HCS08/RS08 集成开发环境设计与实现

中文摘要

Freescale 公司于 2004 年开始推出的新一代 8 位 MCU HCS08 系列及其简化版本的 RS08 系列产品，目前已经有 200 多个型号上市。该系列 MCU 新引入的 BDM 功能，为嵌入式开发提供了全新的调试手段。目前，国内使用该系列 MCU，其开发工具依赖进口，本课题目标是自主开发 HCS08/RS08 集成开发环境。

开发 HCS08/RS08 集成开发环境内容包括：编程调试器、硬件评估系统、各芯片最小系统、PC 方软件等。开发难点主要有：编程调试器的通用性、汇编及 C 语言源程序级调试及 USB 设备驱动程序的开发。

通过测量目标芯片的频率实现稳定的通信，在此基础上实现编程调试器的通用性。涵盖 HCS08/RS08 各型号产品的测试目标板有效地验证了编程调试器的性能。对代码编译后的 list 文件和 dbg 文件进行分析，设计相应的数据结构，实现了代码的单步调试和断点调试。PC 方程序完成对代码的编辑、编译和对目标文件的分析。论文还详细描述了开发 USB 设备驱动程序所涉及的相关基础知识以及具体过程，实现了一个完整的 USB 设备驱动程序。

关键词：HCS08/RS08 系列微控制器，USB，编程调试器，BDM

作　　者：叶旺胜
指导老师：王宜怀

The Design and Implementation of HCS08/RS08 Embeded Developing Platform

ABSTRACT

Freescale's new generation MCUs HCS08 and its simplify version RS08 provide more than 200 types products. BDM technology inside them can help developer to promote their work. But many native users depend on foreign platforms to develop,so we design and implement HCS08/RS08 develop platform by us.

The system include such as programmer and debug pod, hardware test system,PC host software etc.To support all kinds of MCUs of HCS08/RS08 and provide tools to debug C&ASM code are the most challengeable jobs as well as USB's device dirver.

By measurement the frequence of target MCU so the host can communicate to target stably,and we bring forward the programmer&debugger pod to support all kind of MCUs of HCS08/RS08. The hardware test boards including all kinds of HCS08/RS08 MCUs can test and verify the quality of the pod. The suitable data struct help to step in/out the code and make the breakpoint to debug code through analyse the list file and the dbg file. The paper also describes in detail the coherent knowledge of the device driver, bring forward the USB device driver for the programmer and debugger pod.

Key Word:HCS08/RS08 series MCUs,USB,programmer and debugger,BDM

Written by Ye Wangsheng
Supervised by Wang Yihuai

目 录

第一章 概述

Freescale（中文译名飞思卡尔，其前身为 Motorola 半导体部）公司的 MCU 系列产品在业界一直享有盛誉，其新推出的HCS08 系列和 RS08 系列 MCU 是新一代高性价比，高集成度的 8 位 MCU，兼容传统的 HC08 MCU 指令，具有更多的寻址方式，更高的时钟频率，更强的省电性能和更多层次的低功耗工作模式。最为显著的特点是增加了 BDM 调试方式。BDM 是一种全新的 MCU 开发理念，是一种真正意义上的在线开发模式。Freescale HCS08/RS08 系列 MCU 的 BDM 调试方式在该公司先前推出的 16 位 S12 系列 MCU 基础上又做了一些改进，其性能比 16 位 S12 的还要好[1]。

本章首先介绍 HCS08/RS08 系列 MCU 的相关背景知识，然后在此基础上提出一个针对 HCS08/RS08 系列 MCU 的集成开发环境实现方案。最后介绍本文的工作内容和论文结构。

1.1 HCS08 和 RS08 系列 MCU 概述

MCU 是指在一块芯片上集成了中央处理器（CPU）、存储器（RAM/ROM 等）、定时器/计数器及多种输入/输出（I/O）接口的比较完整的数字处理系统[2]。

Freescale 公司于 2004 年开始推出 HC08 系列 MCU 的增强型——HCS08 系列 MCU。HCS08 系列 MCU 使用 HCS08 CPU，与 HC08 CPU 相比，其速度更快，内部总线频率由最高 8MHz 提高到 20MHz，CPU 频率由最高 20MHz 相应提高到 40MHz 。在时钟产生方式上，HCS08 CPU 的外部晶振既可以使用 HC08 的 32～100KHz 石英晶振，也可以使用 1～16MHz 的石英或陶瓷晶振。兆赫兹级的晶振比 32KHz 的晶振更稳定也更容易起振。HCS08 CPU 内部还有中心频率为 243KHz 和 8MHz 2 个内部时钟发生器，校正后频率稳定性可达 0.2％～0.5％。在对频率稳定性要求不高的情况下，可以省去外部时钟电路[3]。在分频因子的处理方式上，HCS08 CPU 采用了 11 位数，连续可调的单一分频因子使分频因子的算法大为简化。

HCS08 指令集兼容 HC08 指令集，对于使用频率很高的 16 位数据传送指令 LDHX，STHX 和 16 位数据比较 CPHX，HCS08 比 HC08 增加了多种寻址方式。对于 LDHX 指令，HCS08 可以使用所有的寻址方式，对于 STHX 和 CPHX 指令，HCS08 增加了扩展寻址和栈指针相对 8 位偏移量寻址。这些扩展的寻址方式有利于提高 C

语言编译效率。

HCS08 CPU 在节电方面比 HC08 有更多的考虑。HC08 只有 WAIT 和 STOP 两种省电方式，而 HCS08 CPU 支持 STOP1，STOP2，STOP3 三种 STOP 模式。不同的 STOP 模式对应不同的停机深度和不同的唤醒方式[4]。

HCS08 MCU 在时钟模块的设计上较 HC08 有了较大改进，结构更加复杂，功能也更加强大。熟悉 HC08 MCU 的读者都清楚 HC08 MCU 是使用锁相环电路将低频频率变成高频频率的。为了使锁相环电路正常工作，HC08 MCU 要求用户必须外接 RC 滤波电路。HCS08 MCU 使用了锁频环技术，在实现频率倍频时采用数字化的处理方式。在使用上，锁频环电路无需外接 RC 电路，对于用户来说更加方便了。使用锁频环电路取代锁相环电路是 HCS08 MCU 对 HC08 MCU 的一大改进[5]。

HCS08 MCU 与 HC08 系列 MCU 相比最突出的一点就是增加了 BDM 调试方式。BDM 系统不占用片内资源，仅需要少量的 I/O 口，提供了更快的连接速度和更加方便的调试功能。HCS08 CPU 指令集增加了一条 BGND 指令用于调试。HCS08 系列 MCU 包含有 BDC（Background debug controller）模块，它使用专用的 BKGD 引脚进行在线编程以及在调试模式下为主机和目标机提供双向的数据传输。HCS08 系列的 MCU 产品中，除 Q 子系列产品产品外，其它子序列的产品还包含有 DBG（On-Chip Debug System）模块。DBG 模块使用一个 8 字节的 FIFO（First In，First Out）队列用于存储地址和数据信息，同时使用两个比较器 A 和 B 控制收集总线信息的方式。BDC 和 DBG 模块为应用程序的开发提供了强大的调试手段[6]。

8 位 MCU 正在向小型化应用发展。在这些小型应用中，也许并不需要使用完整的 HC08 或 HCS08 所具备的丰富功能。为此，Freescale 于 2006 年推出了 RS08 系列产品，特别的设计使得其效率更高，成本更低，可以较大幅度地降低应用系统的功耗，从而更加适合小内存的 MCU。RS08 是一些新兴应用的理想解决方案，例如完全用固态电路实现的简单机电设备或小型便携设备，甚至一次性便携设备。

RS08 内核是非常流行的 HCS08 CPU 的简化版，其内核尺寸比 HCS08 的尺寸小 30％。为了减小面积，RS08 将计数器和地址总线宽度限制为 14 位，使用一个全局中断标志寄存器取代了矢量中断功能，同时还取消了以下功能[7][8]：

① 堆栈指针和 H:X 寄存器及其相关指令和寻址模式。

② 乘法、除法以及 BCD 码指令。

③ 算术逻辑移位运算（保留了逻辑移位和旋转）。

④ 条件码寄存器中的若干位以及相关条件分支指令。

这些被取消的功能由更为简单的结构所代替，这些结构保证了在内存低于 16K 且引脚数目很少的器件上，可以用非常简洁高效的代码实现大多数嵌入式应用。为了进一步提高运算效率，飞思卡尔增加了如下内容：

① 屏蔽程序计数器，用于更为高效的子程序调用。

② 简短微小的寻址模式，允许对最常用的变量和寄存器进行更为有效地访问和操作。

③ 内存分页方案，能够更充分地利用直接寻址模式和新型的简短微小的寻址模式。

1.2 嵌入式开发平台的基本功能

Freescale 公司的 8 位 MCU 的使用非常广泛，学习了解和掌握 Freescale MCU 的知识及应用有利于跟踪学习国外的先进技术与经验，扩大知识面，增强创新能力。目前国内能够掌握与应用 Freescale MCU 的人才相当的少，一个主要原因是相关资料与工具的缺乏。Freescale 提供的开发平台价格昂贵，且为英文界面，不利于推广和普及。国内的复旦大学、清华大学和苏州大学先后开发过 HC08 的开发平台，取得了一定的效果和作用[9]。

嵌入式系统的开发有别于其它系统的开发，有其特定的方式和方法，它是硬件与软件相结合的系统。在实际的开发过程中，硬件与软件相互依托与相互影响增加了系统开发的难度，需要嵌入式开发平台来协助实际应用的开发，这其中，调试功能往往成为系统开发的一大利器。HC08 MCU 仅提供了简单的调试功能，其通过软中断的方式提供调试信息。HCS08 /RS08 MCU 提供了强大的调试机制，可以实现多种调试手段，为应用程序的开发提供极大的帮助。

一个完整的嵌入式开发平台包含以下功能：

① 源程序的编辑与编译。

② 目标芯片的擦除与写入。

③ 代码在线调试。

这其中，程序调试功能的实现较为复杂。程序的调试可以实现单步跟踪和多断点

设置以及多模式的条件触发，功能较为强大。实现 HCS08/RS08 MCU 的 BDM 编程调试器，可以使用另外的 MCU 来实现。利用两个 I/O 引脚实现 BDM 编程调试器的功能。一个 I/O 引脚控制目标 MCU 的 RESET 信号，另一个 I/O 引脚控制目标 MCU 的 BKGD 信号。

在调试器与 PC 机之间选用 USB（Universal Serial Bus 通用串行总线）进行通信。USB 接口使用方便，速度快，可以连接多个不同设备，并且可以为 USB 设备供电，这些优点使得 USB 被业界广泛接受，USB 设备产品日益受到青睐，发展势头如日中天。

本文的设计采用 Freescale 的 HC908JB8 MCU 实现 S08 系列 MCU 的调试功能。该款 MCU 是一款低成本，高性能的 HC08 MCU，符合 USB1.1 规范，提供 1.5Mbps 的低速（low-speed）传输模式，具有 384 字节 RAM 和 16K 字节的 FLASH，总线频率为 6MHz。

1.3 系统开发的必要性

Freescale 的 MCU 产品蜚声国际，在工业控制、汽车电子以及无线通信等领域拥有无可争议的领导地位。目前，我们国家正在进行经济结构调整，要建设成为一个以信息化为代表的新型工业化国家，因此未来对嵌入式人才和应用的需求将是极其巨大的。学习、了解与掌握 Freescale 微控制器技术，有利于借鉴国外更多的先进技术与经验，扩大自己的知识面，增强创新思维能力。而国内能够掌握和应用 Freescale 相关 MCU 的人才不多，远不能满足市场日益扩大的需求，究其原因，一个主要的原因是国内 Freescale 的相关技术资料较少，学习环境不够充分，相关开发工具昂贵，诸如种种都限制了对其 MCU 等相关产品的学习与掌握。

国内目前使用 Freescale 产品的应用研发机构不多。近些年 Freescale 加大了宣传力度，且在一些大学中设立了研究机构。其中，复旦大学、清华大学和苏州大学分别研究开发过类似的平台。复旦大学开发了 M68HC08 系列单片机仿真器，该仿真器可以在目标系统硬件尚未定 MCU 型与制版情况下，先行调试目标系统的硬件、软件设计，为目标系统的研制提供前期基础，但是，一些情况下，难以实现 100%的实时仿真，有些功能在用仿真器调试时十分正常，而到了实际应用系统却不能顺利运行，同时，传统的仿真方式的一些调试功能仅适用于初学者，对于具有一定开发经验并拥有

通用功能模块积累的开发者，往往增加了开发时间[10]；清华大学于 2003 年推出的 8 位 Freescale 单片机 M68HC908 全系列编程器[11]，专门为 Freescale FLASH 型单片机而设计，但只支持 M68HC08 系列中的 14 种单片机，支持 MCU 芯片数量少，而且为英文界面，使用不方便，也不能实现调试。苏州大学研制的 HC08 开发环境，可以较好的支持 HC08 系列 MCU 的应用开发。其采用 MON08 接口实现对目标芯片的擦除写入工作，在实际应用中取得了较好的效果。但其只支持 HC08 系列 MCU，且调试功能较弱，通信速度较慢。

HCS08/RS08 集成开发环境瞄准 Freescale 最新的产品系列，开发适合国内用户需要的 HCS08/RS08 系列 MCU 嵌入式开发平台，为用户提供一个使用这两个系列芯片产品的方便、快捷、高效和便宜的开发环境。

1.4 本文工作

在研究生的学习过程中，我对“嵌入式系统的软、硬件设计”、“Freescale 8 位、16 位及 DSP 等系列的单片机”等课题有了深入的学习和实践，积累了有关电路设计、系统设计、数据通信方面的知识和经验，使得我有信心将 HCS08 和 RS08 系列 MCU 嵌入式集成开发环境的设计作为我的毕业设计课题。

（1）选择设计方案

在设计硬件和软件之前，首先要选择好设计方案。集成开发环境涉及软硬件设计实现且由于新型号的芯片会不断推出，因此需要考虑到后续的更新和功能改进。

（2）硬件平台设计与实现

① 芯片选型。

② 了解芯片的外围电路，设计硬件原理图。

③ 绘制 PCB 电路图，联系厂家制作电路板。

④ 其它元器件的选型与采购等。

⑤ 焊接、测试，完成硬件系统。

（3）MCU 方软件的设计、测试与实现

① USB1.1 协议分析。

② USB 固件程序设计与测试。

③ BDM 通信和命令程序设计与测试。

④ 自定义命令程序设计。

⑤ 整体测试。

（4）PC 方软件的设计、测试与实现

① USB 驱动程序设计与测试。

② 界面的设计。

③ 数据库的建立。

④ 测试模块程序设计。

⑤ 总体的测试。

1.5 本文结构

全文共七章，各章的内容安排如下：

第一章介绍了相关系列 MCU 的概况，给出了课题开发的背景和必要性、论文工作及结构。

第二章描述设计方案及相关知识技术基础，主要是 BDM 通信和 USB 通信的内容。

第三章阐述编程调试器的硬件设计以及相关的目标板的设计。

第四章论述 MCU 方的软件设计方案，给出关键子程序的代码和流程。

第五章给出 PC 方的软件设计实现及处理流程。

第六章介绍了 USB 驱动程序设计的相关背景知识和具体的实现代码。

第七章对全文的工作进行总结，提出需要改进的地方。

第二章 设计方案及技术基础

HCS08/RS08 系列 MCU 最大的特点就是具备了 BDM 调试功能，这也是嵌入式集成开发平台所需要重点实现的功能。本章主要介绍系统的设计方案，以及该方案所涉及到的相关技术基础。

2.1 设计方案

为了更好的利用 HCS08/RS08 系列 MCU 所具有的 BDM 功能以及适应这两个系列不同型号产品的要求，设计实现一个编程调试器,通过接收 PC 机发送的操作命令，与目标 MCU 进行通信，使目标 MCU 完成相应的功能操作。如图 2-1 所示。针对不同的目标 MCU 的操作，由 PC 机通过编程调试器查询目标机的型号，将查询结果与数据库进行关联，从而可以实现对不同型号 MCU 的操作。

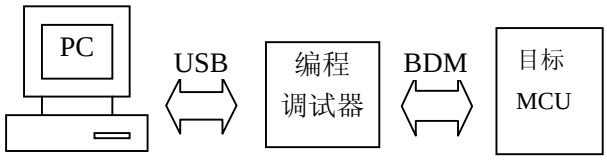

图 2-1 编程调试器实现方案

虽然 BDM 采用单线通信方式，但其速率可以很高，甚至可以和目标机总线频率相一致。为了尽可能的保证整个系统的效率，在 PC 机和编程调试器之间采用 USB 方式进行通信。采用 USB 不但速度快，而且使用方便。USB 日益流行，已成为 PC 机的标准配置。

综上所述，形成最终的解决方案：硬件上使用编程调试器，PC 机与编程调试器采用 USB 进行通信连接，编程调试器与目标MCU 采用 BDM 方式通信。软件上分为 PC 方软件和 MCU 方软件，MCU 方软件发送 BDM 命令和数据给目标 MCU，实现与目标 MCU 的交互，完成编程与调试操作；PC 方软件主要实现对这些操作的控制以及对目标 MCU 的型号判断选择等功能。

2.2 BDM 调试模式

后台调试模式 BDM(Background Debug Mode) 是一项新的调试技术,在 Freescale 的 HC/S12 和 HCS08/RS08 系列 MCU 中得到了广泛的应用。BDM 的基本思想是在片内嵌入一个 BDM 调试模块，该模块通过专用的内部连线可以访问 MCU 的内部资源。

模块以单线方式与外界通信，根据收到的外部命令执行相应的操作，送出相应的信息。HCS08/RS08 包含的背景调试控制器 BDC（Background debug controller）主要实现对 BDM 命令的解码及部分 BDM 命令的执行。

BDC 通过专用的单根数据线为调试目标机提供接口。这种方式为 FLASH 和其它非易失性存储器的编程提供了极大的方便。同时，BDC 还能够在不影响用户程序运行（non-intrusive）的情况下访问存储器的数据以及提供常用的调试手段如修改 CPU 的内部寄存器、设置断点以及单步执行等。

2.2.1 调试方法的历史回顾

早期的调试方法是“擦和烧”的方法，将代码烧写到芯片内部的 EPROM 中，运行程序，如果程序运行不正常，需要从头到尾详细检查代码，然后将修改后的代码再烧写到 EPROM 中。这种方式耗时且效率低下，同时也难以发现隐藏得较深的问题。不久以后，一种新型的器件 ROM 仿真器被引入。ROM 仿真器插在目标板上代替 ROM 芯片。这个器件通过串口、并口或者网口连接到主计算机。ROM 仿真器允许快速的代码下载和擦写。在比较好的系统中，还可以下载包含 ROM 监控的代码，可以通过仿真接口进行通讯监视。但是用户软件在仿真器环境下运行，和脱离仿真器后独立运行是有区别的。在很多情况下，目标板系统往往不能运行或者运行结果和仿真器环境下不一致。与ROM 仿真器相似，下一个在调试上的重要突破是使用在线电路仿真器（In-circuit emulator ，ICE）。ICE 通常是使用特殊型号的外合处理器（bond-out processor）芯片，这种芯片有更多的脚，把典型的内部信号送到片外进行监视和利用。系统或者外部的存储器同样被支持，可以被映射到用户空间。这允许在目标系统没搭建好就进行调试。典型的 ICE 具有复杂的硬件和软件，能够对目标处理器的程序源代码进行完全的访问，硬件端点，跟踪和其它的功能。ICE 可以进行典型的实时跟踪，不占用用户资源，允许在没有目标板的情况下调试。其缺点是：首先价格昂贵；其次外合处理器的推出速度往往慢于相应的处理器的推出速度，而且芯片厂商越来越不愿意为每一款芯片提供售量极少的外合处理器。新的调试手段是 片内调试（On-chip debug， OCD ）。最早的 OCD 是写入到目标系统微码中去的基本的调试监视器，随后加入了一些其它的功能如实时读取 PC ，近乎实时的读取存储器的内容。基本的 OCD 允许代码下载，读写存储器和处理器资源，单步执行，处理器复位及其状态控

制（运行或者停止）。一些芯片用其它的资源来增强它们的 OCD，实现完全的片内调试，允许完全跟踪处理器执行的过程。简单的实时捕捉处理器执行的相关信息，调试器就可以重建（rebuild）完整的程序执行流程[12]。更具威力的调试工具是逻辑分析仪和性能分析仪，它可以在不“明显地”干扰系统运行的情况下了解处理器工作在实时状态时的运行情况。但是，这些工具也是极其昂贵的，一般普通用户难以负担。

2.2.2 HCS08/RS08 和 HC08 调试方法的比较

Freescale HC08 系列 MCU 是 HC05 系列 MCU 的替代产品，目前发展比较成熟，具有多种型号，是面向低端应用的低成本的 MCU。HCS08 和 RS08 系列 MCU 是 Freescale 新推出的高性价比、高集成度的 8 位 MCU。HCS08/RS08 系列 MCU 使用增强的 HCS08 CPU，兼容传统的 HC08 CPU 指令。HCS08/RS08 系列与 HC08 系列相比，最显著的特点是增加了 BDM 调试方式。与 HC08 系列 MCU 采用的监控模式相比，BDM 有着巨大的优势：更少的硬件需求、更快的通信速度、更强的调试功能和更友好的编程接口。使用监控程序法调试 MCU 应用程序时，由于监控程序和应用程序都是在 CPU 上运行，两者不可能同时运行；只有停止应用程序，才能进入监控程序，修改运行测试，故不可能实现对应用程序的动态调试。表 2-1 是 HCS08/RS08 BDM 模式与 HC08 监控模式的比较[13]。

表 2-1 HCS08/RS08 BDM 模式与 HC08 监控模式的比较

功能	HC08 监控模式	HCS08 BDM 模式	RS08 BDM 模式
进入方式	至少需要 4 个 I/O 引脚，IRQ 引脚需要拉高	只需要将 BKGD 引脚拉低	只需要将 BKGD 引脚拉低
固件/硬件	ROM 中内嵌固件	内嵌硬件模块，用户程序不能访问 BDM 寄存器	内嵌硬件模块，用户程序不能访问 BDM 寄存器
通信协议	RS232 串口通信协议，应用标准的 PC 通信波特率	单线传输协议 通信协议中可以带有应答 更快和更宽的速度范围	单线传输协议 通信协议中可以带有应答 更快和更宽的速度范围
指令	只有 5 条软指令，其功能如下 1）可执行用户程序（RUN） 2）可直接访问 CPU 寄存器 3）在 CPU 停止运行时，可通过 Monitor 指令访问存储器	共 30 条指令，其中17 条软指令和 13 条硬指令，其功能为 1）BDM 模式下可执行用户程序 2）软指令可直接访问 CPU 寄存器 3）硬指令可在用户程序运行时访问存储器	共 21 条指令，其中10 条软指令和 10 条硬指令，其功能为 1）BDM 模式下可执行用户程序 2）软指令可直接访问 CPU 寄存器 3）硬指令可在用户程序运行时访问存储器
断点功能	断点模型支持一个断点	BDC 支持一个硬件断点和一	BDC 支持一个硬件断点和一

		条跟踪指令，DBG 模型支持两个功能复杂的额外断点	条跟踪指令
定时器计数器	在监控模式下处于激活状态	在 BDM 模式下不激活	在 BDM 模式下不激活
STOP 和 WAIT模式	在 STOP 和 WAIT模式下不能使用	在 STOP 和 WAIT模式下可以使用	在 STOP 和 WAIT模式下可以使用
接插头标准	16 针 MON08 连接头	6 针 BDM 连接头	6 针 BDM 连接头

2.2.3 BDM 通信协议

BDM 使用单线传输协议，这要求通信双方必须使用相同的时钟频率。对于目标机，其 BDM 通信时钟既可能来自目标机的总线时钟，也可能来自其内部的 BDM 专用时钟。不同的 HCS08/RS08 MCU 其 BDM 专用时钟源可能不一样。

