分　类　号：TP302　　单位代码：10183
研究生学号：2007532051　　密　　级：公　开

吉林大学

硕士学位论文

基于 UEFI 的底层 API 的性能分析及其功能测试的研究与设计

Research and Design on Performance Analyzing and Function Testing of Firmware API Based on UEFI

作者姓名：王 宇 飞
专　　业：计算机软件与理论
研究方向：数据库与智能网络
指导教师：李雄飞 教授
培养单位：计算机科学与技术学院

2010 年 4 月

基于UEFI的底层API的性能分析及其功能测试的研究与设计

Research and Design on Performance Analyzing and Function Testing of Firmware API Based on UEFI

作者姓名：王宇飞

专业名称：计算机软件与理论

指导教师：李雄飞　教授

学位类别：工学硕士

答辩日期：2010年 5 月26日

吉林大学硕士学位论文原创性声明

本人郑重声明：所呈交的硕士学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：王宇飞

日期：2010 年 4 月 2 日

提　　要

在 UEFI 开源社区（www.tianocore.org）中，存在四个与 UEFI BIOS 相关的开源项目，分别为 EDK（EFI Dev Kit）， EDKII， EFI Shell 和 EFI ToolKit。其中，EDKII（EFI Development Kit）是一个开源的 EFI BIOS 的发布框架，其中包含一系列的开发示例和大量基本的底层库函数，因此，对于其 MDE(Module Development Environment) 模块开发环境的分析与测试能够在最大程度上保证开发的稳定性和质量。因而选题具有一定的实用性和先进性。此外，整个分析和测试设计的过程中，能够充分体现出在 UEFI 从事程序设计相对于传统 BIOS 环境下的优势。

本论文计划从以下几个方面进行研究：1、学习研究 UEFI (统一可拓展固件接口)技术；2、学习研究 EDKII 框架和相应的 MDE (模块开发环境)；3、搭建 MDE 库的测试框架 MdeTestPkg；4、编写 MdeTestPkg 下的测试实例，实现对 MDE 库的分析与测试。

通过对现有的 UEFI (统一可扩展固件接口)技术的学习，深入了解 UEFI BIOS 的背景知识。在此基础上，学习研究 EDK II 的整体架构和模块单元开发设计的规范和方法，并用基于 EDK II 搭建 MDE （模块开发环境）的测试框架，编写类库的测试实例。最终的结果是完成 MDE,即模块开发环境框架中的 44 个库类在 DXE 阶段的功能分析与测试，并且由于类库的互通性，使得测试的类库能够在不同的平台架构（如： IA32，X64 和 IPF 等）上成功运行，具有很好的稳定性和健壮性。在本论文中，我只以 NT32 平台架构为例，来说明 MDE 库在 NT32 平台下的测试框架的搭建以及对于 MDE 库类的测试实例的设计，编写和测试。

目　录

第 1 章 绪 论

1.1 UEFI 的概况和优势

1.1.1 UEFI 相对传统 BIOS 的优势所在

目前，英特尔公司对外开放了其 EFI 平台的源代码，估计在不久的将来，新一代 BIOS 平台的体系规范的 UEFI(Unified Extensible Firmware Interface，即统一可扩展固件接口)将会成为国际开放标准，目前最新的版本是 2.3。 UEFI 为平台固件和操作系统定义了一个新的接口模型，此接口不但包括了平台的相关信息，而且能够启动操作系统和加载应用程序，并为其提供实时服务的请求。UEFI 相对于传统的 Legacy BIOS 其优势显而易见，其优点是模块化，规范化和可扩展性，主要有以下几个方面：

- UEFI 使用 C 语言编写，从而能在启动时达到很好的对于其功能的扩展效果。其实 UEFI 就是一个小型的操作系统，机器从打开到启动进入 DXE 启动阶段时，用户可以选择执行 UEFI Shell 作为与机器交互的接口，从而执行进入操作系统前的各种功能；
- 采用了新的驱动/协议方法，而不再使用以前的中断、硬件端口操作方法。而且解决了选择存储的问题；
- 不再支持 X86 架构，输出也不是二进制代码，而是可热插拔的二进制驱动；
- 对于第三方的软件开发，基本上 Legacy BIOS 实现不了，而且 ROM 的大小还会限制开发工作的进行。但是 UEFI 就可以很容易地实现第三方的软件开发。

尽管具有以上的优势， UEFI 距离正式取代 BIOS 还需要相当长的一段时间,而且将来也不会完全取代 BIOS。

1.1.2 国内外研究情况和目前的趋势

UEFI — Unified Extensible Firmware Interface，即统一的可扩展固件接口，定义了连接操作系统和硬件体系之间的接口标准，这个接口使得操作系统能够从预启动操作环境加载到启动后的操作系统中（ Windows 或者 Linux ），具体来说是提供一个明

确的在操作系统与开机时启动固定在硬件中的软件平台之间的接口规范。

BIOS 提供解决硬件的实时需求和执行软件对于硬件的操作要求的功能，是软件程序与硬件之间的一个接口，或者说是转换器。 BIOS 同时也提供操作系统运行前的初始化工作，如检查系统外围设备和连接计算机内的操作系统和各种不同的硬件等等。 BIOS 的历史悠久，在 UEFI 出现之前一直被使用，其由低级语言编写，以 16 位汇编代码、寄存器参数调用方式、静态链接，和 1MB 以下固定编址的形式而存在。CPU 经过了几次革新，但是还在使用老旧的加电启动的 16 位的实模式，并且适用于系统的固件本身之中。正是由于这种长时间的不改变，使得英特尔能够为大型安腾服务器提供运行支持，因此在开发新款 CPU 时考虑加入兼容模式，从而导致系统性能大大降低。基于此，亟待一场革新方案，于是 EFI (可扩展固件接口)技术被提出。

从核心来看， EFI 就是一个被简化的操作系统，介于高级的操作系统和系统底层固件之间。与老的 BIOS 的显著不同之处是 EFI 使用 C 语言编写，从而使更多的软硬件开发工程师参与到 EFI 的开发工作中，以加快平台创新的快速发展。另外，EFI 还支持鼠标点击操作，并由于其具有图像驱动功能，能够提供一个高分辨率的彩色图形环境，从而明显与传统 BIOS 的单调的纯文本界面相区别。由于其独特的先进性，目前其应用领域已经扩展到计算机的应用领域并向家用设备、电子消费领域等扩展[1]。

2000 年，英特尔发布了 EFI,并首先在大型安腾服务器上采用了此项技术，这就是 BIOS 的新一代接口程序。 EFI 首先是由英特尔发布的一种升级方案，其目的是为了在未来的电脑系统中替代 BIOS。新的 EFI 由于采用了模块化、动态链接和 C 语言风格的参数堆栈传递方式的形式来构建系统，因此比老的 BIOS 更具有优势，也更容易实现，相比之下老的 BIOS 在面对需求时，明显显得力不从心了。除此以外，EFI 驱动程序在不同的 CPU 架构上都能够在最大程度上保证兼容性，这是由于 EFI 驱动程序由 EFI 字节代码编写而成，而不是由运行在 CPU 上的代码构成。不同于传统的老的 BIOS 的缺乏文档的，晦涩约定的，固定的，完全基于经验的即成标准，EFI 将成为一个标准的固件接口。于是，由于其广阔的发展前景和在大型安腾服务器平台上的广泛应用，越来越多的人感受到了 EFI 所展现的魅力。 2002 年，英特尔发布 EFI 规范 1.10； 2005 年，在工业界达成共识的基础上，英特尔将其交给了由微软，惠普， AMD 等公司共同参与组织的一个工业联盟，即 UEFI 组织进行管理，并将其核心代码实现了开源，同时 UEFI 联盟负责完成开发，管理和推广 UEFI 规

范的具体工作； 2007 年， UEFI 规范 2.1 被提出。

UEFI 联盟目前已有 86 家企业成员，是 2005 年由英特尔，惠普，微软等厂商共同成立的一个国际组织，这个组织将会定期主持召开 UEFI 技术大会。 UEFI 联盟由四个组组成，分别是：测试工作组（UTWG） -- 负责开发 UEFI 测试组件；规范工作组(USWG) -- 负责 UEFI 的规范制定和改进；行业联络工作组(ICWG) -- 负责业界业务工作联络；和平台初始化工作组(PIWG) -- 负责平台初始化框架规范的制定和改进。其企业成员由三级构成，分别是：接受者，贡献者和推广者，在联盟组织中承担着不同的职责和义务。

前不久, UEFI 联盟公布了两个新规范，分别是：UEFI2.3 和平台初始化规范 1.2。前者的作用是定义了一个接口，这个接口介于操作系统或其他高级软件与系统底层固件之间；后者的作用是保证互操作性，从而保证不同企业（如：固件开发商，维护固件代码的组织和芯片厂商等）提供的固件组件之间的兼容性。

目前，在美国已经成功举办过两次 UEFI 技术大会。 2007 年， UEFI 技术大会第一次会议在中国的南京召开。而且 UEFI 联盟的常务执行副总裁魏东的宣讲，也让国内更多的 IT 厂商更加全面地了解 UEFI 联盟作为一个组织以及他们的神圣职责[2]。

因此，在全世界掀起了一场轰轰烈烈的革命，这场革命的主题就是： UEFI BIOS 将逐步取代传统的 BIOS。

1.1.3 UEFI 目前存在的问题和其局限性

但是， UEFI 也有不足之处。虽然它能为我们提供很多优点，如：结构层次非常清楚、由高级 C 语言开发、模块化并且易于实现等等。但是也有其缺点，主要如下：

1. 虽然 UEFI 使得我们调试网络设备非常方便，但是也为下一步的网络安全堆积了非常大的安全隐患。这是由于 UEFI 在 DXE 阶段引入了支持如： TCP、MTFTP、 UDP 等协议的完整的网络应用；
2. 由于 UEFI 良好的扩展性，使得其内部更多的接口被外界发现，同时由于其核心代码在网站上开源，使得黑客仅仅通过阅读其核心源代码，而无需反汇编代码，就足可以找出 UEFI 技术的漏洞所在；
3. 相比之前 Legacy BIOS 采用的汇编语言， UEFI 采用高级 C 语言进行编写开

发，在极大地降低了开发的门槛的同时，也提供了丰富的尤其是 DXE 阶段 UEFI Shell 的调试接口，因此很可能引起新的对于 BIOS 系统管理模式的攻击；

4. UEFI 作为一个新的技术，其启动阶段非常复杂，并且每一阶段都有一个单独的控制核心。因此对于简单的 PC，特别是嵌入式设备，其启动速度由于各阶段之间的连接可能会变慢。

但是，这只是一些小的问题。相比之下， UEFI 技术的发展前景十分被看好。在未来拥有十分广阔的发展空间，而且将是近三年计算机技术发展的主流趋势，因此基于 UEFI 技术所做的开发都值得我们投入更大的精力和兴趣去研究和发展。

1.2　对 UEFI 架构的分析

1.2.1　UEFI 的固件架构

EFI 定义了一个软件接口，这个软件接口介于固件平台和操作系统之间。此 UEFI 定义的接口包含：平台信息的数据表以及启动过程中和启动后所提供的服务。其中，启动服务包括如下：不同总线，文件和设备服务上所支持的图形和文本控制台和运行时的服务。 EFI 还额外规定了一个微型的命令行处理程序，从而使得在机器启动的过程中，用户可以选择不进入操作系统，而直接通过 EFI Shell 的调用就可以进入命令行处理程序的环境。

