成都理工大学

硕士学位论文

基于DCC和JTAG的ARM硬件仿真调试器的研究与实现

姓名：罗志刚

申请学位级别：硕士

专业：计算机软件与理论

指导教师：洪志全

20080501

基于 DCC 和 JTAG 的 ARM 硬件仿真调试器的研究与实现

作者简介：罗志刚，男，1982 年 10 月生，师从成都理工大学洪志全教授，2008 年 6 月毕业于成都理工大学计算机软件与理论专业，获得工学硕士学位。

摘 要

嵌入式系统开发是当今计算机软件发展的一个热点。嵌入式系统调试器是进行嵌入式开发的关键工具，常用于对嵌入式软件的调试和测试。嵌入式系统调试器由交叉调试器和调试代理组成，其特点在于交叉调试器和调试目标的运行环境相互分离，依赖调试代理来实现其调试会话。随着嵌入式硬件技术的发展，嵌入式应用的不断增长以及嵌入式系统复杂性不断提高，要求嵌入式软件的规模和复杂性也不断提高，嵌入式软件的质量和开发周期对产品的最终质量和上市时间起到决定性的影响，嵌入式软件调试工具的效率成为了人们关注的重点。

本文详细介绍了基于 DCC 和 JTAG 的 ARM 硬件仿真调试器的研究与设计过程。该硬件仿真调试器除了具有下载、断点、单步运行、连续运行、读写内存区域和对寄存器操作等基本调试功能外，还有通过使能 DCC 通道，来进行快速对目标机内存读写的功能。因为读写内存是调试过程中最常用的功能，这样就大大地提高了调试的效率。文中，首先对嵌入式系统开发和嵌入式调试器进行了全面的介绍。然后对当前嵌入式调试中应用最为广泛的 JTAG 技术和 ARM 中的 JTAG 原理作了详细介绍。接着对 ARM 片上调试原理进行了深入分析。最后，深入阐述了 LambdaICE 的设计、实现和测试过程。

本硬件仿真器在设计过程中有两大特色：一是在进行大量数据的内存读写时，采用了 DCC 通道来进行数据传输，这样大大提高了调试器的内存读写速度；二是在保护或恢复上下文时（内核寄存器），采用了批量数据存储指令，这样极大地加快了停止和恢复运行的时间。

关键词：硬件仿真； 调试器； ARM；JTAG；EmbeddedICE；DCC

The Research and Realization of ARM hardware emulation debugger based on JTAG and DCC

Introduction of the author: LuoZhigang, male, was born in October, 1982 whose tutor was Professor HongZhiquan . He graduated from Chengdu University of Technology in Computer Software and Theory major and was granted the Master Degree in June, 2008.

Abstract

Embedded system is a hotspot in the development of computer software nowadays. As a crucial embedded development tool, the embedded system debugger is usually used to debug and test embedded software. A embedded system debugger consists of a cross debugger and a debugger agent, which trait lies on the separation of running environments between the cross debugger and the debugging target, and the dependence on the debugging agent in the debug session. With the development of the embedded hardware technology and the improvement of embedded application and embedded system complexity, the requirement of scale and complexity of embedded software is increasing, the quality of embedded software and development circle play decisive influence on the final quality and marketing time of the products, the efficiency of embedded software debugging tools becomes people's attention focus.

This paper particularly introduces the research and design process of ARM hardware emulation debugger based on DCC and JTAG. Except the basic debugging function of downloading, breaking point, single step, continuous running, memory reading and writing, register operation and so on, this hardware emulation debugger has quick reading and writing target memory by enabling DCC channel. Because memory reading and writing are the most common function in debugging, it improves the debugging efficiency greatly. Firstly, this paper introduces the embedded system development and embedded debugger in the round. The second is the particular introduction of JTAG theory with wide-application JTAG technology and ARM in current embedded debugger. The third is the thorough analysis of debugging principle in ARM chip. At last the design, realization and test process of LambdaICE is expounded.

This hardware emulator has two great features in design process: firstly, the use of DCC channel to transfer data in memory reading and writing with mass data, in this way, we can greatly improve the speed of debugger's memory reading and writing; Secondly,

in protection and restoring of context (the kernel register), the use of batch data storage instruction, so we can greatly quicken the run time of stop and restoring run.

Keywords: hardware emulation; debugger; ARM; JTAG; embeddedICE; DCC

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 成都理工大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

2008年 5 月 30 日

学位论文版权使用授权书

本学位论文作者完全了解 成都理工大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 成都理工大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：

学位论文作者导师签名：

2008 年 5 月 30 日

第 1 章 引 言

1.1 嵌入式系统概述

随着信息化技术的发展和数字化产品的普及，以计算机技术、芯片技术和软件技术为核心的嵌入式系统再度成为当前研究和应用的热点。通信、计算机、消费电子技术（3C）合一的趋势正在逐步形成，无所不在的网络和无所不在的计算（everything connecting，everywhere computing）正在将人类带入一个崭新的信息社会。

1.1.1 嵌入式系统

嵌入式系统是现代科学的多学科互相融合的以应用技术产品为核心，以计算机技术为基础，以通信技术为载体，以消费类产品为对象，引入各类传感器加入，进入 Internet 网络技术的连接，而适应应用环境的专用系统。嵌入式系统最典型的特点是与人们的日常生活紧密相关，任何一个普通人都可能拥有各类形形色色运用了嵌入式技术的电子产品，小到 MP3、PDA 等微型数字化设备，大到信息家电、智能电器、车载 GIS，各种新型嵌入式设备在数量上已经远远超过了通用计算机。

一般认为，嵌入式系统的体系结构可以分为四部分：嵌入式处理器、嵌入式外围设备、嵌入式操作系统和嵌入式应用软件。

1.1.1.1 嵌入式处理器

嵌入式系统的核心是各种类型的嵌入式处理器，它将通用 CPU 中许多由板卡完成的任务集成到芯片内部，从而有利于嵌入式系统在设计时趋于小型化，同时还具有很高的效率和可靠性。

嵌入式处理器的体系结构经历了从 CISC 至 RISC 和 Compact RISC 的转变，位数则由 4 位、8 位、16 位、32 位逐步发展到 64 位。目前常用的嵌入式处理器可分为低端的嵌入式微控制器（micro controller unit，MCU）、中高端的嵌入式微处理器（embedded micro processor unit，EMPU）、用于计算机通信领域的嵌入式 DSP 处理器（embedded digital signal processor，EDSP）和高度集成的嵌入式片上系统（system on chip，SOC）。

目前几乎每个半导体制造商都生产嵌入式处理器，并且越来越多的公司开始拥有自主的处理器设计部门，据不完全统计，全世界嵌入式处理器已经超过 1000 多种，流行的体系结构有 30 多个系列，其中以 ARM、PowerPC、MC 68000、

MIPS 等使用得最为广泛。

1.1.1.2 嵌入式外围设备

在嵌入系统硬件系统中，除了中心控制部件（MCU、DSP、EMPU、SOC）以外，用于完成存储、通信、调试、显示等辅助功能的其他部件，事实上都可以算作嵌入式外围设备。目前常用的嵌入式外围设备按功能可以分为存储设备（如 EPROM、FLASH 等）、通信设备（如 RS-232 接口、USB 接口、Ethernet 接口等）和显示设备（如 LCD）3 类。

1.1.1.3 嵌入式操作系统

为了使嵌入式系统的开发更加方便和快捷，需要有专门负责管理存储器分配、中断处理、任务调度等功能的软件模块，这就是嵌入式操作系统。嵌入式操作系统是用来支持嵌入式应用的系统软件，是嵌入式系统极为重要的组成部分。

嵌入式操作系统根据应用场合可以分为两大类：一类是面向消费电子产品的非实时系统，这类设备包括个人数字助理（PDA）、移动电话、机顶盒（STB）等；另一类则是面向控制、通信、医疗等领域的实时操作系统，如 WindRiver 公司的 VxWorks 和 pSOS、科银京成的 DeltaOS、Microwave 的 OS-9 等。实时系统（real time system）是一种能够在指定或者确定时间内完成系统功能，并且对外部和内部事件在同步或者异步时间内能做出及时响应的系统。

1.1.1.4 嵌入式应用软件

嵌入式应用软件是针对特定应用领域，基于某一固定的硬件平台，用来达到用户预期目标的计算机软件。由于用户任务可能有时间和精度上的要求，因此有些嵌入式应用软件需要特定嵌入式操作系统的支持。

1.1.2 嵌入式系统开发

1.1.2.1 嵌入式系统开发流程

在嵌入式系统的应用开发中，整个系统的简要开发流程如图 1-1。

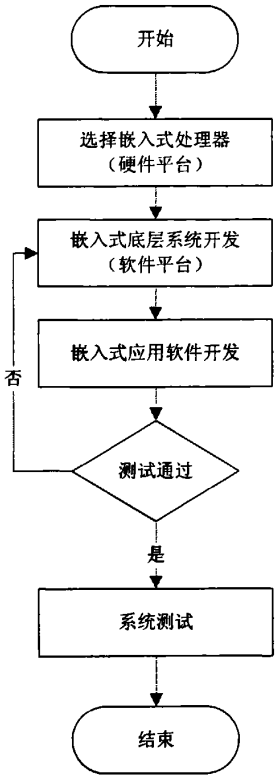

图 1-1 嵌入式系统的开发过程图

由图 1-1 中可以看出，嵌入式系统的开发分为以下三个阶段：

（1）硬件系统的开发，即根据需求和实际情况选择嵌入式处理器平台，相关外围设备，确定系统硬件平台。

（2）底层系统开发，根据需求和实际情况进行板级支持包(BSP)的开发和操作系统的移植、以及驱动和 Bootloader 的开发等，确定软件平台。

（3）上层应用开发，如 Web Server、监控软件等。

嵌入式系统发展到今天，对应于各种微处理器的硬件平台一般都是通用的、固定的、成熟的，这就大大减少了由硬件系统引入错误的机会。嵌入式系统的开发者现在已经从反复进行硬件平台设计的过程中解脱出来，从而可以将主要精力放在满足特定的需求的嵌入式软件的开发上。

1.1.2.2 嵌入式系统软件开发

(1) 底层系统开发

对于嵌入式系统底层软件的开发，难度相对较大，需要了解相关的硬件平台和底层开发的特性。底层系统的开发一般包括 Bootloader、操作系统内核的移植或裁减和设备驱动程序的开发等等。

作为一个嵌入式软件底层开发人员，最重要的就是要了解硬件平台的特性。一般来说，选定的硬件平台往往就是一个关于某个处理器平台的成形开发板。拿到手里之后，嵌入式软件开发人员至少要了解开发板的电路布局以及处理器等的工作特性，才可能进行操作系统的移植或者驱动程序的开发。

在了解硬件平台的特性后，需要弄清底层开发的特性。例如在移植 Linux 内核时，需要了解 Linux 内核的源代码结构以及如何在里面添加一个新的开发板甚至一个新的处理器的支持代码。在 Linux 内核中，同硬件相关的文件主要都在 arch/目录下面，每种不同的处理器和开发板都有单独的配置文件。每一个开发板所具有的不同特性主要在于内存芯片的类型以及在处理器看来的物理地址分配等，还有板上的外接设备的连接方式导致的访问方式的不同。每一个不同的处理器，则具有更多的复杂特性，包括芯片初始化方式、中断处理方式、引脚对应功能等等，需要仔细研究处理器的参考手册，才能进行内核的移植工作。

除此以外，直接编写控制程序，或者在移植的操作系统上开发驱动程序或 Bootloader 程序，也需要在了解硬件设备的工作特性的基础上，根据编写相应底层程序的方法、框架进行开发。由于这部分工作的比较繁琐、复杂，这种底层开发大多数都是由此开发板的供应商来完成。

实际开发中，用户一般只需要对操作系统的内核（如使用嵌入式 Linux 系统）进行配置裁减，就可得到适合的嵌入式操作系统。配置裁减内核相对简单一些，但是也要求开发人员精通内核原理和体系结构。

(2) 应用软件的开发

嵌入式系统的应用软件开发，通常是在已经准备好的底层系统环境之上，开发特定的应用软件。假如底层系统是配置好了的嵌入式 Linux 系统，嵌入式应用开发者就可以在某个嵌入式开发环境中采用 Linux 的软件编写方式，利用 Linux 提供的操作系统接口，来完成特定功能的实现。由于嵌入式 Linux 屏蔽了底层硬件的复杂性，使得开发者通过操作系统提供的 API 函数就可以完成大部分工作，因此大大简化了开发过程，提高了系统的稳定性。所以嵌入式应用软件的开发，主要精力可以集中在应用软件具体的程序流程上面，而不用太过于关心开发板的底层实现。

一般来说，应用软件的开发是在某个嵌入式应用软件开发平台上来完成的，从而使得这个嵌入式系统能够具有某种特定的功能，进而满足市场需求。

(3) 嵌入式软件开发环境

无论是嵌入式底层系统的开发，还是嵌入式应用软件的开发，都离不开良好的嵌入式系统开发环境的支持。由于嵌入式系统受资源限制，不可能附带庞大、复杂的开发环境，因此嵌入式系统软件的开发环境和运行环境往往互相分离，采

用宿主机/目标机模式。宿主机（Host）通常是一台通用计算机（如 PC 机或者工作站），是嵌入式系统软件的开发平台。宿主机的软硬件资源比较丰富，不但包括功能强大的操作系统（如 Windows 和 Linux），而且还有各种各样优秀的开发工具（如 WindRiver 的 WorkBench、科银京成的 LambdaPRO、Microsoft 的 Embedded Visual C++等），能够大大提高嵌入式应用软件的开发速度和效率。目标机（Target）是嵌入式系统的硬件平台，嵌入式系统软件在其中运行。宿主机通过串口、网络接口或特殊的硬件调试接口与目标机通信，从而完成嵌入式软件的开发过程，一般步骤如下：

1) 在宿主机上建立开发环境，进行程序的编码和交叉编译以生成目标平台上可以运行的二进制代码；

2) 下载程序到开发板（目标机）上；

3) 进行交叉调试；

4) 将程序固化到开发板中，并实际运行。

其中，步骤 2）和步骤 3）比较复杂。

对于步骤 2），下载 Bootloader 的动作是根据处理器所支持的方式来实现的，每个平台各不相同（如 s3c2410 的开发板，可以通过 JTAG 方式下载）；而下载 Linux 内核的动作又是根据 Bootloader 所支持的方式来实现的，每个 Bootloader 也各不相同；下载应用程序的方式又是根据操作系统所提供的方式来实现的，是网络方式还是串口方式，通讯协议又是哪种，都无法确定。对于步骤 3）的交叉调试，将在 1.2 节中简要介绍。

1.2 嵌入式系统调试器概述

调试（debug）就是跟踪程序中的错误并加以改正的过程。用于调试程序的工具就是调试器。

许多的编程实践，都是从描述问题逻辑和设计所需的数据结构开始，进而是划分和组织软件模块，最后才着手实现，这样做有助于减少错误和潜在的问题。这也是软件设计与软件工程所存在的价值，一个上百行的程序都应该经过仔细分析和周密设计，而不是等到这个程序将要运行时才来修改。

然而，程序员毕竟是人，编程错误终究难以避免，即使是经过良好设计和良好实现的程序也偶尔会出错。当程序在某处出错而又无法断定为什么出错时，一个行之有效的方法就是用调试器调试代码，运行并观察该程序在哪里发生了错误。因此，调试是开发过程中必不可少的环节。

1.2.1 嵌入式系统调试器

根据调试器和被调试程序的运行环境，软件调试可分为两种方式：一种是本地调试（native debug），这种情况下调试器与被调试的程序往往是运行在同一台机器、相同的操作系统上的两个进程，调试器进程通过操作系统专门提供的调用接口控制、访问被调试进程；另一种是交叉调试（cross debug），这种情况下调试器运行在开发主机上，而被调试程序则运行在目标机上。

交叉调试常常又称为远程调试（remote debug），一般用在嵌入式系统的软件开发中。用于交叉调试的调试器叫做交叉调试器。

进行交叉调试时，开发主机上的交叉调试器以某种方式控制目标机上被调试程序的运行方式，并具有查看和修改目标机上内存单元、寄存器以及被调试进程中变量值等各种调试功能。而提供这种控制功能的就是目标机中称为调试代理的模块，它负责与交叉调试器共同配合以完成对目标机上运行着的进程的调试。一般地，将交叉调试器和调试代理一起称为嵌入式系统调试器。

1.2.2 嵌入式系统调试器的分类

从调试代理的技术实现途径及其应用两个角度，可以将嵌入式系统调试器分为硬件调试器、软件调试器和模拟调试器等 3 类。

硬件调试器与软件调试器在很多方面存在很大的差别。硬件调试器通常适用于嵌入式底层系统的开发。例如，在一块嵌入式开发板上实现一个 Bootloader 程序，或将 Linux 内核移植到一种新型体系结构的 CPU 上等等。在进行硬件调试时，宿主机和目标机之间一般是通过特殊的硬件调试接口来连接的。软件调试器通常适用于嵌入式应用软件的开发，但也可进行嵌入式底层系统的开发。在进行软件调试时，一般使用串口和网络接口来连接宿主机和目标机，调试会话还必须得到目标机上系统软件环境的支持，包括内核、交叉编译器、库程序、shell 交互程序、终端仿真程序等等。

1.2.2.1 硬件调试器

(1) 在线仿真器（in-circuit emulators，ICE）

ICE 是嵌入式系统领域使用得最多，也是功能最强大的调试器之一。ICE 是一个用来设计其他计算机系统的计算机，它代替了目标机上物理的处理器或 MCU，其表现与被代替的目标机处理器完全一样，但是他允许用户查看处理器内部的数据或代码并控制 CPU 的运行[25]。一个在线仿真器通常由仿真探头和仿真器主板组成。仿真探头通过一条电缆与仿真器主板相连，里面包含了一颗与被