MCU 在正常用户模式下复位时，BDM 状态和控制寄存器 BDCSCR 的 CLKSW 位默认为 0，即复位后 BDM 的默认时钟为 BDM 专用时钟。这个 BDM 专用时钟源是固定频率的，因此，在用 BDM 进行调试时，一般选用 BDM 专用时钟源进行调试而不使用总线时钟，因为 BDM 专用时钟源频率稳定，易于保证双方通信速率的匹配，而总线时钟有可能因为用户程序的设置而产生变化，从而影响 BDM 通信的正常进行。因此，在使用 BDM 调试器时需要注意 BDM 时钟源设置。

BDM 的底层操作是读、写一个数据位（bit）的操作。而 BDC 指令集，其基本单位是 1 字节（byte）。因此，在 BDM 通信中，最重要的事情就是可靠的读/写数据位的操作实现。图 2-2 是 BDM 调试器通过 BKGD 引脚向目标机发送 1 位的时序图。从图中可以看出，一个典型的 BDM 位传输过程持续 16 个目标机 BDC 时钟周期，每次位传送均以一个下降沿开始，目标机在传输下降沿开始后的第 10 个周期时读取 BKGD 引脚上的电平值。下一个数据位的传输必须从第 16 个周期之后开始。在谈论到 BDM 的时钟周期时，均指目标机的 BDC 通信时钟周期，与编程调试器上的 MCU 运行频率无关。

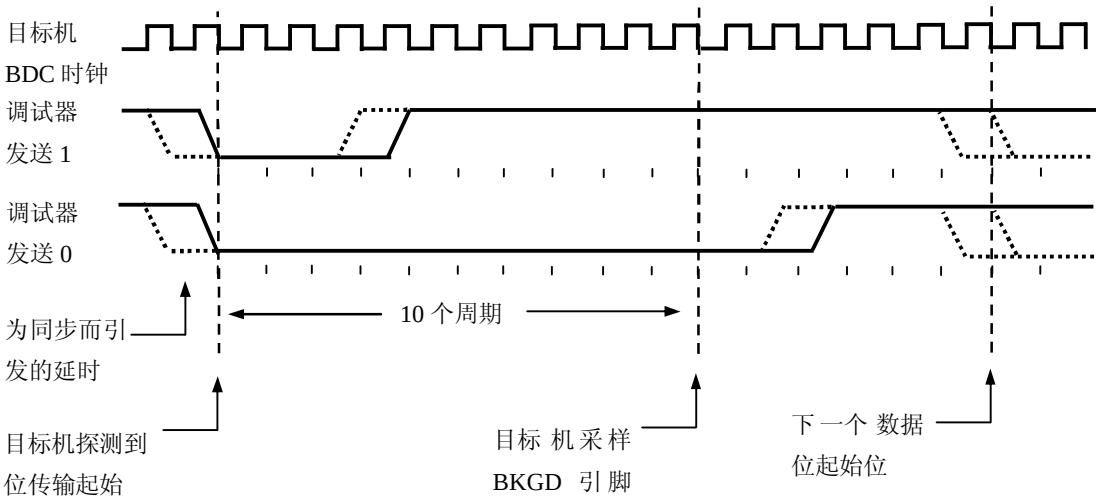

图 2-2 BDM 调试器向目标机发送数据位的时序

图 2-3 是调试器从目标机读取数据位 1 的时序图。从图中可以看出双方协商的过程：首先是 BDM 调试器拉低 BKGD 引脚 4 个目标机 BDC 时钟周期，然后 BDM 调试器释放对 BKGD 引脚的驱动。由于双方在位传输起始点的确定上可能相差 1 个时钟周期，所以目标机无法确定调试器释放 BKGD 的准确时刻。如果目标机立刻驱动 BKGD 为高，有可能与调试器的输出低电平时刻有重叠，就可能造成高低电平短接；而双方 BDM 时钟的差异也会增加对 BKGD 引脚释放时刻判断的不确定性。因此，目标机不是在探测到 BKGD 引脚低电平跳变后的 4 个 BDM 周期内就开始驱动 BKGD，而是延长到第 8 个周期后驱动该引脚。这样，在调试器释放对 BKGD 引脚的驱动时，通信双方的 BKGD 管脚均处于输入状态，BKGD 靠内部上拉电阻上拉维持高电平。由于 RC 时间常数的影响，这个上拉过程将持续一段的时间。在传输开始后的第 8 个周期，目标机开始驱动 BKGD 为高电平，以加速 BKGD 引脚的上升过程。此后，数据已经准备好，BKGD 引脚为稳定的高电平，目标机随后释放对 BKGD 的驱动。调试器在第 10 个周期读取 BKGD 引脚的高电平“1”。到第 16 个周期，数据位传输过程结束，可以开始下一次新的位传输。

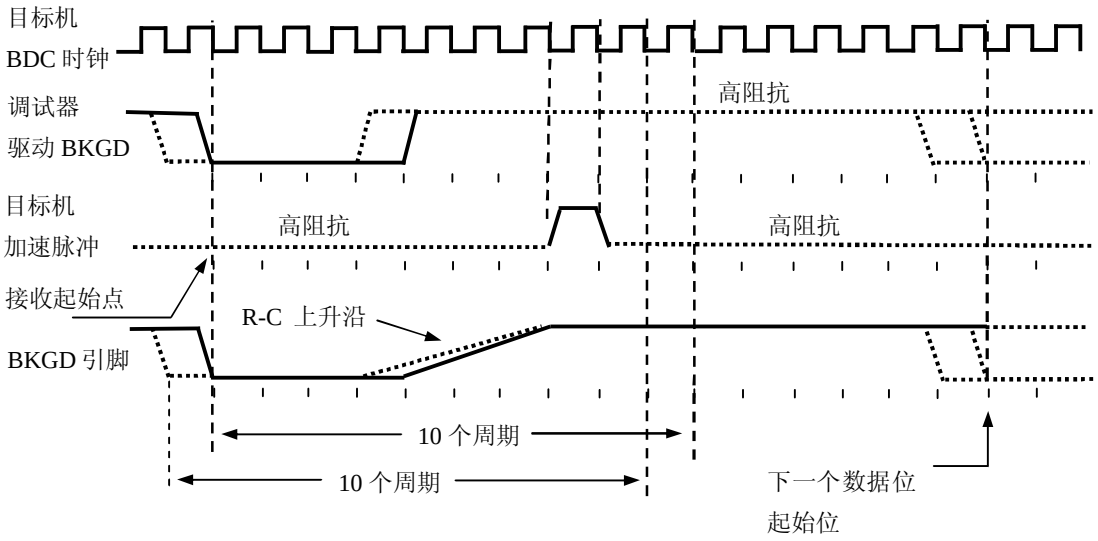

图 2-3 BDM 调试器接收数据位“1”的时序

调试器从目标机读取数据位 0 的过程与读取数据位 1 的过程大体相当。略有不同的是当目标机探测到调试器发出的低电平跳变后，目标机持续驱动 BKGD 为低，直到第 13 个周期后驱动 BKGD 回到高电平并释放。调试器在第 10 个周期读取 BKGD 引脚的低电平“0”。图 2-4 为调试器接收数据位“0”的时序。

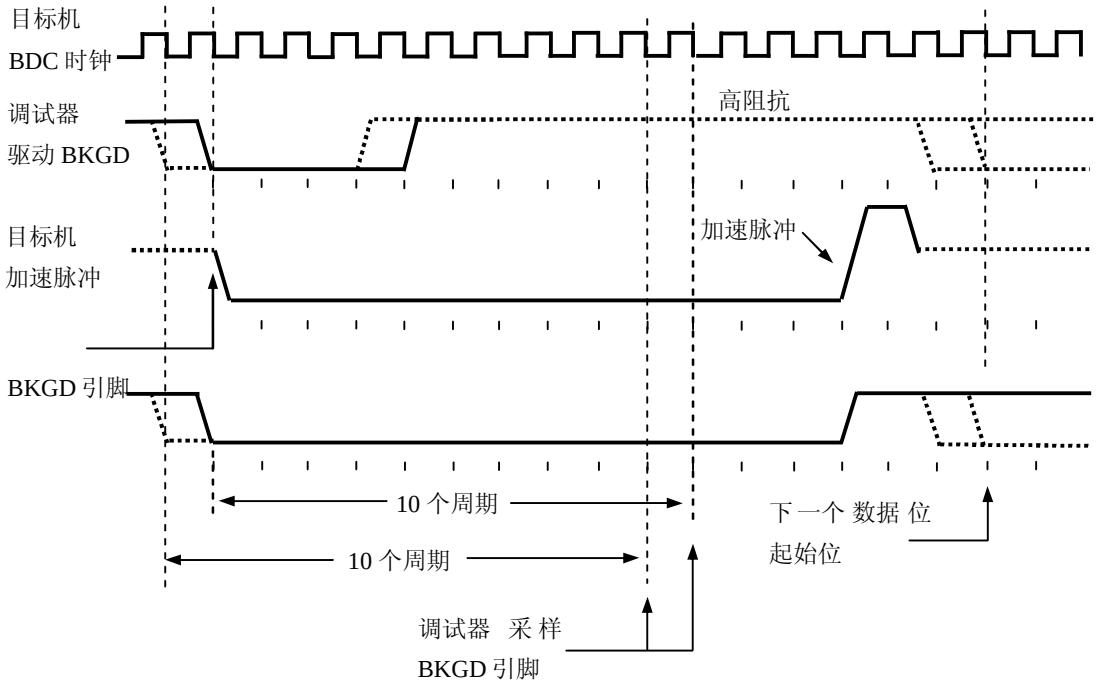

图 2-4 BDM 调试器接收数据位“0”的时序

因为难以保证调试器端与目标机的时钟频率严格相等，而且对发送起始点可能存

在一个周期的误差，所以在位读/写的时序上保留了很大的裕量，以保证 BKGD 的电平稳定后一两个周期后接收端才采样信号，并且发送端也在预期采样点后 4～5 个周期内才释放对 BKGD 引脚的驱动。这就保证了即使在两端时钟有较大偏斜的情况下，也有足够的建立和保持时间。HCS08/RS08 系列芯片的 BDC 模块总是在必要的时候“偷取”（steal）一个周期，这对应用程序的实时操作仅产生微小的影响。因为一个存储器访问操作包含命令（8 位），地址（16 位），以及数据（至少 8 位），以及 16 个周期的命令延时，而每一位的传输时间至少为 16 个周期，所以总的时间至少需要 (32 * 16) +16 = 528 个周期，因此在最差的情况下，BDC 执行一个存储器访问操作，应用程序至少已执行了 528 个周期。BDC 访问存储器时，需要延时 16 个 BDC 时钟周期，而访问 BDC 内部的 BDCSCR、BDCBKPT 寄存器则不需要任何延时。

2.2.4 BDM 指令的组织

BDM 指令分为两类：软指令和硬指令。软指令需要通过 CPU 运行才能实现，执行这些指令必须让 CPU 停止运行应用程序进入背景调试模式。这类指令主要完成读/写 CPU 寄存器、从指定地址处运行应用程序、单步执行应用程序等功能。硬指令由 BDC（Background Debug Controller）模块执行，可以在不干扰 CPU 运行的前提下利用 CPU 运行中的空周期实现对存储器的读/写，实现动态修改存储器内容以及设置断点、设置 BDC 工作状态等功能。

根据指令命令字后面所带参数的性质，BDC 指令集的指令可以分为以下几类：

① 无参数，主要对应一些控制性质的指令，如 BACKGROUND 等。

② 只有写入值，主要对应一些无需寻址（如寄存器）的写指令，如 WRITE_A 等。

③ 无地址，有返回值，主要对应一些无需寻址（如寄存器）的读指令，如 READ_A 等。

④ 带地址和写入值，主要对应存储器写指令，如 WRITE_BYTE 等。

⑤ 带地址，有返回值，主要对应存储器读指令，如 READ_BYTE 等。

⑥ 带状态报告，主要对应一些增强的读、写指令，如 READ_BYTE_WS 等。

表 2-2 列出了 HCS08 MCU 中定义的 BDM 指令集。

表 2-2 HCS08 BDM 指令列表

指令名称	指令类型	指令码及格式	指令描述
SYNC	软指令	无	同步指令
ACK_ENABLE	软指令	D5/d	启用硬件握手协议
ACK_DISABLE	软指令	D6/d	取消硬件握手协议
BACKGROUND	软指令	90/d	令 CPU 进入背景模式(如果 ENBDM=0，则忽略)
READ_STATUS	软指令	E4/SS	读 BDC 状态寄存器 BDCSCR
WRITE_CONTROL	软指令	C4/CC	写 BDC 状态寄存器 BDCSCR
READ_BYTE	软指令	E0/AAAA/d/RD	读取指定地址存储器的值
READ_BYTE_WS	软指令	E1/AAAA/d/SS/RD	读取指定地址存储器的值,并返回 BDM 状态
READ_LAST	软指令	E8/SS/RD	重读上一次指定地址的内容，并返回 BDM 状态
WRITE_BYTE	软指令	C0/AAAA/WD/d	向指定地址存储器写入 1 个字节数据
WRITE_BYTE_WS	软指令	C1/AAAA/WD/d/SS	向指定地址存储器写入 1 个字节数据，并返回 BDM 状态
READ_BKPT	软指令	E2/RBKP	读 BDC 断点寄存器 BDCBKPT 的值
WRITE_BKPT	软指令	C2/WBKP	写 BDC 断点寄存器 BDCBKPT
GO	硬指令	08/d	运行 PC 指针处的程序，CPU 从背景模式进入用户模式
TRACE1	硬指令	10/d	执行 PC 指针处的一条用户程序指令，然后返回背景模式
READ_A	硬指令	68/d/RD	读累加器 A
READ_CCR	硬指令	69/d/RD	读 CCR 寄存器
READ_PC	硬指令	6B/d/RD16	读 PC 寄存器
READ_HX	硬指令	6C/d/RD16	读 HX 寄存器
READ_SP	硬指令	6F/d/RD16	读 SP 寄存器
READ_NEXT	硬指令	70/d/RD	H:X+1->H:X 然后读取 H:X 处的值

READ_NEXT_WS	硬指令	71O/d/SS/RD	H:X+1->H:X 然后读取 H:X 处的值， 并返回 BDM 状态
WRITE_A	硬指令	48/WD/d	设置累加器 A
WRITE_CCR	硬指令	49/WD/d	设置 CCR 寄存器
WRITE_PC	硬指令	4B/WD16/d	设置 PC 寄存器
WRITE_HX	硬指令	4C/WD16/d	设置 HX 寄存器
WRITE_SP	硬指令	4F/WD16/d	设置 SP 寄存器
WRITE_NEXT	硬指令	5O/WD/d	H:X+1->H:X 然后更新 H:X 处的值
WRITE_NEXT_WS	硬指令	51/WD/d/SS	H:X+1->H:X 然后更新 H:X 处的值， 并返回 BDM 状态

2.2.5 BDM 的进入方式

进入 BDM 模式有四种方式：

①在 MCU 复位信号的上升沿，BKGD 引脚低电平。

②从 BKGD 引脚收到 BACKGROUND 命令。

③CPU 执行 BGND 指令。

④CPU 执行时遇到程序断点。

2.3 USB 通用串行总线

USB 于 1995 年面世，目前已发展成为一种解决个人计算机与外设连接问题的事实标准。USB 具有以下特点[14][15]：

⑴ 使用方便

USB 非常通用，很多种外设都可以使用它，不需要为每个外设准备不同的接口和协议。USB 还支持热插拔，当用户连接 USB 外设到一个正在运行的系统时，系统能自动检测外设，自动载入合适的软件驱动，无需用户做更多的操作。USB 设备也不涉及中断冲突的问题，USB 接口单独使用自己的保留中断，不会同其它设备争用计算机有限的资源，同样为用户省去了硬件配置的烦恼。

⑵ 速度够快

速度性能是 USB 技术的突出特点之一。一个全速 USB 接口可以以 12Mb/s 的速度进行通信。实际数据传输速率比这个数值低一些，因为总线必须携带状态、控制和

错误检测信号以及数据，因为所有外设都共用总线。USB2.0 规范允许以 480Mb/s 传输数据，这使得 USB 对打印机和其它需要快速传递大容量数据的外设更具吸引力。USB 也支持 1.5Mb/s 的低速。低速外设通常很便宜。而且，它们的电缆可以更灵活（如鼠标），因为电缆不需要屏蔽。

⑶ 可靠性高

USB 的可靠性来自于硬件设计和数据传输协议两方面。USB 驱动器、接收器和电缆的硬件规范消除大多数的可能引起数据错误的噪声。此外，USB 协议使用了数据错误的检测并能通知发送者，因此它可以重新发送。检测、通知和再发送都由硬件来完成，不需要任何程序。

⑷ 成本低廉

虽然 USB 比以前的接口更复杂，但它的组件和电缆并不贵。带有 USB 接口的设备与带有相同功能的老接口的设备所需要的费用差不多是相同或更少一些。对成本非常低的外设来说，低速选择也降低了对硬件的要求。HCS08/RS08 的编程调试器采用符合 USB1.1 规范的低速通信芯片 JB8，有效的降低了成本。

⑸ 功耗很低

当 USB 外设不被使用时省电电路和代码会自动关闭它的电源，但仍然能够在需要的时候做出反应。这个特征对于利用电池供电的设备尤其有用。

2.3.1 USB 基本概念

首先介绍端点和管道的概念。

1、端点：位于 USB 设备或主机上的一个数据缓冲区，用来存放和发送 USB 的各种数据，每一个端点都有惟一的确定地址，有不同的传输特性（如输入端点、输出端点、配置端点、批量传输端点）

2、管道：一个 USB 管道是主机端驱动程序的一个数据缓冲区与一个外设端点的连接，它代表了一种在两者之间移动数据的能力。一旦设备被配置，管道就存在了。管道有两种类型：数据流管道（其中的数据没有 USB 定义的结构）与消息管道（其中的数据必须有 USB 定义的结构）。管道只是一个逻辑上的概念。

2.3.2 USB 的数据格式

USB 数据是由二进制数字串构成的，首先数字串构成域（有七种域类型），域再

构成包，包再构成事务（IN、OUT、SETUP 三种事务），事务最后构成传输（中断传输、并行传输、批量传输和控制传输）。所有总线操作通讯过程都可以归结为三种包的传输：令牌包、数据包和应答包。

（一）域：是 USB 数据最小的单位，由若干位组成（位个数由具体的域决定），域可分为七个类型：

1、同步域（SYNC），八位，值固定为 0000 0001 。所有的包都起始于同步域，它被用于本地时钟与输入信号的同步，SYNC 的最后两位作为一个记号表明 PID 域（标识域）的开始。

2、标识域（PID），由四位标识符+四位标识符反码构成，表明包的类型和格式，USB 的标识码有 16 种。对于每个包而言，PID 紧跟着 SYNC。主机和所有的外设都必须对接收到的 PID 域进行解码。如果出现错误或者解码为未定义的值，那么这个包就会被接收者忽略。如果外设接收到一个 PID，它所指明的操作类型或者方向不被支持，外设将不会做出响应。

3、地址域（ADDR）：七位地址，代表了设备在主机上的地址，地址 0000000（7 位）被命名为零地址，是设备被复位或上电时，在被主机配置、枚举前的默认地址。由此可以知道一个 USB 主机只能接 127 个设备。外设端点都是由地址域指明的，它包括两个子域：外设地址和外设端点。外设必须解读这两个域，其中有任何一个不匹配，主机发送的令牌就会被忽略。外设地址域(ADDR)指定了外设，它根据 PID 所说明的令牌的类型指明了外设是数据包的发送者还是接收者。

4、端点域（ENDP）：四位，由此可知一个 USB 设备有的端点数量最大为 16 个。它使设备可以拥有几个子通道。所有的设备必须支持一个控制端点 0（EndPoint 0 ）来实现一个缺省的控制方法。这种方法将端点 0 作为输入端点，同时也将端点 0 作为输出端点。USB 系统用这个缺省方法初始化及一般地使用逻辑设备（即设置该设备）。缺省控制支持对控制的传送，一旦设备加电，且又收到一个总线复位命令，端点 0 就是可访问的了。低速设备在端点 0 外，只能有 2 个额外的可选端点。而高速设备可以支持最多 16 个端点（1 个控制端点和 15 个额外端点）。

5、帧号域（FRAM），11 位，每一个帧都有一个特定的帧号，帧号域最大容量 0x7FF，对于同步传输有重要意义（同步传输为四种传输类型之一）。这个域只存在于每帧开始时的 SOF 令牌中。

6、数据域（DATA）：长度为 0~1023 字节，在不同的传输类型中，数据域的长度各不相同，但必须为整数个字节的长度。

7、校验域（CRC）：对令牌包和数据包中非 PID 域进行校验的一种方法。CRC 校验在通讯中应用很泛，是一种很好的校验方法。USB 中的 CRC 校验由硬件负责实现。

（二）包：由域构成的包有四种类型，分别是令牌包、数据包、握手应答包和特殊包，前面三种是重要的包，不同的包的域结构不同。特殊包用于低速设备。

1、令牌包（Token Packet）

可分为输入包（IN）、输出包（OUT）、设置包（SETUP）和帧起始包（SOF）。注意这里的输入包用于设置输入命令，输出包用来设置输出命令，而不是用于放置数据的。其中输入包、输出包和设置包的格式都是一样的，格式为：

PID(8 位)	ADDR(7 位)	ENDP(4 位)	CRC5 (5 位,ADDR 和 ENDP 字段的 CRC)

对于输出包和设置包来说，ADDR 和 ENDP 中所指明的端点将接收到主机发出的数据包，而对 IN 事务来说所指定的端点将输出一个数据包。主机以一定的速率发送 SOF 包，SOF 不引起任何操作。对于帧起始包的格式为：

PID(8 位)	Frame Number(11 位)	CRC5(5 位，DATA 字段的 CRC)

2、数据包（Data Packet）

数据包有 DATA0 和 DATA1 两种类型，这两种包的定义是为了支持数据触发同步。USB 发送数据的时候，当一次发送的数据长度大于相应端点的容量时，就需要把数据包分为好几个包，分批发送， DATA0 包和 DATA1 包交替发送，即如果第一个数据包是 DATA0，那第二个数据包就是 DATA1。但也有例外情况，在同步传输中（四类传输类型中之一），所有的数据包都是为 DATA0。数据包格式为：

PID(8 位)	DATA(0-1023 字节)	CRC16(16 位，DATA 字段的 CRC)

3、握手应答包（Handshake Packet）

结构最为简单的包。握手应答包用来报告数据传输的状态，有三种类型：

① 确认包 ACK：表明数据接收成功

② 无效包 NAK：指出设备暂时不能传送或接收数据，但无需主机介入。可以解释成设备忙。

③ 出错包 STALL：指出设备不能传送或接收数据，但需要主机介入才能恢复。

NAK 和 STALL 不能由主机发出。

握手应答包仅包含一个 PID 域，其长度为 8 位即一个字节。

只有支持流控制的传输类型（控制、中断和批传输）才能返回握手应答包。

4、特殊包（Special Packet）：

特殊包的 PID 名称为 PRE（preamble），用于低速操作。

2.3.3 USB 事务

在 USB 上数据信息的一次接收或发送的处理过程称为事务（Transaction）。事务类型分为输入（IN）事务、输出（OUT）事务和设置（SETUP）事务三类，每一种事务都由令牌包、数据包、握手应答包三个包处理阶段构成。这里阶段的意思是指这些包的发送具有一定的时间先后顺序。事务的三个包处理阶段如下：