如图 1.1 所示：整个 UEFI 架构完全是由高级 C 语言设计编写的，是模块化和层次化的。其中， EFI 启动管理器的作用是不需要专门的启动加载器，而直接加载操作系统。架构是一种固件的架构类型，它可以替代老旧的 Legacy BIOS，是 EFI 固件接口的一种实现。另外， EFI 的扩展还可以通过连接在计算机上的任何不丢失数据的稳定存储设备上进行加载[22]。

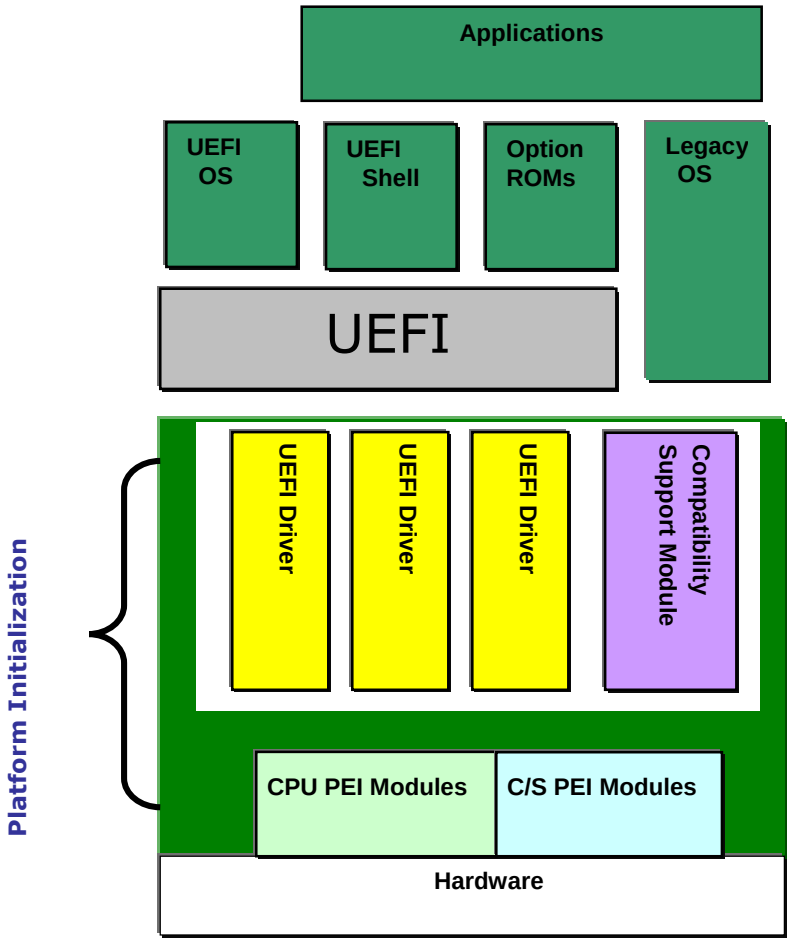

图 1.1　Framework 结构图

UEFI 通过调用 SAL (系统抽象层)和 PAL (处理器抽象层)来操纵系统的各种硬件，而不直接对硬件进行操作。 UEFI 的运行时服务和引导服务也都要调用系统抽象层和处理器抽象层提供的相应功能。 UEFI 还提供了一些用于系统调试的软件，这些工具软件属于 UEFI 的 API 调用。总之， UEFI 在平台固件模型的作用不可或缺，起到了承上启下和屏蔽硬件层物理特性的优势作用。

图 1.2 展示了 UEFI 与其他模块的关系与连接，该结构图清楚地展示了 UEFI 的各个标准协议，接口和模块之间的联系。目前的 UEFI 固件架构由两大模块组成：

- 一部分是向下兼容单元，这是为了保证能够兼容传统的主板而保留下来的。该部分由于其可以直接调用底层平台提供的硬件支持，可以充分地兼容当前市场上差不多所有的计算机平台，从而为 Legacy BIOS 向 UEFI BIOS 的过渡提供了极大地支持；
- 另外一部分就是全新的 UEFI 技术。基于面向对象的设计思想， UEFI 技术在层

与层之间不断地向上提供服务，在不同的服务层间的连接采用了标准的统一化接口。例：在 UEFI 的 Boot Services 启动服务中，包含了启动协议，计时时钟，内存管理，固件驱动和启动服务在内的诸多对象，但对外又重新对这些对象的接口进行了封装，从而为 UEFI 的 OS LOADER 提供了统一的接口，即 UEFI API。使用者不需要关注是哪个对象提供了此项服务，只需要按照标准协议手册去查找标准化服务，以及如何使用即可。

此外， UEFI 还含有很多工业界的如 SMBIOS 和 ACIP （高级配置和电源接口）在内的标准协议构件。

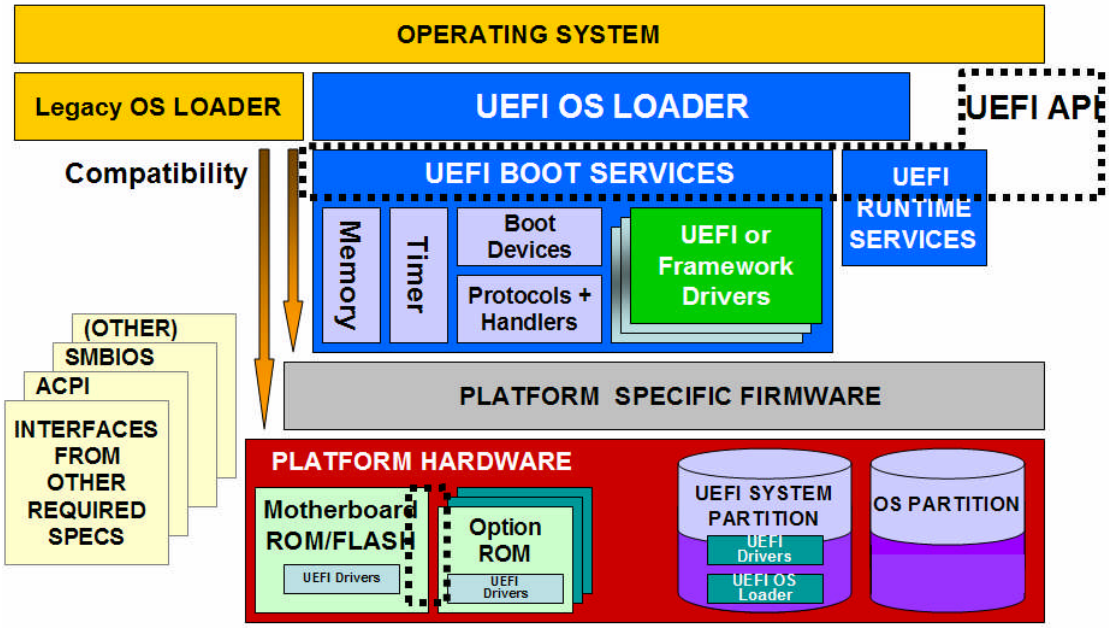

图 1.2　UEFI 与其它模块的关系

1.2.2　UEFI 的初始化框架

如图 1.3 所示：按照阶段来初始化平台， UEFI 的启动过程由四个主要的阶段构成，分别是：

1. SEC—Security，安全阶段；
2. PEI—Pre-EFI，EFI 预初始化阶段；
3. DXE—Driver Execution， 驱动执行阶段；
4. BDS—Boot Device Selection，启动设备选择阶段。

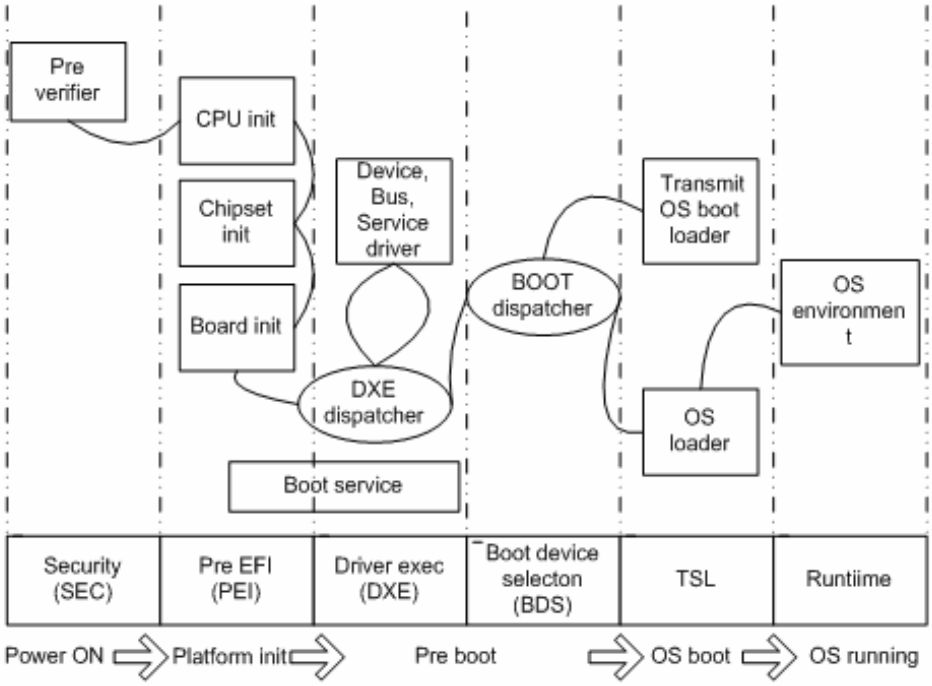

图 1.3　UEFI/Tiano 系统框架

下面详细介绍启动过程中每一阶段的具体细节：

第一个阶段是安全阶段（Security，SEC），是系统上电启动后马上执行的一个步骤。安全阶段负责在系统执行之前检查烧制在芯片上的固件代码，以保证其固件代码的构建是成功的。该阶段通常需要硬件对其的支持，而新一代的芯片组和 CPU 都提供了这些支持功能。

由于选用了高级 C 语言作为 UEFI 的开发语言，所以该阶段的另一个任务就是寻找一块被初始化过的内存，使得 C 源代码可以在预先给定的系统上实现和执行。因此，随后的 PEI 和 DXE 两个启动阶段的划分就都是基于这个标准。

第二个阶段是 EFI 预初始化（Pre-EFI， PEI）阶段，该阶段为用户提供了协议接口和设备驱动的环境界面，是平台设备初始化的一个必不可少的阶段。负责检测启动模块，加载主存模块，并检测和加载 DXE 核心。其目的是以最短的时间找到并初始化内存，很多组件的初始化都将延续到 DXE 阶段后才开始运行。 PEI 阶段发现内存后，就会准备描述平台资源图的状态信息，将其初始化并进入 DXE 阶段[13]。

PEI 转换到 DXE 阶段是单向的，不可返回的。当进入 DXE 阶段后， DXE 的初始化程序装载完成，就成为一个设备齐全的操作环境。此时， PEI 阶段的源代码将不再可用了。

第三个阶段驱动执行环境（Driver Execution， DXE）阶段，是所有系统初始化执行阶段，所以这个阶段在 UEFI 的初始化架构中至关重要，因此将研究底层 API 在 DXE 阶段的性能分析。DXE 阶段包括：DXE 核、DXE 派发程序和很多 DXE 的驱动程序。其中， DXE 核产生一组包括运行时服务和 DXE 服务在内的启动服务；DXE 分派程序的功能是按照一定的次序发现并能够运行这些 DXE 驱动程序；同时 DXE 驱动程序的任务是初始化平台组件，芯片组和处理器，与此同时还要为启动设备、控制台设备和系统服务提供软件抽象。最后，所有的这些组件共同完成平台的初始化工作，并提供一个服务，这个服务是启动一个操作系统所必需的。