代替的 CPU 完全相同的处理器，但是为了调试的目的经过了特殊处理。由于 ICE 对目标机处理器的代替完全是物理上的替代，用户通常要将目标机上的处理器拔出，然后将 ICE 的仿真探头（probe pod）插入目标机的 CPU 插槽中。仿真器主板提供了断点、复杂断点、触发（trigger）、实时跟踪（teal-time trace）、重叠 RAM 和影子 RAM 等众多调试资源，它通过串口（现在出现了 USB 接口）连接开发宿主机上。其中，实时跟踪是 ICE 提供的最有特色的调试手段，它可以在不占用运行时钟周期的情况下获得程序的执行情况，具有非干扰性（nonintrusive）的特点。特别是在强实时系统中，由于无法使用断点，因此实时跟踪就成了唯一有用的调试方式。因此在实时系统的调试中，往往需要使用 ICE。尽管 ICE 有许多优点，但是存在通用性不强与价格昂贵的缺陷，使得 ICE 的应用受到了限制。

(2) 片上调试器（on-chip debuggers，OCD）

由于现代的处理器封装越来越表贴化，仿真器探头的实现也越来越困难。另外，根据统计：在大约 95%的调试过程中，用户仅仅使用了简单断点、单步以及访问处理器资源、内存和外设等一些运行控制方面的基本调试手段。因此，一个很自然的发展趋势就是将实时跟踪和运行控制分开，将运行控制放到目标机系统的 CPU 核（CPU core）内由一个专门的调试控制逻辑模块来实现，并用一个专用的串行信号接口开放给用户，用户可以通过 CPU 核内的调试控制逻辑模块来停止/继续 CPU 的运行，并访问目标机上的各种资源。这种放弃实时跟踪功能，但是提供了大多数 ICE 的调试特性的工具，就是片上调试器[25]。在 OCD 接口中使用串行信号接口是为了减少调试接口的引脚数目。

为了实现主机与目标机处理器的片上调试逻辑之间的连接，可以用一块简单的信号转换电路板来匹配主机通信接口和目标机 CPU 的串行调试接口。这块信号转换电路板称为“片上调试器”或“串行调试器”，信号转换只是它的一个最基本的功能，而其它高级功能的实现由各个厂商在其发布的片上调试器产品中完成。

摩托罗拉公司最早认识到 OCD 技术这个发展趋势，并率先在 683xx 和 68HC16 处理器上创造了 BDM（backgroud debug mode）调试接口，并将其用于它的 Coldfire、PowerPC 等系列微处理器中。而 MIPS、Intel、TI、IBM 和 ARM 等则实现了基于 JTAG（joint test access group）标准的串行调试接口[25]。

与 ICE 相比，BDM 和 JTAG 不存在任何因 CPU 封装或 CPU 速度而带来的问题。尤其是 JTAG，现已成为了 IEEE 的国际标准，即 IEEE1149.1-1990[1]。具有 JTAG 接口的芯片一般都有如下的引脚：测试数据输入（TDI）、测试数据输出（TDO）、测试时钟（TCK）、测试模式选择引脚（TMS），有的还加了一个异步测试复位引脚（TRST）；其片上调试逻辑包括 3 个主要模块：测试访问端口 TAP 控制器、指令寄存器、数据寄存器。虽然 JTAG 调试不占用系统资源，能够调试

没有外部总线的芯片，代价也非常小，但是由于 JTAG 是通过串口依次传递数据，速度比较慢，只能进行软件断点级别的调试，自身还不能完成实时跟踪和多种事件触发等复杂调试功能。因此便有了几种功能更为完善的增强版本。例如 ARM 推出了采用基于 JTAG 版本的 E-Trace，E-Trace 通过 EmbeddedICE 硬逻辑、实时监控、实时跟踪（包括嵌入跟踪微核、跟踪分析仪、跟踪调试软件 3 个部分）3 个增强的辅助片上调试硬件来完成实时调试。

1.2.2.2 软件调试器

(1) ROM monitor

ROM monitor 是指一段驻留在目标机的 ROM 或 Flash 中的小程序，它可以在开发过程中辅助测试与调试用户所编写的嵌入式程序。采用 ROM monitor 方式进行交叉调试需要在目标机上运行 ROM monitor 和被调试程序，宿主机的调试器通过远程调试协议与目标机上的 ROM monitor 建立通信连接。当处理器复位时，ROM monitor 将首先被执行。

在执行完一些必要的初始化后，ROM monitor 一般将等待来自宿主机端的连接，以建立调试会话。ROM monitor 能完成被调试程序的下载、目标机内存和寄存器的读写、设置简单断点以及单步运行等功能。一些高级的 ROM monitor 能完成代码分析(code profiling)、系统分析(system profiling)、ROM 空间的写操作，以及设置各种非常复杂的断点等功能。

(2) 调试桩和调试服务器

调试桩（debugging stub）和调试服务器（debugging server）也是一小段驻留在目标机上的代码。采用这种调试方式进行交叉调试也需要在目标机上运行调试桩（或调试服务器）和被调试程序，宿主机的调试器和目标机的调试桩（或调试服务器）也使用远程调试协议进行连接。

与 ROM monitor 调试方式不同的是，ROM monitor 程序是驻留在目标机的 ROM 中的，系统复位时首先被执行，然后下载被调试程序进行调试；而调试桩和调试服务器不是固化在目标机上的，需要先通过某种工具将它们下载到目标机中。由于调试桩往往是被设计用来独立运行于目标板上的，不需要系统软件环境的支持，因此它必须与被调试程序编译、连接在一起运行，一般用于底层系统软件的调试；而调试服务器通常作为目标机系统上的一个应用程序运行，一般用于应用软件的调试。

以典型的 GNU 调试器 gdb 为例，当使用 gdb 调试底层系统程序时，需要使用 gdb 的调试桩——gdbstub，而使用 gdb 调试应用程序时，需要使用 gdb 的调试服务器——gdbserver。调试桩或调试服务器的职责就是在目标机上实现由宿主

机上的调试器发送过来的调试命令，如读写内存、读写寄存器、设置断点以及运行被调试程序，并将结果返回，以配合宿主机的调试器完成调试[37]。

1.2.2.3 模拟调试器

通常使用的 Simulator 是指令级的模拟器（IIS），它相当于在宿主机上虚拟了一台目标机。该目标机可以是和宿主机的 CPU 不同的类型[37]。利用指令集模拟器进行的交叉调试是一种完全软件模拟的调试方法，根本不需要目标板的支持，就连 I/O 等设备也都是软件模拟的。而实际上软件模拟的结果有时与真实板卡还是有一些差别，硬件的信号、延迟以及对资源的竞争用纯软件的方法根本无法模拟。由于指令集模拟器不需要开发板卡的支持，因此适合于嵌入式系统开发的初级阶段，硬件板卡不是批量生产，数量十分有限。

指令集模拟器也适合于应用程序的调试，因为应用程序与硬件和外围设备关系不是很大。虽然指令模拟器功能有限，但是采用的软件模拟的方法，节省了嵌入式系统开发的成本。

1.3 本文研究目标

在 1.2 节中，我们对目前常用的嵌入式交叉调试方式都做了简单的介绍，对每一种方式的实现原理有了比较清楚的认识：

软件仿真方式使用起来简便、灵活，对硬件依赖程度小。这样在没有目标机硬件的情况下也能够开发调试嵌入式应用软件，实现了软件和硬件同步开发，能够有效缩短产品开发周期。但是由于其纯软件实现的特点，决定了它的强项在于功能仿真，而对嵌入式软件要求较高的实时性这点就难以保证了。所以，嵌入式软件开发最终还是要回到真实的交叉环境中来。

存储监控调试是最常见的也是最经济的一种交叉调试方式。建立交叉环境的过程很简单，只需将一段存储监控程序移植到目标机，利用目标机的硬件（串口、并口、USB 和网卡等）同宿主机调试器通讯，就可以完成各种调试功能。但是同样是纯软件实现的监控程序存在对被调试程序的影响，最主要缺点包括占用内存和硬件通讯设备，消耗处理器时间。

在线实时仿真器 ICE 具有最完善的功能和最优异的实时性能，其他任何调试方式都是无法比拟得。但是由于其高昂的价格，在嵌入式软件开发调试领域正在被后起之秀仿真调试器慢慢超越。仿真调试器可以完成和在线实时仿真器相似的调试功能，但是其成本却低得多。加上众多嵌入式处理器设计生产厂商 Motorola、ARM 等的大力推广，仿真调试器会是今后的一个热点。

因此，本课题研究的目标为：利用 ARM 处理器上自带的 EmbeddedICE 调

试模块，开发一个高效的 ARM 硬件仿真调试器。

基于上述目标，本课题主要研究以下几项内容：

（1）利用 ARM 处理器中 EmbeddedICE 模块进行调试控制（停止、运行、单步、设置断点等）；

（2）利用 EmbeddedICE 中的 DCC 通道进行快速的内存读；

（3）利用 EmbeddedICE 中的 DCC 通道进行快速的内存写；

（4）利用 ARM 的 LDM 指令实现高效的现场环境（寄存器）保护；

（5）利用 ARM 的 STM 指令实现高效的现场环境（寄存器）恢复。

1.4 本文章节安排

第 1 章 介绍了本课题的一些相关背景知识；

第 2 章 深入分析了 JTAG 基本原理和 ARM 中的 JTAG 实现原理；

第 3 章 深入分析了 ARM 内核中 EmbeddedICE 的实现原理；

第 4 章 详细描述了基于 DCC 和 JTAG 的 ARM 硬件仿真调试器 LambdaICE 的总体结构、运行方式、模块间接口等部分的设计思路和实现方法；

第 5 章 简单介绍了 LambdaICE 的测试；

然后是对本文进行的总结；

最后是参考文献和致谢。

第 2 章 JTAG 原理分析

80 年代，伴随着信息技术日新月异的发展，电子设计和嵌入式软件开发领域对专业硬件测试标准的呼声越来越高。在这样的背景下，联合测试行动组（joint test action group,JTAG）于 80 年代后期正是起草了边界扫描测试（boundary-scan testing，BST）规范，于 1990 年正式成为 IEEE 1149.1 工业标准，简称 JTAG 标准。边界扫描测试技术目前已经广泛应用在电子线路设计和软件开发领域。目前生产的多数大规模集成电路（包括 MPU、DSP 等）都提供了 JTAG 功能。

2.1 JTAG 基本原理

2.1.1 JTAG 边界扫描的工作原理

边界扫描测试的基本原理是通过在芯片的各个输入/输出端口增加边界扫描单元（BSC，boundary scan cell）捕获端口信息[1]。每个 BSC 单元由寄存器和两个数据通道构成。一个数据通道用于正常的数据输入和输出，包括 NDI（normal data input）和 NDO（normal data output）；另一个数据通道用于边界扫描测试，包括 TDI（test data input）和 TDO（test data output）。典型的边界扫描实现方式如图 2-1 所示。

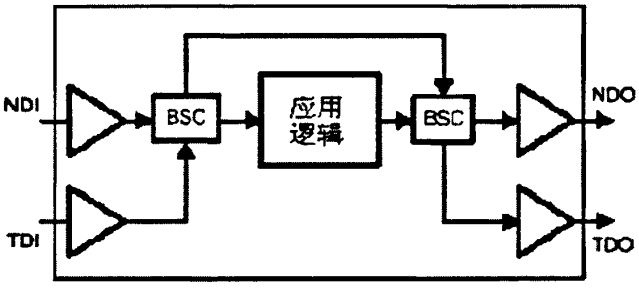

图 2-1 BSC 结构图

具有边界测试功能的芯片在正常情况下的工作流程是：输入的数据从 NDI 经输入端的 BSC 直接发往应用逻辑，然后经过应用逻辑处理后的结果送往输出端 BSC，直接经 NDO 输出结果。在边界扫描状态下，输入端的 BSC 可以有选择的从 NDI 或者 TDI 获得输入数据，数据经应用逻辑处理之后的结果送往输出端 BSC，输出端的 BSC 也可以有选择的将数据发往 NDO 或者 TDO[1]。

在最简单的边界扫描形式下，测试数据连续循环地通过被测单元所有可扫描

的 IC 引脚。这些 IC 引脚是连续的，组成被测电路扫描链（一组串行连接的 TDI 和 TDO 引脚）。换句话说，基本的边界扫描仅仅确定各边界扫描器件是连接正确上。如果采用更复杂的边界扫描形式，例如，将被测单元的寄存器设计成可以累计边界扫描测试数据。累计这些数据后，被测单元根据边界扫描控制器来的串行数据和测试命令处理这些数据。累计的数据能给被测电路提供并行激励，被测电路的响应再被累计，然后通过 TAP 端口输出或者被被测单元测试点上的其它测试设备测量。

可见，对正常使用的芯片边界扫描部件是透明的，不会对芯片产生任何影响。在测试或者调试状态下，通过控制 BSC 单元的状态可将需要检测的应用逻辑单元从系统中隔离出来，这时对应用逻辑单元进行的测试操作便不会对其他单元造成影响。

2.1.2 JTAG 接口的内部结构

JTAG 控制器的电路结构如图 2-2 所示。JTAG 控制器主要由三个部分组成：测试端口控制器（test access port，TAP）、指令寄存器（IR-Instruction Register，包括指令译码器）和数据寄存器（DR-Data Register）[1]。

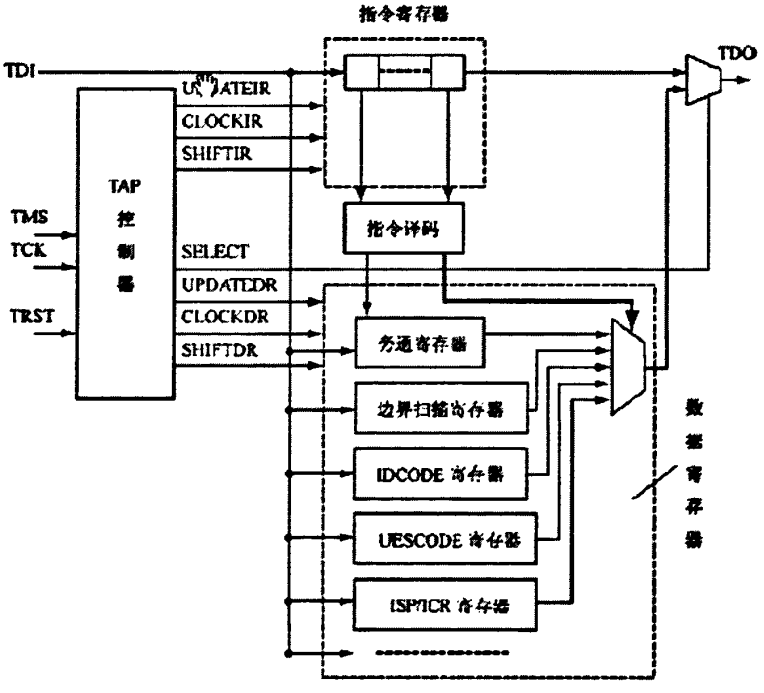

图 2-2 边界扫描测试的内部电路结构

(1) TAP 控制器

这是边界扫描测试核心控制器，有以下 5 个控制信号：

TCK：边界扫描时钟，TAP 的所有操作都是通过这个时钟信号来驱动的。

TMS：JTAG 测试模式选择，用来控制 TAP 状态机的转换。TMS 信号在 TCK 的上升沿有效。

TDI：串行边界扫描输入数据，所有要输入到特定寄存器的数据都是通过 TDI 接口一位一位串行输入的。由 TCK 驱动，在 TCK 的上升沿被采样。

TDO：串行边界扫描输出数据，所有要从特定寄存器的数据都是通过 TDI 接口一位一位串行输出的。由 TCK 驱动，在 TCK 的下降沿被更新。

TRST：JTAG 测试逻辑复位（对 TAP 控制器复位，初始化），低电平有效，当 TRST 输入为低电平时，芯片进入正常工作状态，JTAG 测试逻辑无效。

前四个信号在 IEEE1149.1 标准里是强制要求的，第五个信号在 IEEE1149.1 标准里是可选的，通过 TMS 也可以对 TAP 控制器复位。

(2) 指令寄存器

若执行数据寄存器边界扫描测试，则指令寄存器负责提供地址和控制信号去选择某个特定的数据寄存器；也可以通过指令寄存器执行边界扫描测试，这时，TAP 输出的 SELECT 信号选择指令寄存器的输出去驱动 TDO。

(3) 数据寄存器

JTAG 标准规定，必须具有的两个数据寄存器是边界扫描寄存器（boundary scan register）和旁通寄存器（bypass register）。其它的寄存器是可选的。由指令寄存器选择某个特定的数据寄存器作为边界扫描测试寄存器，当一个扫描路径选定后，其它的路径处于高阻态。边界扫描寄存器是由围绕 IC 管脚的一系列的边界扫描单元 BSC 组成的，正是由它来实现测试管脚信号的输入，输出。旁通寄存器只由一个扫描寄存器位组成，当选择了旁通寄存器，TDI 和 TDO 之间只有一位寄存器，实际上没有执行边界扫描测试，旁通寄存器的作用是为了缩短扫描路径而对不需要进行测试的 IC 进行旁通。

2.1.3 TAP 控制器的状态机

TAP 控制器是边界扫描测试核心控制器。在 TCK 和 TMS 的控制下，可以选择使用指令寄存器扫描或数据寄存器扫描，以及控制边界扫描测试的各个状态。TMS 和 TDI 是在 TCK 的上升沿被采样，TDO 是在 TCK 的下降沿被采样[1]。TAP 控制器的状态机如图 2-3 所示。

