLED_SCL	PB6	模式按键	PD3
	OLED_SDA	PB7	启动按键	PD4
TOF 激光传感器	TOF_TXD	PA9	蜂鸣器	PB0
	TOF_RXD	PA10		

3.5.2 程序设计

一、主程序文件

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "oled.h"
#include "delay.h"
#include <string.h>
#include <stdio.h>
// 定义模式
typedef enum {
    MODE_HIGH_PRECISION = 0,
    MODE_LONG_DISTANCE
} MeasureMode_t;
// 全局变量
volatile MeasureMode_t g_measure_mode = MODE_HIGH_PRECISION;
volatile uint8_t g_measure_flag = 0;
volatile uint8_t g_mode_change_flag = 0;
volatile uint16_t g_distance_mm = 0;
volatile uint8_t g_measure_complete = 0;
// 串口发送缓冲区结构
#define TX_BUFFER_SIZE 64
#define RX_BUFFER_SIZE 32
typedef struct {
    uint8_t buffer[TX_BUFFER_SIZE];
    uint16_t size;
    uint16_t index;
    volatile uint8_t is_sending;
} USART_TxBuffer_t;
typedef struct {
    uint8_t buffer[RX_BUFFER_SIZE];
    uint16_t index;
    volatile uint8_t is_receiving;
} USART_RxBuffer_t;
USART_TxBuffer_t usart1_tx_buffer = {0};
USART_RxBuffer_t usart1_rx_buffer = {0};
// TOF400F 命令定义
// 高精度模式设置命令
```

```
const uint8_t cmd_high_precision[] = {0x01, 0x06, 0x00, 0x04, 0x00, 0x01, 0xCB, 0x0B};
// 长距离模式设置命令
const uint8_t cmd_long_distance[] = {0x01, 0x06, 0x00, 0x04, 0x00, 0x01, 0x09, 0xCB};
// 查询距离命令
const uint8_t cmd_query_distance[] = {0x01, 0x03, 0x00, 0x10, 0x00, 0x01, 0x85, 0xCF};
// 函数声明
void System_Init(void);
void GPIO_Init(void);
void USART1_Init(void);
void NVIC_Configuration(void);
void Beep_Init(void);
void Beep_On(void);
void Beep_Off(void);
void Beep_Beep(uint16_t duration_ms);
uint8_t USART1_SendArray_IT(uint8_t *data, uint16_t size);
void USART1_SetMode(MeasureMode_t mode);
void USART1_StartMeasure(void);
void Parse_Distance_Data(uint8_t *data, uint16_t length);
void OLED_Display_Distance(uint16_t distance_mm);
uint16_t CRC16_Modbus(uint8_t *data, uint16_t length);
// 外部中断处理函数(模式按键和开始按键)
void EXTI3_IRQHandler(void) __attribute__((interrupt("WCH-Interrupt-fast")));
void EXTI4_IRQHandler(void) __attribute__((interrupt("WCH-Interrupt-fast")));
```

二、系统初始化

```
int main(void)
{
    // 系统初始化
    System_Init();
    // OLED 显示初始化
    OLED_Init();
    OLED_Clear();
    // 显示初始信息
    OLED_ShowString(0, 0, "Laser Range Finder");
    OLED_ShowString(0, 2, "Mode: High Precision");
    OLED_ShowString(0, 4, "Press START to measure");
    OLED_ShowString(0, 6, "Dist: --- mm");
        // 设置初始模式
    USART1_SetMode(g_measure_mode);
    while(1)
```