第四个阶段是启动设备选择（Boot Device Selection， BDS）阶段，该阶段和 DXE 阶段共同工作创建一个控制台，并试着从任何可用的启动设备来启动一个操作系统。BDS 阶段是将对于系统的控制权交给操作系统之前的最后运行一个阶段。在此阶段，可以做一些用户喜欢的个人定制工作，如向系统使用者提供用户界面，同时也可以用来修改制定用户喜欢的交互环境。

1.3　论文的研究背景、研究意义和先进性

1.3.1　选题背景

在 UEFI 开源社区(www.tianocore.org)中，有四个与 UEFI BIOS 相关的开源项目，它们是 EFI Dev Kit（EDK）、 EDKII、 EFI Shell 和 EFI ToolKit。其中 EDKII（EFI Development Kit）是一个开源的 EFI BIOS 的发布框架，其中包含一系列的开发示例和大量基本的底层库函数， MDE（Module Development Environment）是附加在 EDK 和 EDKII 中的一个类库，即 EdkPkg 包。因此，对于 MDE 即模块开发环境的底层库函数的分析与测试能够在最大程度上保证开发的稳定性和质量。此外，对于整个 MdePkg 包的分析和测试的设计过程中，充分体现出了 UEFI 从事程序设计相对于传统 BIOS 环境下的优势。随着 UEFI BIOS 的进一步推广，越来越多的 OEM 厂商，甚至个人，都会使用这个类库提供的底层库函数所提供的功能，去开发其他一些基于底层硬件或者操作系统层的新功能。然而，对于当前硬件平台的底层硬件，MDE 库提供的功能是否稳定并且正确，这就需要我们搭建一个测试框架，从整体上对其进行测试。

1.3.2　研究的意义和先进性

目前，英特尔对外发布了 EFI 技术平台的源代码，从此新一代 BIOS 体系规范的统一扩展固件接口 UEFI 将会成为国际开放标准。 UEFI 技术规范为平台固件和操作系统之间定义了一个新的接口模型，此接口由数据表组成，并且包括平台的一些相关属性信息以及对载入程序和操作系统都有效的启动服务请求和实时服务请求，所有的这些共同为启动系统平台并且预加载应用程序的运行定义了完整的服务环境。UEFI 由于其规范化，模块化， 和可扩展的特点，其优势明显胜于老的 BIOS。另外，UEFI BIOS 采用高级 C 语言设计编写，从而其启动时能够充分扩展其的功能。 UEFI 就是一个微型的小操作系统，开机程序执行到 DXE 阶段，用户就可以在启动到操作系统界面前，通过 UEFI Shell 应用程序使用一些扩展功能，如：信息检测，网络应用以及软硬件调试等等。

MDE (Module Development Environment)即模块开发环境，以 EDKII 开发框架作为其依托。在 MDE 库的包里， MdePkg 由开发者设计编写， MdePkg 使得开发者能够在 EDK II 框架下开发自己的模块，同时提供一个非常稳定的并且能够和 UEFI 和 PI 充分兼容的平台，从而使得开发者能够开发 UEFI 驱动， UEFI 程序和 UEFI 操作系统装载器， UEFI 诊断，和 UEFI 平台初始化兼容的硅模块等等。因此，测试 MDE 库的稳定性与正确性是非常重要的，它决定着开发的质量。

本论文所实现的是对于 MDE 包中的 MdePkg 中的 44 个库类在 DXE 阶段的功能分析与测试，如：执行过程是否稳定，其所提供的功能是否正确，并且在所需要的平台架构上是否能够稳定执行。并且由于类库的互通性，使得所要测试的类库能够在不同的平台架构（如： IA32， X64 和 IA64 等）上成功运行，具有很好的稳定性和健壮性。

但在本论文中，最终完成的是：在 UEFI BIOS 的 IA-32 架构硬件平台上，实现了 MDE 库在 NT32 架构平台上的测试框架的搭建以及对于 MDE 库中的所有库类的测试实例的设计，编写和测试。

除此之外，本文还介绍了该项技术在 PEI、 Runtime 等其它阶段和 X64 架构平台上的测试框架 MdeTestPkg 以及 MdeTestPkg 下的测试实例，实现对 MDE 库的分析与测试。

1.4 论文研究的思路内容和整体结构

1.4.1 研究的思路内容

本论文的研究重点主要有以下几个方面：

第一、 EDKII 开发框架

本论文的研究重心在于测试 MDE 库所提供的底层库函数，而这些底层库函数是存在于 EDK II 包中，具体来讲就是 MDE 包以 EDK II 开发框架为依托，因此，对于 EDK II 开发框架的整体架构的研究是论文进展的第一步。

EDKII（EFI Development Kit） 是一个开源的 EFI BIOS 的发布框架，负责为 UEFI 和 PI 的开发提供一个良好的构建和版本跟踪环境。其中包含一系列的开发示例和 44 个基本的底层库函数类，实质上是一个用于装载框架的基本源代码和示例驱动程序的容器。本论文所实现的是模块开发环境库中的 44 个库类在 DXE 阶段的功能分析与测试，如：执行过程是否稳定，其所提供的功能是否正确，并且在所需要的平台架构上是否能够稳定执行。并且由于类库的互通性，使得所要测试的类库能够在不同的平台架构（如： IA32， X64 和 IA64 等）上成功运行，具有很好的稳定性和健壮性。但在本论文中，最终在 UEFI BIOS 的 IA-32 架构硬件平台上，实现了 MDE 库在 NT32 架构平台上的测试框架的搭建以及对于 MDE 库中的所有库类的测试实例的设计，编写和测试。

第二、测试 MDE 库的开发框架 MdeTestPkg 的整体框架架构。

在熟悉了 EDK II 的框架后，我们就要搭建测试 MDE 库的测试框架， MDE 库的测试框架就是 MdeTestPkg 包，其作用就是通过 Porxy 和 TestCase 包所提供的接口去测试 MdePkg 包中所提供的库函数的功能。

MDE（Module Development Environment）是附加在 EDK 和 EDKII 中的一个类库。因此，对于 MDE 即模块开发环境的底层库函数的分析与测试能够在最大程度上保证开发的稳定性和质量。此外，对于整个分析和测试的设计过程中，能够充分体现出在 UEFI 从事程序设计相对于传统 BIOS 环境下的优势。随着 UEFI BIOS 的进一步推广，越来越多的 OEM 厂商，甚至个人，都会使用这个类库提供的底层库函数所提供的功能，去开发其他一些基于底层硬件或者操作系统层的新功能。然而，对于当前硬件平台的底层硬件， MDE 库提供的功能是否稳定并且正确，这就需要我们搭建

一个测试框架，从整体上对其进行测试。

第三、编写 MdeTestPkg 下的测试实例，实现对 MDE 库的分析与测试。

EDK II 库的测试架构被设计在包 MdeTestPkg 中实现。库实例是一个或多个库类的执行。库的执行可能有选择性地用到一个或多个其他的库类，并且也可能有选择地产生一个构造函数和（或者）一个析构函数。对于库实例的构造函数和析构函数的调用语法会根据模块类型的不同而不同。同样地库实例也被设计成与一个特定集合的模块类型相连接。如果库实例被声明成一个基模块类型，那么它可以和任何一种类型的模块相连接。

对于在 MdePkg 中的每一个库，在 MdeTestPkg\TestCase 目录下都有一个相应的测试库；同样地，在 MdeTestPkg\Proxy 目录下都有一个相应的 Proxy 库。因此，在 MdeTestPkg 包中正确设计一个测试实例需要同时修改 MdeTestPkg\TestCase 目录下的文件和 MdeTestPkg\Proxy 目录下的文件。

总的来说，本论文研究的依据主要涉及以下几个方面的内容： UEFI BIOS 概括的分析和理解， UEFI 技术的开发框架 EDKII 整体结构分析，重点在于设计搭建 MDE 库的开发框架 MdeTestPkg，从而实现在开发框架 MdeTestPkg 下设计编写在 DXE 阶段下的测试 MDE 库的测试实例。

1.4.2　整体结构安排

本论文书写共分为五章。其中重点在于第二、三、四章。第二章着重从总体结构上对 EDK II 的总体框架结构进行了分析研究；接下来的两章是全文主体，重点介绍了基于 EDK II 去测试 MDE 库的测试整体框架的搭建，和在所搭建的测试框架中编写测试 MDE 的底层库函数的测试用例的设计编写原则和方法；最后一章是全文的总结，同时给出测试结果和测试过程中遇到的困难以及解决方法。

总的来说，本论文的组织结构如下：

第一章、绪论。本章首先介绍了 UEFI 优势所在，国内外研究情况和其局限性，然后详细系统地介绍 UEFI 的结构和其平台启动过程，最后基于 UEFI 描述本论文研究的背景，意义，先进性以及思路内容和整体结构安排。

第二章、 Tiano 的系统框架和实现原理。深入分析 EDK II 的开发框架，以及以 EDK II 为依托的要搭建 MDE(模块开发环境)库的系统环境要求，以及要实现的目

标；最后详细介绍了 DXE 启动阶段的执行过程及其特点。

第三章、搭建 MDE Library 库的测试框架。这一章是全文的核心所在，首先分析了设计搭建出来的 MdeTestPkg 包的整体组织结构，然后深入分析搭建每一个包的过程，以及所依据的原理和设计思想。其中分三个小节，重点介绍了其中的 Assert 包、Porxy 包、和 TestCase 包的搭建过程。最后针对特定 DXE 阶段对于 MDE 底层库函数的测试，讲述了设计搭建 DXE 包的测试实例的过程。

第四章、 MDE 库在 DXE 阶段的测试用例的设计与编写。这一章在搭建好了 MDE 测试框架的前提下，给出了设计编写底层库函数所要遵循的原则和方法，重点以 TimerLib 库类为例，详细讲述了设计测试用例的过程以及 TimerLib 在 DXE 阶段测试的过程。

第五章、测试结果的统计与总结。模块开发环境包包含着 44 个库类和 72 个库实例。每一个库类都对应着一个库的实例。库的实例是一个或多个库类的执行。本章记录了在实际的工作中对 MDE 库的所有底层库函数类的测试结果，并做了详细的统计和分析。并对全文做了总结和展望，最后提出了进一步的研究方向。

第 2 章　Tiano 的系统框架和实现原理

2.1　EDK II 的开发框架的总体介绍

MDE(Module Development Environment)即模块开发环境，以 EDKII 开发框架作为其依托。在 MDE 库的包里， MdePkg 由开发者设计编写， MdePkg 使得开发者能够在 EDK II 框架下开发自己的模块，同时提供一个非常稳定的并且能够和 UEFI 和 PI 充分兼容的平台，从而使得开发者能够开发 UEFI 驱动， UEFI 程序和 UEFI 操作系统装载器， UEFI 诊断，和 UEFI 平台初始化兼容的硅模块等等。因此，测试 MDE 库的稳定性与正确性是非常重要的，它决定着开发的质量[21]。

其中， EDKII（EFI Development Kit， EFI 开发工具）是 EFI BIOS 的一个开源的发布框架。其中，包含一系列的开发示例和大量的基本底层库函数。开源核心代码包括三种平台架构，分别是： DUET、 NT32 和 Unix。除了开源部分， EDK 同时也是开发、调试和测试 EFI 驱动程序的一个综合平台。