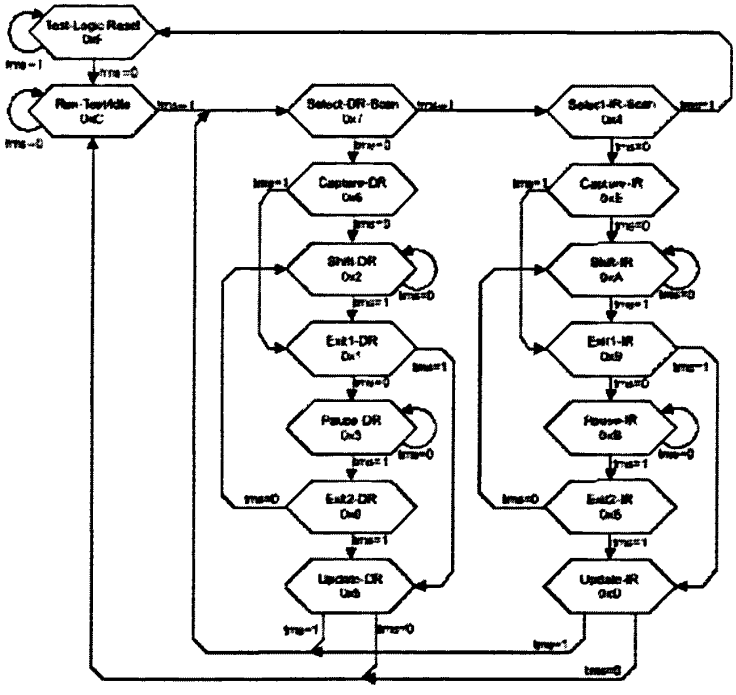

图 2-3 TAP 控制器的状态机

如图 2-3 所示，TAP 控制器共有复位、自检、指令寄存器扫描和数据寄存器扫描 4 部分，共 16 个状态。

(1) 复位

测试复位态（Test-Logic Reset）：设备在正常工作的时候，TMS 维持至少 5 个时钟周期的高电平之后，状态机就处于测试复位态，这时 JTAG 不影响设备的正常运行。当设备需要进行边界扫描测试时，在 TCK 和 TMS 的配合下，TAP 状态机从测试复位态进入其它测试状态。设备在上电、TRST 复位之后会立即进入测试复位态。

(2) 自检

设备测试/暂停态（Run-Test/Idle）：若当前执行自检指令（如 RUNBIST 等）时，设备进入此态之后进行片上系统自检。该状态的另一功能是用于各个边界扫描操作之间的过渡。在一个边界扫描操作完成之后，只要 TMS 保存低电平，TAP 控制器就一直处于测试暂停状态。

(3) 数据寄存器扫描

数据寄存器选择态（Select-DR-Scan）：临时状态，决定 TAP 状态机下一步

的工作状态。若 TMS 置 0，TAP 状态机进入 Capture-DR 态；若 TMS 置 1，TAP 状态机进入 Select-IR-Scan。

数据寄存器捕获态（Capture-DR）：临时状态，在 TCK 上升沿，数据可能被并行送入当前指令选择的测试数据寄存器。若当前指令选择的寄存器不支持数据并行输入或者当前测试项目不需要获得数据，那么测试数据寄存器的数据将保持原有数据不变。若 TMS 置 0，TAP 状态机进入 Shift-DR 态；若 TMS 置 1，TAP 状态机进入 Exit1-DR。

数据寄存器移位态（Shift-DR）：当前指令选择的测试数据寄存器已经放在 TDI 和 TDO 之间，配合使用 TCK 和 TMS 可以将测试寄存器的数据移出。移出操作完成之后，TMS 置 1，TAP 状态机进入 Exit1-DR 态。

数据寄存器扫描退出态 1（Exit1-DR）：临时状态，若 TMS 置 0，TAP 状态机进入 Pause-DR 态；若 TMS 置 1，TAP 状态机进入 Update-DR 态。

数据寄存器暂停移位态（Pause-DR）：暂停数据移位操作，维持测试数据寄存器不变。若退出该状态，TMS 置 1，TAP 状态机进入 Exit2-DR 态。

数据寄存器扫描退出态 2（Exit2-DR）：临时状态，若 TMS 置 0，TAP 状态机进入 Shift-DR 态；若 TMS 置 1，TAP 状态机进入 Update-DR 态。

数据寄存器更新态（Update-DR）：临时状态，在 TCK 的下降沿，通过移位过程获得数据被锁存在测试数据寄存器的输出端口。若 TMS 置 0，TAP 状态机进入 Run-Test/Idle 态；若 TMS 置 1，TAP 状态机进入 Select-DR-Scan 态。

(4) 指令寄存器扫描

指令寄存器选择态（Select-IR-Scan）：临时状态，决定 TAP 状态机下一步的工作状态。若 TMS 置 0，TAP 状态机进入 Capture-IR 态；若 TMS 置 1，TAP 状态机进入 Test-Logic Reset 态。

指令寄存器捕获态（Capture-IR）：临时状态，在 TCK 上升沿，一组固定数据可能被并行送入移位寄存器（shift-register）。若 TMS 置 0，TAP 状态机进入 Shift-IR 态；若 TMS 置 1，TAP 状态机进入 Exit1-IR。

指令寄存器移位态（Shift-IR）：移位寄存器（shift-register）已经放在 TDI 和 TDO 之间，配合使用 TCK 和 TMS 可以设置移位寄存器。移出操作完成之后，TMS 置 1，TAP 状态机进入 Exit1-IR 态。

指令寄存器扫描退出态 1（Exit1-IR）：临时状态，若 TMS 置 0，TAP 状态机进入 Pause-IR 态；若 TMS 置 1，TAP 状态机进入 Update-IR 态。

指令寄存器移位暂停态（Pause-IR）：暂停指令移位操作，维持移位寄存器（shift-register）不变。若退出该状态，TMS 置 1，TAP 状态机进入 Exit2-IR 态。

指令寄存器扫描退出态 2（Exit2-IR）：临时状态，若 TMS 置 0，TAP 状态机进入 Shift-IR 态；若 TMS 置 1，TAP 状态机进入 Update-IR 态。

指令寄存器更新态（Update-IR）：临时状态，在 TCK 的下降沿，通过移位过程获得的指令被锁存在移位寄存器的输出端口。一旦指令被锁定，该指令就成为当前的 TAP 指令。若 TMS 置 0，TAP 状态机进入 Run-Test/Idle 态；若 TMS 置 1，TAP 状态机进入 Select-DR-Scan 态。

TAP 控制器的状态机只有 6 个稳定状态：测试逻辑复位（Test-Logic-Reset）、测试/等待（Run-Test/Idle）、数据寄存器移位（Shift-DR）、数据寄存器移位暂停（Pause-DR）、指令寄存器移位（Shift-IR）、指令寄存器移位暂停（Pause-IR）。其它状态都不是稳态，而只是暂态。

在数据寄存器扫描过程中所做的操作都不会影响当前指令；同样，在指令寄存器扫描过程中所做的操作不会影响当前数据寄存器。

2.2 ARM 中的 JTAG 原理

在上一节已经介绍过，JTAG 控制器主要由三个部分组成：测试端口 TAP（test access port）控制器、指令寄存器（包括指令译码器）、数据寄存器。在本节将以 ARM7TDMI 为例来分析 ARM 内核如何通过各个部分的协调工作，来完成对 ARM 内核的各条扫描链的各种操作，研究成果也可应用到 ARM7 和 ARM9 系列的其他处理器上面。

2.2.1 TAP 控制器、指令寄存器、数据寄存器

TAP 控制器控制着整个 JTAG 的逻辑切换。而整个逻辑的切换过程可以通过 TAP 状态机来描述。状态机的切换过程可以参照前一节的 TAP 状态机图。

通过状态机的切换，可以在 TDI 和 TDO 之间连接的测试数据寄存器为如下几个[3]：

旁路寄存器；设备 ID 寄存器；指令寄存器；扫描链选择寄存器；扫描链 0、1、2、3 。

(1) *旁路寄存器*

目的：为提供 TDI 和 TDO 之间提供一个扫描路径，从而实现对设备的旁路。

长度：1 位。

操作方法：如果 BYPASS 指令是当前指令。

在 CAPTURE-DR 态，BYPASS 寄存器被装载 0。

在 SHIFT-DR 态，TDI 的数据经过 BYPASS 寄存器，在一个 TCK 时钟之后从 TDO 串行输出。因此实现了对设备的旁路。

(2) ARM7TDMI 内核设备 ID 寄存器

目的：用以读取 32 位的内核设备 ID。

长度：32 位。

操作方法：如果 IDCODE 指令是当前指令时，将把设备 ID 寄存器连接在 TDI 和 TDO 之间。ID 寄存器的格式如图 2-4 所示。

在 CAPTURE-DR 态，32 位的设备 ID 将会从设备 ID 寄存器的并行输入端加载到 ID 寄存器。注意，设备 ID 寄存器并没有并行输出端。

在 SHIFT-DR 态，在 CAPTURE-DR 态加载到设备 ID 寄存器的 32 位设备 ID 会从 TDO 串行移出。

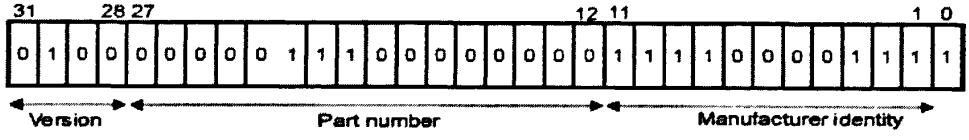

图 2-4 ID 寄存器的格式图

(3) 指令寄存器

目的：变换当前的 TAP 指令。

长度：4 位。

操作方法：

在 CAPTURE-IR 态，会将固定的值 b0001 加载到指令寄存器。该值在 SHIFT-IR 态时被串行由 TDO 移出。复位时，IDCODE 指令被设置为当前指令。

在 TAP 状态机处于 SHIFT-IR 态时，指令寄存器就会在 TDI 和 TDO 之间建立一个串行路径连接。然后在 TCK 的时钟下将数据移入指令寄存器。

在 UPDATE-IR 态，由 TDI 串行移入指令寄存器的指令被设置为当前指令。

(4) 扫描链选择寄存器

目的：用来变换处于激活地位的扫描链。

长度：4 位。

操作方法：如果 SCAN_N 是当前指令，则可以通过设置扫描链寄存器的方法来选择在 TDI 和 TDO 之间建立连接的扫描链。

在 CAPTURE-DR 态，会将固定的值 b1000 加载到该寄存器。这个值会在 SHIFT-DR 时被串行的由 TDO 移出。

在 SHIFT-DR 态，将要选择调试链的数据索引由 TDI 串行移入扫描链选择寄

存器。

在 UPDATE-IR 态,由寄存器中的值来决定选择哪个扫描链作为当然被激活的扫描链。在此以后的所有进一步的指令（如 INTEST）都应用到该扫描链上。直到后续的 SCAN_N 指令执行进而选择其它扫描链、或者发生复位。当复位时，扫描链 0 被设置为激活扫描链。

(5) 扫描链 0、1、2、3

关于扫描链 0、1、2、3 将在 2.2.2 部分中详细介绍。

2.2.2 ARM7TDMI 的扫描链

JTAG 的工作原理就是通过 JTAG 接口，对各个扫描链进行编程，从而实现调试和系统配置功能。

在 ARM7TDMI 内部，有三条扫描链，正是通过对这三条扫描链的，实现了对内核的调试以及对 EmbeddedICE 逻辑进行配置[2]。表 2-1 是 ARM7TDMI 扫描链的编号及功能。

表 2-1 扫描链的配置表

扫描链编号	功能
0	宏单元扫描测试
1	调试
2	EmbeddedICE 逻辑编程
3	外界边界扫描
4	保留
8	保留

在 ARM7TDMI 内部，0、1、2 三条扫描链的配置情况如图 2-5 所示[3]：

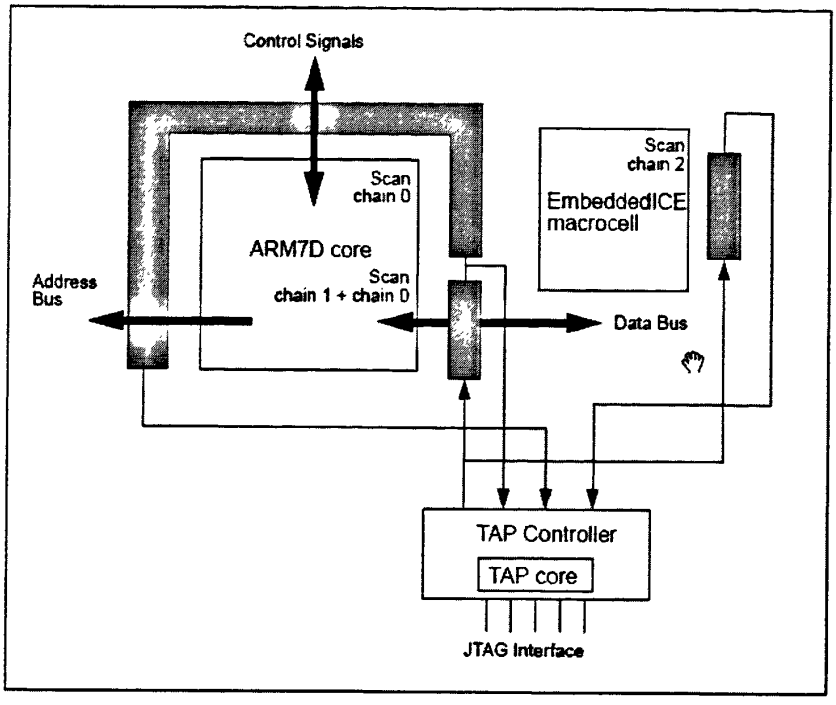

图 2-5 ARM7TDMI 结构图

Scan chain 0：分布在 ARM7TDMI Core 的外围。通过这条链可以控制所有发生在 Core 上的输入和输出。长度为 113bit。

Scan chain 1：可以说是 Scan chain 0 的子链，仅仅分布在数据总线和 breakpoint 上。这条链的功能就是将数据或指令扫描到 Core 上。因此这条链的主要功能是调试功能，所以我们也将他称为调试链（debug chain）。长度为 33bit。为什么会有 breakpoint 这位，我们会在以后详细阐述。

Scan chain 2：这条链是 EmbeddedICE 的配置（编程）链。EmbeddedICE 有的地方称为 ICE-RT，或 IECBREAKER。长度，38bit。

对扫描链的操作方式可以分为以下几种：

INTEST 方式：内核测试。将 TDI 上串行扫描进扫描链中的数据，加载到内核中。所得到的输出单元被捕获后，通过扫描链在 TDO 上串行输出。

EXTEST 方式：外部设备测试。将 TDI 上串行扫描输入的数据加载到内核的输出单元，而不是内核中。系统的输入数据在输入单元被捕获，然后通过扫描链在 TDO 上串行输出。

SYSTEM 方式：扫描单元空闲。这是系统与内核之间进行数据交换的状态，系统将数据加载到内核的输入端，并将内核输出加载系统中。

(1) 扫描链 0

通过扫描链 0 可以对整个 ARM7TDMI 的核外围(包括数据总线和地址总线)进行访问。通过扫描链 0，可以实现对器件之间（inter-device）的测试（EXTEST）和内核的测试（INTEST），该扫描链的长度为 113 位。

从 TDI 到 TDO，扫描链的依次顺序如下：

DB:D0～D31	内核控制信号	AB:A31～A0	EmbeddedICE 控制信号

(2) 扫描链 1

扫描链 1 的主要用途是调试，可以将这条扫描链理解为扫描 1 的子链。完全可以说这条链就是为了调试目的而定义的，之所以单独定义了一条扫描链专门用于调试是因为扫描链 0 过长，不便进行调试，并且效率也不高，该扫描链的长度为 33 位。

从 TDI 到 TDO，扫描链的依次顺序如下：

DB:D0～D31	BREAKPT

(3) 扫描链 2

扫描链 2 是为设置 EmbeddedICE 而定义的。通过这条扫描链可以可以对 EmbeddedICE 进行编程，从而设置断点（Breakpoint）和观察点（Watchpoint）。

从本质上说，扫描链 2 是一个长度为 38 位的移位寄存器，包括：

- 32 位数据
- 5 位地址选择
- 1 位读写控制

(4) 扫描链 3

这条扫描链并不是 ARM7TDMI 的扫描链，它是由具体的芯片生产厂商根据具体的芯片外围逻辑设计，以及集成了不同的内部外设所决定的。因此这里不作详细的介绍。

(5) 控制信号

在除了扫描链扩展之外，ARM 核还进行了额外引脚信号的扩展，这些引脚信号一般在最终的 ARM 处理器芯片上并没有，只是 ARM 核周围的引脚。在这些 ARM 核扩展的额外引脚中最重要的三个引脚是 BREAKPT、DBGRQ、DBGACK，这三个信号的具体情况参考表 2-2 ARM 核信号扩展表。

表 2-2 ARM 核信号扩展表

引脚	说明
BREAKPT	通过该引脚可以强制停止内核运行。在系统运行时，断点指令的执行会该引脚的电平变，此外也可以通过 EmbeddedICE 宏单元声明该引脚的电平，并告知内核
DBGRQ	该信号可以使内核在执行完当前指令后进入调试态。外部硬件可以通过该信号使内核进入调试态
DBGACK	ARM 核的输出信号。如果该信号为高电平，表明内核当前处于调试态。这个信号通常用来判断内核状态

2.2.3 TAP 指令

ARM7TDMI 的 TAP 指令长度为 4bit。没有奇偶校验位。在 TAP 控制器的状态处于 CAPTURE-IR 时，会将固定的值 b0001 加载都指令寄存器中。指令寄存器的最低位被最先扇入（scan in）和扇出（scan out）。下表是 JTAG 指令集的列表 2-3[2]。

表 2-3 ARM7TDMI JTAG 指令集

Instruction	Binary	Hexadecimal
EXTEST	b0000	0x0
SCAN_N	b0010	0x2
SAMPLE/PRELOAD	b0011	0x3
RESTART	b0100	0x4
CLAMP	b0101	0x5
HIGHZ	b0111	0x7
CLAMPZ	b1001	0x9
INTEST	b1101	0xC
IDCODE	b1110	0xE
BYPASS	b1111	0xF

注意：在以下的对指令的描述中，在 TCK 的上升沿对 TDI 和 TMS 采样。而 TDO 上的所有变化都是在 TCK 的下降沿上发生的。

(1) EXTEST (b0000)

执行 EXTEST 指令将被选择的扫描链连接在 TDI 和 TDO 之间，同时使被选择的扫描链处于测试模式。

当 TAP 状态机处于 CAPTURE-DR 状态时，由扫描单元捕获来自内核系统的输入和输出扫描链单元到系统的输出。

当 TAP 状态机处于 SHIFT-DR 状态时，在 CAPTURE-DR 状态捕获的测试数据将从 TDO 串行输出。

EXTEST 指令对于器件间测试非常有用，例如进行电路板上各个器件的连接测试。如果要进行器件间的测试，在 TAP 控制器扫描链 0 后，然后执行 EXTEST 指令。