令牌包阶段：启动一个输入、输出或设置的事务。

数据包阶段：按数据传送方向发送相应的数据。

握手应答包阶段：返回数据的接收情况，在同步传输的 IN 和 OUT 事务中没有这个阶段，这是比较特殊的。

事务的三种类型如下（以下按三个阶段来说明一个事务）：

1、IN 事务

令牌包阶段——主机发送一个 PID 为 IN 的输入包给设备，通知设备要往主机发送数据；

数据包阶段——设备根据情况会作出三种反应（要注意：数据包阶段也不总是传送数据，根据传输情况还会提前进入握手应答包阶段）：

① 设备端点正常，设备向主机发送数据包（DATA0与 DATA1交替）。

② 设备正在忙，无法向主机发送数据包；于是设备就发送 NAK 握手应答包，IN 事务提前结束。

③ 相应设备端点被禁止，设备发送 STALL 出错包，事务提前结束，总线进入空闲状态。

握手应答包阶段——主机正确接收到数据之后就会向设备发送 ACK 包。

2、OUT 事务

令牌包阶段——主机发送一个 PID 为 OUT 的输出包给设备，通知设备要接收数据；

数据包阶段——比较简单，就是主机向设备发数据，DATA0与 DATA1交替。

握手应答包阶段——设备根据情况会作出三种反应：

① 设备端点接收正确，设备向主机返回 ACK，通知主机可以发送新的数据，如果数据包发生了 CRC 校验错误，将不返回任何握手信息；

② 设备正在忙，无法向主机发送数据包；于是设备发送 NAK 握手应答包，通知主机再次发送数据。

③ 相应设备端点被禁止，设备发送 STALL 出错包，事务提前结束，总线直接进入空闲状态。

3、SETUP 事务

令牌包阶段——主机发送一个 PID 为 SETUP 的输出包给设备，通知设备要接收数据。

数据包阶段——比较简单，就是主机向设备发送一个 8 个字节的固定大小的 DATA0包，这 8 个字节的内容就是标准的 USB 设备请求命令。

握手应答包阶段——设备接收到主机的命令信息后，返回 ACK，此后总线进入空闲状态，并准备下一个传输（在 SETUP 事务后通常是一个 IN 或 OUT 事务构成的传输）。

2.3.4 USB 传输

传输由 OUT、IN、SETUP 事务构成。传输有四种类型：中断传输、批量传输、同步传输、控制传输。它们在数据格式、传输方向、数据包容量限制、总线访问限制等方面有着各自不同的特征，其中中断传输和批量转输的结构一样，同步传输的结构最简单，控制传输是最重要的也是最复杂的传输。

1、控制传输

USB 设备初次加载到主机之后，主机通过控制传输来交换信息，读取设备地址和设备描述符，使得主机识别设备，并安装相应的驱动程序。控制传输由 2～3 个阶段构成：初始设置阶段、数据阶段（无数据要求则没有该阶段）、状态阶段。

阶段一：初始设置阶段

USB 设备在正常使用之前，必须先配置。在设置阶段，一个 SETUP 事务用来将信息传送到控制端点。SETUP 事务与 OUT 事务的格式类似，区别在于标识域（PID）不同。设置阶段的 SETUP 事务的作用是执行一个设置的数据交换，并定义此控制传输的内容。

阶段二：数据传输阶段

数据阶段用来传输主机与设备之间的控制数据。有无数据阶段取决于设置阶段的 SETUP 事务的 DATA0 数据包所发送的标准请求命令。数据阶段包含一个或多个 IN 或 OUT 事务。数据阶段的所有事务都具有相同的传输方向，即所有的事务都是 IN 事务或所有的事务都是 OUT 事务。对控制读取而言，数据阶段的事务都是 IN 事务；对控制写入而言，数据阶段的事务都是 OUT 事务。数据阶段的数据量大小和传输方向都在初始设置阶段决定。如果数据量的大小超出指定的数据包的大小，则分成多个数据包进行传输，此时最后一个数据包的数据量可能小于数据包所能传递的最大数据量。

阶段三：状态阶段

状态阶段用来表示整个传输的过程已经完全结束了。状态阶段的数据流向与先前的数据阶段的相反。数据阶段是 OUT 事务，则状态阶段就是一个 IN 事务；数据阶段是 IN 事务，则状态阶段就是 OUT 事务。如果控制传输没有数据阶段，则状态阶段就是一个 IN 事务。状态阶段的事务的数据包标识码 PID 为 DATA1。控制写入传送在状态阶段的事务的数据时相返回状态信息。而对于控制读取传　　送，主机在状态阶段事务的数据时相中发出零长度的数据包之后，功能部件在握手时相返回状态信息。

表 2-3 概括了各种状态下设备发送的握手信号类型。

表 2-3 各状态下发送的握手信号类型

状态响应	控制写入传送（在数据时相发送）	控制读取传送（在握手时相发送）
设备完成	零长度的数据包	ACK 握手应答包
设备出错	STALL 握手应答包	STALL 握手应答包
设备忙	NAK 握手应答包手	NAK 握手应答包

2、中断传输

中断传输由IN或OUT事务组成。用于非周期的、自然发生的、数据量很小的信息的传输如键盘、鼠标等。对于高速设备，允许数据包最大容量为小于或等于64 字节；对于低速设备，只能小于或等于8 字节。具有最大服务周期保证，即在规定时间内保证有一次数据传输。

3、批量传输

批量传输由IN或OUT事务组成，用于大容量数据传输，没有固定的传输速率，也不占用带宽。当总线忙时，USB会优先进行其他类型的数据传输，而暂时停止批量转输。批量传输具有数据传输保证，在必要时可以重试以保证数据的准确性。批量传输操作包括令牌、数据、应答三个阶段。对于输入操作，如果设备不能返回数据，那么必须发出NAK包或STALL 包；对于输出，如果设备不能接收数据，也要返回NAK包或STALL包。

4、同步传输

同步传输不同于其他类型的传输，只包含两个阶段：令牌阶段和数据阶段。因为同步传输不支持重发的能力，所以没有握手应答阶段，另外它也不支持数据的触发同步与重试。同步传输在数据包阶段所有的数据包都为DATA0。

2.3.5 USB标准设备请求

USB标准设备请求命令是用在控制传输中的“初始设置”部分里的数据包阶段。USB标准设备请求命令共有11个，具有相同的结构，大小都是8个字节，由5 个字段构成（字段是标准请求命令的数据部分），结构如下（括号中的数字表示字节数，首字母bm,b,w分别表示位图、字节，双字节）：

bmRequestType(1)	wValue(2)	bRequest(1)	wIndex(2)	wLength(2)
请求特征	根据不同的请求含义改变	请求命令代码	根据不同的请求含义改变，常用于传送索引或偏移	如有数据传送阶段，则为传送的数据的字节数

bmRequestType域说明：

请求特征各位的含义表示如下。

Bit7：传输方向（Direction），0=主机至设备，1=设备至主机。

Bit6-Bit5：类型（Type），0x00=标准，0x01=类，0x10=用户自定义，0x11=保留。

Bit4-Bit0：接收方（Recipient），0=设备，1=接口，2=端点，3=其他，4..31=保留未使用。

这个域表明请求的特性。特别地，这个域表明了下一阶段控制传输的方向。如果 wLength 域为 0，表明没有数据传送阶段，则 bmRequestType 的位 7 的值就会被忽略。

USB 规范定义了一系列所有设备必须支持的标准请求，详见表 2-4。另外，一个设备类可以定义更多的请求，设备厂商也可以定义设备支持的请求。bmRequestType 域也定义了接收者。当指定的接收者是接口或端点时，wIndex 域指定接收的接口或端点。

请求可以被引导到设备，接口或某一个设备端点。

表 2-4 USB 标准设备请求类型

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE 10H	特性选择	0 接口 端点	0	无
10000000B	GET_CONFIGURATION 08H	0	0	1	配置值
10000000B	GET_DESCRIPTOR 06H	描述表种类和索引	0 或 语言标志	描述表 长度	描述表
10000001B	GET_INTERFACE 10H	0	接口	1	替换接口
10000000B 10000001B 10000010B	GET_STATUS 00H	0	0 接口 端点	2	设备、 接口或 端点状态
00000000B	SET_ADDRESS 05H	设备地址	0	0	无
00000000B	SET_CONFIGURATION 09H	配置值	0	0	无
00000000B	SET_DESCRIPTOR 07H	描述表种类和索引	0 或 语言标志	描述表长 度	描述表
00000000B 00000001B 00000010B	SET_FEATURE 03H	特性选择	0 接口 端点	0	无
00000001B	SET_INTERFACE 11H	替换设置	接口	0	无
10000010B	SYNCH_FRAME 12H	0	端点	2	帧号

bRequest 域说明：

这个域标识特定的请求。bmRequestType 域的类型（bit7-bit6）可修改此域的含义。

bmRequestType 域的代码具体含义见表 2-3：

表 2-5 标准设备请求代码

Brequest 码	Value
GET_STATUS	0
CLEAR_FEATURE	1
保留未使用	2
SET_FEATURE	3
保留未使用	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

wValue 域说明：

该域用来传送当前请求的参数给设备，其值取决于具体请求。

wIndex 域说明：

该域的值取决于当前请求的内容。wIndex 域常用来在请求中指定一个接口或端点。

wLength 域说明：

该域表明下一阶段的数据传输长度。传输方向由 bmRequstType 域的 Direction 位指出。wLength 域为 0 则表明无数据传输。在输入请求下，设备返回的数据长度不应多于 wLength 域指定的长度，但可以少于。在输出请求下，wLength 表明主机发出的确切数据量。如果主机发送多于 wLength 的数据，设备做出的响应是无定义的。

2.3.6 USB 的设备状态

根据 USB 设备实现的功能，它可以包含若干种状态，如上电状态、采集状态等，其中有的状态是可见的，有的状态只适用于其设备内部。USB 设备状态有以下几种：

（1）连接状态

USB 设备已经连接在主机或集线器的下行端口上，但 USB 总线还没有提供电流给 USB 设备。

（2）上电状态

USB 设备已连接至主机或集线器的下行端口，且已得到 USB 总线电源，但还没有被复位。

（3）缺省状态

USB 设备会响应主机或集线器下行端口发出的复位信号，并进行复位操作。在复位操作结束后，USB 设备进入缺省状态，这时它可以从总线上获取小于 100mA 的电流，并可使用缺省设备地址对某些 USB 事务作出响应。

（4）地址状态

USB 设备复位结束后，主机会为其分配一个唯一的设备地址，当设备还没有被配置时，这时设备处于地址状态。

（5）配置状态

USB 设备在使用前必须被配置，即正确处理主机发出的 SetConfiguration(x)请求，其中 x 代表一个非 0 的配置值。在配置操作完成后，USB 设备将处于配置状态。

（6）挂起状态

当 USB 设备在 3ms 内没有检测到总线活动时，它将自动进入挂起状态，这时其仍将保持原有的设备地址和配置值。USB 设备进入挂起状态时可以节省系统的功耗。

2.3.7 USB 设备和主机

在 USB 系统中，USB 设备可以被划分为三个功能模块：USB 总线接口、USB 逻辑设备和功能单元。它们的功能如下：

① USB 总线接口负责实现与主机间数据的物理传输。

② USB 逻辑设备负责处理 USB 总线接口与功能单元各端点之间的数据传输。USB 逻辑设备是程序员与 USB 设备打交道的部分。

③ 功能单元负责实现 USB 设备所特定的功能。

前两小节介绍了 USB 设备描述符和设备状态，在 4.2.2 节还将介绍 USB 设备枚举。了解这三个部分的内容是了解 USB 设备运行的关键。

USB 主机也划分为三个功能模块：USB 总线接口，USB 系统软件和客户软件。

USB 总线接口包括 USB 主控制器和根集线器两部分。其中，根集线器为 USB 系统提供连接起点，USB 主控制器负责完成主机和 USB 设备间数据的实际传输。

客户软件负责和 USB 设备的功能单元进行通信，以实现 USB 设备的特定功能，它一般包括 USB 设备驱动程序和界面应用程序两部分。客户软件不能直接访问该功能单元，其与USB 设备间的通信必须经过 USB 系统软件和 USB 总线接口才能实现。客户软件一般由开发人员自行开发。

USB 系统软件负责和 USB 逻辑设备进行配置通信，并管理客户软件启动的 USB 数据传输。它对 USB 主控制器的硬件进行抽象，并提供 USBDI 通信接口，以简化客户软件和 USB 设备功能之间的通信机制。系统软件一般包括 USB 主控制器驱动程序（HCD）、USB 总线驱动程序（USBD）和非 USB 主机软件三部分，这些软件通常由操作系统提供。

总之，主机为 USB 提供了以下功能：

① 检测 USB 设备的连接和断开。

② 管理主机和 USB 设备间的控制通信流。

③ 管理主机和 USB 设备间的数据通信流。

④ 收集总线状态和总线活动信息。

⑤ 控制主控制器和 USB 设备间的电气接口。

2.4 小结

本章主要介绍了编程调试器设计实现过程中的相关技术背景。HCS08 和 RS08 芯片中引入了 BDM 技术，为8 位 MCU 的开发应用带来了新的变革。BDM 是 Freescale 的专有调试接口，Freescale 是第一个把具有处理器调试功能的调试逻辑电路置入处理器核心中的微处理器厂商。BDM 开创了片上集成调试资源的趋势。采用的 JB8 芯片实现的是 USB1.1 规范。通过对 JB8 芯片的使用了解以及对 USB1.1 规范的研究，为今后进一步研究 USB2.0 技术打下良好的基础。

第三章 硬件设计

根据前面介绍的设计方案，HCS08/RS08 系列开发平台的硬件部分可划分为编程调试器和目标板两大部分。编程调试器主要接收 PC 主机发送的命令和数据完成对目标芯片的擦除写入以及调试的功能，同时向 PC 主机返回相关的操作结果信息和调试信息。该编程调试器具有对 HCS08 系列和 RS08 系列的通用性。设计制作 BDM 编程调试器可以使用任意一种 MCU，通过控制目标机的复位引脚和 BKGD 通信引脚，实现编程调试器与目标机的通信。

目标板即是目标芯片所在的电路板。目标板的形式有多种，一种是核心板加扩展板的方式，一种是单板方式。核心板加扩展板方式中，扩展板上集成了很多应用模块的外围电路，例如 LED 模块、串口模块、液晶模块和键盘模块等。扩展板具有通用性，不同芯片对应不同的核心板，核心板和扩展板之间通过两个双排 20 针的插座相连，将核心板上的芯片引脚引到扩展板上的各个应用模块。

3.1 芯片选型

编程调试器主控芯片选择主要基于以下几点考虑：

① 是否便于实现。

② 是否能够提供足够的性能。

③ 是否有足够的开发工具支持。

④ 芯片价格。

由于现在 USB 日益成为计算机外部接口的主流，近些年几乎所有新推出的台式机，笔记本电脑都提供了 USB 接口。USB 具有即插即用的特点，并且可以通过接口为 USB 设备供电。综合考虑，决定采用 Freescale 的 MC68HC908JB8[16]（以下简称 JB8）作为调试器的主控芯片。JB8 的主要特征如下：

① 3MHz 内部总线频率。

② 256 字节的片内 RAM。

③ 8192 字节的片内 Flash 存储器，具有在线编程的功能。

④ 16 位双通道定时器接口模块，具有输入捕捉、输出比较和脉宽调制输出功能。

⑤ 内置 USB 模块，支持 USB1.1 协议。其中端点 0 作为控制传输端点，具有 8

字节的发送缓冲区和 8 字节的接收缓冲区；端点 1 作为数据发送端点，具有8 字节发送缓冲区；端点 2 可作为数据收发端点，具有8 字节发送缓冲区和 8 字节接收缓冲区。

3.2 基本系统的电路设计

基本系统的电路是支撑 JB8 工作的最小系统，包括电源电路、时钟电路以及复位电路。

3.2.1 电源电路

JB8 的电源电路如图 3-1 所示。VCC 上连接的 10uF 电容和 0.1uF 电容主要是为电源进行滤波。发光二极管 LED 作为电源指示灯。

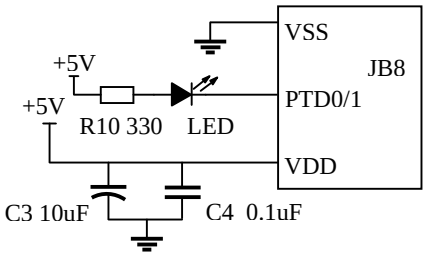

图 3-1 JB8 电源电路

JB8 与大多数 8 位 MCU 一样，电源单一，供电电路较简单。编程调试器采用 USB 总线供电。USB 接口电压标称值为 5V，但实际上 PC 机的 USB 接口电压范围一般为 4.4V～5.25V，另外，由于电缆和其它损失，USB 设备最后得到的电压可能会更低一些。JB8 的输入工作电压为 4.0V～5.5V，能够保证正常的工作。

3.2.2 时钟电路

时钟电路用来为芯片工作提供基本的时钟信号。C1，C2 为滤波电容，R 为反馈电阻，X1 为晶振。晶振用一种能把电能和机械能相互转化的晶体在共振的状态下工作，以提供稳定，精确的单频振荡。JB8 只有一种工作频率，不需要进行设置，其外接晶振为 6MHz，内部总线频率为 3MHz。图 3-2 为 JB8 的时钟电路图。

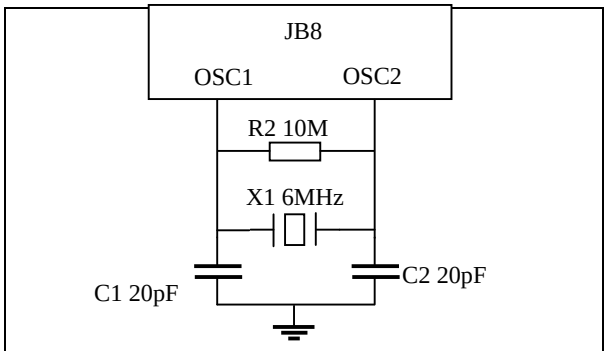

图 3-2 JB8 时钟电路

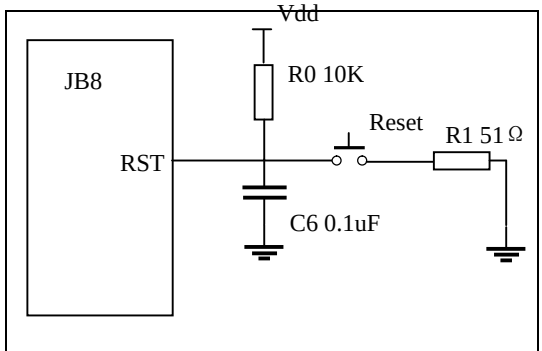

图 3-3 JB8 复位电路

3.2.3 复位电路

1．复位的概念和作用

在设计复位电路之前，有必要了解复位的概念和复位的功能。

复位能够使 MCU 立刻进入到初始状态，并且从用户定义的存储器地址开始执行程序。

对于 JB8 而言，当系统复位时，系统作出以下反应：

① MCU 将停止当前运行的所有程序。

② 把绝大部分的控制寄存器和状态寄存器的值恢复到初始状态值（个别寄存器不受复位影响）。

③ 从复位中断向量地址$FFFE-$FFFF 将用户自定义的复位中断处理程序入口地址送到程序计数器 PC 中。

④ 禁止 MCU 片内的 I/O 模块；I/O 接口被设置成高阻输入并禁止上拉允许。

⑤ 设置中断屏蔽位，屏蔽中断输入。

⑥ 堆栈指针 SP 置为$00FF.

可能导致复位的复位源包括：

① 上电复位（POR），由 Vdd 引脚上的电压正跳变引起的内部复位。

② 低压检测复位（LVI），由于电源电压降低到某个电压阈值以下时产生的内部复位。

③ 看门狗复位（COP），由看门狗定时器溢出引起的内部复位。

④ 非法操作码复位（ILOP），由不在指令集中的操作码引起的内部复位。

⑤ 非法地址复位（ILAD），由错误的地址引起的内部复位

⑥ USB 复位（USB），由 USB 设备复位引起。

2．外部复位电路设计

在设计 MCU 基本系统时，应该考虑到 MCU 的外部复位设计。外部复位电路如图 3-3 所示。其中，外部复位引脚 $\overline{RESET}$ 是由低电平复位的。

3.3 BDM 接口设计

BDM 接口用于连接编程调试器和目标芯片。目前常用的 Freescale 标准的 BDM 接口如图 3-4 所示，各个引脚信号的定义如表 3-1 所示。但由于引脚 Vdd 和 GND 在 BDM 接口两端，操作中容易插反导致目标芯片烧坏。所以对 BDM 接口引脚排列重新定义。

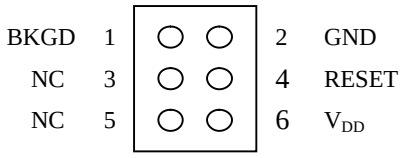

图 3-4 Freescale 的 BDM 接口引脚定义

由于编程调试器只用到了其中四个引脚，所以我们推建使用如图 3-5 所示的 B 型 USB 接口来代替 BDM 接口,这样的连接不仅让用户使用更方便,而且可以有效地防止反插。同时，考虑到具体应用中的各种实际情况，编程调试器也提供了排线接口方便使用。这时需要将 Freescale 定义的 BDM 接口稍作修改，将 BKGD 引脚与邻近的空引脚互换位置，即可防止操作中的误操作带来的影响。

表 3-1 BDM 调试端口信号含义

BKGD	BKGD 是单线背景调试模式引脚，它用来接收和发送背景调试指令
GND	接地
V_{DD}	电源
RESET	目标机复位引脚

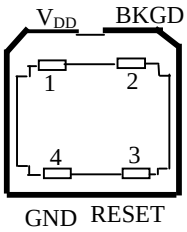

图 3-5 推荐使用的 BDM 调试头引脚定义

3.4 编程调试器电路设计

可以通过将主控芯片 JB8 的引脚直接与目标机的复位引脚及 BKGD 引脚相连，对目标机进行控制，实现单线通信。这种方式虽然实现简单，但是适应性较差，输出不稳定，而且适应的目标机的频率范围较小。为了最大限度的发挥 JB8 的功能，同时使得调试器能有 较高 的稳定性 ， 在目标 机与 JB8 之间 接入一个 三态 缓冲 器 74HC125[17] 。图 3-6 给出了编程调试器与目标机连接的原理图。

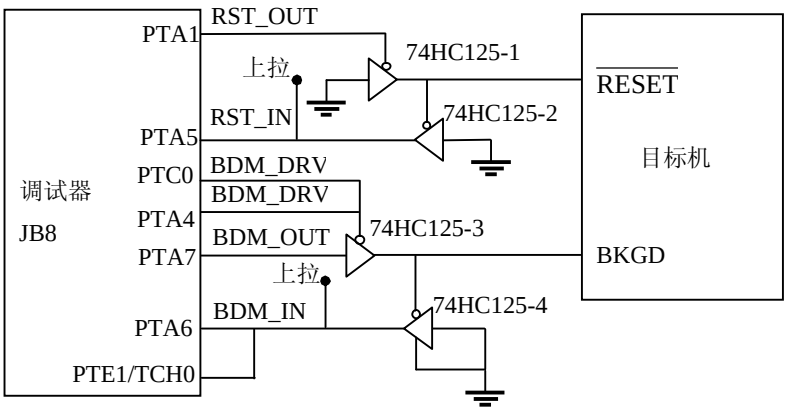

图 3-6 编程调试器与目标机连接原理图

如图所示，对目标机进行复位时，编程调试器拉低 RST_OUT，74HC125 工作，目标机的复位引脚 $\overline{RESET}$ 被拉低，拉低时间超过 100 毫秒则目标机产生复位。编程调试器可以通过 RST_IN 判断目标机的复位引脚电平。

编程调试器发送数据时，先将数据位内容放在 BDM_OUT 上，随后将 BDM_DRV 拉低，以驱动 74HC125-3 工作，数据就可以发送给目标机了。若将 BDM_DRV 拉高，74HC125-3 不工作，此时为高阻抗状态，目标机的 BKGD 也相应进入高阻态。JB8 的 PTC0 和 PTA4 同时接到 BDM_DRV，它们的作用是为了增强驱动能力，以确保稳定性。

在进行硬件设计时，需要充分考虑到硬件功能的最大发挥和软件功能实现上的效率及难易程度。而软件实现语言以汇编语言效率最高，在时序上最精确。因此，在仔细研究 HCS08 汇编指令格式后，决定将进行 BDM 操作的相关引脚分配到同一个输出端口的引脚上。其中，将 JB8 的 PTA7 作为数据输出引脚，PTA6 作为数据输入引脚，在进行 BDM 通信时，可以获得最好的实现效果和实现效率，同时也利于编程的实现和代码的优化。

由于调试器需要能够适应不同型号的芯片，因此需要解决调试器与目标机速率不匹配的问题。BDM 指令集中提供了一条 SYNC 指令用于解决同步问题，4.3.1 节详细描述了 SYNC 指令的实现机制。JB8 的 PTE1/TCH0 引脚是一个复用引脚，除作为普通 I/O 引脚外，还可以作为定时器通道 0 的引脚。在实现 SYNC 指令功能时，可设置为输入捕捉功能，用于捕捉目标机发送的电平跳变。通过测量目标机发送的 128 个低电平周期从而计算出目标机的 BDC 模块的运行频率。