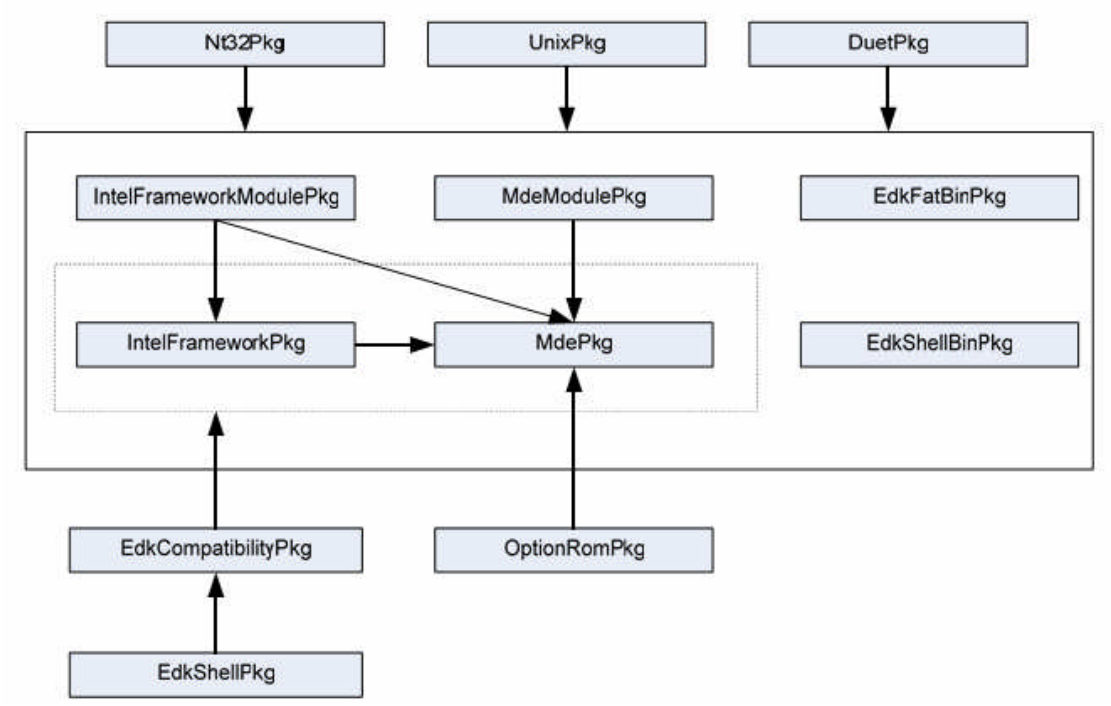

图 2.1　EDKII 主要模块依赖关系图

最初的版本是 EDK, 随后英特尔根据 EFI 客户的反馈信息进行了进一步功能扩

展，研究开发了一个新的功能更为强大的开发平台 EDKII。 EDKII 的主要目标是如何让用户设计出更规范的自定义的模块，其改进的功能是在开发库、类机制以及 PCD 机制方面进行了扩展。

本论文所做的研究就是在 UEFI BIOS 最新发布的 EDKII 平台上进行的，从图 2.1 我们可以看出，EDK II 各模块之间存在非常紧密的依赖关系。例：MdeModulePkg 紧密依赖于 MdePkg，是由于在 MdePkg 中定义了 MdeModulePkg 所使用的协议原型。

从 EDKII 主要模块依赖关系图中，我们可以清楚看到：

- 要想编译 OptionRomPkg， 只需 MdePkg 和 BaseTools 构件就可以完成；
- 要想编译 EdkCompatibiliryPkg， 只要有 MdePkg、 IntelFrameworkPkg 和 BaseTools 就可以完成；
- 要想编译 EdkShellPkg，那么 EdkCompatibilityPkg 和 BaseTools 是不可少的；
- 想要编译 NT32、 Unix、 Duet 模拟器平台，必须依赖于以下模块，分别是：MdePkg、MdeModulePkg、 IntelFrameworkPkg、 IntelFrameworkModulePkg、EdkFatBinPkg、 EdkShellBinPkg 和 BaseTools。

下面具体分析 EDK II，其开发框架中的主要模块及其功能如下所示，在这里对其进行了详细的分析与说明：

- ✓ Conf：保存生成编译环境信息、编译目标定义和编译器参数的路径，配置结束后此路径将会产生三个配置文件。
- ✓ BaseTools：提供编译环境配置文件和二进制编译工具集。
- ✓ MdeModulePkg：提供跨平台的包括 MdePkg 的底层库应用模块示例的模块。
- ✓ MdePkg：包扩 EDKII 框架下 IA32、 X64 和 IA64 平台的底层库函数、工业标准和协议，从而各个平台架构的 UEFI 都可以使用该包下的库函数，使得开发难度简化。
- ✓ EdkFatBinPkg：提供原始的 FAT 驱动，这些驱动针对不同架构的 CPU。但是由于 FAT 版本的问题，这里没有源代码，只有 FAT 的 .efi 可执行文件。
- ✓ EdkShellPkg：是跨平台的开发环境，提供 EFI Shell 下开发应用程序的环境。
- ✓ DuetPkg：可以改造 Legacy BIOS，在硬件平台虚拟出实际的 UEFI BIOS 工作环境，并且提供基于 Legacy BIOS 的 Runtime 环境的支持库。
- ✓ Nt32Pkg：提供支持 UEFI Runtime 的环境，通过在微软的操作系统下加载 32

位模拟器而实现。

- ✓ UnixPkg：提供支持 UEFI Runtime 的环境，通过在 Linux 的操作系统下加载 32 位模拟器而实现。
- ✓ EdkCompatibilityPkg：此包保证了 EDKII 对 EDK 的全部功能的支持，由于其提供兼容 Legacy 和 EDK 库定义的协议和库。
- ✓ OptionRomPkg：包扩了实例程序，其可针对不同架构的 CPU 编译 PCI 兼容镜像。

对于以上每一个模块包都有类似下面的结构。如：其中的包 MdePkg，如下是其包含的子文件夹：

Include\　　--所有公共文件的头文件；

　Ia32\　　--内部专有的支持 IA-32 架构的头文件；

　X64\　　--内部专有的支持 X64 架构的头文件；

　Ipf\　　--内部专有的支持 IA-64 架构的头文件；

　Ebc\　　--内部专有的支持 EBC 架构的头文件；

　Uefi\　　--内部专有的支持 UEFI2.3 技术规范的头文件；

　Pi\　　--内部专有的支持 PI1.2 技术规范的头文件；

　Protocol\　　--专有的包括各种协议规范的头文件；

　Ppi\　　--专有的包括各种 PPIs 规范的头文件；

　Guid\　　--专有的包含各种 GUIDs 规范的头文件；

　IndustryStandard\　--公有的工业规范标准头文件

　Library\　　--包括 MDE 库文件的头文件

Library\　　-- MDE 库的原型

2.2　搭建测试 MDE 库框架前的系统环境配置

在使用 EDKII 进行开发之前，我们不但要更新源代码，将其替换成最新的。还要进行测试开发环境的安装与配置，以搭建测试模块开发环境的框架，同时也要注意 DEBUG 调试环境的安装及配置。

- ➢ 为了正确地配置用于模块开发环境库在 DXE 阶段测试的工作环境，我们需要安装以下工具： Cygwin， ant， java， Masm， xmlbeans， asl， VS.net， svn。

我们把这些工具全部安装到 C:\ 下；

- 利用安装好的 SVN 工具从 SVN 服务器上下载 EDK II 目录树：https://edk2.tianocore.org/svn/edk2/trunk/edk ，并且从以下 svn://sh1sdev135/NewMdeTest/MdeTestPkg URL 网址下载最新的 MdeTestPkg 包；
- 如下设置系统环境变量：

PATH=%PATH%; C:\cygwin\bin;C:\EDKII\Tools\bin

ANT_HOME=c:\ant

XMLBEANS_HOME=c:\xmlbeans

JAVA_HOME=c:\java\jdk*

至此， MDE 库所需的运行环境就设置好了。

2.3　设计测试 MDE 库要实现的目标

MDE (Module Development Environment)即模块开发环境，以 EDKII 开发框架作为其依托。在 MDE 库的包里， MdePkg 由开发者设计编写， MdePkg 使得开发者能够在 EDK II 框架下开发自己的模块，同时提供一个非常稳定的并且能够和 UEFI 和 PI 充分兼容的平台，从而使得开发者能够开发 UEFI 驱动， UEFI 程序和 UEFI 操作系统装载器， UEFI 诊断，和 UEFI 平台初始化兼容的硅模块等等。因此，测试 MDE 库的稳定性与正确性是非常重要的，它决定着开发的质量。

本论文所实现的是 MDE 包中的 44 个库类在 DXE 阶段的功能分析与测试，如：执行过程是否稳定，其所提供的功能是否正确，并且在所需要的平台架构上是否能够稳定执行。并且由于类库的互通性，使得所要测试的类库能够在不同的平台架构（如： IA32， X64 和 IA64 等）上成功运行，具有很好的稳定性和健壮性。

但在本论文中，最终在 UEFI BIOS 的 IA-32 架构硬件平台上，实现了 MDE 库在 NT32 架构平台上的测试框架的搭建以及对于 MDE 库中的所有库类的测试实例的设计，编写和测试。

除此之外，本文还介绍了该项技术在 PEI、 Runtime 等其它阶段和 X64 架构平台上的测试框架 MdeTestPkg 以及 MdeTestPkg 下的测试实例，实现对 MDE 库的分析与测试。

2.4　DXE 阶段的启动原理

DXE（Driver Execution），驱动执行环境阶段，是所有系统初始化所必经的执行阶段，所以这个阶段在 UEFI 的初始化架构中至关重要，因此将研究底层 API 在 DXE 阶段的性能分析。 DXE 阶段包括： DXE 核、 DXE 派发程序和很多 DXE 的驱动程序。其中，DXE 核产生一组包括运行时服务和 DXE 服务在内的启动服务；DXE 分派程序的功能是按照一定的次序发现并能够运行这些 DXE 驱动程序；同时 DXE 驱动程序的任务是初始化平台组件，芯片组和处理器，与此同时还要为启动设备、控制台设备和系统服务提供软件抽象。最后，所有的这些组件共同完成平台的初始化工作，并提供一个服务，这个服务是启动一个操作系统所必需的[22]。

第 3 章　搭建 MDE Library 库的测试框架

3.1　MdeTestPkg 包的组织结构

测试 EDK II 库类的架构 MdePkg 包被设计在包 MdeTestPkg 中实现。我们可以在如下的目录网址中找到： cvs.sc.intel.com:/home/cmplr/cvs/tianoad，其模块名字是 MdkTestPkg：