(2) SCAN_N (b0010)

通过 SCAN_N 指令可以实现在 TDI 和 TDO 之间连接扫描路径选择寄存器。通过对扫描路径选择寄存器的设置可以实现扫描链的切换。

当 TAP 状态机处于 CAPTURE-DR 状态时，会将固定的值 b1000 加载到扫描路径选择寄存器。

当 TAP 状态机处于 SHIFT-DR 状态时，将所需的扫描链的 ID 号移进扫描路径选择寄存器。

当 TAP 状态机处于 UPDATE-DR 状态时，所选的扫描链在 TDI 和 TDO 之间建立连接，直到下一个 SCAN_N 指令发生为止。

在 ARM7TDMI 中，扫描链选择寄存器的长度为 4 位。首先移入移出的是 LSB 位。

(3) SAMPLE/PRELOAD (b0011)

这条指令仅用于产品测试，如用户附加的扫描链（边界扫描链），不允许在 ARM7TDMI 提供的扫描链上使用。

(4) RESTART (b0100)

RESTART 指令在调试态的出口重新启动处理器。RESTART 指令将会在 TDI 和 TDO 之间连接 BYPASS 寄存器。且不用执行 BYPASS 指令。

执行 RESTART 操作后，TAP 状态机进入 Run-Test/Idle 态，然后退出调试装态。

(5) CLAMP (b0101)

CLAMP 指令只能在 chain 0 作为当前链时使用，使用后会在 TDI 和 TDO 之间连接 BYPASS 寄存器，所有的输出信号都会恢复成上一次读入的 chain 0 数据。

当 TAP 状态机处于 CAPTURE-DR 状态时，会将固定的值 b0 加载到 BYPASS 寄存器。

当 TAP 状态机处于 SHIFT-DR 状态时，TDO 第一次输出的是 0。

BYPASS 寄存器在 UPDATE-DR 状态时不受影响

(6) HIGHZ（b0111）

HIGHZ 指令在 TDI 和 TDO 之间连接 BYPASS 寄存器。地址总线、数据总线、nRW, nOPC、 LOCK、MAS[1:0] 以及 nTRANS 都置为悬空的高电平状态，而且外部的 HIGHZ 信号也置为高电平。

当 TAP 状态机处于 CAPTURE-DR 状态时，会将固定的值 b0 加载到 BYPASS 寄存器。

当 TAP 状态机处于 SHIFT-DR 状态时，TDO 第一次输出的是 0。

BYPASS 寄存器在 UPDATE-DR 状态时不受影响。

(7) CLAMPZ（b001）

CLAMPZ 指令在 TDI 和 TDO 之间连接 BYPASS 寄存器。所有 3 态输出端口设置为非激活状态。

当 TAP 状态机处于 CAPTURE-DR 状态时，会将固定的值 b0 加载到 BYPASS 寄存器。

当 TAP 状态机处于 SHIFT-DR 状态时，TDO 第一次输出的是 0。

BYPASS 寄存器在 UPDATE-DR 状态时不受影响。

(8) INTEST（b1100）

INTEST 指令使所选择的扫描链连接在 TDI 和 TDO 之间，并使扫描链处于测试模式。

当 TAP 状态机处于 CAPTURE-DR 状态时，由输出扫描单元捕获来自内核逻辑的输出，输入扫描单元捕获来自系统逻辑的输入数据。

当 TAP 状态机处于 SHIFT-DR 状态时，在 CAPTURE-DR 状态捕获的测试数据将从 TDO 串行输出。

(9) IDCODE（b1110）

IDCODE 指令在 TDI 和 TDO 之间连接设备 ID 寄存器，该寄存器为 32bit 长。当 IDCODE 指令加载都指令寄存器时，所有的扫描单元都被置成处理器正常操作时的系统模式。

当 TAP 状态机处于 CAPTURE-DR 状态时，设备 ID 由 ID 寄存器捕获。

当 TAP 状态机处于 SHIFT-DR 状态时，在 CAPTURE-DR 状态捕获的设备 ID 将从 TDO 串行输出。

(10) BYPASS（b1111）

BYPASS 指令在 TDI 和 TDO 之间连接旁路寄存器，该寄存器 1 位长度。当指令加载到指令寄存器时，所有的扫描单元都被置成处理器正常操作时的系统模式。

当 TAP 状态机处于 CAPTURE-DR 状态时，0 被 BYPASS 寄存器捕获。

当 TAP 状态机处于 SHIFT-DR 状态时，在 CAPTURE-DR 状态捕获的数据 0 将从 TDO 串行输出

当 TAP 状态机处于 UPDATE-DR 状态时，不对 BYPASS 寄存器做任何操作。

第 3 章 ARM 片上调试原理分析

基于 ARM 技术的处理器以其良好的性能，低廉的价格，再加上众多的软硬件厂商的支持，已经占据了全球 32 位 RISC 芯片 75%的市场份额。ARM 体系结构共有六个版本，各个版本都在之前的基础上有所扩展或者修改，但是同种版本处理器的应用程序相互兼容。目前应用最广泛是第五版的 ARM7 系列和 ARM9 系列。后面我们就以 ARM7TDMI 为例分析 ARM 片上调试的工作原理，研究成果也可应用到 ARM7 和 ARM9 系列的其他处理器上面。

3.1 ARM7TDMI 的 EmbeddedICE

通过前面对 ARM7TDMI JTAG 的介绍，我们大致了解了 ARM7TDMI 上面 JTAG 的工作原理和方式。本节将详细介绍 ARM7TDMI 片上调试功能的核心：EmbeddedICE 的结构、工作原理以及编程方式。通过深入分析 EmbeddedICE 原理为后面实现片上调试功能打下基础。

3.1.1 EmbeddedICE 结构

在 2.2.2 节介绍扫描数据链 Chain 2 的时候我们曾经简要提到过 EmbeddedICE（也称为 ICEBeraker）。EmbeddedICE 为 ARM7TDMI 核提供了片上调试功能，可以通过数据链 2（Chain 2）对其进行编程控制。EmbeddedICE 主要由 2 个实时监控单元（Real-Time Watch Point Unit）以及相关的外围控制单元构成。

EmbeddedICE 的结构以及其内部的寄存器如图 3-1 所示[2]。

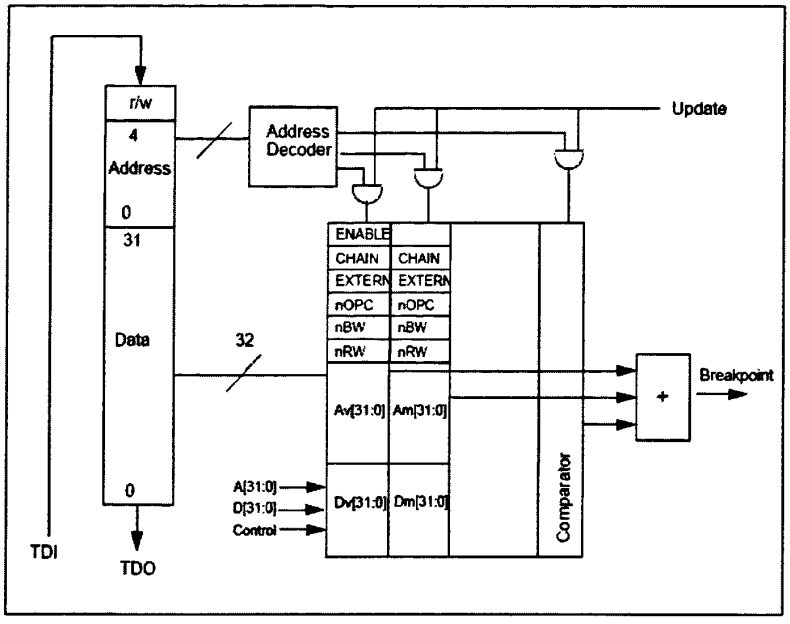

图 3-1 EmbeddedICE 结构图

各寄存器组可以单独编程，并且拥有各自独立的地址。EmbeddedICE 寄存器地址分配如表 3-1 所示[3]。

表 3-1 EmbeddedICE 寄存器地址分配表

Address	Width	Function
00000	3	Debug Control
00001	5	Debug Status
00100	6	Debug Comms Control Register
00101	32	Debug Comms Data Register
01000	32	Watchpoint 0 Address Value
01001	32	Watchpoint 0 Address Mask
01010	32	Watchpoint 0 Data Value
01011	32	Watchpoint 0 Data Mask
01100	9	Watchpoint 0 Control Value
01101	8	Watchpoint 0 Control Mask
10000	32	Watchpoint 1 Address Value
10001	32	Watchpoint 1 Address Mask
10010	32	Watchpoint 1 Data Value
10011	32	Watchpoint 1 Data Mask
10100	9	Watchpoint 1 Control Value
10101	8	Watchpoint 1 Control Mask

当 EmbeddedICE 处于监控状态时，实时监控单元中的一个或者两个实时地监控系统中三种总线的数据情况：数据总线、地址总线以及各种的运行控制信号总线。通过比较器判断三组是否同时匹配实时监控单元中的对应数据，一旦出现总线数据与监控单元内的数据完全匹配的情况，EmbeddedICE 马上发出暂停内核信号(BREAKPT)，内核在当前指令执行完以后转入调试状态。

3.1.2 实时监控单元

实时监控单元由 3 对寄存器组成，每对寄存器包括一个数据寄存器和一个掩码寄存器：

(1) 地址总线数据寄存器和地址总线掩码寄存器

(2) 数据总线数据寄存器和数据总线掩码寄存器

(3) 控制数据寄存器和控制掩码寄存器

数据寄存器和掩码寄存器都是 32 位寄存器，需配合使用。掩码寄存器某一位设置为 1 时，对应位置的数据寄存器的比较结果始终为 1；掩码寄存器某一位设置为 0 时，只有在对应位置的数据寄存器和总线数据一致的情况下比较结构为 1。例如，硬件断点只比较地址完全匹配的情况。将地址寄存器设为断点地址，掩码可以设置为 0，仅在规定位置触发。将数据总线寄存器掩码设置为 0xFFFFFFF，这样任何数据都可以触发。

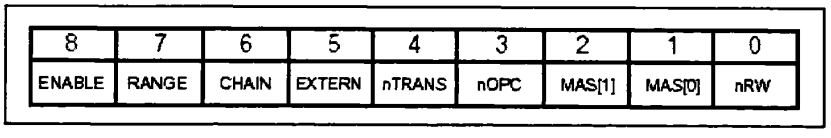

图 3-2 控制寄存器结构图

如控制寄存器结构图所示，控制寄存器有 8 个状态位，分别是 nRW、MAS[0:1]、nOPC、nTRANS、EXTERN、CHAIN、RANGE 和 ENABLE：

nRW：检查总线的活动方向，1 表示写周期，0 表示读周期；

MAS[0:1]：检查总线宽度。如表 3-2 所示。

表 3-2 总线宽度表

bit 1	bit 0	Data size
0	0	byte
0	1	halfword
1	0	word
1	1	(reserved)

nOPC：检查当前周期，0 表示取指周期，1 表示数据访问周期。

nTRANS：检查用户模式，0 表示用户模式，1 表示非用户模式。

EXTERN：观察点外部输入逻辑使能，0 表示允许观察点 1 接收 DBGEXT 输入，1 表示运行观察点 1 接收 DBGEXT 输入。

CHAIN：链接到其他观察点输出。

RANGE：链接观察点 1 和观察点 2，两个观察点可以同时检查输入数据。通常用于范围检查。

ENABLE：观察点允许位，1 表示允许观察点，0 表示禁止观察点。只有在允许观察点的情况下，观察点触发后才产生 BREAKPT 信号。

3.1.3 外围控制单元

EmbeddedICE 外围控制单元包含调试控制寄存器（debug control）、调试状态寄存器（debug status）、调试通讯控制寄存器（debug comms control register）和调试通讯数据寄存器（debug comms data register）。

调试控制寄存器和调试状态寄存器用于处理内核调试信号：TBIT, nMREQ, DBGACK, DBGRQ 和 DBGACK。图 3-3 是 TBIT, nMREQ, DBGACK, DBGRQ 和 DBGACK 结构图[3]。

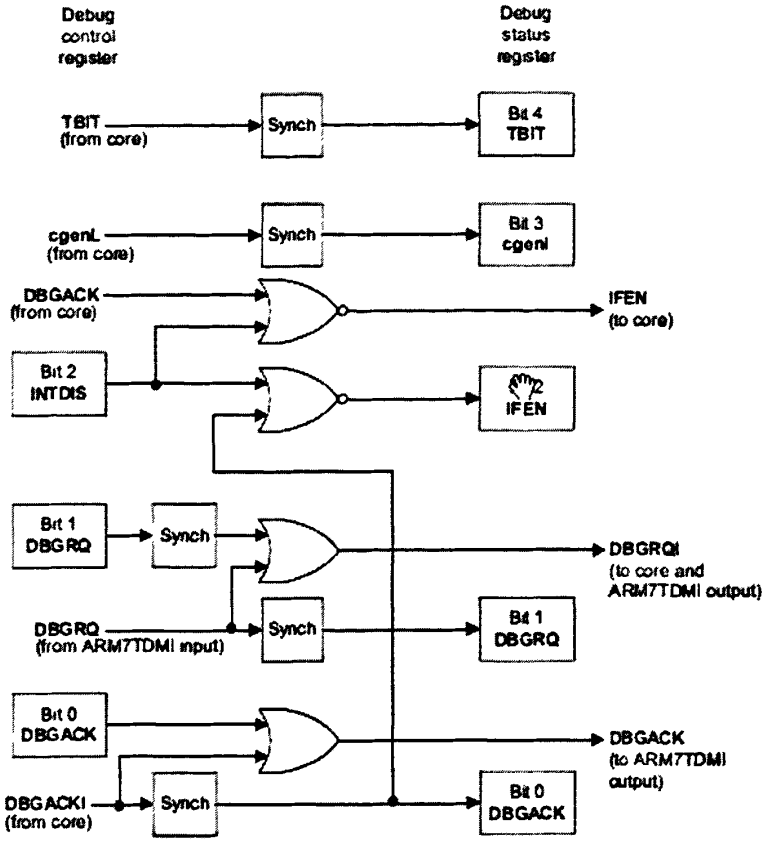

图 3-3 EmbeddedICE 外围控制单元图

调试控制寄存器（debug control）有 6 个控制位：

DBGACK：第 0 位，强制设置内核 DBGACK 状态，告知其它模块内核处于调试状态；

DBGRQ：第 1 位，强制设置内核 DBGRQ 状态。在内核正常运行的时候设置该位，使内核进入调试状态；

INTDIS：第 2 位，在调试状态关闭外部中断。FIQ、IRQ 在 DBGACK 为 1 或者 INTDIS 为 1 的情况下不能被响应；

SBZ/RAZ：第 3 位，该位必须置 0；

监控模式使能：第 4 位，如果该位置 1，当到达观察点或者断点时内核进入调试模式；该位置 0，当到达观察点或者断点时内核进入中断响应模式。

禁止 EmbeddedICE-RT：第 5 位，禁止比较器输出。

调试状态寄存器（debug status）有 5 个状态位：

DBGACK：第 0 位，DBGACK 同步值；

DBGRQ：第 1 位，DBGRQ 同步值；

IFEN：第 2 位，内核中断使能的同步值；

nMREQ：第 3 位，标示调试期间的存储器访问是否结束；

TBIT：第 4 位，TBIT 同步值。

调试通讯控制寄存器（debug comms control register）和调试通讯数据寄存器（debug comms data register）用于和宿主机上面的调试器通讯，将在 3.1.4 调试通信通道中详细介绍。

3.1.4 调试通信通道

ARM7TDMI 处理器的 EmbeddedICE 逻辑，包含一个调试通信通道(debug communications channel，DCC)[3]。通过调试通信通道，可以实现目标机和宿主机上的调试器进行通信。

调试通信通道主要由以下几部分组成：

(1) 32 位通信读寄存器；

(2) 32 位通信写寄存器；

(3) 32 位通信数据控制寄存器，用于处理器和调试器之间的同步握手。

这些寄存器在 EmbeddedICE 逻辑中有固定地址映射，处理器可以使用 MCR 和 MRC 指令访问协处理器 P14 来对这几个寄存器进行读写。而调试器可以通过扫描链 2 来访问这些寄存器。关于它们的地址映射信息，可以参看介绍扫描链 2 部分。

3.1.4.1 DCC 控制寄存器

DCC 控制寄存器用来实现处理器于调试器之间的同步握手。该寄存器的格式如图 3-4 所示。

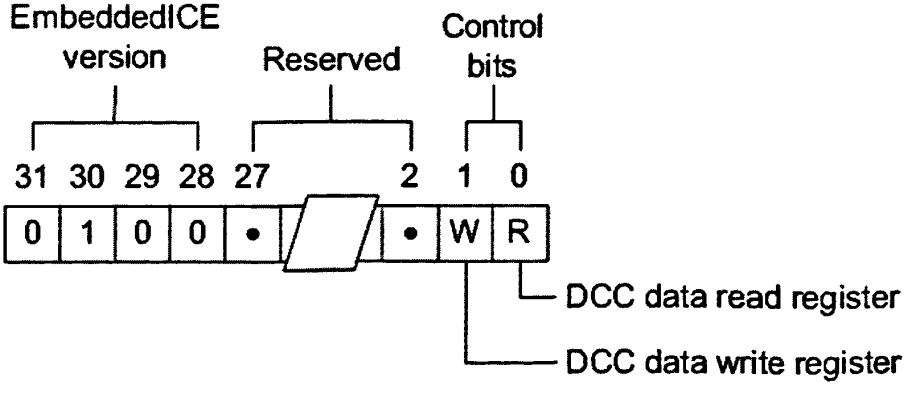

图 3-4 DCC 控制寄存器结构图

Bits[31:28]：EmbeddedICE 版本号，ARM7TDMI EmbeddedICE 版本号为 0001；

Bits[27:2]：保留数据区；

Bit[1]：对宿主机而言是通信写寄存器写入标志。当宿主机发现该位为 1 时，表明通信写寄存器有新的数据到来。对处理器而言是通信写寄存器可写标志，当处理器发现该位为 0 时，表明通信写寄存器可写；