3.5 USB 接口设计

编程调试器与 PC 机采用 USB 接口进行通信。图 3-7 给出了 USB 接口连接原理图。其中，SN75240 是一个噪声抑制器，是专门为 USB 接口电路设计的，用以抑制 USB 数据线上的瞬时电气噪声。此外，还需要将 USB 接头的屏蔽层接地，这样可以更好地屏蔽电磁干扰，保证数据的可靠性。

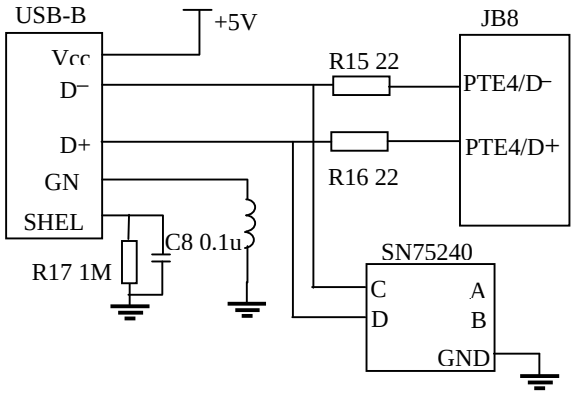

图 3-7 USB 接口电路

3.6 目标板设计

HCS08 系列和 RS08 系列 MCU 内部的 BDC 模块，其命令接口均一致。因此，对于这两个系列的 MCU，编程调试器所执行的动作均一致。为了更好的检验编程调试器的稳定性和适应性，设计了不同具体芯片的目标板来进行测试。用来测试的有具体项目中使用的实际电路板、教学用的最小系统板以及芯片裸机。进行实际电路板测试的有 HCS08GB60[18]和 HCS08GT32 以及 HCS08QG8[19] 3 种型号芯片，进行最小系统板测试的有 HCS08GB60、HCS08LC60[20]、HCS08QD4[21]、HCS08RG60[22]、HCS08AW60[23]以及 RS08KA2[24] 6 种型号芯片，进行芯片裸机测试的有 HCS08QG8、HCS08QD4 和 RS08KA2 这 3 种型号芯片。芯片裸机测试即表示将编程调试器与被测试芯片的管脚直接连接进行操作。这些测试芯片涵盖了 HCS08 和 RS08 目前的产品范围。HCS08 的 AW 系列为通用型系列，比较适应于工业和汽车电子类应用；G 系列为通用型高性能系列；Q 系列是比较有特色的一个系列，兼顾了精简封装和丰富的片内功能模块这两方面；R 系列是低功耗产品，供电电压 1.8V～3.6V，典型值是 2.7V，电压低于 1.8V 时会报警，低电压复位电平为 1.2V，适应电池供电的应用场合；L 系列是 2007 年 2 月新推出的低压低功耗产品，工作电压为 3V，内部带有液晶驱动接口。RS08 系列目前只推出了 KA 系列 KA1 和 KA2 两种产品，主要区别是内部 FLASH 的大小不同。

图 3-8 给出了编程调试器与目标板的连接实物图。

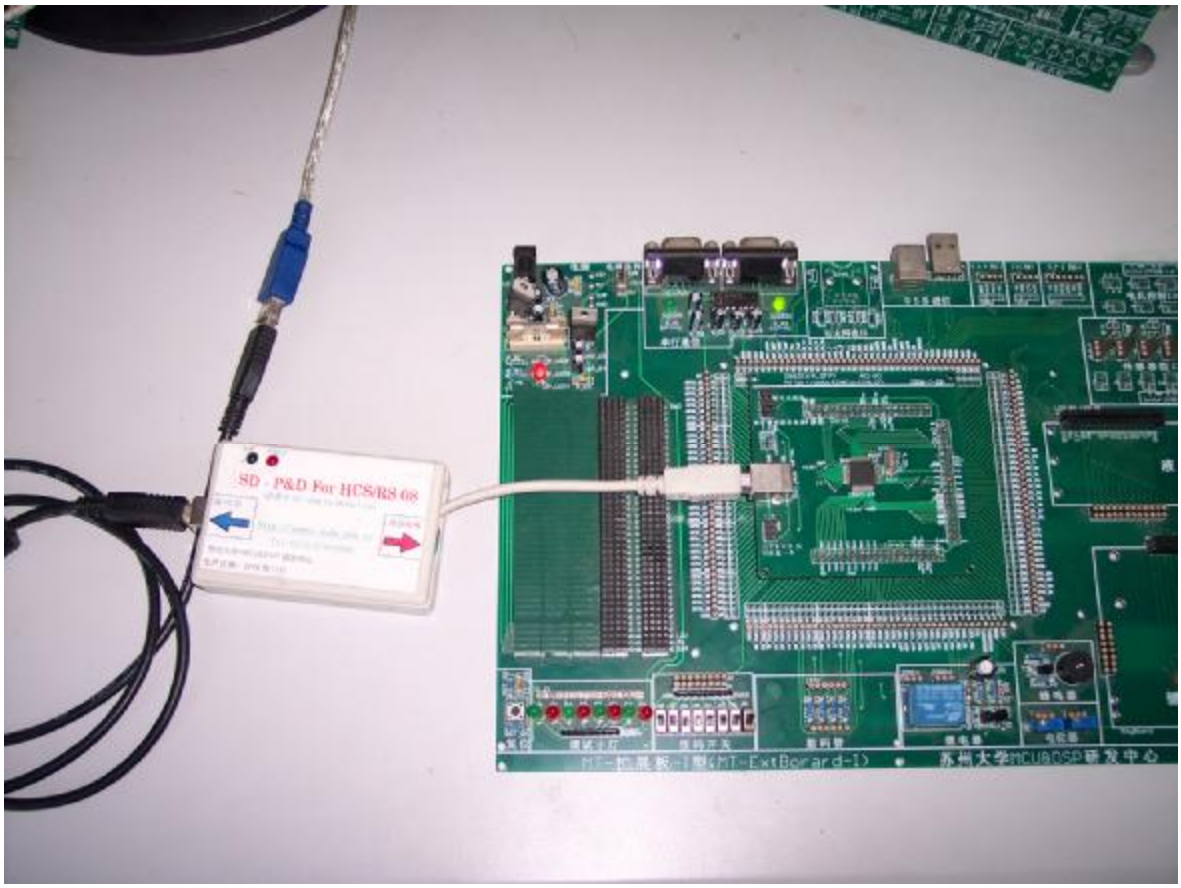

图 3-8 编程调试器与目标板的连接实物图

3.7 小结

在硬件电路设计过程中，需要考虑干扰的问题，确保硬件电路的稳定性。其中，晶振电路是一个很强的噪声源。由于工作频率高，易对系统其它电路造成干扰。其中，干扰信号的主要是时钟信号的 3 倍频率[25]。晶振电路同时也易受其它电路的干扰造成系统死机。解决的方法一般有：在保证性能的前提下尽量降低时钟频率；在布线时保证与时钟电路相关的走线尽可能短；将时钟电路的外接器件都用地线围绕起来以及采用卧式晶振等。

在 PCB 板设计的时候，还有考虑到整个电路板的大小。因为编程调试器一般都放在桌面上使用，因此，电路板的面积应尽可能小，器件的布置尽可能紧凑。

第四章 MCU 方软件设计

MCU 方软件负责接收 PC 发送的命令和数据，完成对目标芯片的各种操作，同时将操作的完成情况反馈给 PC 机。MCU 方的软件设计对整个系统的运行效率起到决定性的作用。

4.1 总体设计

MCU 方的程序设计既可以采用汇编语言，也可以采用 C 语言。两种语言各有所长。考虑到本设计中对时序的要求较为严格，JB8 芯片的 RAM 及 FLASH 容量大小有限制以及今后功能扩充带来的代码量增加，故采用汇编语言作为开发语言。

图 4-1 给出了编程调试器端的简易程序流程图。按如下步骤执行：

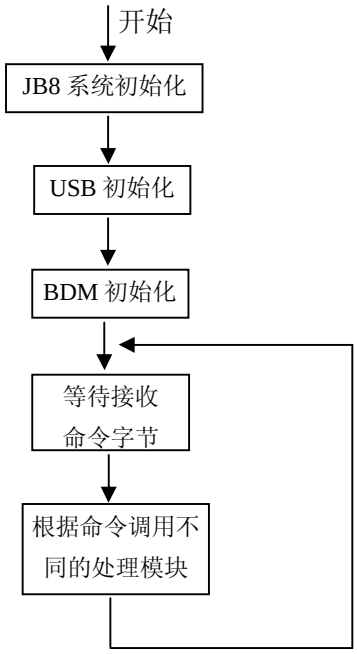

图 4-1 MCU 方程序流程图

① 进行芯片系统初始化工作，完成堆栈、定时器等的准备工作。

② USB 模块的初始化。

③ 对执行 BDM 功能的引脚进行初始化。

④ 接收 PC 机发送过来的命令字节。

⑤ 根据命令字调用相应的处理模块，同时将处理结果反馈给 PC 机。

这些处理模块主要有：擦除模块、写入模块和调试模块等。在处理完相应的模块后，程序跳转到④处继续循环执行。下面对各主要功能模块进行详细介绍。

4.2 USB 模块设计

JB8 芯片中包含的 USB 硬件模块实现 USB1.1 规范[26]的低速功能，通信速率为 1.5Mb/s。本节讨论 JB8 通过 USB 模块与 PC 机进行数据通信。USB 软件模块设计主要涉及对相关寄存器的设置和操作。由于 JB8 中 USB 模块的相关寄存器模块较多，在此不作详细说明，可参看附录 A 中介绍。

USB 通信可分为三个阶段：初始化阶段、枚举阶段和数据传输阶段[27]。

4.2.1 USB 初始化

在 MCU 方主程序开始阶段就要完成 USB 的初始化工作。初始化工作主要是对 USB 模块的控制端点和传输端点进行设置，清除全部中断标志位，允许 USB 响应各种中断等。这一部分内容较为简单。需要注意的是，在初始化例程中，中断寄存器 0 的 EOPIE 位和 SUSPND 位不能置位。

4.2.2 USB 设备枚举

编程调试器实际上是一个 USB 设备。任何 USB 设备在与 PC 主机通信之前，都需要进行枚举。主机对 USB 设备的识别过程叫做枚举，它是主机的设备驱动程序与设备通信的最基本的信息交换。这个过程由如下动作组成：分配一个地址给设备，从设备读取描述数据，分配和载入一个设备驱动程序以及从接收到的数据中出现的选项中选择一个配置。然后设备就被配置好了，并且准备好使用它的配置中支持的任何终端来传输数据。

主机枚举是通过给终端 0 发送包含标准 USB 请求的控制传输。所有的 USB 设备必须支持控制传输、标准 USB 请求和终端 0。对一个成功的枚举来说，设备必须对每一个请求相应，返回请求的信息以及采取其它请求的动作。

（1）描述符

描述符是数据结果，或是信息的格式化块，它可以使主机知道这个设备。每个描述符包含了关于这个设备整体的信息或者一个元素的信息。所有的 USB 外设必须响应对标准 USB 描述符的请求。在枚举的时候，主机使用控制传输来从设备请求描述符。在枚举过程中，被请求的描述符逐渐涉及设备的小的元素：首先是整个设备，然后是每个配置，接着是每个配置的接口，最后是每个接口的终端（Endpoint，也称为端点）。

更高层的描述符通知主机任何其它低层的描述符。每个设备有一个并且只有一个描述设备整体信息的设备描述符，用于指定设备支持的设备配置号。每个设备有一个或多个配置描述符，这些描述符包含了设备使用的电源和这个配置支持的接口号等信息。每个接口描述符有零个或多个终端描述符，这些描述符包括了与一个终端通信所需要的信息。

还有一种描述符类型，字符串描述符，可以保存诸如制造商或设备的名字的文本。

其它描述符可以保存指向这个字符串描述符的指针。主机可以使用 Get_Descriptor 请求来读取字符串描述符。

（2）设备枚举

一个完整的设备枚举过程包括以下步骤：

① Get Device Descriptor，主机命令要求得到设备描述符。

② Set Address ，设置设备的新地址。设置结束后，设备进入地址状态，主机以后会在新的指定地址处访问设备。

③ Get Device Descriptor，主机再次发送请求得到设备描述符的更多信息。

④ Get Configuration Descriptor，主机要求得到设备的配置描述符。

⑤ 读取全部 Configuration Descriptor 。接着主机要求得到设备全部的配置描述符、接口描述符和终端描述符。

⑥ 主机从描述符中知道了能够知道的信息后，分配和载入一个设备驱动。主机会再次要求设备重新发送设备描述符和配置描述符；接着主机发送设置设备配置 SETUP 包，设备处理此事件，将允许所有节点进入工作状态；最后主机请求得到设备和接口的配置，如果设备成功应答，枚举过程结束。

设备枚举是在 USB 中断服务例程中执行的。JB8 的 USB 模块有三种中断类型：

① 当每个端点接收或发送事务完成时，产生事务结束中断。

② 当挂起的 USB 总线被激活时，产生唤醒中断。

③ 当检测到包结束信号时，产生包结束中断。

所有的 USB 中断共享同一个中断向量，由中断程序根据中断标志位区分中断类型。

图 4-2 为 USB 中断例程的流程图。其中，包的概念指的是 USB 进行数据传输的基本单元，所有的数据都是经过打包后在总线上进行传输的。端点 0 的 IN，OUT 事务处理以及 SETUP 事务处理用于完成设备枚举过程。

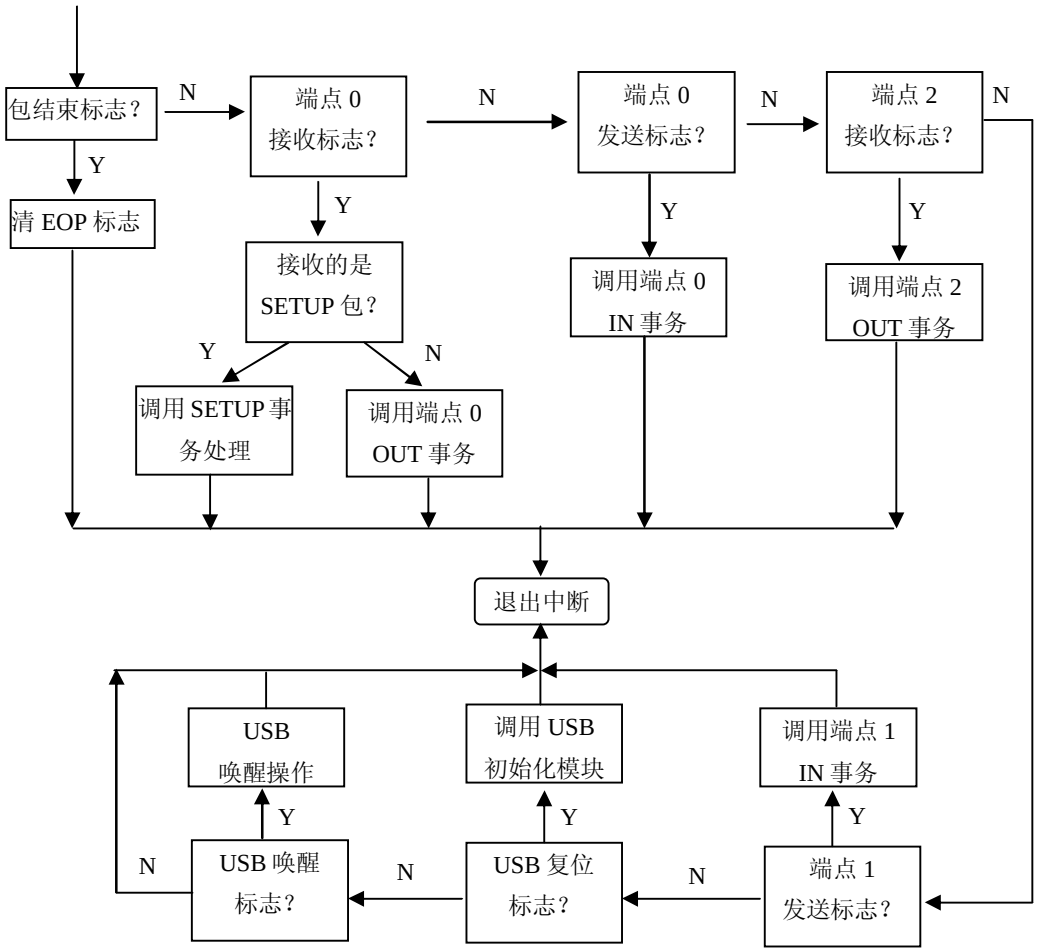

图 4-2 USB 中断处理流程图

下面代码给出了 JB8 的 USB 中断处理子程序。

```
*************************************************************
*功能:USB 中断子程序                          *
*入口:无                                *
*出口:无                                *
*************************************************************

USB_isr:
   SEI                          ;关闭中断
   PSHA
   CLR   SuspendCounter         ;USB 总线活动了

   BRCLR #RXD0F,UIR1,isr1       ;不是 EP0 接收中断，跳转
   ;******以下一段处理 EP0 接收******
   BCLR  #RX0E,UCR0             ;禁止 EP0 接收
   BRCLR #SETUP,USR0,EP0_OUT
   JSR   USBEP0_SETUP           ;SETUP 包处理
   BSET  #RXD0FR,UIR2           ;清 RXD0F
   BSET  #RX0E,UCR0             ;允许 EP0 接收
   BRA   isr_EXIT
```

```
EP0_OUT:                        ;EP0 的 OUT 包处理
   BCLR  #RX0E,UCR0
   BSET  #RXD0FR,UIR2           ;清 RXD0F
   BSET  #RX0E,UCR0             ;允许 EP0 接收
   BRA   isr_EXIT

isr1:
   BRCLR #TXD0F,UIR1,isr2       ;不是 EP0 发送中断，跳转
   ;******以下一段处理 EP0 发送******
   JSR   EP0_IN
   BSET  #TXD0FR,UIR2           ;清 TXD0F
   BSET  #TX0E,UCR0
   BRA   isr_EXIT

isr2:
   BRCLR #RSTF,UIR1, isr_EXIT ;不是 USB 复位中断，跳转
   ;******以下一段处理 USB 复位******
   BSET  #RSTFR,UIR2            ;清 RSTF
   JSR   initUSB
   BSET #ENABLE1,UCR3           ;允许端点 1
   BSET #ENABLE2,UCR3           ;允许端点 2
   BSET #TXD0IE,UIR0            ;允许端点 0 发送数据产生中断
   BSET #RXD0IE,UIR0            ;允许端点 0 接收数据产生中断
   BSET #RX0E,UCR0              ;允许端点 0 接收数据

isr_EXIT:
   PULA
   CLI
   RTI
```

4.2.3 USB 数据传输

USB 定义了四种数据传输方式，如表 4-1 所示，每种传输方式对应特定类型的端点。实际上，给定类型的端点总是使用对应类型的传输。

表 4-1 USB 数据传输方式

传输方式	功能	纠错	传输字节	延时保证
控制	用于发送和接收 USB 定义的结构化信息	是	<=8,16,32,64	尽最大能力保证不延时
批量	用于发送或接收小块无结构数据	是	<=8,16,32,64	无
中断	与批量方式相似，但包括一个最大延迟	是	<=64	以保证的最小速率轮询
同步	用于发送或接收有周期保证的大块无结构数据	否	<=1023	每 1 毫秒帧中的固定部分

模块使用端点 1 发送数据。发送数据时，先将数据放入端点 1 的输出缓冲区，置

位控制寄存器 UCR1 中的输出序列位 TES1Q、输出使能位 TX1E 和输出数据大小位 TP1SIZ0～TP1SIZ3。由于输出缓冲区的大小为 8 字节，所以通过对需要发送的数据量大小的判断调用一次或多次端点 1 发送子例程即端点 1 的 IN 事务，可将数据全部发送出去。

模块使用端点 2 接收数据。端点 2 接收中断标志位触发接收子例程的执行，在接收子例程中，先禁止端点 2 接收使能，在将接收数据从缓冲区中读取后，清 RXD2F 位，随后才放开接收使能位，从而确保数据接收的正确。

以下给出了 USB 数据接收的代码。

```
*****************************************************************
*功能:通过 USB 从 PC 接收 RxSize 个数据                    *
*入口:RxBuffer:接收数据缓冲区                          *
*出口:RxSize:接收的数据个数                            *
*****************************************************************
getUSB:
      PSHH
      PSHX
getUSB0:
      CLI
      BRCLR #B_RXD2F,UIR1,.          ;等待 PC 机发送来数据
      BCLR  #B_RX2E,UCR2             ;禁止端点 2 接收
      MOV   #$00,UIR2
      BSET  #B_RXD2FR,UIR2           ;清除接收中断标志

      ;首先检查接收到的数据个数是否正确，不对则重新等待
      LDA USR1
      AND #$0F
      STA RxSize

      ;将 UE2D0-UE2D7 中的 8 个数据存入 RxBuffer
      MOV   UE2D0,*RxBuffer
      MOV   UE2D1,*RxBuffer+1
      MOV   UE2D2,*RxBuffer+2
      MOV   UE2D3,*RxBuffer+3
      MOV   UE2D4,*RxBuffer+4
      MOV   UE2D5,*RxBuffer+5
      MOV   UE2D6,*RxBuffer+6
      MOV   UE2D7,*RxBuffer+7

      BSET  #B_RX2E,UCR2             ;允许端点 2 接收

      PULX
      PULH
      RTS
```

4.3 BDM 功能模块设计

HCS08/RS08 支持后台调试模式 BDM，内部集成的 BDC（Background Debug Controller）模块负责 BDM 指令的解码以及硬指令的执行。HCS08（除 Q 系列外）还集成有 DBG（On-chip debug module）模块，提供了更强大的调试功能，DBG 模块的访问及操作一般通过 BDM 模式来完成。通过 BDC 和 DBG，可以实现应用程序的擦除与下载、实现应用程序的动态调试以及配置与修改 MCU 内部资源，对应用程序进行加密处理等。

4.3.1 目标机频率测试

由于 BDM 使用单线传输方式，这就要求通信双方必须使用相同的时钟频率。所以，首先需要确定目标机的 BDM 通信频率才能进行正常的通信。因此，软件设计实现的第一步就是测量目标机的频率。由于主控芯片通过发送足够长度时间的低电平来表征同步请求 SYNC 指令，所以需要事先预估目标机的频率范围。通过对 HCS08 系列 MCU 的工作频率进行分析，将目标机可能的最低工作频率设为 400KHz 效果较为理想。主控芯片执行以下步骤：

① 计算目标机 BDC 的最低工作频率，将目标机 BKGD 引脚拉低至少 128 个目标机的 BDC 时钟周期。

② 发送一个瞬时高电平信号。

③ 撤销对目标机 BKGD 引脚的驱动使其进入高阻态。

④ 监听目标机发送到 BKGD 引脚上的 SYNC 应答信号。

目标机在探测到主控芯片发送的 SYNC 请求后，执行以下步骤。

① 等待 BKGD 引脚上的信号变高。

② 检测到信号变高以后目标机延时 16 个 BDC 周期等待调试器发送瞬时高电平信号结束。

③ 将 BKGD 引脚拉低 128 个 BDC 时钟周期。

④ 发送一个瞬时高电平信号，然后撤销对 BKGD 引脚的驱动使其进入高阻态。

写入调试器使用的芯片为 JB8，主频 3MHz。JB8 通过记录目标机发送 128 个 BDC 周期的低电平信号所对应的定时器计数值，从而计算出目标机的 BDC 工作频率。JB8 的定时器使用总线频率进行计数。例如，目标机发送 128 个 BDC 周期的低电平，JB8

的计数器计数 96 次，则可根据公式“128*JB8 总线频率/计数值”得出目标机的 BDC 工作频率为 128*3/96＝4MHz。

测出目标机 BDC 工作频率后，查找相应的跳转表获取对应的通信时序例程，就可以与目标机进行正确的通信了。由于这一功能的汇编实现代码较长，这里略去。

4.3.2 收发 1 字节的 BDM 实现

BDM 指令的底层操作是读、写一个数据位（bit）的操作。因此，在 BDM 通信中，最重要的事情就是可靠的读/写数据位的操作实现。根据 BDM 收发 1 位数据的时序图，可以较为方便地编写出以固定频率通信的的收发 1 字节的子程序。但是，为了适应 HCS08 系列产品的要求，需要能够满足不同频率下的稳定的数据收发。在对时序图以及汇编语句格式仔细研究后，根据通信时的时序周期和汇编语句所需要的周期数的关系，分段编写实现了在不同频率段下收发数据的子程序。