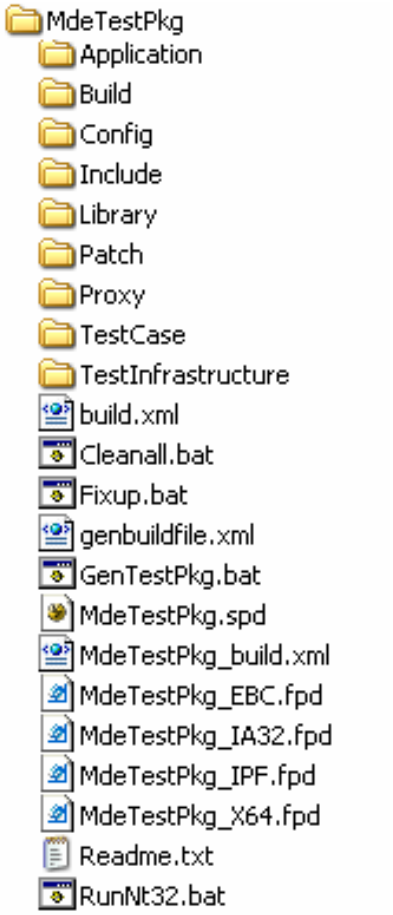

图 3.1　树状组织结构

如下，图 3.1 展示了 MdeTestPkg 包的树状组织结构的第一层。其中，包括很多目录和单一的文件：

- Application：测试框架提供的测试程序；

- Build：在 MdeTestPkg 的 build 过程中所用到的文件；
- Config：在 SCT 运行环境中需要用到的配置文件；
- Include：所有包的包含文件；
- Library：把 Proxy 驱动和测试实例的访问点包含进来所用到的库；或者能够在 MdeTestPkg 中提供服务；
- Patch：当把 MdeTestPkg 和 EDK II 在每一轮集成起来的过程中，能够为 EDK II 解决任何问题的补丁；
- Proxy：能够把库实例在 proxy 和测试实例之间注册到一个共享空间的“封装库”。例如：在 DXE 阶段的配置表；
- TestCase：能够在 BIOS 的不同测试阶段和不同的平台架构中共享的的库接口的测试实例；
- TestInfrastructure：用于建立测试架构的测试框架的驱动程序，或者在特定的场景下能够提供支持；

在 MdeTestPkg 中的根目录下的单一文件，如下描述：

- 这些 XML 文件和 ANT build 系统相关；
- Fixup.bat 用于在 Proxy 驱动和测试实例的内部产生软连接，以使得测试实体能够被不同的阶段所共享；
- GenTestPkg.bat 用于建立一个 TestPkg，这个 TestPkg 能够把所有的输出二进制文件数据在不同的架构中组织起来，参考图 3.2；
- Cleanall.bat 用于删除在 Proxy 驱动程序和测试实例之间的软连接；同时也会删除产生出来的 TestPkg；
- MDETestPkg.spd 是包的表面域的描述文件；
- MdeTestPkg_*.fpd 是平台的表面域的描述文件。 * 是一个通配符，代表 IA32, X64, IPF 或者 EBC 架构。每一个 .fpd 文件描述所有的 Proxy 驱动程序，测试实例，支持驱动，库等等，所有的这些都将在这个特定的架构中被构建；
- RunNt32.bat 是用来在 Nt32 平台运行 SCT 的脚本。如果这些平台还没有被构

建出来，那么这些脚本是不能执行 SCT 二进制 Nt32 平台的。如果这样,那么错误信息将会被输出到屏幕。

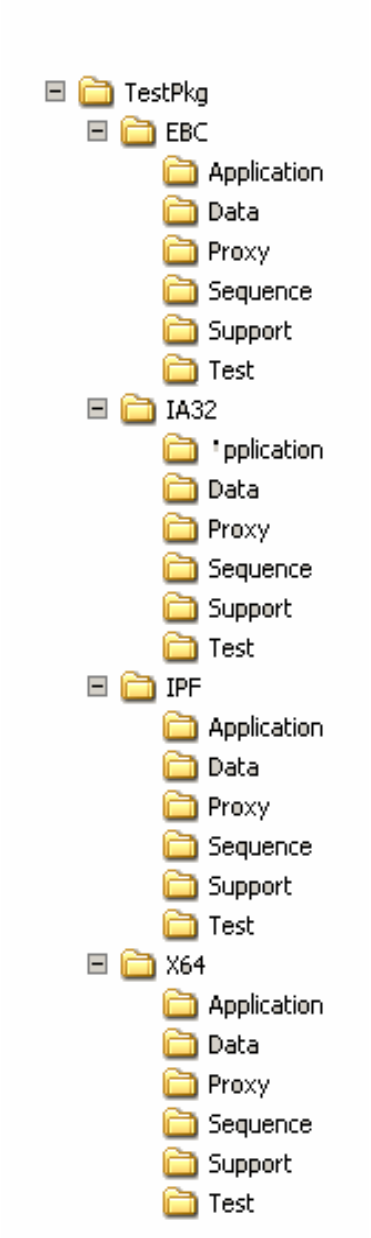

图 3.2　TestPkg 组织结构

3.1.1　库实例的描述与架构

库实例是用于执行一个或多个库类。库的执行可能有选择性地用到一个或多个其它的库类，并且也可能有选择地产生一个构造函数和（或者）相应的析构函数。对于

库实例的构造函数和析构函数的调用语法会根据模块类型的不同而不同。同样地，库实例也被设计成与一个特定集合的模块类型相连接。如果库实例被声明成一个基模块类型，那么它可以和任何一种类型的模块相连接。

如果不可能在特定 CPU 架构（如： IA32、 X64、 IA64）上为某一特定模块类型从一个库类中执行一个库函数，那么这个执行必须保证能够产生一个 ASSERT()。

允许库实例在其执行过程中利用 PCD 实体，但是必须符合一定的限制条件[27]。其限制条件和其它的一些限制条件在下面列出：

1. 库实例不允许利用 FixedPcdGetXXX() 的 PCD 库函数。
2. 对于带有包含全局变量的基模块类型的库实例，必须声明其为 CONST。 这就要求来自 Base 库中的函数可以从 SEC XIP 和 PEI XIP 代码中利用，这些代码直接从 FLASH 中执行。如果尝试着写入全局变量，那么将会尝试着写入 FLASH 设备中。
3. 在 DXE Runtime, DXE SMM, 或者 DXE SAL 阶段类型的库实例可能在执行 ExitBootServices() 后不会访问动态 PCD 实体。
    a) 在 DXE Runtime 阶段和 DXE SAL 阶段可用的库实例同样可利用 BINARY Patch 类型的 PCD 实体必须能够描述这个库的消费者必须能够处理 SetVirtualAddressMap() 交易。对于不能代表系统地址的 PCD 实体或者 FixedAtBuild PCD 实体也同样没有问题。

下面的表也列出了由包产生的库实例。每一个表为给定的包定义了所有的库实例。在表的模块类型这一类，黑体字类型的第一个元素，是被用于构建库实例的模块类型。如果下面所述都是正确的话，那么模块可能和一个库实例相连接：

1. 模块依赖于在库类这一列中所列出的库类之一；
2. 模块和库实例具有同样的模块类型；
3. 如果库实例在模块类型这一列中包含不只一个模块类型，那么库实例可能与任何一个在所列出的模块类型中的模块和消费了列在库类这一列中的库类相连接；
4. 如何在库类这一列中列出库实例产生出的库类。对于库实例能够产生多于一个的库类，这是合法的，以下是几个例子： CPU 这一列列出了库实例支持的 CPU 架构。如果这一列的实体值都是‘All’， 那么库实例可以被烧制成模块开发环境包所支持的所有 CPU 架构。

3.1.2　ASSERT 测试

大多数函数需要检查在规范说明书中描述的实时前提，因此需要求助于 ASSERT 机制。 EDK II 模块开发环境库测试框架提供测试 ASSERT() 脚本的机制，这从而意味着不论我们的代码会不会产生 ASSERT()，如果条件与环境相匹配它就可以进行验证。当 ASSERT() 发生时，我们的测试实例并不会真正地 ASSERT，但是将会报告断言已经产生了。

3.1.2.1　测试机制介绍

为了提供测试 ASSERT() 的测试机制，设计把以下的组件加入到 EDK II 模块开发环境库的测试框架中：

- 库实例： BaseDebugLibForTest。这个实例的库类是 DebugLib；
- 库类： DebugTestLib；
- 库实例： PeiDebugTestLib/DxeDebugTestLib。 DebugTestLib 在 Pei 和 Dxe 阶段的实例；
- 驱动： DebugTestLibJumpBuffer。这个驱动是一个支持驱动，其作用是为一个用于在 SetJump/LongJump 时记录 CPU 上下文场景的共享内存安装协议。当前只支持 IA32， X64 和 IPF。
- PEIM： DebugTestLibJumpBuffer。这个 PEIM 为用于记录 CPU 上下文场景的共享内存安装 PPI。目前只支持 IA32， X64 和 IPF。

3.1.2.2　设计 ASSERT()测试实例

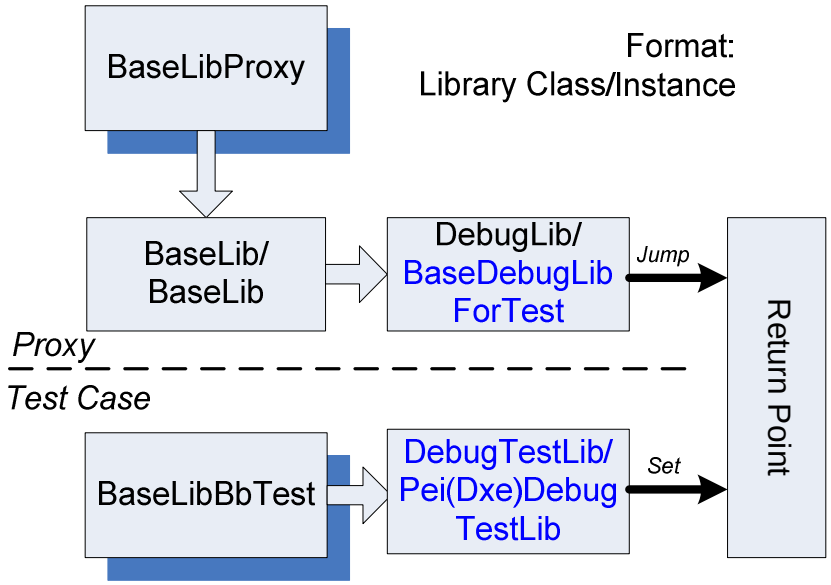

Figure: DebugTestLib Usage Model

图 3.3　Debug 测试库使用模型

图 3.3 给出了一个例子，用于说明在 BaseLib 中如何设计编写 ASSERT() 测试实例。

1. 用在 BaseLibProxy 中的 DebugLib 的库实例应该被 BaseDebugLibForTest 代替来跟踪 ASSERT()。

2. 库类 DebugTestLib 应该和实例 PeiDebugTestLib 或者 DxeDebugTestLib 一块被加入到 BaseLibBbTest 中。

3. 测试实例需要以如下方式在测试实体中编写：

```
UINTN ReturnValue;
SET_RETURN_POINT (ReturnValue);
  if (ReturnValue == 0) {
    BaseLibraryInterface->StrLen (NULL);  // Sentences which triggers
ASSERT().
    …
    AssertionType = EFI_TEST_ASSERTION_FAILED;
  } else if (ReturnValue == 1) {
    AssertionType = EFI_TEST_ASSERTION_PASSED;
  } else {
    //
    // Must never happen
    //
    AssertionType = EFI_TEST_ASSERTION_FAILED;
    CpuDeadLoop ();
  }
```

4. 在 *TestCommon.c 中，如下设计更改队列 gSupportProtocolGuid[]：

```
EFI_GUID gSupportProtocolGuid[] = {
  DEUBG_TEST_LIB_JUMP_BUFFER_GUID, // support guid
  EFI_NULL_GUID
};
```