Bit[0]：对处理器而言是通信读寄存器写入标志。当处理器发现该位为 1 时，表明通信读寄存器有新的数据到来。对宿主机而言是通信读寄存器可写标志，当处理器发现该位为 0 时，表明通信读寄存器可写。

3.1.4.2 通过 DCC 通信

调试器可以通过 DCC 发送信息给处理器；处理器也可以通过 DCC 向调试器发送信息。这样便实现了调试器与处理器之间的通信。图 3-5 是主机端调试器和目标机处理器通过 DDC 通讯的示意图。

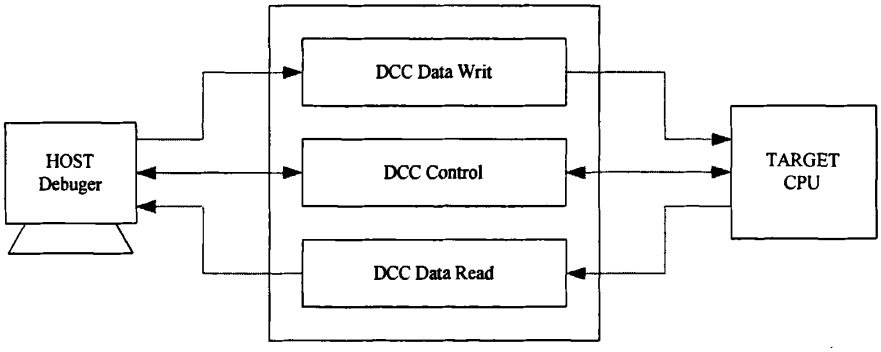

图 3-5 DDC 通讯的示意图

3.2 ARM7TDMI 片上调试的实现

3.2.1 调试系统

关于通过 JTAG 对以 ARM7TDMI 为内核的处理器进行调试的方法，图 3-6 很好的表达其中的逻辑关系。

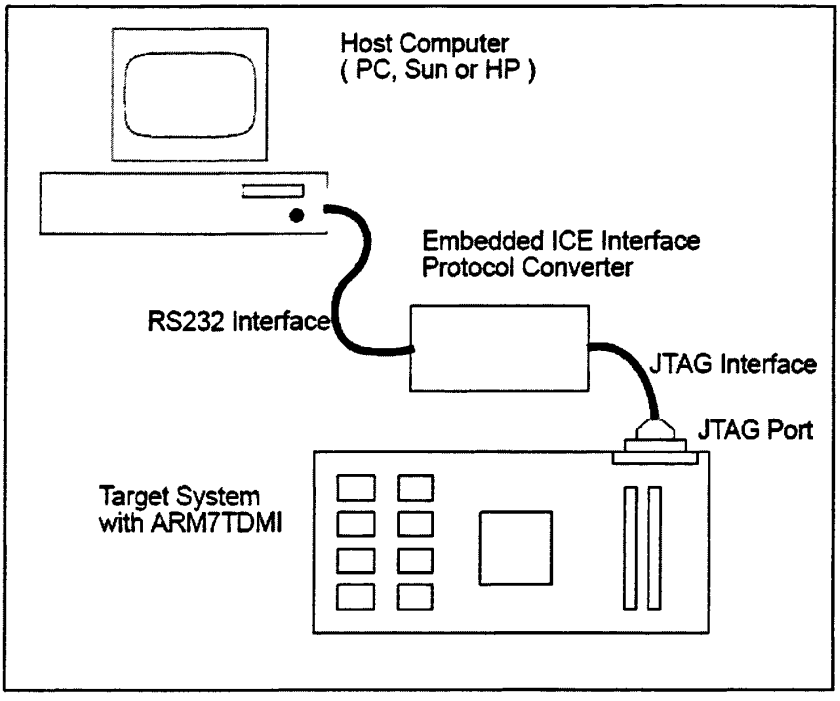

图 3-12 仿真调试系统结构图

整个系统清晰的分为三个部分：宿主机、协议转换器以及目标机。

（1）宿主机：运行调试器（如 GDB、ADS），宿主机发出高级的调试命令，如设置断点，查看寄存器，查看内存，执行指令等。

（2）协议转换器：负责解析调试命令，然后转化为 JTAG 接口支持的低级指令。如我们常见的 BDI，MultiICE（ARM 公司提供给用户的调试工具，但是只有在 ARM 的调试器 ADW 上才能使用）。当然，最典型的还是 BDI。这并不是说 BDI 比 MultiICE 要好，这里典型的意思是更能清晰的代表层次体系，因为 MultiICE 的协议转换是在宿主机上完成。

（3）目标机：包含以 ARM7TDMI 为内核的处理器系统。

3.2.2 ARM7TDMI 的调试模式

ARM7TDMI 调试接口是基于 IEEE 标准 1149.1－1990 以及标准测试访问接口和边界扫描体系结构（standard test access port and boundary-scan architecture）。

ARM7TDMI 为提供先进的调试特性而做了相应的硬件扩展。这使开发应用软件、操作系统以及硬件本身都更加容易。增加这些硬件扩展之后，使得内核在调试时可以处于两种不同的模式，分别为：

（1）暂停模式（**Halt Mode**）：在这种模式下，内核进入调试状态（debug

State)。这时内核被停下来，并且与系统的其他部分隔离。运行在宿主机上的调试器（GDB）通过操作 JTAG 口，完成对内核、内存以及系统其他设备的操作。待调试完成之后，调试器恢复内核和系统的状态，程序继续开始运行。

（2）监控模式（Monitor Mode）：与暂停模式不同的是，在监控模式下为了保证系统对中断或者异常事件的处理能力，当内核其他部件已经暂停的情况下，异常复位部件仍然可以正常工作。当发生数据访问异常或者指令取指异常，这时内核并不进入调试状态，可以继续执行异常复位例程。

3.2.2.1 暂停模式（Halt Mode）

(1) 调试阶段

下列事件都有会引起内核进入暂停模式：

1) 断点（breakpoint），读取断点处指令；

2) 观察点（watchpoint），对设置观察点处进行数据访问；

3) 调试请求（DBGRQ），外部调试请求。

在内核进入调试状态后，内核与外部系统隔离，这时可以通过 JTAG 接口来串行检测内核状态。这样就使得在内核与外部系统隔离的情况下仍然可以对内核进行调试。此外，通过扫描链 1 还可以把指令串行地插入到指令流水线(instruction pipeline)。例如在调试状态时，通过往指令流水线插入 STM 指令来输出 ARM7TDMI 寄存器的内容，并通过 TDO 串行的移出。

(2) 时钟

ARM7TDMI 内核有两个时钟，或者说可以在两个时钟下工作。

1）MCLK：存储器时钟。当系统正常运行，或者说系统运行在系统速度（System Speed）下的时钟。

2）DCLK：测试时钟，即当内核进入调试状态后的运行时钟。在 JTAG 接口的测试时钟 TCK 下产生，但并不是说 DCLK 与 TCK 相同或成某种分频的比例关系。

3.2.2.2 监控模式（Monitor Mode）

ARM7TDMI 逻辑允许在进行系统调试时，并不完全地将内核停下来；而是当内核正在被调试器询问地时，它仍然可以继续执行关键的中断服务例程。通过设置调试控制寄存器的第 4 位（监控模式使能），可以使能监控模式。当 EmbeddedICE 逻辑被设置成 Monitor 模式，断点和观察点就会使内核产生指令预取异常和数据异常，并分别执行中断向量表中各自的异常处理程序。设置成监控模式后，内核就不会因为发生断点和观察点而进入调试模式。如果要将

ARM7TDMI 设置成 Monitor 模式进行调试，一些限制是用户必须要注意的。

Monitor 模式下断点和观察点必须与数据无关；不支持外部断点和观察点；不支持暂停模式（Halt Mode）和监控模式（Monitor Mode）的混和模式；不提供范围功能；断点和观察点的发生仅仅依赖于以下条件：

(1) 指令地址或数据地址。

(2) 外部观察点条件位（EXTERN0, EXTERN1）。

(3) 用户或特权模式访问（nTRANS）。

(4) 观察点的数据读写（nRW）

(5) 访问观察点的数据宽度（MAS[1:0]）

3.2.3 断点和观察点

软件调试中最基本的功能是断点和观察点的设置。在正常情况下一般通过替换中断位置的代码，设置一条软件中断指令的方式实现断点功能；通过硬件指令设置观测点。而 ARM7TDMI 由于内置了 EmbeddedICE，利用其提供的硬件监测功能，可以很方便的实现断点和观察点功能。通过对 EmbeddedICE 编程，在不改动被调试文件代码的基础上实现断点和观察点设置，可以进行一些常规方式无法实现的固化调试等功能。下面将介绍如何对 EmbeddedICE 编程实现断点和观察点。

3.2.3.1 断点

断点可以分成硬件断点和软件断点两类：

(1) 硬件断点

硬件断点一般通过监视地址的方式实现。它不受存储器种类限制，可以在代码的任何位置设置。但是由于受 EmbeddedICE 实时监控单元数量的限制，系统中最多有 2 个硬件断点同时存在。

对观察点单元进行设置，进而产生硬件断点的过程如下：

1）对观察点地址值寄存器编程，将产生硬件断点的指令地址写入该寄存器。

2）如果运行在 ARM 状态,需要设置观察点地址屏蔽寄存器的位[1:0]为 b11。如果运行在 THUMB 状态，则将观察点地址屏蔽寄存器的位[1:0]设置为 b01。

3）如果需要数据相关的硬件断点（数据相关是指除了需要匹配地址值外，还要匹配该地址处的指令代码），则编程观察点数据寄存器；且要确保将数据屏蔽寄存器的全部位清零（0x00000000）。如果不需要数据相关，则将数据屏蔽寄存器写为 0xFFFFFFFF（全部置位）。

4）观察点控制寄存器 nOPC 位清零。

5）观察点控制屏蔽寄存器 nOPC 位清零。

6）当需要区分用户模式和非用户模式时，编程 nTRANS 的值和相应的屏蔽寄存器。

7）若需要，则以同样的方法编程 EXETERN，RANGE 和 CHAIN 位。

8）将所有未使用的控制值的屏蔽位置位。

注意：在监控模式（Monitor Mode）下，在改变 EmbeddedICE 的寄存器之前，必须设置 EmbeddedICE 禁止位，在编程完成后在清除该位。

(2) 软件断点

软件断点可以通过监视从任何地址读取的位图实现。因此 EmbeddedICE 可以支持任何数量的软件断点。软件断点一般只能在 RAM 中设置，因为要产生软件断点处的指令需要被这个特定的位图替换，从而在该处产生一个软件断点。

通过编程观察点单元，可以实现在处理器读取特定的位图时会引起软件断点。操作过程如下：

1）将观察点地址屏蔽寄存器全部置位（0xFFFFFFFF），使任何地址处的特定位图都会引起软件断点。

2）将该特定位图写到观察点数据寄存器。如果处于 **THUMB** 模式，则将高 16 位和低 16 位重复写入特定的 16 位位图。例如，特定的位图是 0xDEEE，则编程 0xDEEEDEEE 到数据寄存器。

3）将观察点数据屏蔽寄存器全部清零（0x00000000）。

4）观察点控制寄存器 nOPC 位清零。

5）观察点控制屏蔽寄存器 nOPC 位清零。

6）当需要区分用户模式和非用户模式时，编程 nTRANS 的值和相应的屏蔽寄存器。

7）若需要，则以同样的方法编程 EXETERN，RANGE 和 CHAIN 位。

8）将所有未使用的控制值的屏蔽位置位。

设置软件断点的操作过程可以分为设置软件断点和清除软件断点两个步骤：

1) 设置

a) 读取要设置软件断点处地址的指令，并将其保存到别处。

b) 在该地址处写入引发软件断点的特定位图。

2) 清除

如果要清除断点，则恢复该地址原来的指令。

3.2.3.2 观察点

观察点的实现方式和硬件断点的实现方式非常相似：通过数据访问时产生观

察点实现。它同样不受存储器种类限制，可以在代码的任何位置设置。但是由于受EmbeddedICE实时监控单元数量的限制，系统中最多有2个观察点同时存在。

通过编程观察点单元，可以实现在处理器读取特定地址的数据时会引起观察点触发。操作过程如下：

(1) 用数据访问的地址来编程观察点单元地址值寄存器。

(2) 将观察点地址屏蔽寄存器的值全部清零。

(3) 如果需要数据相关（除了要匹配地址值外，还要匹配读或写的数据值）的观察点，要确保清除观察点单元数据屏蔽寄存器。如果不需要数据相关，则要将数据屏蔽寄存器全部置位。

(4) 按需要编程观察点控制寄存器，即：

1) nOCP=1；

2) nRW=0（读）或nRW=1（写）；

3) 根据数据宽度设置MAS[1:0]；

(5) 编程控制屏蔽寄存器：

1) nOCP=0；

2) nRW=0；

3) 根据数据宽度设置MAS[1:0]＝0；

4) 所有其他位置位；

(6) 当需要区分用户模式和非用户模式时，编程nTRANS的值和相应的屏蔽寄存器。

(7) 若需要，则以同样的方法编程EXETERN，RANGE和CHAIN位。

第 4 章 LambdaICE 的设计与实现

本章主要从 LambdaICE 的总体设计、运行设计、接口设计和核心代码的实现四方面，由粗到细的方式来介绍 LambdaICE 的实现原理。

4.1 总体设计

本节主要从 LambdaICE 的系统组成、运行环境和系统结构三方面对 LambdaICE 的总体设计进行介绍。因为是在一个较高的层次来介绍 LambdaICE，故在本节中不会关心实现细节方面的内容。

4.1.1 交叉调试系统组成

交叉调试是嵌入式调试的一个显著标志，通常由宿主机、目标机和调试代理 3 个部分组成。

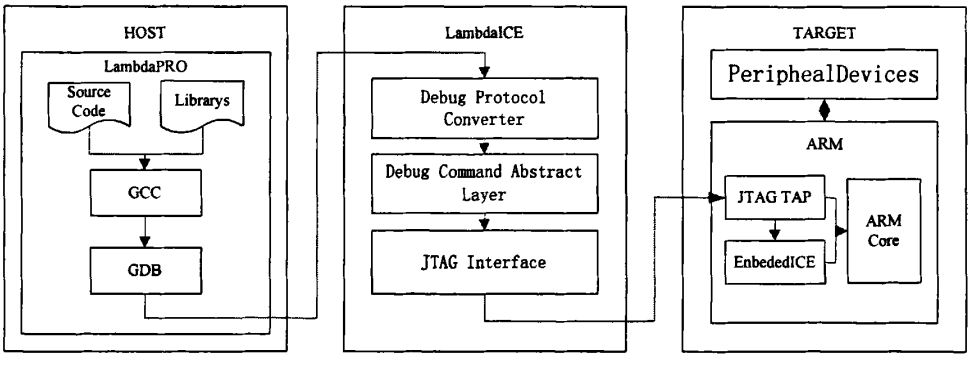

图 4-1 交叉调试环境结构图

如图 4-1 所示，由 LambdaICE 组成的交叉调试环境由如下 3 个部分组成：

（1）宿主机（HOST）：宿主机就是平常使用的装有 Windwos 操作系统的 PC 机，并在上面运行由北京科银京成技术有限公司开发的 LambdaPRO 集成开发环境。

（2）LambdaICE 硬件仿真器：LambdaICE 是片上调试方式（OCD）实现的硬件仿真器。支持 ARM7 和 ARM9 提供的片上调试功能；提供网络和 USB 两种方式与宿主机进行通信。

（3）目标板（TARGET）：支持任何采用 ARM7 或 ARM9 为内核的处理器目标板，通过 JTAG 口与 LambdaICE 硬件仿真器连接。

4.1.2 LambdaICE 的运行环境

(1) 目标机端环境

LambdaICE 支持任何一款支持以 ARM7 或 ARM9 为处理器内核的目标板，唯一的硬件要求是目标板要提供标准 14 针或者 20 针 JTAG 接口。

(2) 仿真调试器环境

LambdaICE 仿真调试器以科银京成的实时操作系统 DeltaOS 为基础，运行在 ATMEL 公司生产的 AT91RM9200 处理器上（ARM9 内核）。为了获得较好的仿真效果，LambdaICE 的硬件配置较高：主频 180M，8MB 的 SDRAM，1MB 的 NorFlash 用来作为程序存储，10/100M 以太网接口，全速 USB2.0 接口和 RS-232 串行接口。

(3) 主机端环境

主机端运行由北京科银京成技术有限公司开发的基于 Eclipse 架构的专业嵌入式软件集成开发环境 LambdaPRO。LambdaPRO 以 LambdaTOOL 为集成开发框架，支持多种嵌入式实时操作系统，包括科银京成的实时操作系统 DeltaOS、开源嵌入式实时操作系统 eCos 等。针对不同版本的 LambdaPRO，所支持的嵌入式操作系统会有所不同。

(4) 连接方式

宿主机中的 LambdaPRO 可以通过两种方式与 LambdaICE 连接：一是以太网接口，通过这种方式可以实现远程调试功能；另一种是通过 USB 接口，在这种方式下可以实现即插即用，非常方便用户的使用。以上两种方式都可以得到高速的下载调试效率。