以下是主控芯片发送 1 字节的汇编代码，适应的目标机通信频率为 3.667 MHz-4.667 MHz。

```
**************************************************************************
*功能:向目标机发送 1 字节数据，目标机的适应频率为 3.667 MHz- 4.667 MHz*
*入口:累加器 A 中的数据为要发送的数据                                    *
*出口：无                                                                *
**************************************************************************
BDM_TX5:
        AIS   #-1                   ;开辟临时变量
        LDX   #8                    ;1 字节有 8 位
        STX   1,sp

        LDHX  #BDM_PORT             ;HX 指向 BDM 通信端口（PTA）
        MOV   #0,BDM_DRV2_PORT      ;[4]驱动 BDM 通信端口
BDMTX5:
        ; bit 7 (MSB)
        CLR   ,X                    ;[2]驱动 BDM 为低
        STA   ,X                    ;[2]A 中的第 7 位发送到 BDM
        ORA   #BDM_OUT_MASK         ;[2] 置累加器 A 的最高位
        CPHX  #0                    ;[3] 该指令用于延时 3 个总线周期
        STA   ,X                    ;[2]置 BDM 为高
        LSLA                        ;[1]下一位
        DBNZ  1,sp,BDMTX5           ;[6]

        MOV   #BDM_DRV2_MASK,BDM_DRV2_PORT     ;停止驱动 BDM

        AIS   #1
        RTS
```

主控芯片 JB8 发送一位，使用 4 条语句，需要 9 个周期，如下所示：

```
STA   ,X                ;[2]A 中的第 7 位发送到 BDM
ORA   #BDM_OUT_MASK     ;[2] set MSB of A
CPHX  #0                ;[3] wait 3 cycles
STA   ,X                ;[2]置 BDM 为高
```

目标机在探测到通信起始位后第 10-13 个周期（这里是目标机的周期,即 BDM 周期）取值，由于同步因素，目标机可能会延迟 1 个周期才探测到通信起始位。调试器主控芯片 JB8 总线频率为 3MHz。所以，主控芯片运行 9 个周期，而目标机的 BDC 模块则可能运行 11-14 个周期。从而求出目标机的通信频率为 3*（11～14）/9=3.667～4.667MHz。可据此编写出适合其余通信频率的代码。

主控芯片读取目标机 BKGD 端的信息时，先拉低目标机 BKGD 端 2-4 个周期后再释放对其的驱动，然后在传输起始点后的第 10 个周期时采样目标机 BKGD 引脚，读取目标机发送的数据位。

以下是主控芯片从目标机读取 1 字节数据的汇编代码，适应的目标机通信频率为 3.38 MHz - 4.87 MHz，可据此编写出适合其余通信频率的代码。

```
**************************************************************************
*功能:从目标机接收一字节数据                                             *
*入口:无                                                                 *
*出口:A 中为接收的数据                                                   *
**************************************************************************
    BDM_RX5:
        AIS   #-2               ;开辟临时变量
        LDA   #8                ;循环次数
        STA   1,SP
        BSET  #RESET_OUT,RESET_PORT     ;[4]
        LDHX  #BDM_PORT                 ;HX 指向 PTA
        LDA   #BDM_DRV1_MASK+RESET_OUT_MASK

    BDMRX5:
        STX   ,X                ;[2]驱动 BDM 为低
        STA   ,X                ;[2]让 BDM 为高阻态
        CPHX  #0                ;[3]延时 3 个周期
        NOP                     ;[1]

        LDX   ,X                ;[2]取值 ( 第 6 位为 BDM_IN )
        ASLX
        ASLX                    ;此时接收的位值—>C
        ROL   2,SP              ;C—>SP[2]的 bit0
        CLRX                    ;清 X
        DBNZ  1,SP,BDMRX5

        LDA   2,SP
```

```
        AIS   #2                ;释放临时变量
        RTS
```

4.3.3 目标机复位

HCS08/RS08 系列 MCU 提供了一个新的特性。主机可以通过 BDM 的方式设置目标机的 SBDFR 寄存器（地址一般为$1801）中的 BDFR 位使目标机复位。该寄存器只能由 BDC（Background Debug Controller）模块访问，用户程序不能访问，从而防止用户程序错误地复位芯片。此外，还可以通过拉低目标机复位引脚足够的时间长度（超过 512 个总线时钟周期）令目标机复位。子程序 BDM_Reset 用于实现该功能。在 MCU 复位信号的上升沿，如果 BKGD 引脚为低电平，则目标机复位后进入 BDM 模式。

BDC 的 BDCSCR 寄存器中的 CLKSW 位用于选择和表明 BDC 的时钟源。BDC 可以选择总线时钟或备选 BDC 时钟（Alternate BDC Clock Source ）作为时钟源。目标机复位直接进入 BDM 模式时 BDC 使用总线频率作为工作频率；若目标机程序运行后进入 BDM 模式，则 BDC 使用备选 BDC 时钟频率作为工作频率。此时可通过设置 CLKSW 位选择总线频率为 BDC 的工作频率。在进行程序调试时一般选用备选 BDC 时钟作为 BDC 的工作时钟，因为总线频率可能会由于应用程序的设置更改而变化。通过设置 BDCSCR 寄存器的 CLKSW 位选择 BDC 的时钟源后，需要进行连接测试测出当前的目标机的 BDC 工作频率以便选择合适的收发例程进行通信。目标机在每一次复位后也都需要进行连接测试以测出目标机 BDC 的当前工作频率。

4.3.4 收发例程选择

测出目标机的 BDC 当前工作频率后，需要选择对应目标机 BDC 当前工作频率的收发例程进行通信。通过相关的跳转表查找对应的收发例程入口地址，选择合适的收发例程。因为是根据目标机 BDC 的当前工作频率选择合适的收发例程，所以在每一次目标机复位或重新选择 BDC 工作频率后都需要进行连接测试以便选择合适的收发例程。发送例程的入口地址保存在变量 bdm_tx 中，接收例程的入口地址保存在变量 bdm_rx 中。

4.3.5 BDM 指令实现

BDM 指令的基本单位是字节。实现了收发 1 字节数据和收发例程选择后，紧接着就可以实现各 BDM 指令。以下给出读取 BDCSCR 寄存器值的 BDM 指令的实现代码作为示例。该指令的编码结构为：E4/SS。其中，E4 为指令编码，SS 表示目标机返回的内容。

```
************************************************************
*功能:读取目标机 BDCSCR 寄存器的值                    *
*入口:无                                  *
*出口:A=目标机 BDCSCR 寄存器的值                      *
************************************************************
BDM_READ_STATUS:            ;读取目标机 BDCSCR 寄存器的值
    JSR   bdm_tx_prepare    ;准备发送

    LDA   #$E4
    LDHX  bdm_tx            ;取得发送例程入口地址
    JSR   ,X                ;发送命令 READ_STATUS

    JSR   bdm_cmd_wait16    ;延时 16 个目标机 BDC 时钟周期
    JSR   bdm_tx_finish     ;发送结束

    CLRA
    LDHX  bdm_rx            ;取接收例程入口地址
    JSR   ,X                ;接收
    RTS
```

4.3.6 目标机擦除及写入

在实现了 BDM 指令后，就可以在此基础上实现各种应用了。编程调试器在对目标机的 FLASH 进行擦写操作时，一般将目标机的 BDC 时钟设置为总线时钟，这样可以与应用程序操作 FLASH 的相关设置保持一致，同时可以实现以最快的速度对 FLASH 进行擦除与写入。要实现对目标机的擦除写入功能，高端程序需要发送一些相关的参数给 JB8。这些参数包括目标机的内存起始地址 RamStart（2 字节），目标机 FLASH 时钟分频寄存器的值（Fcdiv），目标机内存数据接收缓冲区地址高低位（DataAddrH，DataAddrL）。

子程序 MassEraseFlash 对目标机的 FLASH 进行整体擦除。首先对目标机的 FPROT，FCDIV 寄存器进行初始化，允许进行整体擦除，设置目标机的 FLASH 工作频率。随后发送整体擦除指令对目标机 FLASH 进行整体擦除。这里需要详细查阅芯片手册，了解在不同的 FLASH 工作频率下对应的擦写时钟脉冲以及完成操作

所需的周期数，从而计算出完成操作所需的时间。在整体擦除完成后，还需要进行空校验，以确保整体擦除操作的成功。在进行空校验操作时，命令的执行时间需要适当放宽。芯片手册没有说明空校验操作的执行时间，但是在实际过程中，如果空校验命令的执行延时不够，会导致测试 FBLANK 位时出错。

空校验成功后，需要立即改写 NVOPT 寄存器（地址$FFBF）的值，以免目标机下次复位后进入加密状态。

目标机代码下载写入功能实现原理为：先编写一段负责对 FLASH 进行写入的代码，将其编译成 S19 代码后，将该段 S19 代码通过 BDM 指令写入目标机的内存，驻留于固定地址。在目标机内存中开辟一数据缓冲区，专门用于接收需要写入 FLASH 的数据。将数据（一般每次为 128 字节）写入数据缓冲区后，发送 BDM 指令执行写入子程序，将数据写入目标机的 FLASH。子程序 Ram2Flash 的作用是将高端发送过来的写入子程序代码写入目标机的内存指定位置。高端发送的字节序列的第 1，2 个字节为写入子程序在目标机内存中的驻留起始地址，第 3 个字节为写入子程序 S19 代码的字节个数，第 4 个及其以后的字节为写入子程序 S19 代码。子程序 RecWrite1Page 的作用是从高端接收一页数据写入目标机内存，然后执行目标机内存中原先驻留的写入子程序，将这一页数据写入 FLASH。在接收一页数据时，首先接收相关的参数信息，即写入 FLASH 的起始地址和一页数据所包含的数据个数。在接收数据完成后，随即将数据写入目标机内存中的接收数据缓冲区，发送命令使目标机内存中驻留的写入子程序运行将数据缓冲区中的数据写入 FLASH。写入完成后，写入子程序还需要计算写入数据的校验和，将校验和和执行结果状态写入（目标机的）HX 寄存器中。编程调试器可以通过读取目标机的 HX 寄存器值以获知执行写入的结果。由于目标机中驻留的写入子程序执行时目标机脱离了 BDM 模式而进入了用户模式，所以 JB8 需要先令目标机重新进入 BDM 模式才能正确读取目标机的 HX 寄存器值，否则读出结果全为 0。

对于中断向量的写入也采取这种方式进行。

4.3.7 调试

调试功能只能在目标机处于 FLASH 区解密的状态下使用。如果目标机处于加密状态，则不能读取任何信息。BDC 模块中有一个断点寄存器 BDCBKPT 用于设

置断点地址。BDCBKPT 寄存器由专门的 BDM 指令进行设置和读取。当程序运行到断点地址时，目标机从用户模式进入 BDM 模式，此时可以通过 BDM 指令读写目标机的 RAM 和 CPU 相关寄存器的内容，以及读取 FLASH 区内容。通过对累加器 A，程序计数器 PC，堆栈指针 SP 的修改，用户可以动态地干预程序的运行。

HCS08 MCU 除包含 BDC 模块外，还包含一个 DBG 模块。DBG 模块提供了更为复杂和高级的调试功能。DBG 模块的访问及操作一般通过 BDM 模式来完成，也可以在用户模式下访问该模块。DBG 模块支持多达 9 种不同类型的触发条件来满足不同的调试需求，其所包含的比较器 A，B 还可作为断点寄存器使用，为程序的多断点调试提供了实现的手段。

图 4-3 给出了调试操作的流程图。BDC 模块和 DBG 模块可以单独使用，也可以配合使用。调试程序常常离不开断点的设置。断点类型有两种：TAG 类型断点和 FORCE 类型断点。FORCE 类型的断点迫使 CPU 在访问断点地址时立即进入 BDM 模式，而 TAG 类型断点使得断点地址处的指令代码在进入执行队列时被标记，当该指令被执行到时，CPU 以 BGND 指令替代从而进入 BDM 模式。

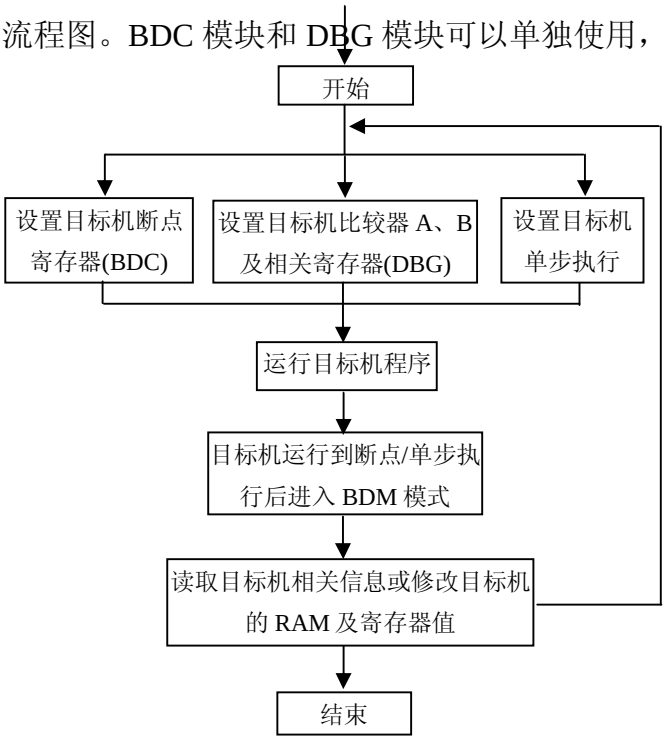

图 4-3 MCU 方调试程序流程图

通常在 BDM 模式下设置断点寄存器 BDCBKPT。设置 BDC 状态和控制寄存器 BDCSCR 中的断点使能位 BKPTEN 后，就可以使用断点功能进行断点调试了。目标机执行到断点处，进入 BDM 模式，此时程序暂停执行。通过 BDM 指令读取目标机 RAM、FLASH 区域及各寄存器的值，可以确切了解程序运行的情况。

通常情况下，一个硬件断点无法满足程序调试的需要。而使用 DBG 模块可以

扩展出两个额外的硬件断点，即利用比较器 A、B 作为地址比较器，可以获得更好的调试效果。

DBG 模块使用一个容量为 8 字节的 FIFO（First In，First Out），用于存储地址和数据总线信息；而比较器 A、B 用于控制何时收集总线信息。比较器 A、B 定义了触发条件，这是 DBG 模块设置的关键；DBG 控制寄存器 DBGC 和 DBG 触发寄存器 DBGT 用于定义相应触发模式的选择。比较器 A 只能用于指定 16 位的地址信息，而比较器 B 既可以用于指定 16 位的地址信息，也可用于比较 8 位的数据信息。HCS08 CPU 的数据总线被分为读总线和写总线，因此还可以在控制寄存器中指定在读操作还是写操作时产生触发。

可以通过设置 A OR B、TAG 类型方式来匹配两个硬件地址，以达到设置硬件断点的目的。比较器 A 和 B 用于匹配地址信息的捕捉。当程序指针 PC 运行到 A 或 B 指定的地址时，将引发一个断点操作。

在调试程序时，还可以利用 DBG 模块提供的功能来了解感兴趣的地址处数据的读/写情况。通过使用 FIFO 既可以捕获程序分支信息，也可以捕获指定地址的数据变化。使用 FIFO 的一般步骤为：先设置好触发模式及其它的控制位，然后启动调试器。当 FIFO 被填满后，可以按照先入先出的顺序来读取 FIFO 中保存的调试信息。在 FIFO 未填满时，程序通过向 DBGC 寄存器中的 ARM 位写入 0 的方法强行终止当前调试程序的运行过程。强行终止时会向 FIFO 中写入一个无效数据。FIFO 中的有效数据量可以从 FIFO 的状态寄存器中获得。有两种类型的调试模式：一种称为始调试模式（Begin Trace），另一种称为末调试模式（End Trace）。始调试模式在触发条件满足时开始向 FIFO 中填写数据。当 FIFO 填满时，向 CPU 产生一个断点请求，CPU 进入 BDM 模式。末调试模式则一直向 FIFO 中不断刷新相关的数据，在触发条件满足时，停止刷新 FIFO 中的数据，同时向 CPU 产生一个断点请求，CPU 进入 BDM 模式。一般而言，始调试模式关注的是在特定条件满足后 CPU 的后继动作，而末调试模式关注的是在特定条件到达之前 CPU 的动作。此外，还可以通过 FIFO 来获知某一地址被写入一个特定值后程序的运行情况以及生成 CPU 指令执行地址的记录信息。

在 BDM 指令集中，有一条用于单步执行的指令 TRACE1。该指令需要在 BDM 模式下执行。TRACE1 指令将 CPU 从 BDM 模式切换到用户模式后，执行由程序

计数器 PC 指定地址处的一条用户程序指令后再返回 BDM 模式。通过 TRACE1 指令，可以一步步跟踪程序的执行情况。

4.4 小结

MCU 方固件程序可以分为两个主要部分：一是 USB 通信实现，二是 BDM 通信实现。USB 通信部分主要实现 USB 中断处理子程序和标准设备请求相关子程序以及数据收发子程序。标准设备请求要重点实现 GetDescriptor 请求、ClearFeature 请求、SetAddress 请求和 SetConfiguration 请求。BDM 通信部分实现重点一是目标芯片频率测量，二是收发数据的时序要求的精确度。

第五章 PC 方软件设计

PC 方软件提供了高端用户界面，与 MCU 方程序配合运行。PC 方软件与 MCU 方软件同步协同开发。上一章介绍了有关 MCU 方程序设计的相关问题，本章着重讨论 PC 方软件的设计。

5.1 总体设计

PC 方程序主要提供用户界面，该界面是一个多窗口的应用程序，包含菜单、状态栏、文件管理窗口、信息输出窗口等人机接口。

PC 方程序按功能主要分为四个部分：

① 工程管理，负责工程文件管理。

② 信息输出，用于工程活动中各种信息的输出显示。

③ 程序下载，完成对目标机的芯片擦除与代码写入。

④ 程序调试，实现目标机的代码调试功能。

程序代码实现主要有五个大类，简介如下：

（1） 控件类

表 5-1 给出了控件类的功能和隶属功能模块。

表 5-1 控件类

类名	功能	隶属模块
CLeftView	视图管理器 Tab 控件	工程管理
CListEditView	输出信息控件	信息输出
CHexEdit	16 进制编辑器控件	
CSizingControlBar	实现与 MSDEV 界面类似的 WorkSpace 和 OutputBar 浮动窗口类	
CMaskEdit	文本编辑类	

（2） 对话框类

对话框类负责对话框的实现与和消息处理，如表 5-2 所示。

表 5-2 对话框类

类名	功能	隶属模块
CProgressDlg	进度条对话框，用于下载等模块	下载
CMcuSelectDlg	芯片选择对话框	写入调试
CConBar	信息显示对话框	信息输出
CMemBar	内存显示对话框	
CRegBar	寄存器显示对话框	
CWatchBar	变量显示对话框	

(3) 模块类

模块类主要包含工程管理、下载、调试模块的一些算法，如表 5-3 所示。

表 5-3 模块类

类名	功能	隶属模块
CProject	工程管理类	工程管理
CUSBInstance	写入调试类	写入调试

(4) 主框架、视图、文档类

主框架、视图、文档类不属于任何模块，是Visual C++的窗口应用程序必需的类，如表 5-4 所示。

表 5-4 主框架、视图、文档类

类名	功能	隶属模块
CMainFrame	主框架类，MDI 程序必需	其它
CChildFrame	子框架类，MDI 程序必需	
CSdIDEDoc	文档类，MDI 程序必需	
CSdIDEView	视图类，MDI 程序必需	
CSdIDEApp	实例类，Windows 程序必需	

(5) 全局函数

全局函数主要是与其它模块无耦合且调用频率非常高的子函数，由于数量比较繁多，限于篇幅本文不一一列出。

5.2 目标芯片库

HCS08 系列和 RS08 系列 MCU 各具有多种型号，而且未来会推出更多的新型号。不同型号的 MCU 其属性参数有所不同。这些属性参数包括内部 RAM 及 FLASH 的大小和起始地址，及一些相关寄存器地址。这些参数在对目标芯片进行擦除写入及调

试操作时极为重要。因此，可以将各型芯片的相关属性参数存入数据库表中。本设计建立了一张库表 ParaTable，将 HCS08 和 RS08 各型芯片的相关属性参数存储其中，如表 5-5 所示。

表 5-5 ParaTable 表结构

字段名称	字段类型	字段大小	对应的目标 MCU 参数
McuType	文本	25	目标 MCU 类型
McuFlaStartAddr	文本	4	目标 MCU 的 FLASH 区起始地址
McuFlaEndAddr	文本	4	目标 MCU 的 FLASH 区结束地址
McuRAMStartAddr	文本	4	目标 MCU 的 RAM 区起始地址
McuWritePage	备注	—	页写入程序代码
vectorsize	文本	4	向量区大小
vectoradr	文本	4	向量区首地址
Pagesize	文本	4	目标 MCU 的页大小

5.3 擦除、写入功能的实现

PC 方的擦除、写入操作主要是配合 MCU 方程序，来达到对目标 MCU 的芯片擦除及代码写入。第四章介绍了 MCU 方擦除及写入功能的实现，以下具体介绍 PC 方程序的擦除写入过程。

对目标 MCU 的 Flash 区的整体擦除分 4 步进行：

（1）使目标 MCU 复位进入 BDM 模式

PC 方发送目标 MCU 复位进入 BDM 模式指令（#216），令目标机进入 BDM 模式。

（2）对目标 MCU 进行频率测试

PC 方发送频率测试指令（#204），编程调试器 MCU 方程序接收该命令后，与目标 MCU 进行测试，并返回 3 个字节的测试结果（R1、R2、R3）。PC 方根据 R1 来判断测试是否成功（0 表示成功，否则失败）。若成功，R2R3 的值则表示目标机频率的一个测试值，进行换算后可得到目标机的总线频率；若不成功，R1 的值给出出错的原因，此时 R2R3 的值无意义。

（3）发送参数

PC 方发送传参指令（#206），然后发送 5 个参数给低端。这 5 个参数分别为：目标机内存起始地址高位和低位、目标机的 FLASH 时钟分频寄存器的值、目标机接收缓冲区首址高位和低位。这些参数主要为随后的擦除写入操作服务。

（4）整体擦除

PC 方发送整体擦除命令（#208），对芯片进行整体擦除操作。

对目标 MCU 的 Flash 区的写入分以下几步进行：

（1）发送写入子程序

PC 方发送写入子程序命令（#210），通知主控 MCU 接收写入代码。接下来 PC 方就将目标 MCU 的写入代码发送给 MCU 方程序。编程调试器 MCU 方程序将该段写入代码通过 BDM 指令写入目标机的 RAM 中。不同型号的 MCU 有不同的写入代码，其编译后的 S19 代码存于 ParaTable 表中。

（2）发送数据

PC 方发送一页数据写入指令（#212），紧接着发送一个 3 字节的信包，其内容为：写入数据的 FLASH 地址高位和地址低位及写入数据的个数。随后发送具体的一页数据内容。编程调试器 MCU 方程序在接收到这些信息后，将这些信息写入目标机，随后发送程序运行指令，令驻留在目标机 RAM 中的写入代码执行，将数据写入目标机的 FLASH 中。

5.4 目标代码分析

编程调试器和目标芯片的 RAM 空间有限，不可能一次接收完所有的数据。因此需要在下载之前将整个目标代码文件进行分析，以一定的数据结构组织起来。经过多方面的考量，采用单链表结构存储目标代码文件。分析过程如下：

① 将目标代码文件过滤一遍，去掉 S0、S9 记录以及空行，只留下 S1 记录。

② 将所有 S1 记录根据包含的地址信息从低到高进行排序，这一步是至关重要的，因为经过编译得到的目标代码文件很有可能是无序的，但为了保证分页后，页间地址和页内地址都是连续的，排序是一定要做的。