```
    {
        // 处理模式切换
        if(g_mode_change_flag)
        {
            g_mode_change_flag = 0;
            // 切换模式
            g_measure_mode = (g_measure_mode == MODE_HIGH_PRECISION) ?
                             MODE_LONG_DISTANCE : MODE_HIGH_PRECISION;
            // 发送模式设置命令
            USART1_SetMode(g_measure_mode);
            // 更新 OLED 显示
            OLED_Clear();
            OLED_ShowString(0, 0, "Laser Range Finder");
            OLED_ShowString(0, 2, g_measure_mode == MODE_HIGH_PRECISION ?
                            "Mode: High Precision": "Mode: Long Distance");
            OLED_ShowString(0, 4, "Press START to measure");
            OLED_ShowString(0, 6, "Dist: --- mm");
        }
        // 处理测量请求
        if(g_measure_flag)
        {
            g_measure_flag = 0;
            g_measure_complete = 0;
            // 发送测量命令
            USART1_StartMeasure();
            // 显示测量中提示
            OLED_ShowString(0, 6, "Dist: Measuring...");
        }
        // 处理测量完成
        if(g_measure_complete)
        {
            g_measure_complete = 0;
            // 蜂鸣器提示
            Beep_Beep(200);
            // 显示测量结果
            OLED_Display_Distance(g_distance_mm);
        }
    }
}
void System_Init(void)
{
    // 设置系统时钟
```

```
    SystemInit();
    // 初始化各模块
    GPIO_Init();
    USART1_Init();
    NVIC_Configuration();
    Beep_Init();
    Delay_Init();
}
void GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 使能 GPIO 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                           RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE);
    // 配置模式按键(PD3)和开始按键(PD4)为上拉输入
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    // 配置蜂鸣器控制引脚(PB0)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_ResetBits(GPIOB, GPIO_Pin_0);  // 初始关闭蜂鸣器
}
```

三、USART1 初始化

```
void USART1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    // 使能 USART1 和 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);
    // 配置 TX 引脚 (PA9)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    // 配置 RX 引脚 (PA10)
```

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    // USART1 配置
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_
None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART_Init(USART1, &USART_InitStructure);
    // 使能接收中断
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
    // 使能 USART1
    USART_Cmd(USART1, ENABLE);
}
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    // 配置 USART1 中断
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    // 配置外部中断 3(模式按键 PD3)
    NVIC_InitStructure.NVIC_IRQChannel = EXTI3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    // 配置外部中断 4(开始按键 PD4)
    NVIC_InitStructure.NVIC_IRQChannel = EXTI4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    // 配置外部中断线
    EXTI_InitTypeDef EXTI_InitStructure;
    // 模式按键(PD3)中断配置
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource3);
```

```
    EXTI_InitStructure.EXTI_Line = EXTI_Line3;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;  // 下降沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);
    // 开始按键(PD4)中断配置
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource4);
    EXTI_InitStructure.EXTI_Line = EXTI_Line4;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;  // 下降沿触发
    EXTI_Init(&EXTI_InitStructure);
}
```

四、蜂鸣器控制函数

```
void Beep_Init(void)
{
    // PB0 已在上面的 GPIO_Init 中初始化
    // 蜂鸣器初始状态为关闭
    Beep_Off();
}
void Beep_On(void)
{
    GPIO_SetBits(GPIOB, GPIO_Pin_0);
}
void Beep_Off(void)
{
    GPIO_ResetBits(GPIOB, GPIO_Pin_0);
}
void Beep_Beep(uint16_t duration_ms)
{
    Beep_On();
    Delay_ms(duration_ms);
    Beep_Off();
}
```

五、USART 发送函数

```
uint8_t USART1_SendArray_IT(uint8_t *data, uint16_t size)
{
    // 检查是否正在发送
    if (usart1_tx_buffer.is_sending)
```

```
        return 1;
    // 检查数据长度是否有效
    if (size == 0 || size > TX_BUFFER_SIZE)
        return 1;
    // 复制数据到缓冲区
    memcpy(usart1_tx_buffer.buffer, data, size);
    usart1_tx_buffer.size = size;
    usart1_tx_buffer.index = 0;
    usart1_tx_buffer.is_sending = 1;
    // 启用发送中断,启动传输
    USART_ITConfig(USART1, USART_IT_TXE, ENABLE);
    return 0;
}
void USART1_SetMode(MeasureMode_t mode)
{
    if(mode == MODE_HIGH_PRECISION)
    {
        USART1_SendArray_IT((uint8_t *)cmd_high_precision, sizeof(cmd_high_
precision));
    }
    else
    {
        USART1_SendArray_IT((uint8_t *)cmd_long_distance, sizeof(cmd_long_
distance));
    }
    // 等待发送完成
    while(usart1_tx_buffer.is_sending);
    Delay_ms(100);  // 给模块一点处理时间
}
void USART1_StartMeasure(void)
{
    // 发送查询距离命令
    USART1_SendArray_IT((uint8_t *)cmd_query_distance, sizeof(cmd_query_
distance));
}
```