注意： ASSERT() 的测试实体被加入到队列 gBbTestEntryField[] 中。

另外：设计采取以下步骤，在 DXE 阶段编写 ASSERT() 测试实例：

1. 在测试实例的 MSA 文件中加入库类“DebugTestLib”；

2. 为这个测试实例在 FPD 文件中加入库实例“DxeDebugTestLIb”；

3. 为测试 Proxy 驱动程序在 FPD 文件中用 BaseDebugLibForTest 代替 DebugLib 的其它库实例；

4. 为这个测试 Proxy 在 FPD 文件中加入库实例“DxeDebugTestLib”；

5. 把 DxeDebugTEstLibJumpBuffer.efi 拷贝到 SCT binary 的支持目录下；

要在 PEI 阶段下设计 ASSERT() 测试实例，需要采取以下步骤：

1. 在测试实例的 MSA 文件中加入库类“DebugTestLib”；

2. 为这个测试实例在 FPD 文件中加入库实例“PeiDebugTestLIb”；

3. 为测试 Proxy 驱动程序在 FPD 文件中用 BaseDebugLibForTest 代替 DebugLib 的其它库实例；

4. 为这个测试 Proxy 在 FPD 文件中加入库实例“PeiDebugTestLib”；

5. 在测试平台的 FPD 文件中，如： Nt32 平台或者 Lakeport 平台，加入 PeiDebugTestLibJumpBuffer 作为支持组件，加入 proxy 和测试实例作为所烧制出来的固件代码的测试组件。

3.1.3 Proxy 驱动

Proxy 驱动是能够封装测试过程中的库，并把库接口注册到一个共享内存的模块。接下来的小节将会详细介绍设计出的 Proxy 驱动的源代码架构。

每一个 Proxy 驱动都由以下三个文件夹构成[23]：

- .MSA： Module Surface Area
- .H： C 源头文件
- .C： C 源代码文件

3.1.3.1　Proxy 驱动源代码树状组织结构的设计

图 3.4 给出了 Proxy 驱动的树状组织结构图。Common 目录含有一个封装测试库的 C 文件。DXE 目录含有 MSA 文件和一个 C 文件“InstanceName.C”。其中 InstanceName.C 描述了在这个 Proxy 驱动中我们要测试哪个库实例。

图 3.4 Proxy 驱动源代码树状组织

在 Common 目录下的 C 文件执行被 *ProxyMain 库所保留的两个接口，分别是：InitializeProxyInterface() 和 GetBbTestParametersFromProxy()，来导入它的测试 GUID 和库接口。每一库接口有一个相应的 Proxy 驱动。每一 Proxy 驱动与一个测试实例相关联，这个测试实例由 GUID 唯一定义。每一 Proxy 驱动以一个实例对应一个结构的方式封装所有的库接口，并且把这个结构注册到一个可以被测试实例访问的的共享内存中。

除了 Common 目录，Proxy 驱动可能有多个其它的目录，这是由这个 Proxy 驱动由多少个阶段来决定的。例如，如果测试库是一个 BASE 库，那么 Proxy 驱动就应该包括 PEI, PEI Core，DXE, DXE Core，DXE SMM 和 DXE Runtime 阶段。这就意味着在这个 Proxy 驱动目录中除了 Common 子目录我们还应包含六个额外的目录。

3.1.3.2　Module Surface Area (MSA)文件

MSA 文件，即模块表面域文件详细说明了在每一阶段相关目录（如：DXE 目录和 PEI 目录）的模块的访问域。它与 EDKII 的主题需求相符合。

3.1.3.3　头文件

在 Proxy 驱动中的头文件位于 Include\Proxy 目录下，并且被用于存储全局定义， GUIDs 和函数原型。在 Proxy 驱动中，头文件包含了在需要测试的模块开发环境库的每一函数的函数类型定义以及一个存储在共享内存中的结构。

以 TimerLib Proxy 为例，其驱动的头文件如下所示：

```
#ifndef __TIMER_LIB_PROXY_H__
#define __TIMER_LIB_PROXY_H__

//
// TODO: Define function type for each function belong to
//       the library which will be capsulated in this proxy
//       driver.
//
typedef
UINTN
(EFIAPI *TIMER_LIB_MICRO_SECOND_DELAY) (
  IN UINTN  MicroSeconds
);

typedef
UINTN
(EFIAPI *TIMER_LIB_NANO_SECOND_DELAY) (
  IN UINTN  NanoSeconds
);

typedef
UINT64
(EFIAPI *TIMER_LIB_GET_PERFORMANCE_COUNTER) (
  VOID
);

typedef
```

```
    UINT64
    (EFIAPI *TIMER_LIB_GET_PERFORMANCE_COUNTER_PROPERTIES) (
      IN UINT64  *StartValue,  OPTIONAL
      IN UINT64  *EndValue     OPTIONAL
    );

    //
    // TODO: Structure type define for the interface of this proxy
    //       driver which is bridge between the test case to Mde
    //       library.
    //
    typedef struct {
      TIMER_LIB_MICRO_SECOND_DELAY       MicroSecondDelay;
      TIMER_LIB_NANO_SECOND_DELAY        NanoSecondDelay;
      TIMER_LIB_GET_PERFORMANCE_COUNTER    GetPerformanceCounter;
      TIMER_LIB_GET_PERFORMANCE_COUNTER_PROPERTIES
    GetPerformanceCounterProperties;
    } TIMER_LIBRARY_INTERFACE;

    //
    // TODO: Used this GUID to identify this proxy interface.
    //       And add a new GUID define into MdeTestPkg.spd
    //
    extern EFI_GUID gTimerLibTestGuid;

    #endif
```

3.1.3.4　源文件

Proxy 驱动的源文件位于 Proxy*LibProxy\Common 中。它包含了用于 GetBbTestParametersFromPorxy 和 InitializeProxyInterface 执行的代码。

同样以 TimerLib Proxy 为例，如下所示是其驱动的源代码“InitializeTimerLibraryInterface.c”：

```
    #include "Proxy\TimerLibProxy.h"

    //
    // TODO: Each proxy source code should implement this function.
    //       Description of this function is on page 47.
    //
    EFI_STATUS
    EFIAPI
    GetBbTestParametersFromProxy (
      OUT EFI_GUID     *TestGuid,
      OUT UINTN        *TestProtocolLength  OPTIONAL
      )
    {
      if (TestGuid == NULL) {
        return EFI_INVALID_PARAMETER;
      }

```

```
  CopyMem (
    TestGuid,
    &gTimerLibTestGuid,
    sizeof (EFI_GUID)
    );

  if (TestProtocolLength != NULL) {
    *TestProtocolLength = sizeof (TIMER_LIBRARY_INTERFACE);
  }

  return EFI_SUCCESS;
}

//
// TODO: Each proxy source code should implement this function
//       to Initialize all members of proxy interface.
//       Description of this function is on page 48.
//
VOID
InitializeProxyInterface (
  VOID** ProxyInterface
  )
{
  TIMER_LIBRARY_INTERFACE *Interface;

  Interface = (TIMER_LIBRARY_INTERFACE *) *ProxyInterface;

  Interface->MicroSecondDelay     = MicroSecondDelay;
  Interface->NanoSecondDelay      = NanoSecondDelay;
  Interface->GetPerformanceCounter = GetPerformanceCounter;
  Interface->GetPerformanceCounterProperties =
GetPerformanceCounterProperties;
}
```

3.1.4　测试实例

测试实例是一个包含很多测试实体的模块。每一测试实体会从共享内存中取得测试库的接口，然后执行相关的测试实例。

同样地，测试实例也大致包含如下三个文件夹：

- .MSA：　Module Surface Area（模块表面域文件）
- .H：　C 源头文件
- .C：　C 源代码文件

3.1.4.1　测试实例的源代码树状组织结构的设计

图 3.5 给出了测试实例的树状组织结构。 Common 目录有测试实体的所有源代

码，包括： *Common.c， *Functionality.c， *Conformance.c， *Stress.c 等等。 DXE 目录含有 MSA 文件和一些基于特定阶段的源文件。

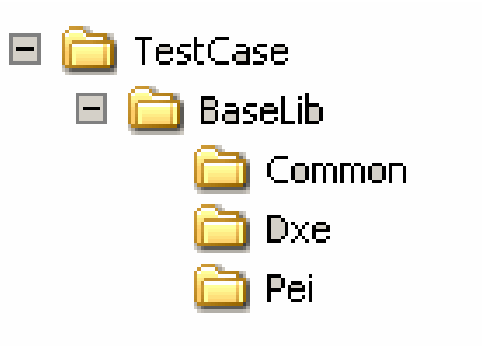

图 3.5 测试实例源代码的树状组织图

在 Common 目录下的源文件，都会被特定执行。大多数情况下，他们被分成 *Common.c, *Functionality.c, *Conformance.c 和 *Stress.c 等等，因而覆盖了所有的 functionality 测试， conformance 测试和 stress 测试[16]。

除了 Common 目录，测试实例可能还有多个其它的目录，这是由此测试实例由多少个阶段来决定的。例如，如果测试库是一个 BASE 库，那么测试实例就应该包括 PEI, PEI Core， DXE, DXE Core， DXE SMM 和 DXE Runtime 阶段。从而意味着在这个测试实例目录中，除了 Common 子目录我们还应包含六个额外的目录。

3.1.4.2 Module Surface Area (MSA)文件

MSA 文件，即模块表面域文件详细说明了在每一阶段相关目录（如： DXE 目录和 PEI 目录）的模块的访问域。它与 EDKII 的主题需求相符合。

3.1.4.3 头文件

头文件被用于储存全局定义， GUIDs 和函数原型，头文件包含以及宏定义。在一个测试实例中，头文件为每一测试实例和测试断言提供 GUIDs，并且详细指定测试实例所依赖的 EFI SCT 包含文件集合。