③ 当前面两步准备工作做好之后，才能开始进行分页的操作。

链表的结点类型如下：

```
typedef struct USBS19 {
   BYTE *PageData;           //页数据的指针
   unsigned int PageAddr;       //页的起始地址
   struct USBS19 * next;        //下一页的指针
}USBS19;
```

由于目标代码写入的起始地址并不一定是页对齐地址，因此构建链表时应该分成两步：

① 构建链表的第一个结点，将经过排序的第一行 S1 记录包含的地址作为该结点的页起始地址（不是页对齐地址）；该页数据长度必须经过计算得到，而不是芯片手册上规定的页大小。

② 构建链表的其他结点，将页对齐地址作为结点的页起始地址；页数据的长度就是芯片手册上规定的页大小（通常为 128 字节）。

由于编译出来的 S19 文件没有特定的规律，一页目标代码有可能是完整的一行，也可以是行的一部分，还有可能分布在多行。因此在构建分页链表时必须充分考虑到这几种情况。

5.5 调试功能实现

调试功能实现对汇编和 C 语言的源码级调试。两者实现过程大体相似，但 C 语言调试的实现相对复杂。

汇编程序和 C 程序在编译后都会产生调试信息文件（后缀名为.dbg）。dbg 文件包含有相应的信息以供调试时使用。

5.5.1 dbg 文件结构

图 5-1 为一个 dbg 文件的结构，接下来以该文件为例，来介绍 dbg 文件结构。一个调试文件包含了整个 C 工程的调试信息，它包含很多个段，每一段包含一个源文件的调试信息。调试文件的格式是 ASCII，文件中的每一行都是一个独立的命令，图 5-1 中的阴影部分都是命令字，这些命令字就是分析调试文件提取调试信息的关键字，下面就介绍几个主要的命令字的含义。

（1）VERSION<#.#>

表示调试文件的版本号，本例中为 1.1。

（2）CPU<name>

用来表明目标 CPU，本例为 HCS08。

（3）DIR<name>

表明下面所有的 FILE 命令的直接路径，也就是整个工程的目录，<name> 表示路径名，“DIR E:\example\ ”表示本例的 C 工程的目录是“E:\example\”。

（4）FILE<name>

表示一个文件段的开始， <name>表示文件名。该命令下的所有命令都应用于该文件中，直到文件结束或下一个 FILE 命令开始。该文件完整的路径是上面 DIR 路径和这里的文件名的结合。

（5）FUNC<name><addr><type>

表示一个 C 函数段的开始。该命令下的所有的命令都应用于该函数中，直到 FUNCEND。<name>表示函数名，<addr>表示该函数中第一个语句对应目标 MCU 中的存储区的开始地址，以十六进制表示。本例中“FUNC main 8035 fV”表示“main”函数的开始，该函数对应目标 MCU 中的存储区的开始地址为$8035。

```
VERSION 1.1
CPU HCS08
DIR E:\example\
FILE main.c
FUNC main 8035 fV
BLOCK 16 8038
DEFLOCAL I 0 A[12:3:2]I
LINE 16 8038
LINE 18 8038

LINE 33 8096

LINE 48 8117
LINE 26 8117
BLOCKEND 0 811C
FUNCEND 811C
FILE setup.c
FUNC _HC08Setup 811D fV
BLOCK 10 811D
LINE 10 811D

LINE 37 813F
BLOCKEND 0 8141
FUNCEND 8141
FILE delay.C
FUNC Delay1000 8142 fV
BLOCK 11 8145
DEFLOCAL u 0 I
LINE 11 8145

LINE 13 8165
BLOCKEND 0 816D
FUNCEND 816D
FILE main.c
DEFGLOBAL x 42 D
DEFGLOBAL p 46 pI
DEFGLOBAL I1 48 I
START 8000
```

图 5-1 dbg 文件结构

（6）FUNCEND<addr>

表示一个函数段的结束。<addr>表示该函数返回指令对应目标 MCU 中的存储区的地址，以十六进制表示。

（7）DEFGLOBALE<name><addr><type>

定义全局变量。<name>表示变量名，<addr>表示该变量对应目标 MCU 的存储区的首地址，以十六进制表示，<type>表示变量类型，关于变量类型将在第 5.4.6 节介绍。本例中的“DEFGLOBAL x 42 D”表示实型全局变量 x，在目标 MCU 中 RAM 区的地址为$42。

（8）LINE<line no><addr>

表示源文件行号和该行代码对应目标 MCU 存储区的首地址。<line no>表示行号，是十进制表示，<addr>表示地址，是十六进制表示。

（9）DEFLOCAL<name><offset><type>

表示一个函数范围内的局部变量。<name>表示局部变量名，<offset>表示与 H:X 寄存器的偏移量，它是用十进制表示的，<type>表示该变量类型。

5.5.2 调试方式

HCS08 具有强大的调试功能，除断点调试和单步调试外，内部专有的调试模块提供更为复杂的调试实现，提供两个比较器和多达九种的触发模式以及数据读/写匹配选择。PC 方程序配合低端 MCU 方程序实现调试功能。在具体设计时，将全部调试功能一起通过菜单选择方式实现，则存在两个问题：一是用户界面会显得凌乱，不利于使用；二是代码实现会较为繁杂，效率不高。通过分析全部的调试功能，根据实际开发应用中的情况，可以将 HCS08 所具有的调试功能分为常用和不常用两部分。其中，单步调试、断点调试、监视点调试这些功能是较为常用的功能。在调试界面中，这些常用功能可从右击弹出的菜单中直接进行选择，很方便地供用户使用。除此之外，在菜单中还提供有称为“专家模式”的选项，在该选项中，用户可以对目标机芯片内部相关的调试寄存器直接进行赋值操作，以实现任何一种调试功能。“专家模式”要求用户对目标芯片结构和编程方式有一定的了解，在操作上不够直观，但是在功能实现和界面易用结合方面达到了有效的平衡。

（1）断点设置

从当前打开的文件中选择需要设置断点的行，根据相应的.dbg 文件找到对应的地址值即该行对应的断点地址，同时红色高亮显示当前选择的代码行。

图 5-2 为设置断点的程序流程图， 将当前工程的调试文件打开，从该文件的第一行开始分析，直到在和当前设置断点所在文件的文件名相同的“FILE”行下找到这样的“LINE”行，该行的<line no> 等于当前设置断点的行号，然后提取该“LINE”行的<addr>，将它转化为十六进制后即为当前的断点地址。若找不到，则表示选择的的不是代码行。

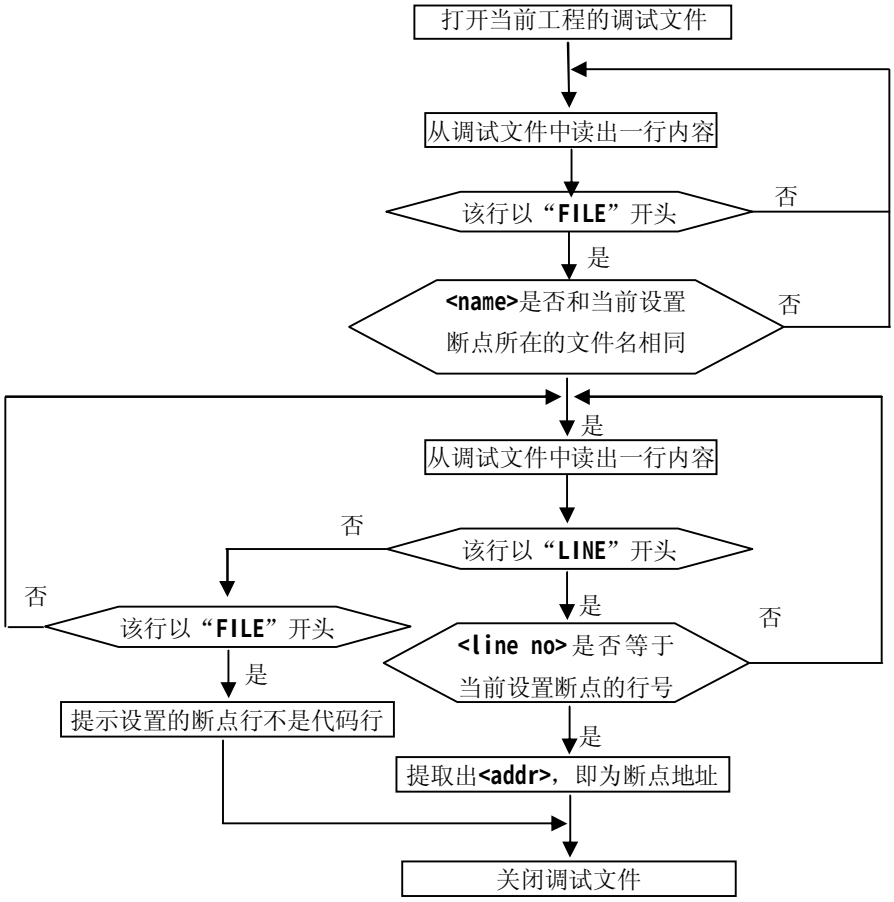

图 5-2 设置断点流程图

（2）单步调试

HCS08/RS08 提供了一条 BDM 指令 TRACE1，由 BDM 模式进入用户模式后执行由程序计数器 PC 指定地址处的一条用户程序指令后再返回 BDM 模式。对于汇编程序，一行代码就是一条指令，单步调试易于实现；而对于 C 语言程序，一行代码编译后往往对应多条汇编指令，在单步调试时需要确定该行 C 代码所对应的汇编指令条数。

为了实现单步调试，需要对程序编译后的的.dbg 文件进行分析。.dbg 文件包含了各文件编译后的相关信息，各文件信息按编译顺序分段排序。在文件段的内部，以块的方式组织文件所包含的函数信息。为此，设计了三个数据结构分别用于处理文件段信息、函数块信息以及各代码行信息。下面给出了三个数据结构的具体信息：

```
//代码行处理结构
typedef struct LineInfo{
int linenum;              //      行号
int lineaddr;             //      程序行所对应的地址
BOOL LiveFlag;            //      标记该节点在应用规则二时是否应该被删除
struct LineInfo *LineNext;
}LineInfo,*pLineInfo;

//函数块处理结构
typedef struct BlkInfo{
int BlkMaxAddr;           //Debug    文件中某块的最大地址
int BlkMinAddr;           //Debug    文件中某块的最小地址
CString FuncName;
struct BlkInfo *BlkNext;
pLineInfo LineNext;
}BlkInfo;

//文件段处理结构
typedef struct{
CString FileName;         //   文件名
int  FileMaxAddr;         //   文件对应的最大 PC 地址
int  FileMinAddr;         //   文件对应的最小 PC 地址
struct BlkInfo *BlkNext;  //   指向文件所包含的块信息
}FileInfo;
```

除 dbg 文件以外，程序编译后产生的 list 文件也是进行调试操作时必要的参考文件。list 文件详细地给出了指令编码格式与地址和代码行之间的对应关系。因此，专门建立了一个数据结构用于处理 list 文件信息，该数据结构与 list 文件的结构一起在下面给出。

```
(0041)
    8043 3F 00        clr 0x00
(0042)          Light_P=0x00;    /*指示灯 灭 */
(0043)          Light_P=0xFF;    /*指示灯 亮 */
    8045 A6 FF        ldA #-1
    8047 B7 00        stA 0x00
```

```
typedef struct ListInfo{
 int LineAddr;
 CString Code;
   struct ListInfo *next;
}ListInfo,*pListInfo;
```

单步调试中，又分为单步步入与单步步过两种方式。以下分别介绍这两种方式的实现。在目标机已经进入调试状态后，以单步步入的方式执行一条 C 语句的流程如下：

① 如果该语句所对应的.dbg 文件信息行不是块的最后一行并且通过.lst 文件信息行知道该条 C 语句不是子函数调用语句，转②；否则(即该.dbg 信息行是块的最后一行或者是子函数调用语句)转③。

② 从.lst 文件中将 C 语句编译后所对应的最后一条汇编指令的地址设置为断点，发送 GO 命令使目标机从运行到断点处，并且获取目标机各寄存器的值。

③ 发送 TRACE1 命令使得目标机执行一条汇编指令并且获取目标机程序计数器 PC 的值。

④ 以获取的 PC 值（即地址值）从.dbg 文件中查找判断文件中是否含有该地址值；如果没有找到，则表示 C 语句对应的汇编指令码还没执行完成，转③；否则转⑤。

⑤ 到此认为单步步入执行一条 C 语句完成。

以单步步入的方式执行一条 C 语句的流程如下：

① 如果该语句所对应的 dbg 文件信息行不是块的最后一行，转②；否则(即该语句所对应的 dbg 文件信息行是块的最后一行)转③。

② 从 list 文件中将 C 语句编译后所对应的最后一条汇编指令的地址设置为断点，发送 GO 命令使目标机运行到断点处，并且获取目标机各寄存器的值。

③ 发送 TRACE1 命令使得目标机执行一条汇编指令并且获取目标机程序计数器 PC 的值。

④ 以获取的 PC 值（即地址值）从 dbg 文件中查找判断文件中是否含有该地址值；如果没有找到，则表示 C 语句对应的汇编指令码还没执行完成，转③；否则转⑤。

⑤ 到此认为单步步入执行一条 C 语句完成。

（3）监视点调试

监视点调试可以用于对目标机 RAM 区域和可在程序运行过程中更改内容的寄存器进行监视。举例而言，某寄存器或 RAM 区域的值在程序运行期间需保持不变，如果运行期间其值发生了改变，则目标机会立即进入 BDM 模式，从而可以快速和方便的定位问题发生的根源。这种方式称为 A AND B data 方式。监视点调试还可应用于另一种称为 A AND B not data 的情况，即某寄存器或 RAM 区域的值在运行期间不能为某一值，如果程序运行时其值改变为该值，则目标机会立即进入 BDM 模式。

监视点调试时需将比较器 A 的内容设置为要监视的地址，比较器 B 设置为相应的值即可。

5.5.3 变量数据的获取与处理

在调试时，可以在右边的变量跟踪窗口的“变量名”一栏中输入变量名，按回车便可在该窗口的右边看到当前该变量的值，以后该变量值会随着调试的进一步执行而自动更新。下面主要介绍一下变量是怎么处理的。

首先介绍上面在分析调试文件时提到的变量类型<type>。表 5-6 详细描述了调试文件中变量类型和实际变量类型的对应关系和变量大小。

表 5-6 dbg 文件中的数据类型和 C 数据类型对照

.dbg 文件中表示的类型	C 数据类型	大小（单位：字节）
C	signed char	1
S	short	2
I(i)	int	2
L	long	4
D	double or float	4
c	unsigned char	1
s	unsigned short	2
i	unsigned int	2
l	unsigned long	4

程序中设了两个数据链，这两个数据链的每个结点存放了一个变量的信息，都是一个结构体类型（VarInfo）数据，该结构体包含如下的信息：

```
struct VarInfo{
    char    VarName[32];  //    变量名
    int     Address;      //    变量地址,全局变量为绝对地址，局部变量为偏移量
    char    VarType[2];   //    变量类型
    BYTE    data[4];      //    变量值对应的字节
    bool    LocalVar;     //    是否是局部变量
    int     Index;        //    变量在变量跟踪窗口中显示的位置
    VarInfo *next;        //    指向下一个结点
  };
```

上面结构体中，要注意 Address 的值，若变量为全局变量，即 LocalVar＝false，则 Address 是该变量在目标 MCU 中的绝对地址，若变量为局部变量，即 LocalVar＝true，则 Address 则是该变量的地址相对于寄存器 H:X 中的值的偏移量，也就是该变量地址等于 H:X 的内容加 Address；数组 data 是用来接收从 MCU 方发回的该变量的字节；Index 是变量在变量跟踪器窗口中显示的位置，这在显示变量值时使用。

上面提到的两个数据链中，一个数据链包含当前所有变量的信息，该数据链的前部分是全局变量信息，后部分是当前断点所在的程序的局部变量的信息，该数据链由

结构体 VarInfo 指针变量 CurrentVar 指向其头结点，结构体 VarInfo 指针变量 EGloadVar 指向该链前部分的最后一个全局变量，以下称该链为 CurrentVar 链；另一个数据链包含当前要显示的所有的变量（在变量跟踪窗口中的变量）的信息，该数据链由结构体 VarInfo 指针变量 ShowVar 指向其头结点，以下称该链为 ShowVar 链。下面将介绍这两个数据链的生成，及如何取得并处理变量值的。

（1）CurrentVar 链的生成

设置好断点，点击“调试”—>“开始调试”，目标 MCU 成功进入监控后，程序将生成 CurrentVar 链，生成该链主要是分析调试文件中的“DEFGLOBALE”行（全局变量行）和“DEFLOCAL”行（局部变量行），对于上面的例子，假如当前设置的断点在“main.c”文件的“main”函数里，那全局变量行有以下几行：

```
DEFGLOBAL x 42 D
DEFGLOBAL p 46 pI
DEFGLOBAL I1 48 I
```

其中 x、p 和 I1 都为全局变量，x 为实型变量，它在目标 MCU 中的起始地址为$0042；p 为整型指针变量，它的起始地址为$0046，在生成指针变量结点时，是将它分成两个结点，一个存储是指针变量本身 p，另一个存储指针所指的地方*p；I1 为整型变量，它的起始地址为$0048。

当前局部变量行有以下一行：

```
DEFLOCAL I 0 A[12:3:2]I
```

上面的一行表示 I 为局部变量，它是一个二维的整型数组，一维下标为“3”，二维下标为“2”，它所占据的存储空间大小为 12 个字节，它在目标 MCU 中的地址为当前寄存器 H:X 的值偏移 0 的位置，在生成数组结点时，是将它分成 n 个结点的，n 等于所有的下标之积，例如这里的 I 就是分成 6（3*2＝6）个结点，即从 I[0][0]到 I[2][1]各有一个结点。

将上面的所有的变量生成数据链即为当前的 CurrentVar 链，图 5-3 即为该链的结构。其中“?”是指在生成结点时该处的值是未知的。该链 EGloadVar 所指结点及该结点之前的全局变量在程序调试的过程中是一直不变的，EGloadVar 所指结点后面的结点要根据当前的断点所在的程序的不同而实时更新的，也就是在“执行”、“单步步入”和“单步步过”时都要更新该数据链。

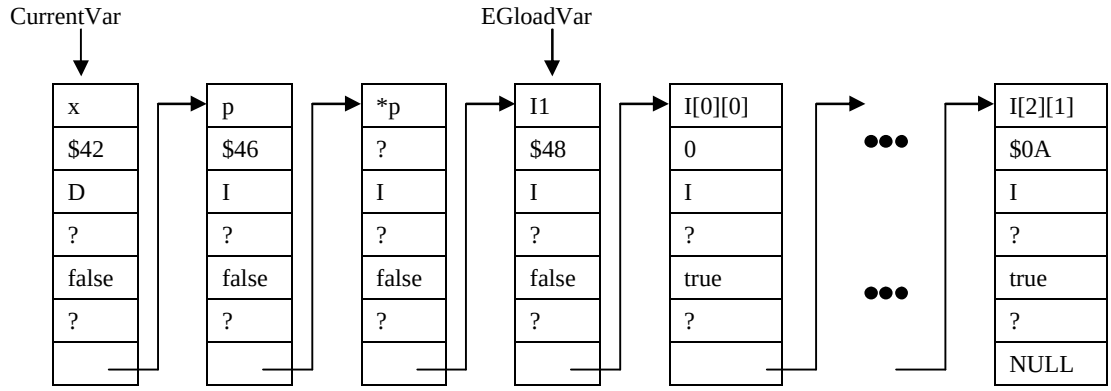

图 5-3 当前变量链结构图

（2）ShowVar链的生成

该链的生成和更新是用户在变量跟踪窗口中的变量名栏输入变量名并按回车键时发生的。在用户在变量跟踪窗口中的变量名栏输入变量名并按回车键后，程序首先判断该变量在 CurrentVar链中是否存在，若不存在，表示输入的变量名在当前的环境下不存在，若存在，则看 ShowVar链中有没有这样的结点，该结点的 Index 值和输入的变量的 Index 相同，若有，则将该结点更换成输入的变量结点，若没有，则直接在 ShowVar 链后面加上该变量结点。假设变量跟踪窗口中输入一些上面的变量 x、p、*p、I1、I[1][1]和 I[2][0]，并按回车，在变量的右边将会出现这些变量的值，当前的 ShowVar链的结构如图 5-4 所示。当程序使用“设置断点”和“执行”或“单步调试”使程序继续向下调试时，变量值也相应的更新。

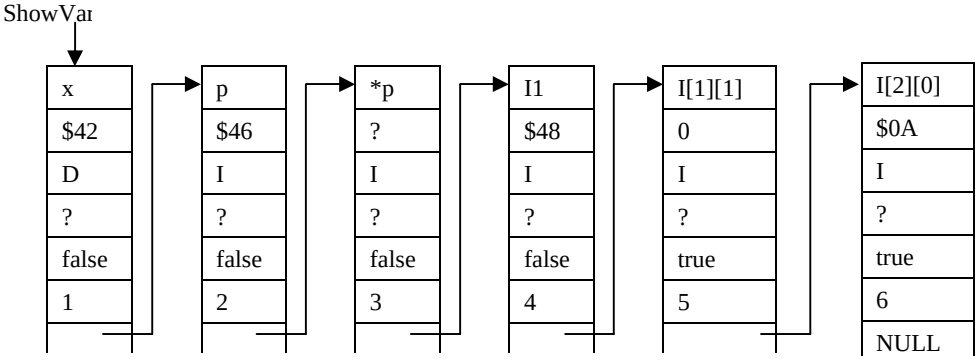

图 5-4 显示变量链结构

（3）变量的读取及处理

每次目标 MCU 发生断点中断时，PC 方都要读取 ShowVar链中的所有变量字节，然后将这些字节转换成变量对应的数据类型的数据并在相应的位置显示出来。

读完 ShowVar 链中所有变量的字节后，这些字节都是相对独立的，需要将这些字节转化为对应的数据类型。例如若一个结点对应的变量类型是实型的，则可以调用下面的 GetDoubleData 函数将该结点中 data 数组中的数据转化为一个实型数据，然后将该实型数据返回，并显示在变量跟踪窗口对应的 Index 位置处。其他类型的转换也要调用对应的转换函数，这里就不一一列举。

```
double CUSBInstance::GetDoubleData(VarInfo *p)
{
 float data=0.0;
 char source[4];
 for(int i=0;i<=3;i++)
 {
  source[i]=p->data[3-i];
 }
 CopyMemory(&data,&source[0],4);
   return data;
}
```

图 5-5 为读取 ShowVar链中的所有变量字节的流程图，P 为结构体 VarInfo 指针变量。

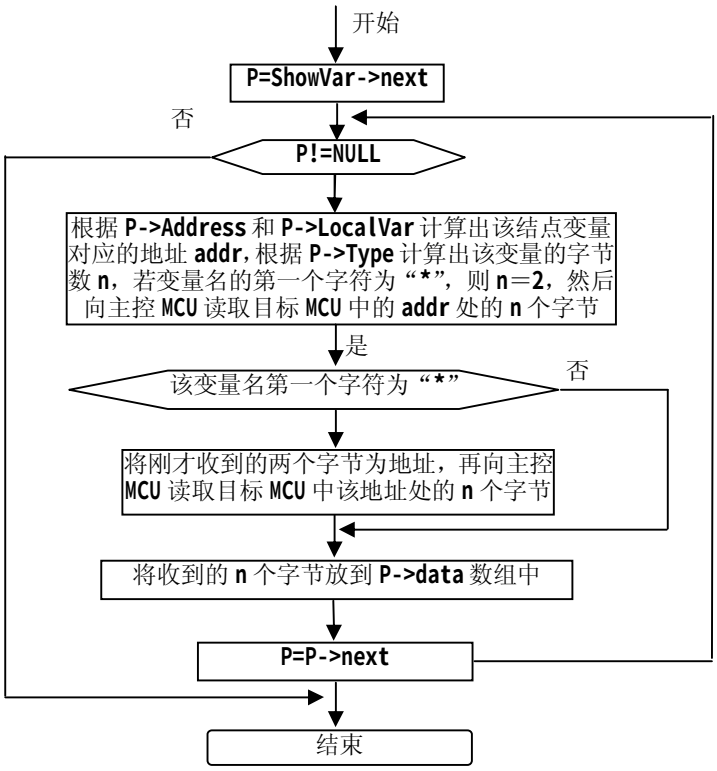

图 5-5 变量的读取与处理流程图

图 5-6 给出了程序进行调试时的情形。在图的左边，从上到下分别为寄存器窗口、变量窗口和存储器窗口。寄存器窗口给出了芯片内部各寄存器的值，变量窗口显示了程序代码中定义的变量的值，存储器窗口则可以显示 RAM 区或 FLASH 区的内容，一次可以显示 64 字节的内容。