六、数据解析函数

```
void Parse_Distance_Data(uint8_t *data, uint16_t length)
{
    // 检查数据长度
```

```
    if(length < 7)  // 至少需要 7 个字节
        return;
    // 检查地址和功能码
    if(data[0] != 0x01 || data[1] != 0x03)
        return;
    // 检查数据长度字段
    if(data[2] != 0x02)
        return;
    // 提取距离数据(大端格式)
    g_distance_mm = (data[3] << 8) | data[4];
    // 验证 CRC
    uint16_t received_crc = (data[5] << 8) | data[6];
    uint16_t calculated_crc = CRC16_Modbus(data, 5);
    if(received_crc == calculated_crc)
    {
        // 数据有效
        g_measure_complete = 1;
    }
}
uint16_t CRC16_Modbus(uint8_t *data, uint16_t length)
{
    uint16_t crc = 0xFFFF;
    uint8_t i, j;
    for(i = 0; i < length; i ++)
    {
        crc ^= data[i];
        for(j = 0; j < 8; j ++)
        {
            if(crc & 0x0001)
            {
                crc >>= 1;
                crc ^= 0xA001;
            }
            else
            {
                crc >>= 1;
            }
        }
    }
    return crc;
}
void OLED_Display_Distance(uint16_t distance_mm)
```

```
{
    char buffer[32];
    // 清空显示区域
    OLED_Fill(0, 48, 127, 63, 0);
    // 显示距离
    sprintf(buffer, "Dist: %d mm", distance_mm);
    OLED_ShowString(0, 6, buffer);
    // 如果距离超过阈值,显示警告
    if(g_measure_mode == MODE_HIGH_PRECISION && distance_mm > 1000)
    {
        OLED_ShowString(0, 4, "Warning: Out of range !");
    }
}
```

七、中断服务函数

```
// USART1 中断服务函数
void USART1_IRQHandler(void)
{
    // 发送中断处理
    if (USART_GetITStatus(USART1, USART_IT_TXE) != RESET)
    {
        if (usart1_tx_buffer.is_sending)
        {
            if (usart1_tx_buffer.index < usart1_tx_buffer.size)
            {
                // 发送下一个字节
                USART_SendData(USART1, usart1_tx_buffer.buffer[usart1_tx_buffer.index]);
                usart1_tx_buffer.index ++;
            }
            else
            {
                // 发送完成,禁用发送中断
                USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
                usart1_tx_buffer.is_sending = 0;
            }
        }
        else
        {
            // 没有数据要发送,禁用中断
```

```
            USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
        }
        USART_ClearITPendingBit(USART1, USART_IT_TXE);
    }
    // 接收中断处理
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        uint8_t received_data = USART_ReceiveData(USART1);
        // 将数据存入接收缓冲区
        if(usart1_rx_buffer.index < RX_BUFFER_SIZE)
        {
            usart1_rx_buffer.buffer[usart1_rx_buffer.index ++] = received_data;
            // 检查是否收到完整的一帧数据
            if(usart1_rx_buffer.index >= 7)  // 一帧数据至少 7 字节
            {
                Parse_Distance_Data(usart1_rx_buffer.buffer, usart1_rx_buffer.
index);
                usart1_rx_buffer.index = 0;  // 重置缓冲区索引
            }
         }
         else
         {
            // 缓冲区溢出,重置
            usart1_rx_buffer.index = 0;
         }
         USART_ClearITPendingBit(USART1, USART_IT_RXNE);
     }
}
// 模式按键中断处理函数
void EXTI3_IRQHandler(void)
{
     if(EXTI_GetITStatus(EXTI_Line3) != RESET)
     {
          // 设置模式切换标志
          g_mode_change_flag = 1;
          // 清除中断标志
          EXTI_ClearITPendingBit(EXTI_Line3);
     }
}
// 开始测量按键中断处理函数
void EXTI4_IRQHandler(void)
{
```

```
    if(EXTI_GetITStatus(EXTI_Line4) != RESET)
    {
        // 设置测量标志
        g_measure_flag = 1;
        // 清除中断标志
        EXTI_ClearITPendingBit(EXTI_Line4);
    }
}
```