以 TimerLib 测试实例为例，下面列出了 TimerLibBbTestCommon.h。它应包含 Guid.h 和相应的 Proxy 驱动包含文件。以 Proxy\TimerProxy.h 为例的源代码。它同样也为测试实例和访问点包含了 GUIDs。它声明了所有的测试实体和全局变量。如下所示：

```
#ifndef __TIMER_LIB_BB_TEST_COMMON_H__
#define __TIMER_LIB_BB_TEST_COMMON_H__

#include "Guid.h"
#include <Proxy\TimerLibProxy.h>

#define TIMER_LIB_BB_TEST_REVISION        0x00010000


//
// TODO: Generate new GUID for the category of this test
//

//
// GUIDs for the test case and entry points
//
// {E4C4D568-9866-4f31-8215-E1F4BA106D1F}
#define TIMER_LIB_BB_TEST_GUID            \
  { 0xe4c4d568, 0x9866, 0x4f31, { 0x82, 0x15, 0xe1, 0xf4, 0xba,
0x10, 0x6d, 0x1f }}

// {C35D6218-F7A4-4541-B0A2-B776318388EE}
#define MICROSECONDDELAY_BB_TEST_ENTRY_GUID        \
  { 0xc35d6218, 0xf7a4, 0x4541, { 0xb0, 0xa2, 0xb7, 0x76, 0x31,
0x83, 0x88, 0xee }}

// {5E246166-DDE0-4ae5-B478-2111CD7729F9}
#define NANOSECONDDELAY_BB_TEST_ENTRY_GUID         \
  { 0x5e246166, 0xdde0, 0x4ae5, { 0xb4, 0x78, 0x21, 0x11, 0xcd,
0x77, 0x29, 0xf9 }}

// {9CF2E261-1D19-4e14-920B-E35274270B72}
#define GETPERFORMANCECOUNTER_BB_TEST_ENTRY_GUID         \
  { 0x9cf2e261, 0x1d19, 0x4e14, { 0x92, 0xb, 0xe3, 0x52, 0x74,
0x27, 0xb, 0x72 }}

// {FA9F3345-96A9-450a-9207-26489476C730}
#define GETPERFORMANCECOUNTERPROPERTIES_BB_TEST_ENTRY_GUID
\
  { 0xfa9f3345, 0x96a9, 0x450a, { 0x92, 0x7, 0x26, 0x48, 0x94,
0x76, 0xc7, 0x30 }}


//
// TODO: Test entry point declaration
//

//
// Function declarations
//
EFI_STATUS
EFIAPI
MicroSecondDelayBbTestEntry (
  IN VOID                     *This,
  IN VOID                     *ClientInterface,
  IN EFI_TEST_LEVEL            TestLevel,
```

```
  IN EFI_HANDLE                 SupportHandle
  );

EFI_STATUS
EFIAPI
NanoSecondDelayBbTestEntry (
  IN VOID                       *This,
  IN VOID                       *ClientInterface,
  IN EFI_TEST_LEVEL              TestLevel,
  IN EFI_HANDLE                 SupportHandle
  );

EFI_STATUS
EFIAPI
GetPerformanceCounterBbTestEntry (
  IN VOID                       *This,
  IN VOID                       *ClientInterface,
  IN EFI_TEST_LEVEL              TestLevel,
  IN EFI_HANDLE                 SupportHandle
  );

EFI_STATUS
EFIAPI
GetPerformanceCounterPropertiesBbTestEntry (
  IN VOID                       *This,
  IN VOID                       *ClientInterface,
  IN EFI_TEST_LEVEL              TestLevel,
  IN EFI_HANDLE                 SupportHandle
  );

EFI_STATUS
EFIAPI
GetPerformanceCounterPropertiesBbTestEntry (
  IN VOID                       *This,
  IN VOID                       *ClientInterface,
  IN EFI_TEST_LEVEL              TestLevel,
  IN EFI_HANDLE                 SupportHandle
  );

//
// TODO: Below is common section for all test case head file
//

//
// Common Support Service Functions
//
EFI_STATUS
EFIAPI
CommonAllocatePool (
  IN UINTN      Size,
  OUT VOID      **Buffer
  );

```

```
EFI_STATUS
EFIAPI
CommonFreePool (
  IN VOID     *Buffer
  );

//
// Global variables for Test Case Field
//
extern EFI_BB_TEST_PROTOCOL_FIELD gBbTestProtocolField;
extern EFI_GUID                   gSupportProtocolGuid[];
extern EFI_BB_TEST_ENTRY_FIELD    gBbTestEntryField[];

#endif
```

3.1.4.4　源文件

源文件一般被命名为 xxxLibBbTestCommon.c。大体上说，它应详细说明这个测试实例的基本信息，包括：测试协议域，支持协议 GUID,和测试实体域。它同样也会执行 GetBbTestParametersFromTestCase() 接口来导入这个信息。

以下详细说明了测试实例的源文件：

1. 测试定义

由于测试实例可以在 EFI SCT 下运行来进行 DXE 阶段的测试，一个独立的测试将会被可扩展固件接口的黑盒测试协议结构所定义。这个结构包含关于测试的元数据，包括：修订版，类别，和可阅读的文本描述。其中，每一测试都要求必须有一个协议域。

以 TimerLib 测试实例为例，以下代码是其协议域：

```
EFI_BB_TEST_PROTOCOL_FIELD gBbTestProtocolField = {
  TIMER_LIB_BB_TEST_REVISION,
  TIMER_LIB_BB_TEST_GUID,
  L"Timer Library BB Test",
  L"Timer Library Black-Box Test"
};
```

每一测试都要求有一个全球唯一标识符(GUID)。这个 GUID 被包含在测试协议域中，作为以下形式的结构：

```
EFI_GUID gSupportProtocolGuid[] = {
  EFI_STANDARD_TEST_LIBRARY_GUID,
  EFI_NULL_GUID
};
```

2. 测试实例定义

一个独立的测试可能包含着几个测试实例，这些测试实例反过来可能包含几个测

试断言。每一个测试实例都被测试实体域所详细指定。

以下，仍以 TimerLib 测试实例为例，来说明测试实体域的代码：

```
EFI_BB_TEST_ENTRY_FIELD gBbTestEntryField[] = {
 {
    //
    // TODO: Generate a new GUID to identify this test entry
    //
   MICROSECONDDELAY_BB_TEST_ENTRY_GUID,
   L"MicroSecondDelay",
   L"MicroSecondDelay BB Test",
   EFI_TEST_LEVEL_DEFAULT,
   gSupportProtocolGuid,
   EFI_TEST_CASE_AUTO,
   MicroSecondDelayBbTestEntry
 },
 {
   NANOSECONDDELAY_BB_TEST_ENTRY_GUID,
   L"NanoSecondDelay",
   L"NanoSecondDelay BB Test",
   EFI_TEST_LEVEL_DEFAULT,
   gSupportProtocolGuid,
   EFI_TEST_CASE_AUTO,
   NanoSecondDelayBbTestEntry
 },
   GETPERFORMANCECOUNTER_BB_TEST_ENTRY_GUID,
   L"GetPerformanceCounter",
   L"GetPerformanceCounter BB Test",
   EFI_TEST_LEVEL_DEFAULT,
   gSupportProtocolGuid,
   EFI_TEST_CASE_AUTO,
   GetPerformanceCounterBbTestEntry
 },
 {
   GETPERFORMANCECOUNTERPROPERTIES_BB_TEST_ENTRY_GUID,
   L"GetPerformanceCounterProperties",
   L"GetPerformanceCounterProperties BB Test",
   EFI_TEST_LEVEL_DEFAULT,
   gSupportProtocolGuid,
   EFI_TEST_CASE_AUTO,
   GetPerformanceCounterPropertiesBbTestEntry
 },
{
   EFI_NULL_GUID
 }
};
```

3. 参数传输

每一测试实例应该包括 GetBbTestParametersFromTestCase() 的执行。这个接口以库作为其原型，并被连接到这个库的每一个测试实例进行执行。它会被 DxeBbTestCaseMain（）调用来输入每一个测试阶段所用到的测试参数。

以下部分仍以 TimerLib 的测试实例为例，来说明其

GetBbTestParametersFromTestCase()的执行过程。

```
EFI_STATUS
EFIAPI
GetBbTestParametersFromTestCase (
  OUT EFI_GUID                 **TestGuid,
  OUT EFI_BB_TEST_ENTRY_FIELD    **TestEntryField,
  OUT EFI_BB_TEST_PROTOCOL_FIELD **TestProtocolField
  )
{
  //
  // Check input parameters except TestProtocolField
  // because it is not used in PEI test cases.
  //
  if ((TestGuid         == NULL) ||
      (TestEntryField    == NULL)) {
    return EFI_INVALID_PARAMETER;
  }

  //
  // TODO: Assign the test parameters
  //
  *TestGuid         = (EFI_GUID *) (&gTimerLibTestGuid);
  *TestEntryField    = (EFI_BB_TEST_ENTRY_FIELD *)
gBbTestEntryField;

  //
  // TestProtocolField is assigned as NULL in PEI phase.
  //
  if (TestProtocolField != NULL) {
    *TestProtocolField = (EFI_BB_TEST_PROTOCOL_FIELD *)
(&gBbTestProtocolField);
  }

  return EFI_SUCCESS;
}
```

4. 测试代码

在测试实例的 Common 目录下的 xxxBbTestFunction.c，执行在其功能性部分的 gBbTestEntryField[] 下详细指定的测试实体。(假定在这个例子中，我们只进行功能性测试实体)。每一测试实体将会首先从共享内存中取得被测的库接口。

以下的源代码是测试实例 TimerLib 的测试实体的一个执行过程：

```
EFI_STATUS
EFIAPI
MicroSecondDelayBbTestEntry (
  IN VOID           *This,
  IN VOID           *ClientInterface,
  IN EFI_TEST_LEVEL  TestLevel,
  IN EFI_HANDLE      SupportHandle
  )
{
```

```
  EFI_TEST_ASSERTION      AssertionType;
  TIMER_LIBRARY_INTERFACE  *TimerLibraryInterface;
  UINTN                   DelayMicroSecond;
  UINTN                   DelayMicroSecondByCounter;
  UINT64                  Counter1;
  UINT64                  Counter2;
  INT64                   Counter;
  UINT64                  Frequence;

  TimerLibraryInterface = (TIMER_LIBRARY_INTERFACE *)
ClientInterface;

  //
  // TODO: Add the validation code here
  //

  //
  // Checkpoint1: Should work with a valid delay time.
  //
  DelayMicroSecond = 5000000ull;

  //
  // Get the current value of Performance Counter
  // and put it into counter1
  //
  Counter1 = TimerLibraryInterface->GetPerformanceCounter ();

  //
  // Delay for delayMicroSecond
  //
  TimerLibraryInterface->MicroSecondDelay (DelayMicroSecond);

  //
  // Get the value of Performance Counter and put
  // it into counter2
  //
  Counter2 = TimerLibraryInterface->GetPerformanceCounter ();

  //
  // Calculate to the delay time and the delay time should
  // be positive
  //
  Frequence =
TimerLibraryInterface->GetPerformanceCounterProperties (
                                    NULL,
                                    NULL
                                    );
```

```
    Counter = Counter1 - Counter2;
    if (Counter < 0) {
      Counter = -Counter;
    }

    DelayMicroSecondByCounter = (UINTN) DivU64x64Remainder (
                                    MultU64x64 (
                                      Counter,
                                      1000000ull
                                      ),
                                    Frequence,
                                    NULL
                                    );

    //
    // Compare the DelayMicroSecondByCounter and DelayMicroSecond,
    // DelayMicroSecondByCounter should bigger than or equal to
    // DelayMicroSecond.
    // And the difference should be less than 100 MicroSeconds.
    //
    if ((DelayMicroSecondByCounter >= DelayMicroSecond     ) &&
        (DelayMicroSecondByCounter <  DelayMicroSecond + 100)) {
        AssertionType = EFI_TEST_ASSERTION_PASSED;
    } else {
        AssertionType = EFI_TEST_ASSERTION_FAILED;
    }

    LibRecordAssertion (
      mStandardLibHandle,
      AssertionType,
      gTimerLibBbTestFunctionAssertionGuid001,
      L"TimerLib_MicroSecondDelay ",
      L"%a:%d:Delay %d micro-second, the delay time from counter is
    %d micro-second, the difference should be less than 100",
      __FILE__,
      __LINE__,
      DelayMicroSecond,
      DelayMicroSecondByCounter
      );

    //
    // CheckPoint2:Should work with a boundary delay time.
    //
    DelayMicroSecond = 0;

    //
    // Get the current value of Performance Counter
    // and put it into counter1
    //
    Counter1 = TimerLibraryInterface->GetPerformanceCounter ();

    //
    // Delay for delayMicroSecond
    //
```

```
    TimerLibraryInterface->MicroSecondDelay (DelayMicroSecond);

    //
    // Get the value of Performance Counter and put
    // it into counter2
    //
    Counter2 = TimerLibraryInterface->GetPerformanceCounter ();

    //
    // Calculate to the delay time and the delay time should be
    // positive
    //
    Counter = Counter1 - Counter2;
    if (Counter < 0) {
      Counter = -Counter;
    }

    DelayMicroSecondByCounter = (UINTN) DivU64x64Remainder (
                                    MultU64x64 (
                                      Counter,
                                      1000000ull
                                      ),
                                    Frequence,
                                    NULL
                                    );
    //
    // Compare the DelayMicroSecondByCounter and DelayMicroSecond,
    // DelayMicroSecondByCounter should bigger than or equal to
    // DelayMicroSecond.
    // And the difference should be less than 100 MicroSeconds.
    //
    if ((DelayMicroSecondByCounter >= DelayMicroSecond     ) &&
          (DelayMicroSecondByCounter <  DelayMicroSecond + 100)) {
      AssertionType = EFI_TEST_ASSERTION_PASSED;
    } else {
       AssertionType = EFI_TEST_ASSERTION_FAILED;
    }

    LibRecordAssertion (
      mStandardLibHandle,
      AssertionType,
      gTimerLibBbTestFunctionAssertionGuid002,
      L"TimerLib_MicroSecondDelay ",
      L"%a:%d:Delay %d micro-second, the delay time from counter is
%d micro-second, the difference should be less than 100",
      __FILE__,
      __LINE__,
      DelayMicroSecond,
      DelayMicroSecondByCounter
      );

    return EFI_SUCCESS;
  }
```