4.1.3 LambdaICE 系统结构

LambdaICE 是一个硬件仿真调试器，在交叉调试系统中充当调试代理的角色，根据宿主机发送的调试命令，完成对目标的调试控制。

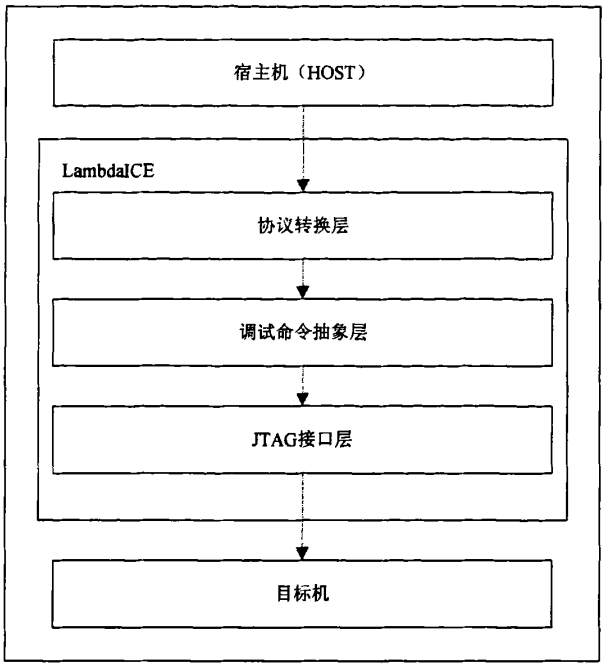

图 4-2 LambdaICE 系统结构图

如图 4-2 所示，LambdaICE 由 3 个模块组成：

(1) JTAG 接口（JTAG Interface）

(2) 调试命令抽象层（Debug Command Abstract Layer）

(3) 协议转换器（Debug Protocol Converter）

4.1.3.1 JTAG 接口层

JTAG 接口是 LambdaICE 最底层模块。它直接和目标机的 JTAG 通讯，向上层模块提供最基本的 JTAG TAP 状态机操作命令，包括状态机复位、TAP 状态的转换、JTAG 命令的执行和 JTAG 数据寄存器的读/写等操作。LambdaICE 对 ARM 内核的任何操作最终都将转化为对状态机的一系列操作。

事实上，本模块是对 JTAG 的 TAP 状态机操作的封装。按照 TAP 状态机的定义，状态机只涉及各个状态之间的切换关系，与被扫描对象的实现方式无关。也就是说，JTAG 命令接口层如果基于上述原则设计，那么与 ARM 内核的内部实现将没有直接联系，该接口也无需任何改动就可以很方便的应用到其他的 JTAG 的设备上面，如 PowerPC 构架的目标板上等。

4.1.3.2 调试命令抽象层

调试命令抽象层是 LambdaICE 的核心模块。它在底层 JTAG 接口的基础上，将调试需要的基本命令再进行了一层封装。该模块和 ARM 内核的联系就非常紧

密，是直接通过 JTAG 口对 ARM 内核中的各条扫描链、EmbeddedICE 等设备执行读写操作。

抽象层包括实现调试代理需要的基本功能：

(1) 寄存器读写

(2) 内存读写

(3) 运行控制命令

硬件仿真方式实现上述功能与软件监控调试的方式实现有很大的差别，这是由于两者的实现机理不同导致。下面以软件和硬件仿真方式实现一条读取 r3 寄存器的命令为例，说明两者的实现机制的差别。

由于调试代理运行于特权模式，能够执行全部指令，所以通过软件方式实现读取任何寄存器的操作非常简单：系统在进入调试代理服务程序之后会立即执行现场保护操作，保存当前执行程序的上下文。调试代理如果要获得 r3 寄存器的信息只需要在现场保护的记录中查找上下文的 r3 寄存器信息即可。

ARM 内核没有提供直接访问寄存器的扫描链，但是扫描链 0 和 1 能够扫描处理器用于读写操作的 32 数据总线。要获得 r3 寄存器的数据，必须设法先将 r3 寄存器的数据打到数据总线上面，再通过扫描数据总线的方式获得。通常的做法是：首先在数据总线上打入一条读 r3 寄存器的指令，然后执行该指令。当让处理器执行完该指令停止后，r3 寄存器的数据就在数据总线上，这样就可以通过访问扫描链 0 或者 1 获得。

从上面的例子可以看出硬件仿真调试方式获取数据的一个特点就是：所有数据（寄存器数据或者内存数据）都是通过执行读写指令（寄存器、内存读写）将数据打到数据总线上获得。

调试命令抽象层是整个 LambdaICE 的 3 个模块中结构最复杂，功能最灵活的部分，代码量占到 LambdaICE 代码总量的 50%以上。

4.1.3.3 协议转换层

协议转换层是协议转发组件，负责接收主机端 LambdaPRO 发送过来的调试命令，然后调用调试命令抽象层的相应接口，来对目标机进行调试控制。

和调试命令抽象层对应，协议转换层也提供 3 种类型的调试命令：

(1) 数据读写命令：包括内存读写和寄存器读写命令

(2) 运行控制命令：包括状态查询命令和控制命令

(3) 辅助命令：一些为了方便用户使用而添加的功能

在调试命令抽象层基础上实现的协议转换层，无需关心底层硬件平台的差异，这样便于实现一个通用的协议转换层。

4.2 运行设计

LambdaICE 调试环境是一个由多个组件组成，主机端调试器、仿真器和目标机端协同工作，并接受用户命令的复杂的交叉调试环境。用户应该如何使用 LambdaICE，以及 LambdaICE 应该如何运行都是一个非常重要的问题，不仅关系到最终完成的 LambdaICE 在操作上是否方便高效，更重要的是，关系到 LambdaICE 的设计与实现。本节从 LambdaICE 总体运行流程出发，依次介绍本文研究的重点：读内存、写内存、上下文保护和上下文恢复的操作流程。其他操作流程不是本文关心的重点，这里不再赘述。

4.2.1 LambdaICE 总体运行流程

软件的总体运行流程是在一个非常高的层面来解析软件的运行过程，从中可以清楚的了解软件是如何运作的。

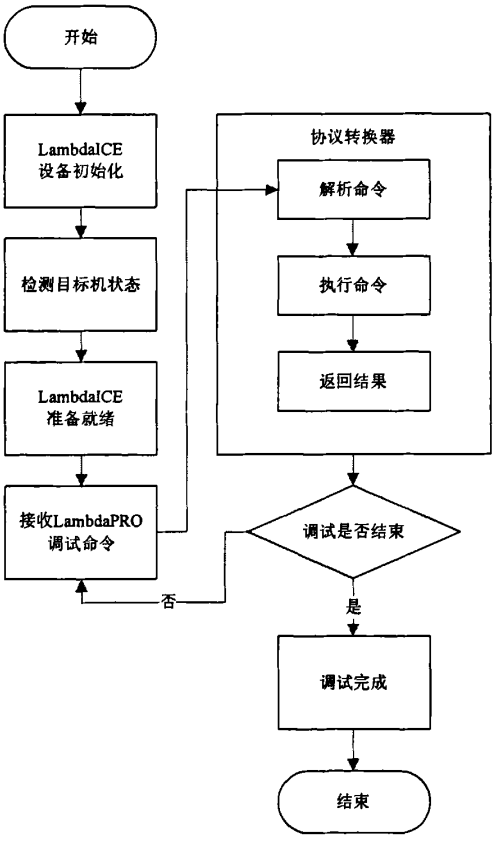

图 4-3 LambdaICE 总体运行流程图

如图 4-3 所示，LambdaICE 的总体运行流程如下：

（1）LambdaICE 上电之后进行自检、设备初始化；

（2）检查目标板内核版本信息（通过查询 JTAG 设备 ID_Code）。目前 LambdaICE 支持 ARM7 和 ARM9 系列内核，如果检测到是其他内核就通知调试器发现不支持的内核；

（3）待全部初始化工作完成之后，LambdaICE 进入就绪状态，等待调试器命令；

（4）接收到调试器命令后交给协议处理程序；

（5）协议转换器将调试器命令转换成相应的抽象层命令执行；

（6）将执行结果重新封装后返回调试器；

（7）检查调试过程是否结束，否则回到状态(3)；

（8）调试结束。

4.2.2 LambdaICE 读内存流程

读目标机内存是调试过程中运用最为频繁的动作之一，它的效率是影响调试效率的主要因素。

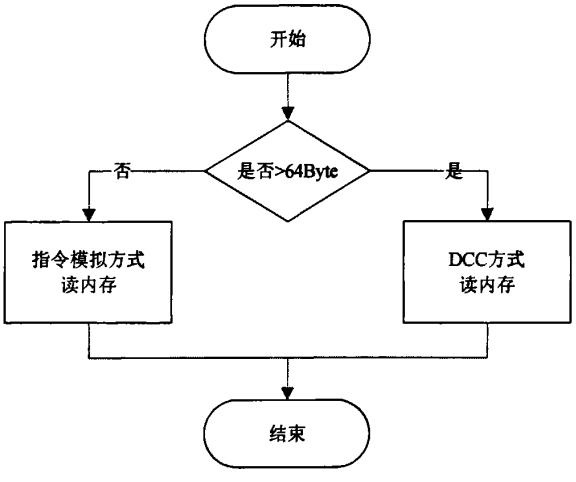

图 4-4 LambdaICE 读内存流程图

如图 4-4 所示，LambdaICE 在读内存时，首先要判断要读写的大小是否大于 64 个字节。如果是大于 64 字节，则采用 DCC 方式读内存，否则采用指令模拟方式读写内存。

4.2.2.1 指令模拟方式读内存流程

指令模拟方式读内存是当今硬件调试器中运用最为广泛的读目标机内存方式，它的特点是实现简单。

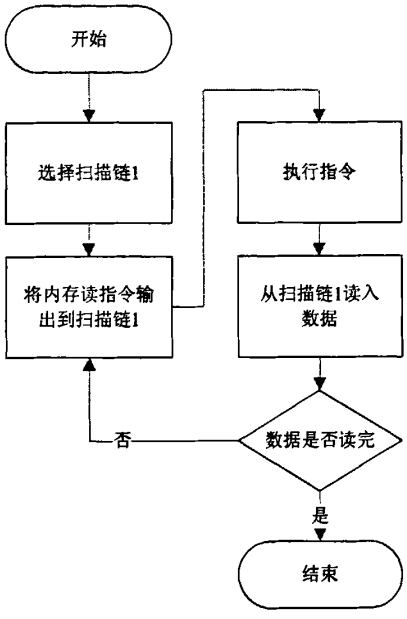

图 4-5 指令模拟方式读内存流图

LambdaICE 通过指令模拟方式读内存的过程如图 4-5 所示：

（1）在使用指令模拟方式读写内存时，首先要选中 ARM 内核提供的扫描链 1；

（2）将读内存指令通过 ARM 内核提供的 JTAG 口传输到 ARM 内核的扫描链 1 上；

（3）当读内存指令输出到扫描链 1 上后，ARM 内核会自动执行这条指令；

（4）在读内存指令的执行过程中，ARM 内核会将读到的数据放到数据总线上，故要通过扫描链 1 将读到的数据传送出来；

（5）如果要读多个数据，则要重复执行(2)到(4)的过程。

4.2.2.2 DCC 方式读内存流程

DCC 通道是 ARM 内核专门为调试而设计的一种通信方式，通过 DCC 通道读目标机内存，可以大大提高读内存的速度，从面提高调试效率。

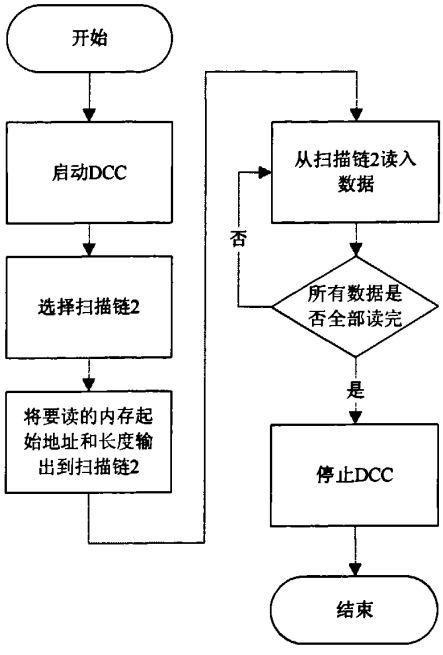

图 4-6 DCC 方式读内存流程图

LambdaICE 通过 DCC 方式读内存的过程如图 4-6 所示：

（1）当通过 DCC 方式读内存时，首先要在目标机中启动 DCC Handler；

（2）因 DCC 相关的寄存器是通过 ARM 内核提供的扫描链 2 来访问的，故要先选中扫描链 2；

（3）根据设计好的协议，将要读的内存起始地址和长度通过扫描链 2 传给 DCC Handler；

（4）DCC Handler 接到读内存命令后就会根据得到的起始地址和长度去读内存，并把读到的数据传输到扫描链 2 上，因此要从扫描链 2 上循环地将所有数据传出；

（5）当数据全部读完后，为了执行其它调试命令，要停止 DCC Handler 的运行。

4.2.3 LambdaICE 写内存流程

写内存也是调试过程中运用最为频繁的一个动作之一，它的效率最终影响调试的效率。

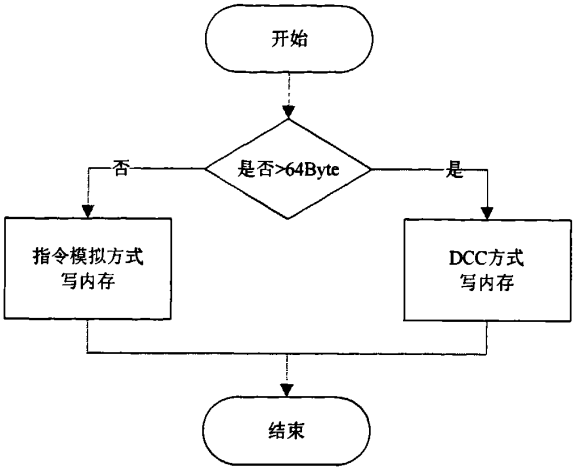

图 4-7 LambdaICE 写内存流程图

如图 4-7 所示，LambdaICE 在写内存时与读内存相似，首先要判断要写的大小是否大于 64 个字节。如果是大于 64 字节，则采用 DCC 方式写内存，否则采用指令模拟方式写内存。

4.2.3.1 指令模拟方式写内存流程

指令模拟方式写内存是当今硬件调试器中运用最为广泛的写目标机内存方式，它的特点是实现简单。

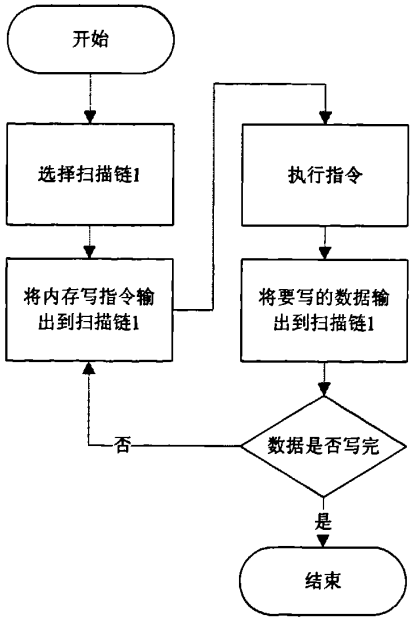

图 4-8 指令模拟方式写内存流程图

LambdaICE 通过指令模拟方式写内存的过程如图 4-8 所示：

（1）在使用指令模拟方式写内存时，首先要选中 ARM 内核提供的扫描链 1；

（2）将写内存指令通过 ARM 内核提供的 JTAG 口传输到 ARM 内核的扫描链 1 上；

（3）当写内存指令输出到扫描链 1 上后，ARM 内核会自动执行这条指令；

（4）在写内存指令的执行过程中，ARM 内核会先把数据从数据总线上读入，再写到内存中去，故要通过扫描链 1 将数据传送到总线上去；

（5）如果要读多个数据，则要重复执行(2)到(4)的过程。

4.2.3.2 DCC 方式写内存流程

DCC 通道是 ARM 内核专门为调试而设计的一种通信方式，通过 DCC 通道写目标机内存，可以大大提高写内存的速度，从而提高调试效率。

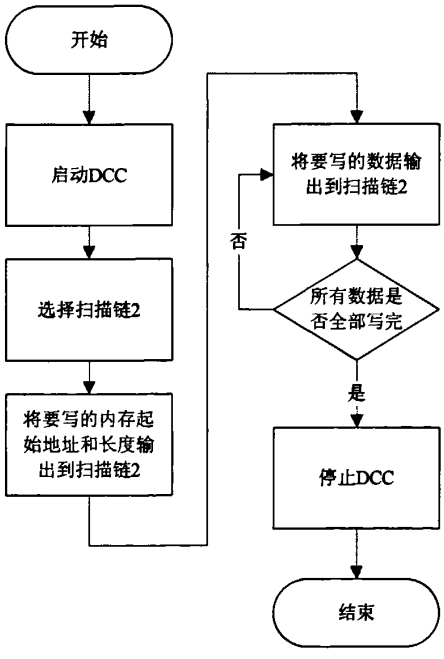

图 4-9 DCC 方式写内存流程图

LambdaICE 通过 DCC 方式写内存的过程如图 4-9 所示：

（1）当通过 DCC 方式写内存时，首先要在目标机中启动 DCC Handler；

（2）因 DCC 相关的寄存器是通过 ARM 内核提供的扫描链 2 来访问的，故要先选中扫描链 2；

（3）根据设计好的协议，将要写的内存起始地址和长度通过扫描链 2 传给 DCC Handler；

（4）DCC Handler 接到写内存命令后就会根据要写的内存长度不断地从 DCC 的数据寄存器中读数据，并连续不断地从起始地址开始写入内存，因此要把要写入的数据连续不断地输出到扫描链 2 上；

（5）当数据全部传输完后，为了执行其它调试命令，要停止 DCC Handler 的运行。

4.2.4 利用 STM 指令实现高效的上下文保护流程

在目标机停止运行后的首要动作就是保护目标机的上下文，因为它的使用频率也是非常高的，故它的效率对最终的调试效率影响也是很大的。

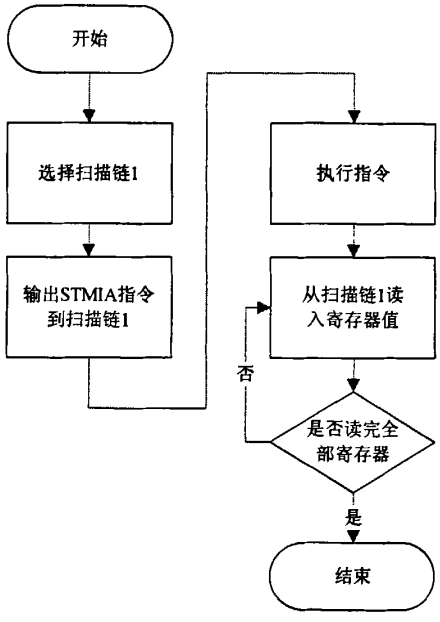

图 4-10 利用 STM 指令实现高效的上下文保护流程图

LambdaICE 通过 STM 指令实现高效的上下文保护流程如图 4-10 所示：

（1）因为此方式进行上下文保护的基本原理也是基于指令模拟的，故首先要选中 ARM 内核提供的扫描链 1；

（2）将 STMIA 指令通过 ARM 内核提供的 JTAG 口传输到 ARM 内核的扫描链 1 上；

（3）当 STMIA 指令输出到扫描链 1 上后，ARM 内核会自动执行这条指令；

（4）在 STMIA 指令的执行过程中，ARM 内核会先把所有寄存器的值连续地传送到总线上，因此要不断地从扫描链 1 读入寄存器的数据，直到所有要读的寄存器读完。