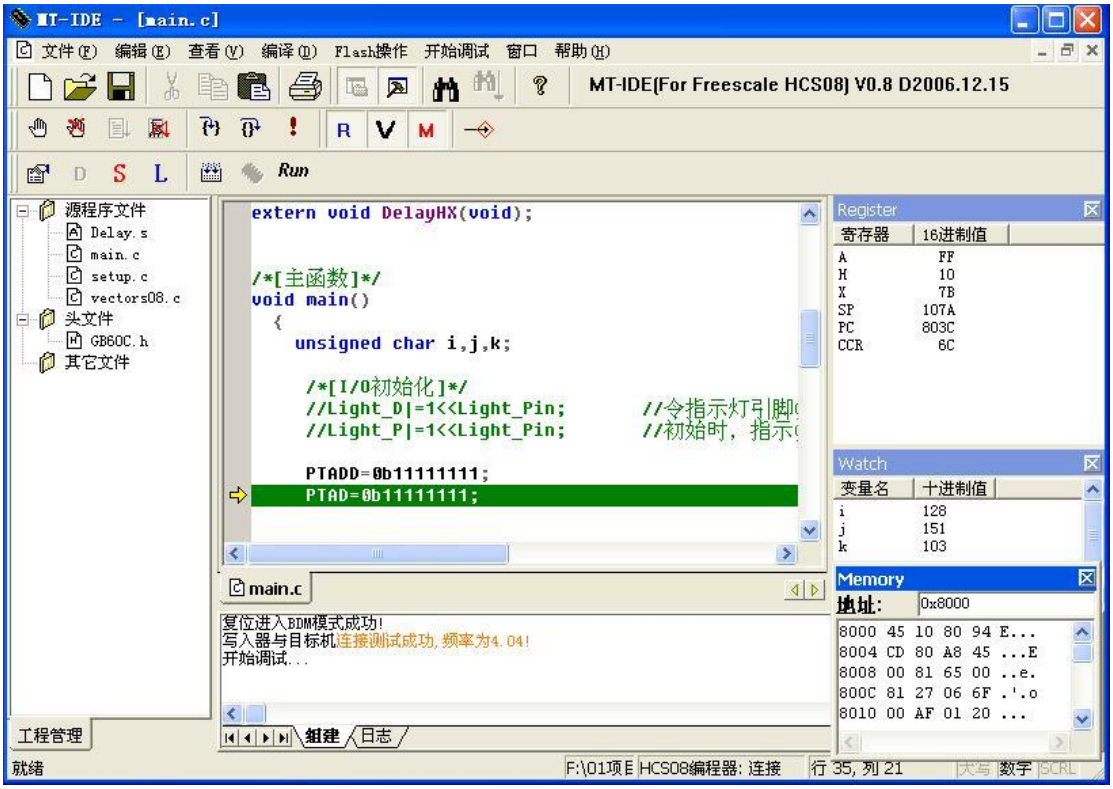

图 5-6 程序调试

5.6 小结

PC 方程序主要是提供了用户操作的界面。用户可以在该集成开发环境界面下编辑、编译和调试程序。多断点和多功能的调试手段给用户带来极大的便利。在实际项目中，使用了该集成开发环境进行开发。实践证明，该集成开发环境所提供的调试功能有效地提高了开发效率。由于篇幅所限，不能将调试过程步骤一一列出。

第六章 USB 驱动程序设计

USB 日益流行，已成为电子工程师必须掌握的计算机外设接口技术。USB 设备需要和主机结合才能正常工作。因此，学习和掌握主机端的驱动程序设计就显得尤为重要。由于该集成开发系统是在 Windows 环境下设计实现的，所以这里介绍 Windows 环境下的 USB 驱动设计。

6.1 Windows 驱动模型

Windows 环境下的驱动程序模型经过不断的演化发展到目前的 WDM（Windows Driver Model ）模型。WDM 旨在实现对新硬件支持的基础上，进一步降低驱动程序的数量和复杂性，以简化驱动程序的开发。

6.1.1 WDM 概述

WDM 属于操作系统的内核模式，其驱动程序由运行于内核模式的系统级代码组成。以某种观点看，Windows 2000/XP 都是由一个操作系统核心和多个驱动程序组成。图 6-1 的 Windows 2000 系统结构试图着重驱动程序开发者所关心的特征[28]。

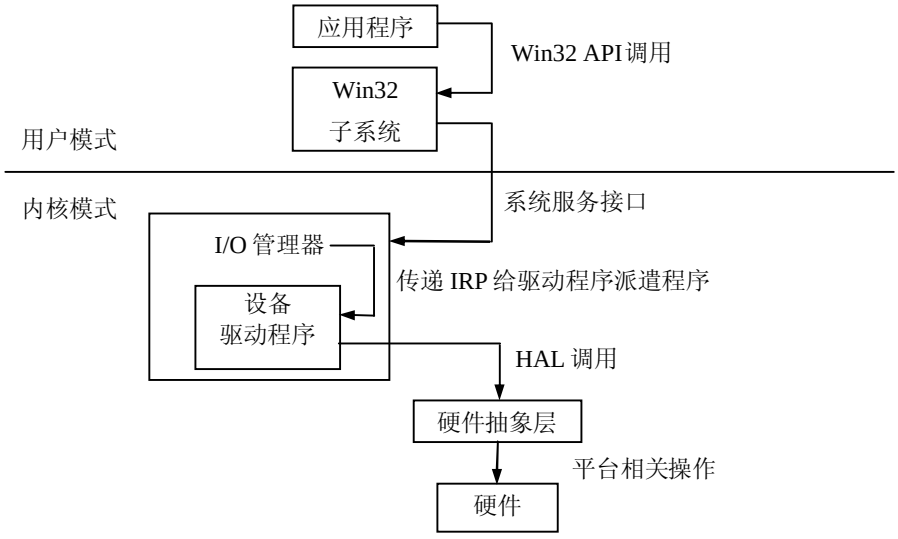

图 6-1 Windows2000 系统结构

从图中可以看出驱动程序要处理的就是从 I/O 管理器发送过来的 IRP（I/O Request Packet），根据不同的 IRP 作出不同的处理（或者是通过调用系统提供的函数来读写端口）。I/O 管理器的主要工作就是接收 I/O 请求（通常来自于用户模式的应用程序），创建 IRP，将 IRP 传递给合适的驱动程序，并且跟踪它们直到完成，同时为每个 I/O

操作的原始请求者返回状态。Windows 2000 系统可以使用多种驱动程序，图 6-2 给出了其所包含的设备驱动种类[29]。从图中可以看出，WDM 驱动程序是一种 PnP 驱动程序，遵循即插即用协议。WDM 驱动程序还细分为类驱动程序（class driver ）和微小驱动程序（minidriver）。类驱动程序管理属于已定义类的设备；微小驱动程序给类驱动程序提供厂商专有的支持。

WDM 引入设备对象的概念来描述设备，主要包含物理设备对象（Physical Device Object，PDO）、功能设备对象（Function Device Object，FDO）和过滤设备对象（Filter Device Object，filter DO）。其中，PDO 对应实际的物理设备，FDO 和 filter DO 是相应驱动程序的处理对象。一个物理设备有且只有一个 PDO 和一个 FDO，却可以拥有多个 filter DOs 。WDM 驱动程序直接操作的不是硬件本身，而是相应的 PDO、FDO 和 filter DO。当用户发出请求时，操作系统的 I/O 管理器将其打包形成一个 IRP 结构，并把它发送到驱动程序，并通过识别 IRP 中的设备对象来区分它是发送给哪一个设备。

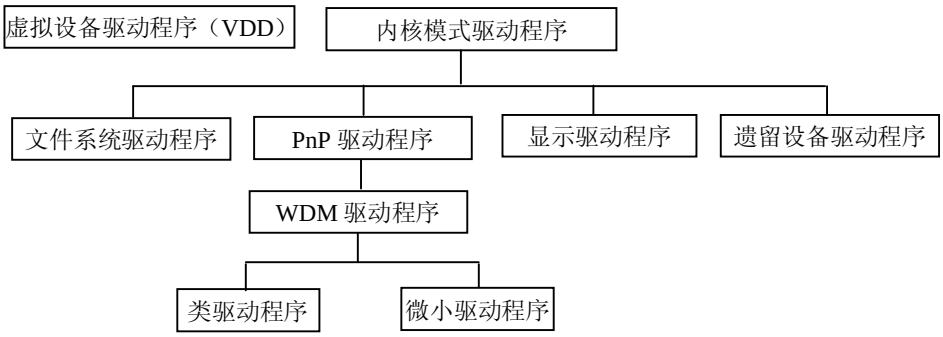

图 6-2 Windows2000 中的设备驱动种类

除支持即插即用外，WDM 驱动程序还支持设备的电源管理，使得设备同系统电源使用情况和设备所处的工作状态相关。WDM 环境提供了系统总线驱动程序，通过系统总线驱动程序，设备驱动程序实现对物理设备的底层控制和资源配置。驱动程序依靠 PDO 存取系统总线驱动程序，FDO 通过向 PDO 发送 IRP，实现和总线驱动程序的通信[30]。WDM 驱动程序采用分层结构，可以和其它驱动程序相联系，接收建立在其上的驱动程序提供的服务，也可以向其它驱动程序发送 IRP 请求。图 6-3 给出了 WDM 设备对象和驱动程序的层次结构。图中左边是一个设备对象堆栈，设备对象是操作系统为帮助软件管理硬件而创建的数据结构。操作系统的 PnP 管理器按照设备驱动程序的要求构造设备对象堆栈。总线驱动程序的一项任务就是枚举总线上的设备，

并为每个设备创建一个 PDO。一旦总线驱动程序检查到新硬件的存在，PnP 管理器就创建一个 PDO，之后就开始描绘图 6-3 所示的结构[31]。

创建完 PDO 后，PnP 管理器参照注册表中的信息查找与这个 PDO 相关的 IRP 器和功能驱动程序。系统安装程序负责添加这些注册表项，控制硬件安装的 INF 文件负责添加其它表项。这些表项定义了各驱动程序在设备对象堆栈中的次序。PnP 管理器根据这些信息完成整个堆栈。

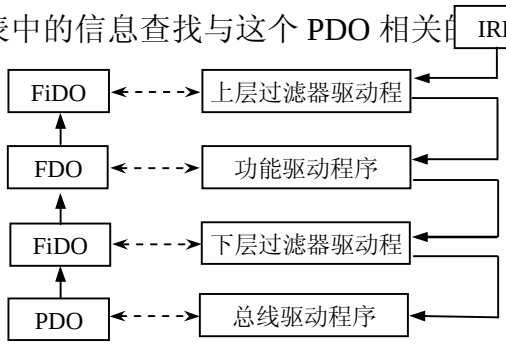

图 6-3 WDM 设备对象和驱动程序的层次结构

层次结构可以使 I/O 请求过程更加明了，参见图 6-3 的右侧。影响到设备的每个操作都使用 I/O 请求包。通常，IRP 先被送到设备堆栈的最上层驱动程序，然后逐渐过滤到下面的驱动程序。每一层驱动都可以决定如何处理 IRP，这取决于设备以及 IRP 所携带的内容。

在单个硬件的驱动程序堆栈中，不同位置的驱动程序扮演了不同的角色。总线驱动程序管理计算机与 PDO 所代表的设备的连接。功能驱动程序管理 FDO 所代表的设备。过滤器驱动程序用于监视和修改 IRP 流。

6.1.2 WDM 的重要概念和数据结构

（1）I/O 请求包 IRP

IRP 是 I/O 管理器在响应一个 I/O 请求时从非分页系统内存中分配的一块可变大小的数据结构内存[32]。IRP 是 WDM 驱动程序中很关键的一个数据结构。由于驱动程序的分层性，IRP 正是在各层驱动程序之间来回穿梭的“信使”。很多 I/O 例程和 PnP 例程以 IRP 为操作对象。当应用程序对驱动程序有读写动作时，系统先要分配并初始化一个 IRP 包，然后传给驱动程序。驱动程序处理 IRP 后将其向下面的总线驱动程序传递，同时要接收总线上传的 IRP。IRP 是一个预先定义好的数据结构，有一个固定的首部和可变数目的 I/O 堆栈单元。每个堆栈单元是一个 IO_STACK_LOCATION结构，它也是由系统定义的固定结构。首部结构和当前 IRP 栈单元中的信息指出驱动程序要完成的动作。IRP 数据结构中有一部分内容由操作系统负责处理，驱动程序不能

访问，其余部分由驱动程序访问和操作。表 6-1 给出了这部分内容。

表 6-1 IRP 首部结构中驱动程序可见的部分

IO_STATUS_BLOCK IoStatus	包含 I/O 请求的状态
PVOID AssociatedIrp.SystemBuffer	如果执行缓冲区 I/O，这个指针指向系统缓冲区
PMDL MdlAddress	如果直接 I/O，这个指针指向用户缓冲区的存储器描述符表
PVOID UserBuffer	I/O 缓冲区的用户空间地址
BOOLEAN Cancel	说明 IRP 是否被取消

每个堆栈单元都对应一个将处理该 IRP 的驱动程序。堆栈单元含有 IRP 的大多数重要的信息，其中 Major Function 是 IRP 代码（如读操作 IRP 的 IRP_MJ_READ），IRP_MJ_PNP 等 IRP 使用 MinorFunction 域指定请求具体的相关功能。表 6-2 给出了 I/O 堆栈单元的数据结构[33]。

表 6-2 I/O 堆栈单元数据结构

MajorFunction	MinorFunction	Flags	Control
Parameter			
DeviceObject			
FileObject			
CompletionRoutine			
Context			

（2）*驱动程序对象* DriverObject

装入系统的每个驱动程序都有一个驱动程序对象。驱动程序对象由操作系统创建，是驱动程序的一个镜像，其内部包含指向不同驱动程序例程的指针。它作为参数传递给驱动程序的入口函数 DriverEntry，并且在其中完成整个驱动程序的初始化。当 I/O 管理器需要定位驱动程序的其它函数时，它使用与设备相关联的驱动程序对象来进行查找。驱动程序对象采用的数据结构是 DRIVER_OBJECT。与其它对象不一样的是，I/O 管理器没有提供对其进行操作的函数，其各字段的值均由 DriverEntry 例程直接设置。驱动程序对象的作用如下：

① 当 I/O 管理器在加载驱动程序的时候创建一个驱动程序对象，如果这个驱动程序在初始化的时候失败，I/O 管理器将删除这个驱动程序的对象。

② 在驱动程序初始化的过程中，DriverEntry 例程加载驱动程序的其它驱动程序函数的指针到驱动程序对象中。

③ 当一个 IRP 被发送到一个指定的设备，I/O 管理器使用相关的驱动程序对象找到正确的驱动程序的派遣例程。

④ 如果请求包含一个设备硬件的操作，I/O 管理器使用驱动程序对象找到驱动程序的 Start I/O 例程。

⑤ 如果驱动程序被卸载，I/O 管理器使用驱动程序对象找到驱动程序的 Unload 例程，当 Unload 例程返回后，I/O 管理器就删除这个驱动程序对象。

图 6-3 给出了驱动程序对象的结构。

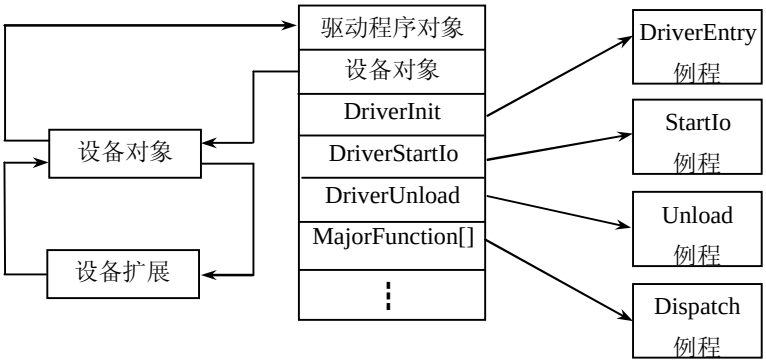

图 6-3 驱动程序对象的结构

（3）设备对象 DeviceObject

每个驱动程序都要为其控制的设备建立一个设备对象：PDO、FDO 或 filter DO，以代表其操纵的功能单元。设备对象是驱动程序对设备的数据描述，它作为一个结构体存在于系统内核中。这些设备对象采用相同的数据结构：DEVICE_OBJECT，它保存了设备的特征和状态信息。通常，一个驱动程序要为多个设备服务。例如，有两个相同的 USB 设备接入计算机时，同一个驱动程序要创建两个设备对象来描述他们。I/O 管理器会将设备对象作为参数传给其驱动程序的大多数例程。WDM 驱动程序中设备对象的创建工作是由 AddDevice 例程来完成的。图 6-4 给出了设备对象的结构。

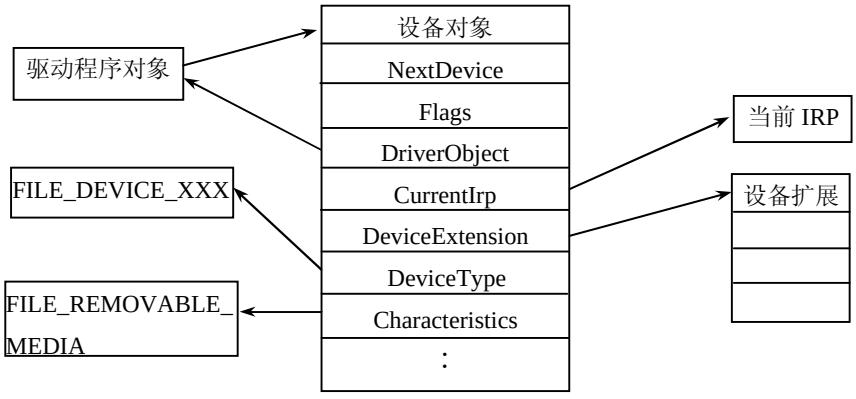

图 6-4 设备对象的结构

（4）设备扩展 DeviceExtension

设备扩展是与设备对象相关联的另一重要的数据结构（DEVICE_EXTENSION），用于保存特定设备的一些专用信息。它通常由驱动开发人员根据具体情况自行定义，其长度和内容均不固定，但一般都含有设备对象的反向指针。

6.1.3 WDM 驱动程序的组成

从一定的观点来看，驱动程序是一些例程的集合。其中的一些例程是必须包含的，另一些是可选的。因此，具体驱动程序不同，其所包含的例程也不同。图 6-5 描述了一个 WDM 驱动程序的基本组成。它包括以下 5 个例程。

① 驱动程序入口例程：处理驱动程序的初始化。

② 即插即用例程：处理 PnP 设备的添加、删除和停止。

③ 分发例程：处理应用程序发出的各种 I/O 请求。

④ 电源管理例程：处理电源管理请求。

⑤ 卸载例程：处理驱动程序的卸载。

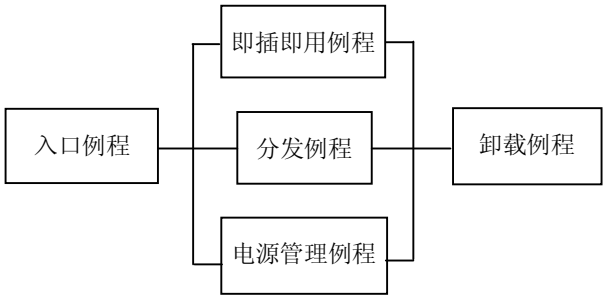

图 6-5 WDM 驱动程序的组成

一个复杂的 WDM 驱动程序还可以包含许多其它的例程，如中断服务例程、延时过程调度例程、故障检测例程等。这些例程在开发 USB 设备驱动程序时不是很常用，因此略去不谈。

6.2 USB 设备驱动实现

USB 总线成功的关键是使用户感到了使用 USB 设备的方便。即插即用概念的应用使得 USB 设备的安装过程得到了简化。USB 驱动程序高度依赖其总线驱动程序（USBD.SYS），而不直接使用硬件抽象层 HAL 函数。USB 驱动程序为了向其硬件设

备发送一个请求，首先创建一个 USB 请求块（USB Request Block ，URB），然后把 URB 提交到总线驱动程序，最后由 USBD.SYS 把请求送到总线上。

6.2.1 USB 即插即用功能的实现

即插即用（PnP）是 WDM 驱动程序的本质特征之一。即插即用包括如下两层含义：一是当一个设备接到主机上时，总线通知系统调用相关的设备驱动程序，当设备移走时，自动释放设备对象和相关资源；二是要识别和处理多个设备同时在主机上的应用。设备对象是 PnP 操作的核心，基本的 PnP 操作是围绕设备对象的建立和删除实现的。本文设计实现的编程调试器仅支持比较简单的 PnP 功能，更多的 PnP 功能可参考相关手册和文档。驱动程序的派遣例程响应主功能码为 IRP_MJ_PNP 的 IRP，并根据次功能码来实现具体的 PnP 功能。表 6-3 中列出了编程调试器驱动程序支持的 PnP 次功能码。

表 6-3 本文驱动程序支持的 PnP 次功能码

次功能码	说明
IRP_MN_START_DEVICE	启动设备
IRP_MN_REMOVE_DEVICE	删除设备
IRP_MN_STOP_DEVICE	停止设备

下面给出了处理相关 PnP 次功能码的源程序：

```
//处理 IRP_MN_START_DEVICE 次功能码
NTSTATUS MyUSB_DriverDevice::OnStartDevice(KIrp I)
{
 NTSTATUS status = STATUS_UNSUCCESSFUL;
 AC_STATUS acStatus = AC_SUCCESS;
 I.Information() = 0;

 acStatus = m_Lower.ActivateConfiguration(1);  //   激活 USB 设备配置

 switch (acStatus)
 {
  case AC_SUCCESS:
   GetStringDescriptors();              //          调用获取设备描述符例程
              status = STATUS_SUCCESS;
   break;
  default:
   break;
 }
   return status;
}

//处理 IRP_MN_STOP_DEVICE 次功能码
```

```
NTSTATUS MyUSB_DriverDevice::OnStopDevice(KIrp I)
{
 NTSTATUS status = STATUS_SUCCESS;
 m_Usb.DeActivateConfiguration();              //   中止 USB 设备当前活动配置
 return status;
}

//处理 IRP_MN_REMOVE_DEVICE 次功能码
NTSTATUS MyUSB_DriverDevice::OnRemoveDevice(KIrp I)
{
 m_Usb.ReleaseResources();                     //   释放动态分配的资源
 return STATUS_SUCCESS;
}
```

6.2.2 USB 驱动程序接口

(1) USBDI 的定义

编程调试器驱动程序作为上层驱动要完成必要的 USB 通信和管理功能，需要使用 USB 驱动程序接口 USBDI 与下层进行沟通。USBDI 是 Windows 操作系统提供的 USB 总线驱动程序接口，它实现了 USB 数据传输的底层协议。USB 设备驱动程序可以直接使用该接口与其物理设备进行通信，而不用考虑诸如总线如何列举、数据如何传输等细节问题。图 6-6 描述了一个包含 USBDI 的 WDM 驱动程序分层结构。其中，USBDI 介于 USB 设备驱动程序和 USB 主机驱动程序栈之间[34]。