注意：省略了 OLED 驱动函数和延时函数。

3.5.3 实物测量效果

利用设计好的激光测距仪做了一个沙盘地形分析装置，如图 3.11 所示。二维运动机构可以带动激光测距仪模块在二维待测区域内逐点扫描，实现连续分布点的距离测量功能，将接收到的数据生成三维模型。本装置的 X 轴可移动长度为 30 cm，Y 轴可移动为 24 cm，对其进行 128×128 像素点扫描，电机转速为 18.75 转/分钟，扫描速度 0.375 cm/s。生成的三维地形如图 3.13 所示。

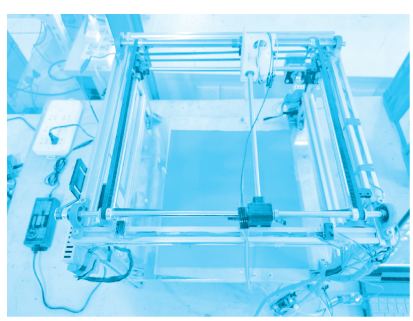


图 3.11 沙盘地形分析装置

图 3.12 自制三维地形实物

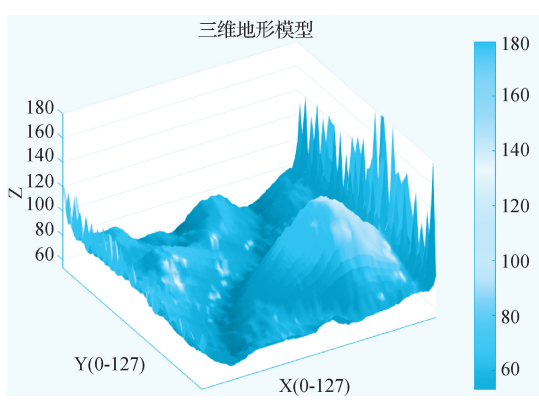


图 3.13 测量数据重构三维地形模型

根据生成的三维表面图可知，有明显的山峰、河流和盆地，与图 3.12 的沙盘实物图相比，地形一致，实现了三维地形的识别与重构功能。

3.6 项目小结

本项目基于 STM32F103VET6 微控制器和 TOF（飞行时间）激光测距传感器，成功设计并实现了一套高精度、实时的激光测距系统。系统核心是 STM32F103VET6，它通过其片上 USART 模块与 TOF 传感器进行稳定可靠的串口通信。

通过本项目，深入掌握了基于串口中断的异步通信数据处理方法，以及 TOF 传感器的工程应用流程，为后续开发更复杂的嵌入式测控系统积累了宝贵经验。未来可考虑增加多传感器融合、无线传输或复杂环境下的滤波算法以进一步提升系统性能。

实战练习

基础任务

1. TOF 测距技术与传统的超声波测距、红外测距相比有何优缺点？

2. 编写程序，通过 USART1 输出班级、学号、姓名等个人信息。

3. 编写串口收发程序，收到大写英文字母，输出对应的小写英文字母；收到其他值，输出“error”。

进阶挑战

编写一个键盘密码开机程序，接收正确的密码组合后开机，接收错误密码时提示错误，连续 6 次输出密码后，锁定开机界面，24 小时后方可重新登录。