4.2.5 利用 LDM 指令实现高效的上下文恢复流程

在目标机恢复运行前的重要工作就是恢复目标机停止时的上下文，因为每次恢复运行前都要恢复目标机的上下文，故它的效率对最终的调试效率影响也是很大的。

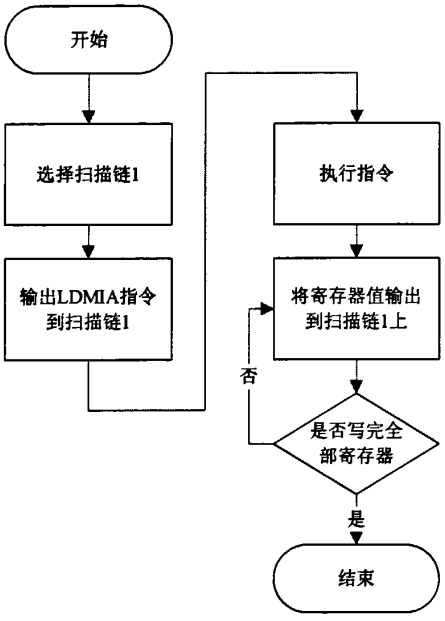

图 4-11 利用 LDM 指令实现高效的上下文恢复流程图

LambdaICE 通过 LDM 指令实现高效的上下文恢复流程如图 4-11 所示：

（1）因为此方式进行上下文恢复的基本原理也是基于指令模拟的，故首先要选中 ARM 内核提供的扫描链 1；

（2）将 LDMIA 指令通过 ARM 内核提供的 JTAG 口传输到 ARM 内核的扫描链 1 上；

（3）当 LDMIA 指令输出到扫描链 1 上后，ARM 内核会自动执行这条指令；

（4）在 LDMIA 指令的执行过程中，ARM 内核会连续地从数据总线上读入数据，并把读到的数据写入相应的寄存器中，因此要不断地把要恢复的寄存器值输出到扫描链 1 上，直到所有要恢复的寄存器全部写完。

4.3 接口设计

本节主要就 LambdaICE 的 3 个模块的主要对外接口 API 进行简单介绍。通过对模块的每一个对外接口的介绍，可以清晰的了解该模块的每一个功能。

4.3.1 JTAG 接口层接口设计

JTAG 接口层的主要接口如表 4-1 所示：

表 4-1 JTAG 接口层主要接口表

接口序号	接口名称	功能描述
1	fnJTAG_TargetReset	复位目标机处理器
2	fnJTAG_TAPReset	复位目标机 TAP
3	fnJTAG_SetTapStat	设置 TAP 状态机到另一个状态
4	fnJTAG_GetTapStat	获得当前 TAP 状态机状态
5	fnJTAG_ExchangeData	JTAG 接口数据交互
6	fnJTAG_ExchangeTAPInst	JTAG 接口指令交互

4.3.2 调试命令抽象层接口设计

调试命令抽象层主要接口如表 4-2 所示：

表 4-2 调试命令抽象层主要接口表

接口序号	接口名称	功能描述
1	fnDCAL_SetScanChain	设置扫描链
2	fnDCAL_IDCode	读取 IDCode
3	fnDCAL_ICE_Read	读取扫描链 2
4	fnDCAL_ICE_Write	写扫描链 2
5	fnDCAL_StopCore	停止内核
6	fnDCAL_Restart	内核重新运行
7	fnDCAL_StartDCC	启用 DCC

8	fnDCAL_ExecuteInst	执行处理器指令
9	fnDCAL_SaveContents	保存处理器当前上下文
10	fnDCAL_RestoreContents	恢复处理器当前上下文
11	fnDCAL_ReadMemByWord	指令模拟字方式读内存
12	fnDCAL_ReadMemByHWord	指令模拟半字方式读内存
13	fnDCAL_ReadMemByByte	指令模拟字节方式读内存
14	fnDCAL_WriteMemByWord	指令模拟字方式写内存
15	fnDCAL_WriteMemByHWord	指令模拟半字方式写内存
16	fnDCAL_WriteMemByByte	指令模拟字节方式写内存
17	fnDCAL_CheckPower	检查目标机是否上电
18	fnDCAL_ResetTarget	复位目标机
19	fnDCAL_DCCWriteMemByWord	DCC 字方式写内存
20	fnDCAL_DCCWriteMemByHWord	DCC 半字方式写内存
21	fnDCAL_DCCWriteMemByByte	DCC 字节方式写内存
22	fnDCAL_DCCReadMemByWord	DCC 字方式读内存
23	fnDCAL_DCCReadMemByHWord	DCC 半字方式读内存
24	fnDCAL_DCCReadMemByByte	DCC 字节方式读内存
25	fnDCAL_EnCache	打开 Cache
26	fnDCAL_DisCache	关闭 Cache
27	fnDCAL_CleanCache	清 Cache
28	fnDCAL_SetWPTRegs	设置观察点寄存器

4.3.3 调试协议转换层接口设计

调试协议转换层主要接口如表 4-3 所示：

表 4-3 调试协议转换层主要接口表

接口序号	接口名称	功能描述
1	fnDPCL_ReadReg	读取指定寄存器
2	fnDPCL_WriteReg	改写指定寄存器
3	fnDPCL_Step	单步执行

4	fnDPCL_CheckStat	检查目标机状态
5	fnDPCL_Continue	连续执行
6	fnDPCL_StartDCC	启动 DCC
7	fnDPCL_Pause	停止处理器
8	fnDPCL_HardBkpt	设置硬件断点
9	fnDPCL_Watch	设置观察点
10	fnDPCL_TargetReset	复位目标机
11	fnDPCL_IDCode	读取 IDCode
12	fnDPCL_DCCWriteMem	DCC 方式写内存
13	fnDPCL_DCCReadMem	DCC 方式读内存
14	fnDPCL_WriteMem	指令模拟字节方式写内存
15	fnDPCL_ReadMem	指令模拟字节方式读内存

4.4 DCC Handler 的实现原理

DCC Handler 是一个运行在目标机端的用于提高 LambdaICE 对目标机内存读写速度的小程序。DCC Handler 借助 EmbeddedICE 中的调试通信通道(debug communications channel，DCC)，以一定的通信协议来与 LambdaICE 交换数据。与指令模拟方式相比，这样可以大大地提高 LambdaICE 与目标机的数据交互能力。

4.4.1 DCC Handler 通信协议

根据 DCC 的硬件实现原理，每次通信最大能传输 32Bit 数据。据此，DCC Handler 与 LambdaICE 交互数据的协议设计如下：

(1) 读内存

第 1 个 32 位数据：Bit[31]固定为 b1，表示读内存；Bit[30-29]为读内存方式选择位，b00 表示 32 位方式，b01 表示 16 位方式，b10 表示 8 位方；Bit[28-0]为本次读内存长度。

第 2 个 32 位数据：Bit[31-0]本次要读内存的起始地址。

(2) 写内存

第 1 个 32 位数据：Bit[31]固定为 b0，表示写内存；Bit[30-29]为写内存方式选择位，b00 表示 32 位方式，b01 表示 16 位方式，b10 表示 8 位方；Bit[28-0]为本次写内存长度。

第 2 个 32 位数据：Bit[31-0]本次要写内存的起始地址。

第 3－N 个 32 位数据：为要写入的数据。

4.4.2 DCC Handler 实现

DCC Handler 代码采用汇编编写，且要与目标机无关，编译成二进制文件，为作一个数据块放在 LambdaICE 中。因目标机有大小端，故得要有大端方式和小端方式两个二进制文件。当要调试目标机时，首先判断目标机是大端还是小端，然后把相应数据（DCC Handler 代码）通过扫描链 1 下载到目标机中，并让其运行。这样 LambdaICE 就可以通过调试通信通道(Debug Communications Channel，DCC)实现对目标机内存的快速读写。DCC Handler 的代码如下：

```
.macro ReceivePacket   reg
RLoop:
    mrc p14,0,r5,c0,c0,0
    tst r5,#0x1
    beq RLoop
    mrc p14,0,\reg,c1,c0,0
.endm

.macro SendPacket reg
SLoop:
    mrc p14,0,r5,c0,c0,0
    tst r5,#0x2
    bne SLoop
    mcr p14,0,\reg,c1,c0,0
.endm
```

在如上所示代码中是两个宏函数，第一个是通过 DCC 通道接收数据的宏函数，第二个是通过 DCC 通道发送数据的宏函数。

```
_start:
    /*Receive packet header*/
    ReceivePacket   r0
    ReceivePacket   r1

    /*Parse packet header*/
    mov     r2, r0, lsr #29
    bic  r0, r0, #0xE0000000
    tst   r2, #4
    beq      ReadMem

    /*WriteMem*/
WriteMem:
    cmp     r2, #5
    blt   WriteWords
    beq      WriteHWords
    cmp     r2, #6
    beq      WriteBytes
    swinv      0x00FFFFFF
    nop
    b.

    /*ReadMem*/
ReadMem:
    cmp      r2, #1
    blt      ReadWords
    beq      ReadHWords
    cmp      r2, #2
    beq      ReadBytes
    swinv      0x00FFFFFF
    nop
    b.
```

如上所示代码是 DCC Handler 的入口，主要完成的功能是接收 LambdaICE 发送给目标机的通信协议，并解析，然后跳转到相应的处理接口中。

```
/*WriteWords*/
WriteWords:
    ReceivePacket   r3
    str     r3, [r1]
    add     r1, r1, #4
    sub     r0, r0, #4
    cmp     r0, #0
    bgt     WriteWords
    b     _start

    /*WriteHWords*/
WriteHWords:
    ReceivePacket   r3
    strh    r3, [r1]
    add     r1, r1, #2
    sub     r0, r0, #2
    cmp     r0, #0
    bgt     WriteHWords
    b     _start

    /*WriteBytes*/
WriteBytes:
    ReceivePacket   r3
    strb    r3, [r1]
    add     r1, r1, #1
    sub     r0, r0, #1
    cmp     r0, #0
    bgt     WriteBytes
    b     _start
```

如上所示代码是写内存接口。WriteWords 接口是以 32 位方式写内存；WriteHWords 接口是以 16 位方式写内存；WriteBytes 接口是以 8 位方式写内存。

```
/*ReadWords*/
ReadWords:
    ldr     r3, [r1]
    SendPacket  r3
    add     r1, r1, #4
    sub     r0, r0, #4
    cmp     r0, #0
    bgt     ReadWords
    b       _start

    /*ReadHWords*/
ReadHWords:
    ldrh    r3, [r1]
    SendPacket  r3
    add     r1, r1, #2
    sub     r0, r0, #2
    cmp     r0, #0
    bgt     ReadHWords
    b       _start

    /*ReadBytes*/
ReadBytes:
    ldrb    r3, [r1]
    SendPacket  r3
    add     r1, r1, #1
    sub     r0, r0, #1
    cmp     r0, #0
    bgt     ReadBytes
    b       _start
```

如上所示代码是读内存接口。WriteWords 接口是以 32 位方式读内存；WriteHWords 接口是以 16 位方式读内存；WriteBytes 接口是以 8 位方式读内存。

第 5 章 LambdaICE 的测试

对 LambdaICE 的测试主要进行了两个阶段的测试，即单元测试和系统测试。

5.1 单元测试

由于整个 LambdaICE 使用 GNU 交叉环境开发，在单元测试中我们采用 GammaCP 进行覆盖测试。GammaCP 是一个运行在 Windows 平台下的 gcc 程序覆盖测试工具，它提供以下功能：

(1) 语句覆盖测试；

(2) 分支覆盖测试；

(3) 决策覆盖测试。

经过不断地增加和优化测试案例，LambdaICE 的 3 个模块的覆盖率达到了 95%以上。最后尚未被覆盖的代码主要是一些错误处理代码，构造案例来完全覆盖这些代码非常困难。

5.2 系统测试

系统测试从测试设计到测试完成，历经 3 个多月。测试主要分为三个大的方面来进行测试：功能测试，联机测试和回归测试。在功能测试中，被测对象的设计较为充分，设计的仿真调试案例比较完善，被测对象的规模也较大；在联机测试中，不仅测试调试器的常规操作，也进行许多非常规操作的测试，应该说本次测试是较为充分的。在回归测试中将前期发现并解决的所有问题设计成新的测试案例，重新对 LambdICE 完整地测试一遍确保问题彻底解决。

本次测试的前期发现 LambdaICE 存在较多不完全正确的地方。通过与研发人员的不断交互，在回归测试中，以前存在的绝大部分问题都得到解决。整个回归测试过程中，LambdaICE 的运行非常稳定，经过修改后的 LambdaICE 能够正确的实现调试功能。

本次测试耗费较长时间，主要按照测试计划、测试设计、测试实现和测试执行四个步骤来进行。测试耗费的时间主要在测试的实现和测试执行上面。在测试实现过程中，由于本次测试计划中涉及的被测对象规模较大，所以进行了自动生成被测对象和测试案例以及自动生成测试报告的设计工作，该工作耗费了较长时间。在测试执行过程中，由于被测对象的规模较大，加上测试的功能点较多，也占用了整个测试过程中的很多时间。

5.3 测试结果

本文研究的重点是利用 DCC 通道提高 ARM 硬件仿真器 LambdaICE 读写目标机内存的效率，故测试结果中启用 DCC 通道后 LambdaICE 读写目标机内存速度成为了本文关注的焦点，表 5-1 是测试结果。

表 5-1 读写内存速度测试结果

目标板类型	读写内存方式	速度
ARM7	指令模拟方式	130 KByte/S
ARM7	DCC 方式	690 KByte/S
ARM9	指令模拟方式	150 KByte/S
ARM9	DCC 方式	700 KByte/S

对比表 5-1 中指令模拟方式和 DCC 方式读写内存的速度可知，当启用 DCC 通道后，LambdaICE 读写目标机内存的效率大大提高，达到了本文研究的目标。

通过对 LambdaICE 严格的单元测试和系统测试之后，LambdaICE 的质量和可靠性有一定的保证，我们对 LambdaICE 也有了可以度量的信心。当然无论测试有多么全面，肯定会有遗留的 bug 没有发现，在以后的工作中我们将继续跟踪 bug，不断完善 LambdaICE。

结 论

嵌入式系统开发是当今计算机软件发展的一个热点。随着嵌入式硬件技术的发展，嵌入式应用的不断增长以及嵌入式系统复杂性不断提高，要求嵌入式软件的规模和复杂性也不断提高，嵌入式软件的质量和开发周期对产品的最终质量和上市时间起到决定性的影响，嵌入式软件调试工具的效率成为了人们关注的重点。为此，本课题利用 ARM 处理器上自带的 EmbeddedICE 调试模块，研究实现了一个高效的 ARM 硬件仿真调试器。

从 5.3 节的测试结果中可以看出，当打开 DCC 通道后，ARM7 和 ARM9 类型的目标板内存读写速度分别为 690KByte/S 和 700KByte/S，而采用传统的指令模拟方式的内存读写速度仅为 130KByte/S 和 150KByte/S。由此可知，通过 DCC 通道对目标机进行内存读写，可以大大提高调试效率，达到了本文研究的目标。

本文就创新点而言主要有以下几点：

（1）在进行大量数据的内存读写时，采用了 DCC 通道来进行数据传输，这样大大提高了调试器的内存读写速度；

（2）在保护或恢复现场时（内核寄存器），采用了批量数据存储指令，这样极大地加快了停止和恢复运行的时间。

目前，LambdaICE 对多内核的支持还不十分完善（只支持 ARM7 和 ARM9 系列处理器）。相信，随着我们对硬件调试技术研究的不断深入，能够在不久的将来推出一个功能更加强大，更加稳定的硬件调试器，为嵌入式应用的开发，提供一把利剑。

致 谢

本文是在导师洪志全教授的悉心指导下完成的，他渊博的知识、丰富的实践经验、严谨的治学态度、精益求精的工作作风、对学科发展方向的敏锐眼光和对科学的献身精神给予了我极大的启迪和引导，他是我终生学习的榜样。洪老师是我进入嵌入式领域的领路人，而且在我的课程学习、研究方向、研究方法、论文写作等方面都进行了精心的指导。他对我的严格要求，使我对科学研究的精神、方法和内在规律有了非常深刻的领会，这些收获是我今后工作和继续学习的重要基础。洪老师除了在学习上给予我帮助，在平时的工作中也潜移默化地影响着我，使我明白了诚实、正直的为人道理和踏踏实实、有始有终做好每一件事的处事态度。在此，我向洪老师致以最诚挚的感谢和最崇高的敬意！