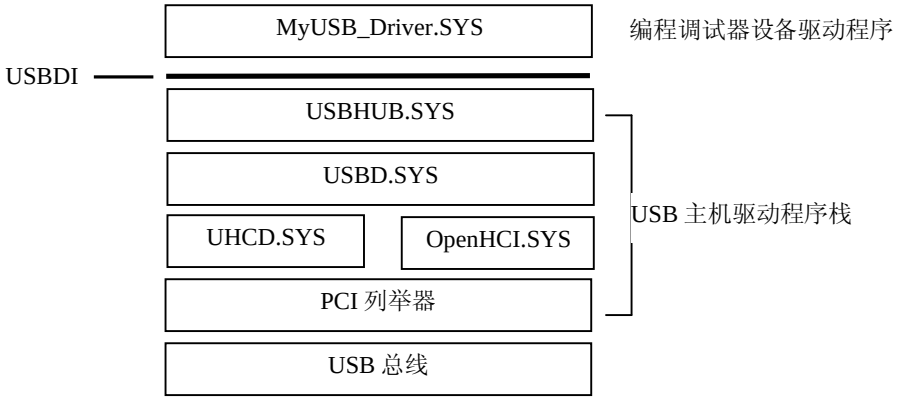

图 6-6 USBDI 和 WDM 分层结构

USBDI 定义驱动程序通过一个或多个管道（Pipe）访问功能设备。管道是主机和设备之间单向或双向的数据传输通道，有控制传输、中断传输、块传输、等时传输四种传输管道。管道如同设备的断点，USBDI 将设备看作是由配置、接口和管道组成的，如图 6-7 所示。

图 6-7 说明了设备和 USBDI 表现给客户驱动程序的逻辑实体。一个设备的管道连接点称为端点。多个端点可以组合在一起，构成一个接口。一个或多个接口组成一个配置。一个设备通常只含有一个设置和一个接口，但也可以含有多个配置和接口。在任意时刻只能有一个配置是可用的，且该配置中的所有接口和端点都是活动的。Windows 使用 USB 设备描述符、厂商 ID 和产品 ID 域形成硬件 ID。这个硬件 ID 用于尝试发现匹配的设备驱动程序并安装 INF 文件。如果没有匹配，则用选定的配置中的每个接口描述符的类域形成一个兼容 ID。这些兼容 ID 用于搜索特定的安装 INF 文件，对设备中的每个接口创建一个设备。

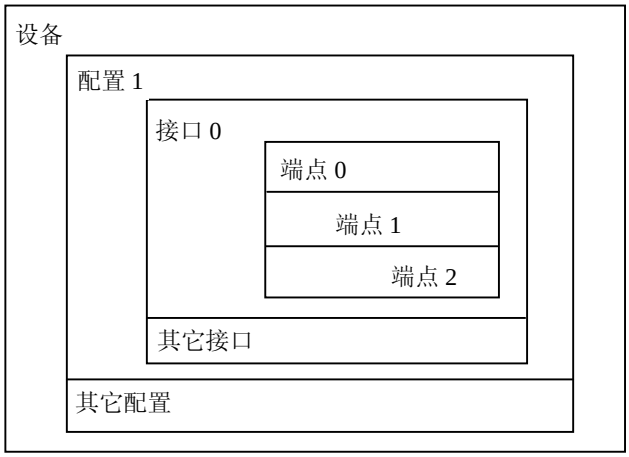

图 6-7 USBDI 定义的逻辑设备结构

控制端点是双向的，中断端点是单向到主机的，块端点和等时端点也是单向的，沿任一方向。编程调试器有一个配置及一个接口，接口包括 3 个端点：一个控制端点，一个读中断传输端点和一个写中断传输端点。

（2）USBDI 定义的各种描述符

与图 6-7 的 USB 设备逻辑结构一致，USBDI 定义了设备描述符、配置描述符、接口描述符和端点描述符等数据结构。

① 设备描述符

设备描述符定义在 USB_DEVICE_DESCRIPTOR 结构中，该结构定义如下所示。

```
typedef struct _USB_DEVICE_DESCRIPTOR {
  UCHAR  bLength ;                  //描述符长度=18
  UCHAR  bDescriptorType ;          //描述符类型，为 1 代表是设备描述符
  USHORT  bcdUSB ;                  //USB 设备说明版本的 BCD 编码
  UCHAR  bDeviceClass ;             //设备类代码
  UCHAR  bDeviceSubClass ;          //设备子类
  UCHAR  bDeviceProtocol ;          //设备协议
  UCHAR  bMaxPacketSize0 ;          //厂商 ID
  USHORT  idVendor ;                //产品 ID
  USHORT  idProduct ;               //设备发行版本号
  USHORT  bcdDevice ;               //指向可读文本的厂商字符串指针
  UCHAR  iManufacturer ;            //指向可读文本的产品字符串指针
```

```
  UCHAR  iProduct ;              //指向可读文本的序列号字符串指针
  UCHAR  iSerialNumber ;         //默认控制端点 0 的最大传输包
  UCHAR  bNumConfigurations ;    //设备的配置数
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR ;
```

② 配置描述符

配置描述符定义在 USB_CONFIGURATION_DESCRIPTOR 结构中，该结构的定义如下所示。

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
  UCHAR  bLength ;               //描述符长度＝9
  UCHAR  bDescriptorType ;       //配置描述符类型，为 2
  USHORT  wTotalLength ;         //此配置信息的总长（包括配置、接口和端点的描述符）
  UCHAR  bNumInterfaces ;        //此配置所支持的接口个数
  UCHAR  bConfigurationValue;    //在 SetConfiguration 请求中用作参数来选定此配置
  UCHAR  iConfiguration ;        //描述此配置的字符串描述表索引
  UCHAR  bmAttributes ;          //电源配置特性
  UCHAR  MaxPower ;              //此配置下的总线电源消耗，单位为 2mA
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR ;
```

③ 接口描述符

接口描述符定义在 USB_INTERFACE_DESCRIPTOR 结构中，该结构的定义如下所示。

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
  UCHAR  bLength ;               //描述符长度＝9
  UCHAR  bDescriptorType ;       //接口描述符类型，为 4
  UCHAR  bInterfaceNumber ;      //接口号，当前配置支持的接口数组索引（从零开始）
  UCHAR  bAlternateSetting ;     //可选设置的索引值
  UCHAR  bNumEndpoints ;         //此接口用的端点数，如果为 0，表示只用缺省控制端点
  UCHAR  bInterfaceClass ;       //设备类值，零值为将来的标准保留
  UCHAR  bInterfaceSubClass ;    //子类码
  UCHAR  bInterfaceProtocol ;    //协议码
  UCHAR  iInterface ;            //描述此接口的字符串描述表的索引值
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR ;
```

④ 端点描述符

端点描述符定义在 USB_ENDPOINT_DESCRIPTOR 结构中，该结构的定义如下所示。

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
  UCHAR  bLength ;               //描述符字节数＝7
  UCHAR  bDescriptorType ;       //端点描述符类型，为 4
  UCHAR  bEndpointAddress ;      //端点的地址及方向
  UCHAR  bmAttributes ;          //传输类型
  USHORT  wMaxPacketSize ;       //端点传输的最大数据包
  UCHAR  bInterval ;             //中断和同步传输的巡检时间
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR ;
```

（3）URB 的构造和发送

USB 设备驱动程序从不直接与硬件通信，相反，它仅靠创建 URB 并把 URB 提交到总线驱动程序就可以完成硬件操作。可以把 USBD.SYS 看作是接受 URB 的实体，向 USBD 的调用被转化为带有主功能代码为 IRP_MJ_INTERNAL_DEVICE_CONTROL 的 IRP。然后 USBD 再调度总线时间，发出 URB 中指定的操作。

为创建一个 URB，首先需要为 URB 分配内存，然后调用初始化例程把 URB 结构中的各个域填入请求要求的内容。URB 有 30 多个不同的功能，也被编成代码称为 URB 功能码，URB 功能是依照 USB 设备协议中的标准设备请求设计的。URB 结构是一个联合，如下所示。

```
typedef struct _URB{
   union{
          struct _URB_HEADER UrbHeader;
          struct _URB_SELECT_INTERFACE UrbSelectInterface;
          struct _URB_SELECT_CONFIGURATION UrbSelectConfiguration;
             :
        }
  }URB, *PURB;

struct _URB_HEADER{
   USHORT Length;
   USHORT Function;
   USBD_STATUS Status;
      :
  };
```

URB 结构含有 18 个不同的结构。每个功能代码使用其中的一个 URB 结构详细说明它的输入或输出参数。所有的 URB 结构以一个公共的 URB_HEADER 结构开始，在调用 USB 设备接口之前，须填写它的 Length 和 Function 域。为了更容易地构造合适的 URB，微软的 DDK 开发包提供了各种构造宏（如 UsbBuildGetDescriptorRequest），填写预先分配好的 URB。还有一些宏为 URB 分配内存并填写它[35]。以下代码构造了一个 URB。

```
urb=ExAllocatePool(NonPagedPool,sizeof(struct
_URB_CONTROL_DESCRIPTOR_REQUEST));
if(urb){
         siz=sizeof(USB_DEVICE_DESCRIPTOR);
         deviceDescriptor=ExAllocatePool(NonPagedPool,siz);
         if(deviceDescriptor){
                  UsbBuildGetDescriptorRequest(urb,(USHORT)sizeof(struct
```

```
_URB_CONTROL_DESCRIPTOR_REQUEST),USB_DEVICE_DESCRIPTOR_TYPE,0,0,
deviceDescriptor,NULL,SIZ,NULL);
//该 USBDI 接口函数用于设置 urb 的参数，使其能够从设备中读取指定的描述符
}
}
else ntStatus=STATUS_NO_MEMORY;
```

构造好 URB 后就把它发出去。USB 内部驱动程序不使用 I/O 栈单元的功能码，而是将栈单元的 Parameters.Others.Argument1 域设置为 URB 的指针。以下代码将 URB 发送出去。

```
event=ExAllocatePool(NonPagedPool,sizeof(KEVENT));
if(event){
      irp=IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
                deviceExtension->StackDeviceObject,NULL,0,NULL,0,TRUE,event,&ioStatus);
     nextStack=IoGetNextIrpStackLocation(irp);
     nextStack->Parameters.Others.Argument1=Urb;
     ntStatus=IoCallDriver(deviceExtension->StackDeviceObject,irp);
     if(ntStatus==STATUS_SUSPEND){
        ntStatus=KeWaitForSingleObject(event,Suspended,KernelMode,FALSE,&timeout);
        }
      ExFreePool(event);
    }
else
   ntStatus=STATUS_NO_MEMORY;
```

IoBuildDeviceIoControlRequest 构造 IRP 包，其 第一个 参数 IOCTL_INTERNAL_USB_SUBMIT_URB 指明此 IRP 用于发送 URB。随后在下一个驱动程序的 I/O 栈单元的 Parameters.Others.Argument1 域中存储 URB 包的指针。IRP 包构造好后，IoCallDriver 将其发送出去。

（4）数据的读写

有了以上的基础，就可以进行数据的读写了。以下给出数据读写的代码。

```
currentIrp=IoGetCurrentIrpStackLocation(Irp);              //获取当前 I/O 栈单
元
deviceExtension=DeviceObject->DeviceExtension;
inBufLen=currentIrp->Parameters.DeviceIoControl.inBufLen;  //输入数据的大小
outBufLen=currentIrp->Parameters.DeviceIoControl.outBufLen;//输出数据的大小
pInterfaceInfo=deviceExtension->Interface;

siz=sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER);
urb=ExAllocatePool(NonPagedPool,siz);                      //为 URB 分配内存
    //ioBuffer 指向读/写数据缓冲区首址
ioBuffer=Irp->AssociatedIrp.SystemBuffer;
pPipeNum=(PULONG)ioBuffer;                                 //获取端点号
pPipeInfo=&(pInterfaceInfo->Pipes[*pPipeNum]);
pcTempBuffer=(char*)ioBuffer+sizeof(ULONG);
```

```
if(Read)
 length=outBufLen-sizeof(ULONG);
else
   length=inBufLen-sizeof(ULONG);

UsbBuildInterruptOrBulkTransferRequest(urb,(USHORT)siz,pPipeInfo->PipeHandle,
pcTempBuffer,NULL,length,USBD_SHORT_TRANSFER_OK,NULL);
//设置 urb，使其能够使用块管道或中断管道收发数据
ntStatus=UrbSend(DeviceObject,urb);                    //发送 URB
Irp->IoStatus.Status=ntStatus;
if(NT_SUCCESS(ntStatus))
     Irp->IoStatus.Information=outBufLen;
else
     Irp->IoStatus.Information=0;
ExFreePool(urb);

return(ntStatus);
```

下图给出了编程调试器在接入主机时进行设备枚举时的过程。

```
Bus Hound 5.00 capture.

  Device - Device ID (followed by the endpoint for USB devices)
           (22) MyUSB_Driver Driver
           (23) USBIO - Device 1
  Phase  - Phase Type
           CTL   USB control transfer
           DI    Data in
  Descr  - Description of the phase
  Data   - Hex dump of the data transferred
  Cmd... - Position in the captured data

Device  Phase  Description       Data
------  -----  ---------------   -----------------------------------------------
  22.0  CTL    GET DESCRIPTOR    80 06 00 01  00 00 12 00
  22.0  DI     ........4.......  12 01 10 01  00 00 00 08  34 12 00 00  00 01 01 02
               ..                00 01
  22.0  CTL    GET DESCRIPTOR    80 06 00 02  00 00 00 04
  22.0  DI     .. .............  09 02 20 00  01 01 00 c0  00 09 04 00  00 02 ff 01
               ................  ff 00 07 05  81 03 08 00  0a 07 05 02  03 08 00 0a
  22.0  CTL    SET CONFIG        00 09 01 00  00 00 00 00
  22.0  CTL    GET DESCRIPTOR    80 06 00 01  00 00 12 00
  22.0  DI     ........4.......  12 01 10 01  00 00 00 08  34 12 00 00  00 01 01 02
               ..                00 01
  22.0  CTL    GET DESCRIPTOR    80 06 00 03  09 01 04 00
  22.0  DI     ....              04 03 09 04
  22.0  CTL    GET DESCRIPTOR    80 06 01 03  09 01 04 00
  22.0  DI     ..So              14 03 53 6f
  22.0  CTL    GET DESCRIPTOR    80 06 01 03  09 01 14 00
  22.0  DI     ..Soochow Univer  14 03 53 6f  6f 63 68 6f  77 20 55 6e  69 76 65 72
               sity              73 69 74 79
```

```
22.0  CTL    GET DESCRIPTOR    80 06 02 03  09 01 04 00
22.0  DI     ".HC              22 03 48 43
22.0  CTL    GET DESCRIPTOR    80 06 02 03  09 01 22 00
22.0  DI     ".HCS08/RS08 Pro  22 03 48 43  53 30 38 2f  52 53 30 38  20 50 72 6f
             grammer & Debugg  67 72 61 6d  6d 65 72 20  26 20 44 65  62 75 67 67
```

图 6-8 编程调试器设备枚举过程

6.3 小结

Windows 环境下的驱动程序开发模式和开发方法与一般的应用程序有很大的不同，而且开发难度较大，不易入门。同时，有关 Windows 设备驱动开发的书籍较少，且多为译本，内容有一定的深度，需要读者具有相当的专业知识。嵌入式的系统学习和工作，或多或少不可避免的要与驱动打交道，需要对驱动程序尤其是底层尽可能地理解透彻，这需要下很大的苦功夫才能达到。

第七章 开发体会与总结

7.1 体会

HCS08/RS08集成开发环境实现规模不算很大，麻雀虽小，五脏俱全，却也很好的体现了嵌入式的许多特点。经过这些年的学习与实践，本人对嵌入式技术应用设计开发有了一些体会与认识。下面结合本人的毕业设计提出个人的几点体会，希望能给相关技术人员起到抛砖引玉的作用。

（1）要重视高端程序设计技能的锻炼

嵌入式系统一般而言主要与低端底层的系统打交道，因此对嵌入式学习的重心自然落在对底层相关知识的理解与掌握上。但嵌入式系统的特点也决定了嵌入式的开发需要在宿主机上进行。在嵌入式系统的开发过程中，需要编制各种测试界面进行系统测试。测试界面的功能以及使用的难易对系统的实现产生很大影响。对可视化面向对象编程语言的熟练掌握将有助于这一问题的解决。

（2）要保证硬件电路的正确性与合理性

嵌入式系统往往是软硬件协调工作的。硬件的正常和正确工作是整个系统正常工作的前提。在开发过程中，往往会遇到各种各样的问题，有些是单纯的软件问题，有些是单纯的硬件问题，还有些问题是软硬件相互影响造成的问题，而这类问题最难发现也最不好解决，是拖延开发进度的重要因素之一。

在设计硬件电路时，需要协调好各模块之间的位置摆放，对于易受干扰的部件模块需要采取有效的抗干扰措施。信号线和电源线应尽可能互相隔离，需要流经较大电流的线路应尽可能加宽。同时，在器件的选择上，也要充分考虑到器件的稳定性以及功耗等各方面的性能指标，需要认真查看相关技术手册。

（3）软件设计应具有编程规范

好的程序不仅仅体现在程序的正确性与运行效率上，更体现在可读性与可维护性上。子程序的长短、变量的命名、文件的大小、注释的位置等等都是在编写程序的过程中需要注意的地方。子程序应该有相关的程序说明，对子程序的功能，输入输出参数都应该详细说明。变量的命名最好采用微软的“匈牙利命名法”，将属性+类型+对象描述的顺序组合起来，以使程序员操作变量时对变量的类型和其它属性有直观的了

解。

（4）测试要充分

测试是开发过程中不可或缺的一环。嵌入式系统的测试一般包含硬件和软件的测试。在测试过程中，需要软硬件交互测试，以确保软硬件协调工作的稳定正常。

软件的测试应先进行各子模块的测试，再进行集成的测试。集成测试过程中，各子模块相继加入测试。每加入一个子模块，需要集成测试一次。对于有问题或变更了功能的子模块，需要进行回归测试，才能确保系统的稳定正常运行。

7.2 总结

本文开发的 HCS08/RS08 集成开发环境，采用 HC08JB8 芯片设计编程调试器，完成了硬件设计、硬件部板、固件编程、高端编程以及系统软硬件测试。本文主要工作详细总结如下：

① 实现 HCS08 系列芯片的擦除、写入和调试操作。

② 实现 RS08 系列芯片的擦除、写入和调试操作。

③ 实现调试过程中寄存器和变量的跟踪。

④ 实现程序的编辑、编译和调试的结合。

该集成开发环境已在苏州大学 Freescale 实验室投入使用，同时还提供给部分企业单位使用。实践证明该系统操作简单，稳定性好，对学习和科研生产活动提供了很大的帮助。尤其是其强大的调试功能带来前所未有的革新令人耳目一新。

在实际运用中，也发现了存在的一些不足，主要是集中在速率这一问题上。由于编程调试器采用 JB8 芯片，其采用 USB1.1 的低速（low speed）通信，速率较低，影响了写入以及调试时的速度，没能充分发挥 HCS08/RS08 芯片 BDM 功能的巨大优势。下一阶段可采用具有 USB1.1 的全速（full speed）或 USB2.0 的高速（high speed）功能的 USB 芯片，进一步提高通信速率，充分发挥 BDM 的高速优势。

参考文献

[1] Background Debug Module (BDM)[M].Freescale Inc.
[2] 王宜怀,刘晓升.嵌入式应用技术基础教程[M].清华大出版社, 2005.
[3] Scott Page. 深入了解 HCS08 的内部时钟源 模块，飞思卡尔应 用笔记, AN3041[DB/OL].http://www.freescale.com.
[4] 邵贝贝 等著.单片机认识与实践[M].北京航空航天大学出版社，2006.
[5] HCS08 Family Reference Manual Volume 1[M].Freescale Inc.
[6] Introduction to Background Debug Mode[DB/OL].飞思卡尔应用笔记，AN3335. http://www.freescale.com.
[7] RS08 Core Reference Manual[M].Freescale Inc.
[8] Getting Started with RS08[DB/OL]. 飞思卡尔应 用笔记， AN3266.http:// www.freescale.com.
[9] 刘雪兰.M68HC08系列MCU嵌入式开发平台的设计与实现[D].苏州大学，2006.
[10] 王宜怀.嵌入式应用在线编程开发系统的研制[J].计算机工程，2002,(28) P22-24.
[11] 8 位 Motorola 单片机 MC68HC908 全系列编程器用户手册[DB/OL]..清华大学 Motoro la单片机于DSP应用开发研究中心,2003.
[12] Arnold Berger著，吕骏 译.嵌入式系统设计[M].电子工业出版社，2002.
[13] HCS08 RS08 Background Debug Mode versus HC08 Monitor Mode[DB/OL].飞思卡尔应用笔记，AN2497.http://www.freescale.com.
[14] Jan Axelson 著.USB大全[M].中国电力出版社，2001.
[15] Don Anderson 著,孟文 译.USB系统体系(第二版)[M].中国电力出版社,2003.
[16] MC68HC908JB8 Technical [M].Freescale Inc.
[17] MC74HC125A Data Sheet[M].Freescale Inc.
[18] MC9S08GB60 Data Sheet[M].Freescale Inc.
[19] MC9S08QG8 Data Sheet[M].Freescale Inc.
[20] MC9S08LC60 Data Sheet[M].Freescale Inc.
[21] MC9S08QD4 Data Sheet[M].Freescale Inc.
[22] MC9S08RG60 Data Sheet[M].Freescale Inc.
[23] MC9S08AW60 Data Sheet[M].Freescale Inc.
[24] MC9RS08KA2 Data Sheet[M].Freescale Inc.
[25] Bonnie Baker著，李喻奎 译.嵌入式系统中的模拟设计[M].北京航空航天大学出版社，2006.
[26] Universal Serial Bus Specification Revision 1.1[DB/OL]. http://w ww.usb.org.
[27] 陈启美 等著.计算机USB接口技术[M]. 南京大学出版社，2003.
[28] Chris Cant著，孙义等译.Windows WDM设备驱动程序开发指南[M].机械工业出版社，2000.
[29] 武安河 著.Windows 2000/XP WDM 设备驱动程序开发[M].电子工业出版社，2005.
[30] 张惠娟 等著.Windows 环境下的设备驱动程序设计[M].西安电子科技大学出版社，2002.

[31] Walter Oney 著.Programming the Microsoft Windows Driver Model[M]. Microsoft Press，1999.
[32] Peter G Viscarola著，新智工作室译.实用技术: Windows NT与Windows 2000设备驱动及开发[M].电子工业出版社，2000.
[33] Microsoft DDK Documentation[DB/OL].www.microsoft.com.
[34] 胡晓军 等著.USB接口开发技术[M].西安电子科技大学出版社，2005.
[35] 王成儒 等著.USB2.0原理与工程开发[M].国防工业出版社，2004.

附录 A JB8 芯片 USB 模块寄存器

寄存器	地址	第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
UADDR	$0038	USBEN	UADD6	UADD5	UADD4	UADD3	UADD2	UADD1	UADD0
UIR0	$0039	EOPIE	SUSPND	TXD2IE	RXD2IE	TXD1IE	0	TXD0IE	RXD0IE
UIR1	$003A	EOPF	RSTF	TXD2F	RXD2F	TXD1F	RESUMF	TXD0F	RXD0F
UIR2	$0018	EOPFR	RSTFR	TXD2FR	RXD2FR	TXD1FR	RESUMFR	TXD0FR	RXD0FR
UCR0	$003B	T0SEQ	0	TX0E	RX0E	TP0SIZ3	TP0SIZ2	TP0SIZ1	TP0SIZ0
UCR1	$003C	T1SEQ	STALL1	TX1E	FRESUM	TP1SIZ3	TP1SIZ2	TP1SIZ1	TP1SIZ0
UCR2	$0019	T2SEQ	STALL2	TX2E	RX2E	TP2SIZ3	TP2SIZ2	TP2SIZ1	TP2SIZ0
UCR3	$001A	TX1ST	TX1STR	OSTALL0	ISTALL0	0	PULLEN	ENABLE2	ENABLE1
UCR3	$001B	0	0	0	0	0	FUSBO	FDP	FDM
USR0	$003D	ROSEQ	SETUP	0	0	RP0SIZ3	RP0SIZ2	RP0SIZ1	RP0SIZ0
USR1	$003E	R2SEQ	TXACK	TXANK	TXSTL	RP2SIZ3	RP2SIZ2	RP2SIZ1	RP2SIZ0
USB 端点 0 数据寄存器,UE1D0-UE1D7 的地址是从$0020-$0027									
USB 端点 1 数据寄存器,UE1D0-UE1D7 的地址是从$0028-$002F									
USB 端点 2 数据寄存器,UE1D0-UE1D7 的地址是从$0028-$002F									