我的周围是一群风华正茂的有志青年，终日沉溺于学术，偶尔游历于山水。指点江山直抒胸臆，青梅煮酒，纵论英雄。他们永远是我高歌猛进的力量之源。谢谢你们，我亲爱的同学！

感谢我的父母和家人，在我最困难的时候，他们总会在生活上给我无微不致的关心和精神上的鼓励，还有他们对我的学业的关心和理解，常令我感动不已。

最后，但绝非不重要，要衷心感谢为评阅本论文而付出辛勤劳动的各位专家和学者，在我即将离开母校之际，他们所提的宝贵意见和诚恳批评将使我受益匪浅。

参考文献

[1] IEEE. IEEE Standard Test Access Port and Boundary-Scan Architecture[R]. IEEE, 1990.
[2] ARM 公司. ARM Architecture Reference Manual[R]. ARM 公司，2005.
[3] ARM 公司. ARM7TDMI Technical Reference Manual[R]. ARM 公司，2004.
[4] ARM 公司. ARM7TDMI-S Technical Reference Manual[R]. ARM 公司，2001.
[5] ARM 公司. ARM710T Technical Reference Manual[R]. ARM 公司，2004.
[6] ARM 公司. ARM720T Technical Reference Manual[R]. ARM 公司，2004.
[7] ARM 公司. ARM740T Technical Reference Manual[R]. ARM 公司，2004.
[8] ARM 公司. ARM9TDMI Technical Reference Manual[R]. ARM 公司，2000.
[9] ARM 公司. ARM920T Technical Reference Manual[R]. ARM 公司，2001.
[10]ARM 公司. ARM940T Technical Reference Manual[R]. ARM 公司，2000.
[11]ARM 公司. The ARM-THUMB Procedure Call Standard[R]. ARM 公司，2000.
[12]ARM 公司. ARM PrimeCellTM Vectored Interrupt Controller(PL190)[R]. ARM 公司，2000.
[13]周立功. ARM 嵌入式系统基础教程[M]. 北京：北京航空航天大学出版社，2005.30-71.
[14]周立功. ARM 微控制器基础与实战[M]. 北京：北京航空航天大学出版社，2003.1-33.
[15]王田苗. 嵌入式系统设计与实例开发[M]. 北京：清华大学出版社，2003.
[16]杜春雷. ARM 体系结构与编程[M]. 北京：清华大学出版社，2003.
[17]谭浩强. C 语言程序设计教程[M]. 北京：高等教育出版社，1997.
[18]桑楠. 嵌入式系统原理及应用开发技术[M]. 北京：北京航空航天大学出版社，2002.
[19]罗蕾. 嵌入式实时操作系统及应用开发[M]. 北京：北京航空航天大学出版社，2005.
[20]Jean J.Labrosse. 嵌入式实时操作系统 uc / OS-II(邵贝贝译)[M]. 北京：北京航空航天大学出版社，2003.34-281.
[21]李芳敏. VxWorks 高级程序设计[M]. 北京：清华大学出版社，2004.
[22]孔祥营,柏桂枝. 嵌入式实时操作系统 VxWorks 及其开发环境 Tornado[M]. 北京：中国电力出版社，2002.23-211.
[23]Steve Furber 著,田译等译. ARM SOC 体系结构[M]. 北京：北京航空航天大学出版社，2002.
[24]Jean J.Labrosse. Embedded Systems Building Blocks,2E[M]. 北京：机械工业出版社，2002.
[25]陈渝,李明,杨晔. 源码开放的嵌入式系统软件分析与实践——基于 SkyEye 和 ARM 开发平台[M]. 北京：北京航空航天大学出版社，2004.
[26]陈智育,温彦军,陈琪. VxWorks 程序开发实践[M]. 北京：人民邮电出版社，2004.
[27]毛德操,胡希明. Linux 内核源代码情景分析[M]. 浙江：浙江大学出版社，2001.
[28]赵炯. Linux 内核 0.11 完全注释[M]. 北京：机械工业出版社，2004.
[29]陈文智. 嵌入式系统开发原理与实践[M]. 北京：清华大学出版社，2005.
[30]李伯成. 单片机及嵌入式系统[M]. 北京：清华大学出版社，2005.
[31]Jonathan W.Valvano. 嵌入式微计算机系统实时接口技术[M]. 李曦，周学海，方潜生，熊悦等译.北京：机械工业出版社，2003.
[32]李华，孙晓民. MCS-51 系列嵌入式微处理器实用接口技术[M]. 北京：北京航空航天大学出版社，1993.
[33]任晓东，文博. CPLD/FPGA 高级应用开发指南[M]. 北京：电子工业出版社，2003.

[34]Wayne Wolf 著,孙玉芳,梁彬,罗保国,谢谦等译. 嵌入式计算系统设计原理[M]. 北京:机械工业出版社，2002.
[35]毛德操,胡希明. 嵌入式系统-采用公开源代码和 StrongARM/XScale 处理器[M]. 浙江:浙江大学出版社，2003.
[36]陈莉君. Linux 操作系统内核分析[M]. 北京：人民邮电出版社，2000.
[37]尹立孟. 嵌入式应用交叉调试器的设计与实现[D]. 成都：电子科技大学，2001.
[38]张彦明. 嵌入式操作系统远程调试器的研究与实现[D]. 西安：西北工业大学，2001.
[39]何先波. 嵌入式系统软件开发环境中调试器的设计与实现[D].成都：四川大学，2001.
[40]魏勇. 嵌入式交叉调试技术的研究与实现[D].成都：电子科技大学，2005.
[41]张静. 任务级调试的设计与实现[D]. 成都：电子科技大学，2003.
[42]郑家玲,张云峰. 嵌入式系统的内核载入过程分析[J]. 微型机与应用，2002，21(11):59-60.
[43]陈定君,郭晓东. 嵌入式软件仿真开发系统的研究[J].电子学报,2000,28(3):137-139.
[44]曾杰,蒋泽军,王丽芳,张彦明. 嵌入式远程调试器的设计与实现[J]. 计算机测量与控制，2005，13(7):731-733.
[45]DeltaCore 使用手册[J]. 成都：北京科银京城技术有限公司成都研发中心，2005.
[46]DeltaCore 参考手册[J]. 成都：北京科银京城技术有限公司成都研发中心，2005.

基于DCC和JTAG的ARM硬件仿真调试器的研究与实现

作者： 罗志刚
学位授予单位： 成都理工大学

相似文献(10条)

1.学位论文 左刚 基于Motorola16位微控制器的嵌入式开发系统设计及实现 2004

近年来随着嵌入式技术在各个领域的普及以及消费者对于嵌入式产品的迫切需求，使嵌入式开发吸引了越来越多的关注。但是由于一些众所周知的原因，长期以来国内的一些开发人员只好使用国外的嵌入式开发产品。这样，在提高了开发效率、缩短了开发周期的同时，导致最终产品的成本颇高。Motorola公司的16位MCU提供了一种全新的片上调试模式：后台调试模式(BackgroundMode)，并提供了相关的开发资料，这样就使开发自主的嵌入式开发系统成为可能。本文在深入分析了Motorola公司的相关资料以及第三方厂商为Motorola所设计的调试硬件的基础上，使用Motorola的16位微控制器MC689S12DP256B设计并实现了目标芯片为Motorola16位处理器的嵌入式开发系统。

本嵌入式开发系统大致分为几个模块：调试模块、编译模块和PC端主控模块。此外，本嵌入式开发系统还针对目前主流的开发工具的一些不足，提出并实现了相关的解决方案。

本文首先介绍了嵌入式调试的背景资料，主要是嵌入式调试技术的发展历史以及演变；然后介绍了与该系统相关的硬件设计，并着重阐明了基于Motorola16位微控制器的硬件调试技术的实现；之后讲述了嵌入式编译器和汇编器的相关原理与实现，主要集中介绍了汇编器对于嵌入式开发的多种灵活的支持；接着介绍了本开发系统的实现所参照的Motorola16位微控制器，以及选择该种处理器的原因；然后，在总体上阐述了本嵌入式开发系统的设计及实现的过程；最后本着测试和验证的目的，介绍了使用本嵌入式开发系统进行多任务应用程序的开发过程。

2.学位论文 高峰 基于嵌入式开发板S1S65000的Linux系统移植和驱动开发 2004

嵌入式ARM微处理器S1S65000是专为带网络功能的数码相机设计的，文章介绍了以S1S65000嵌入式开发板为基础，将Linux系统(内核版本2.4.18)移植到开发板上，然后开发引导装载程序BootLoader和以太网卡、CF卡、CAMERA等驱动程序。

文章首先在第二章介绍了嵌入式系统的硬件和软件、开发嵌入式系统时如何选择硬件和软件。第三、四章介绍了嵌入式微处理器S1S65000和S1S65000的开发板。

第五章介绍了如何将Linux内核移植到S1S65000开发板上，这部分包含开发环境的建立、调试和纠错环境、内核移植、文件系统的选择等内容。

第六章介绍了如何开发系统的装载引导程序BootLoader，着重介绍了BootLoader的原理、流程和烧写FLASH的方法。

第七、八、九章分别重点介绍了以太网卡、PCMCIA接口的CF卡、带JPEG编码模块的CAMERA等设备驱动的开发方法。这几章包括网络设备驱动程序的概念和结构、PCMCIA和CF卡的知识、JPEG编码等丰富的内容。

最后，第十章介绍了整个网络数码相机系统的构成和原理、应用程序的开发、技术特点和优点、尚存在的问题等，并展望了系统适合应用的领域。

文章针对嵌入式系统的Linux内核移植和驱动开发、系统集成做重点介绍，对嵌入式系统的开发者特别是消费类嵌入式网络设备的开发者有很高的参考价值。

3.期刊论文 耿玉菊 嵌入式系统开发技术分析 -牡丹江教育学院学报2009,""(1)

基于嵌入式系统的概念,阐述了嵌入式系统的关键技术及嵌入式开发.首先分析嵌入式系统的技术特点,分别从嵌入式处理器和嵌入式操作系统两方面介绍;在此基础上阐述嵌入式软件的开发过程,并结合嵌入式软件开发的实践,着重阐述嵌入式软件的一些开发技巧.

4.学位论文 刘华 基于ARM-Linux的嵌入式开发关键技术的研究与应用 2007

随着软硬件技术的不断发展，嵌入式系统的应用越来越广泛，嵌入式技术也全面渗透到日常生活的每一个角落。掌上汉语学习机系统的开发是为满足对汉语学习有需要的人群。随着经济、技术的不断提高，中国逐渐走向国际化，汉语学习人员的人数也不断增加，开发出一款掌上汉语学习机系统是有充分的市场需求的。

系统的开发环境是基于ARM-Linux开发平台，并应用了嵌入式开发的相关关键技术，包括嵌入式图形用户界面系统MiniGUI和嵌入式数据库SQLite，系统开发的目的是在这些技术的基础上，为需要学习汉语的人员提供一款界面友好、功能丰富的语言学习工具。

本文主要讲述的是掌上汉语学习机系统的软件开发过程以及开发过程中涉及的开发环境及开发技术。随着手持设备的硬件条件的提高，嵌入式系统对轻量级GUI的需求越来越迫切，图形用户界面的支持是实现一个完善的语言学习系统的基础，本文首先从图形用户界面的历史、技术特点、结构模型、发展状况等方面做了介绍，然后介绍了MiniGUI的体系结构和版本，以及移植的方法和过程，还详细阐述了应用程序开发中对MiniGUI函数库的使用和调用方法。同时一个简单实用的数据库的支持会为系统中的数据处理和组织提供方便，本项目中有六个字典和其它学习程序，对数据的处理也相当多，本文介绍了嵌入式数据库的知识和Sqlite数据库在项目中的使用和开发技术。最后，以系统中的单位换算模块为例详细介绍了应用程序的开发过程。本文的内容涉及了嵌入式Linux软件开发的主要技术，在ARM-Linux嵌入式开发领域具有很强的实践意义。

5.学位论文 程君 基于ARM平台的GDB远程调试环境的研究与移植 2007

嵌入式系统开发工具在开发过程中所起的作用日益突出，相关研究、技术也随之不断更新。随着硬件性能不断提升，很多智能家电、智能手机、甚至高端游戏机都采用了嵌入式系统作为平台进行开发。作为嵌入式开发的关键，调试环节成为嵌入式系统研发的主要瓶颈。在嵌入式硬件性能不断提升的同时，嵌入式软件规模也不断扩大，因此调试难度也与日俱增。

本文首先简要说明了嵌入式软件的开发过程，回顾嵌入式交叉调试技术发展的各种技术。然后分析调试器整个框架和核心，介绍了调试器相关理论和设计思想，并分别研究、对比几种调试技术实现途径和方法，并对调试器中关键流程进行详细阐述。

然后，针对GDB所提供i386和SPARC架构下远程调试环境代码进行分析，抽象出调试桩GDB进行远程调试的核心流程，并根据具体硬件平台差异在ARM处理器上进行代码和远程调试协议移植。本文编写过程中所使用的硬件平台是由使用ARM7处理器的S3C4510b开发板。进入测试阶段，又在S3C4480开发板上进行了测试，对这套模式的可用性进行了验证。

6.期刊论文 袁明.张连芳.董淼.赵宇.郑武 面向对象技术在嵌入式开发中的应用 -计算机应用研究2003,20(2)

随着信息技术的发展,对嵌入式系统的研究与开发也成为当前的一个热点.由于PC机上应用的GUI占用资源太多,不适合嵌入式的应用,因此嵌入式系统对轻量级 GUI 的需求越来越迫切.首先介绍了嵌入式系统及其相关概念,并针对图形用户界面在嵌入式系统中的重要性,从技术角度对其进行了详细介绍;最后结合当前流行的面向对象技术,介绍了该技术在开发TianCai GUI 2.0过程中的应用.

7.学位论文 赵星星 嵌入式实时操作系统移植技术研究与应用 2007

随着嵌入式系统在各个领域的不断蓬勃发展，嵌入式操作系统对不同硬件平台的系统移植技术的研究就成为了嵌入式开发中的一个重要问题。嵌入式操作系统的移植与嵌入式微处理器和嵌入式操作系统的体系结构密切相关，所以随着嵌入式操作系统种类的不断增加，微处理器体系结构的不断变化，嵌入式操作系统的移植就越来越复杂。在这种情况下，提出一种通用的嵌入式操作系统移植技术来指导和简化嵌入式操作系统的移植工作是很有必要的。

本论文深入研究了嵌入式操作系统在不同平台的移植的理论与技术，在此基础上总结分析出嵌入式操作系统移植技术中的共同点，提出了一种通用的嵌入式操作系统移植技术。这种通用的嵌入式操作系统移植技术总结了嵌入式操作系统移植过程中所涉及的所有技术，对嵌入式操作系统移植过程中，从建立交叉开发环境到移植成功后的测试都给出了详细的指导。通用嵌入式操作系统移植技术包括了嵌入式操作系统移植过程中涉及的以下六个方面内容：

■嵌入式系统硬件平台分析
■嵌入式开发工具环境配置技术

■bootloader的移植技术
■嵌入式操作系统的内核移植技术
■嵌入式操作系统的内核调试技术
■嵌入式操作系统移植测试技术最后，在通用嵌入式操作系统移植技术的指导下成功实现了CRTOS(ChineseReal-time Operating System)内核到HHARM2410评估板的移植，并对移植后的CRTOS进行了功能测试、实时性测试和存储性测试，取得了很好的移植效果。

8.会议论文 叶林辉.张春红.勾学荣.于斌 基于嵌入式平台SIP终端的设计与实现 2006

SIP(session initiate protocol)是VoIP(voice overinternet protocol)协议之一。将SIP软终端移植到嵌入式系统中使SIP终端有好的移动性和便携性，也使得SIP协议更为推广。嵌入式技术是当前微电子技术与计算机技术中的一个重要分支。本文介绍了Linux下的SIP软终端Linphone1.2.0移植到OMAP5910的过程，并介绍了嵌入式开发的一般过程。重点讨论了嵌入式系统、bootloader等关键技术。

9.学位论文 李向蔚 嵌入式系统软件开发平台配置管理技术的研究与实现 2004

嵌入式系统的专用和资源约束等特性要求嵌入式操作系统必须是可定制的,而嵌入式系统开发平台作为嵌入式应用开发必不可少的一个组成部分,其重要性越来越突出.如何构建开放、灵活的嵌入式开发平台,实现对开发资源的重用,并降低由于配置工具的差异性、复杂性带来的操作系统定制难度,一直是个难题.本文正是基于这个目标,对嵌入式开发平台及操作系统配置管理技术展开了深入的理论研究和实践探索.嵌入式操作系统受应用需要、资源有限、目标平台差异等方面的制约,需要不同的操作系统构件支持,其定制环节是现今嵌入式软件开发不可缺少的一步.而研究操作系统结构组成是研究嵌入式操作系统可配置技术的关键.论文首先从可配置性视角对嵌入式Linux和CRTOS的结构上进行了详细分析.接着对几种嵌入式操作系统的定制技术进行研究,抽象其共性,从通用性、可重用性、方便性出发,提出了一种基于构件模块的操作系统定制过程模型OSTAILOR,并从结构和原理上对其进行阐述.软件体系结构是嵌入式软件开发平台构件集成的粘合剂.论文接着分析比较了几种平台相关的软件体系结构,选定具有"即插即用"特性的工具软总线作为开发平台配置的基础,并提出了一种基于工具软总线的开发平台配置剪裁器设计方案.在上述基础上详细介绍了所实现的配置剪裁器的核心模块所采用的主要数据结构、算法、函数流程和接口.最后给出了配置剪裁器评测结果.

10.学位论文 刘云霞 嵌入式集成开发环境的研究设计与测试 2007

嵌入式系统是计算机应用研究领域的重要分支之一。根据嵌入式软件交叉开发的特点，普通集成开发环境(Integrated Developping Environment，以下简称IDE)软件不能满足嵌入式软件开发要求，而由于嵌入式软件运行的目标机及其安装或移植的RTOS具有多选择性，目前还没有通用的嵌入式IDE。

海尔嵌入式IDE在Delphi环境下开发，实现支持C/C++，以及汇编语言开发的，面向多种目标机的嵌入式集成开发环境。本文从用户界面的设计原则出发，参照多种嵌入式IDE的用户界面，设计出海尔嵌入式IDE的用户界面。通过研究分析代码编辑软件应具有的主要功能，采用开放源码的第三方控件SynEdit作为海尔嵌入式IDE的代码编辑组件，设计实现海尔嵌入式IDE个性化的代码编辑模块，缩短嵌入式软件开发时间，为用户带来方便。

本文重点是研究分析GCC编译、GDB调试的原理，采用后台调用GCC，GDB的方式，根据用户设置动态生成Make命令文件实现编译模块，并通过本项目组开发的JTAG调试软件下载到目标板，进行各种调试。

本文还研究分析了嵌入式软件测试流程及测试方法和策略，针对海尔嵌入式IDE开发流程，给出V型测试模型，对每个开发阶段进行测试，保证海尔嵌入式IDE的功能性、稳定性等需求，使海尔嵌入式IDE具有一定的参考价值。