3.2　DXE 目录的设计实现

在 Proxy 和 TestCase 目录下，都存在特定的 Dxe 目录，这个目录与相同根目录下的 Common 目录所不同的是，在这个目录下我们存放的是用于特定 DXE 阶段的测试实例和 Proxy 接口的源代码。

本小节设计定义了将测试实例导入到 DXE 阶段进行 MDE 库测试的准则，并且介绍如何在 EFI SCT 下执行 DXE 阶段的测试实例。 MDE 库测试框架的目标之一是能够在不同的阶段做到代码共享，分别是： PEI 阶段， DXE 阶段， SMM 阶段和 Runtime 阶段，等等。这里，仍以 TimerLib 测试实例为例。

EDK II 库测试框架模拟在所有支持它的架构和编译器中的 EDK II 库的使用模型，它也包含了所有支持它的阶段。例如，模块类型是“BASE”的库将会在 6 个阶段被测试： PEI 阶段， DXE 阶段， SMM 阶段， Runtime 阶段， PEI Core 阶段 和 DXE Core 阶段[15]。搭建出来的对于 MdePkg 包的测试框架的主要思想如下：

1. 把库封装成一个独立的模型，我们把这个独立的库模型叫做“Proxy 驱动”；
2. 执行 Proxy 驱动的实体访问点，把库接口加入到在 Proxy 驱动和测试实例之间的共享空间中。例如：配置表；
3. 通过来自共享空间的库接口来执行测试实体，然后取得测试结果。

DXE 测试框架如图 3.6 所示。在其它的阶段，除了对于共享空间的选择或者测试报告的产生，工作流程大致都是相同的。

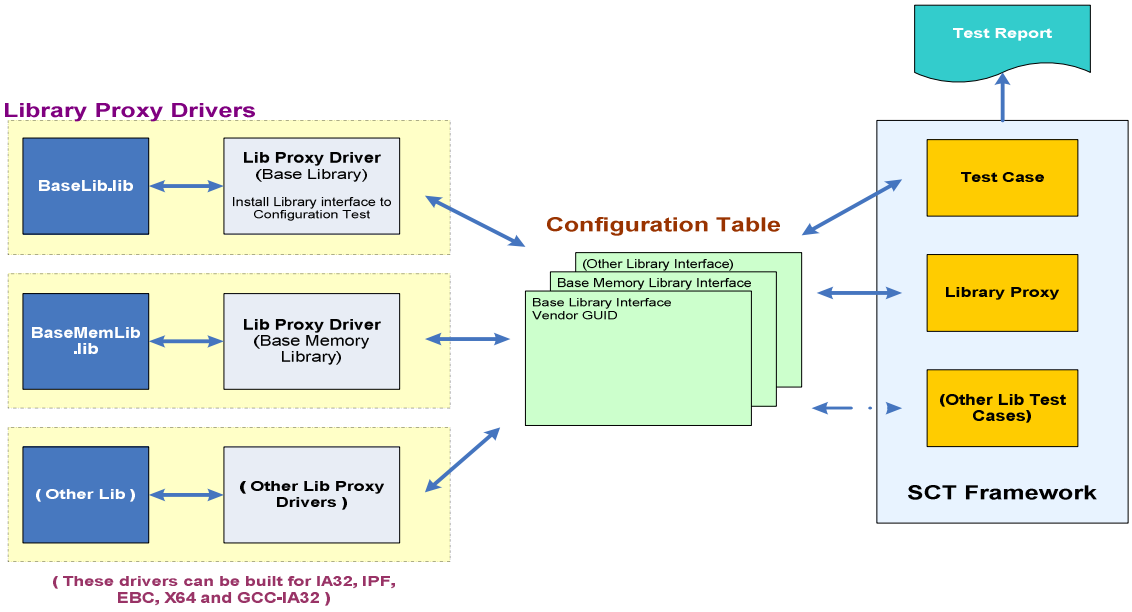

图 3.6 EDK II 库测试框架（DXE）

3.2.1　Proxy MSA file

在 DXE 阶段，这些库类被用在每一个 Proxy MSA 文件中： BaseLib，BaseMemoryLib， DebugLib， UefiBootServicesTableLib， UefiDriverEntryPoint，UefiLib， UefiTestLib， MemoryAllocationLib， DxeBBProxyMainLib， IoLib，PrintLib。

如下所示的是 TimerLib 的 Proxy 驱动的 MSA 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<ModuleSurfaceArea
xsi:schemaLocation="http://www.TianoCore.org/2006/Edk2.0
http://www.TianoCore.org/2006/Edk2.0/SurfaceArea.xsd"
xmlns="http://www.TianoCore.org/2006/Edk2.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <MsaHeader>
    <!--
    TODO: Give a special name to this proxy
    -->
    <ModuleName>DxeTimerLibProxy</ModuleName>
    <ModuleType>DXE_DRIVER</ModuleType>
    <!--
    TODO: Generate a new GUID to identify this MSA file
    -->

<GuidValue>4376A876-BC59-423f-BCEF-0FCB4CE42BBB</GuidValue>
    <Version>1.0</Version>
    <Abstract>FIX ME!</Abstract>
    <Description>FIX ME!</Description>
    <Copyright>Copyright (c) 2006, Intel Corporation</Copyright>
    <License>All rights reserved. This program and the
accompanying materials
      are licensed and made available under the terms and
conditions of the BSD License
      which accompanies this distribution.  The full text of the
license may be found at
      http://opensource.org/licenses/bsd-license.php
      THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS
IS" BASIS,
      WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER
EXPRESS OR IMPLIED.</License>
    <Specification>FRAMEWORK_BUILD_PACKAGING_SPECIFICATION
0x00000052</Specification>
```

```
  </MsaHeader>
  <ModuleDefinitions>
    <SupportedArchitectures>IA32 X64 IPF
EBC</SupportedArchitectures>
    <BinaryModule>false</BinaryModule>
    <OutputFileBasename>DxeTimerLibProxy</OutputFileBasename>
  </ModuleDefinitions>
  <LibraryClassDefinitions>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>BaseLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>BaseMemoryLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>DebugLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>UefiBootServicesTableLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>UefiDriverEntryPoint</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>UefiLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>UefiTestLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>MemoryAllocationLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>DxeBBProxyMainLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>IoLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>TimerLib</Keyword>
    </LibraryClass>
    <LibraryClass Usage="ALWAYS_CONSUMED">
      <Keyword>PrintLib</Keyword>
    </LibraryClass>
  </LibraryClassDefinitions>
  <SourceFiles>
    <!--
    TODO: Put the common source files and support files here
    -->
```

```
    <Filename>InstanceName.c</Filename>
    <Filename>Common\InitializeTimerLibraryInterface.c</Filename>
  </SourceFiles>
  <PackageDependencies>
    <Package PackageGuid="5e0e9358-46b6-4ae2-8218-4ab8b9bbdcec"/>
    <Package PackageGuid="E2BF700A-1704-424A-9185-602085099A9E"/>
  </PackageDependencies>
  <Protocols>
    <Protocol Usage="ALWAYS_CONSUMED">
      <ProtocolCName>gEfiLoadedImageProtocolGuid</ProtocolCName>
    </Protocol>
    <Protocol Usage="ALWAYS_CONSUMED">
      <ProtocolCName>gEfiStandardTestLibraryGuid</ProtocolCName>
    </Protocol>
    <Protocol Usage="ALWAYS_CONSUMED">
      <ProtocolCName>gEfiBbTestGuid</ProtocolCName>
    </Protocol>
    <Protocol Usage="ALWAYS_CONSUMED">
      <ProtocolCName>gEfiWbTestGuid</ProtocolCName>
    </Protocol>
  </Protocols>
  <Guids>
    <GuidCNames Usage="ALWAYS_CONSUMED">
      <GuidCName>gEfiDxeServicesTableGuid</GuidCName>
    </GuidCNames>
    <GuidCNames Usage="ALWAYS_CONSUMED">
      <GuidCName>gTimerLibTestGuid</GuidCName>
    </GuidCNames>
    <GuidCNames Usage="ALWAYS_CONSUMED">
      <GuidCName>gEfiNullGuid</GuidCName>
    </GuidCNames>
  </Guids>
  <Externs>
    <Specification>EFI_SPECIFICATION_VERSION 0x00020000</Specification>
    <Specification>EDK_RELEASE_VERSION 0x00090000</Specification>
    <Extern>
      <ModuleEntryPoint>DxeBBProxyMain</ModuleEntryPoint>
    </Extern>
    <Extern>
      <ModuleUnloadImage>DxeBBProxyUnload</ModuleUnloadImage>
    </Extern>
  </Externs>
</ModuleSurfaceArea>
```

3.2.2　Test Case MSA file

在 DXE 阶段，这些库类被用在每一个测试实例的 MSA 文件中： BaseLib，

BaseMemoryLib， DebugLib， UefiBootServicesTableLib， UefiDriverEntryPoint，UefiLib， UefiTestLib， DxeBBProxyMainLib， IoLib， PrintLib。

如下所示的是 TimerLib 的测试实例的 MSA 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<ModuleSurfaceArea
xsi:schemaLocation="http://www.TianoCore.org/2006/Edk2.0
http://www.TianoCore.org/2006/Edk2.0/SurfaceArea.xsd"
xmlns="http://www.TianoCore.org/2006/Edk2.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <MsaHeader>
    <!--
    TODO: Give a special name to this test case
    -->
    <ModuleName>DxeTimerLibBbTest</ModuleName>
    <ModuleType>DXE_DRIVER</ModuleType>
    <!--
    TODO: Generate a new GUID to identify this MSA file
    -->

<GuidValue>84107E02-1660-43b5-8E6C-595775FF0F0D</GuidValue>
    <Version>1.0</Version>
    <Abstract>FIX ME!</Abstract>
    <Description>FIX ME!</Description>
    <Copyright>Copyright (c) 2006, Intel Corporation</Copyright>
    <License>All rights reserved. This program and the
accompanying materials
      which accompanies this distribution.  The full text of the
license may be found at
      http://www.opensource.org/licenses/eclipse-1.0.php
      THE PROGRAM IS DISTRIBUTED UNDER THE ECLIPSE PUBLIC LICENSE
(EPL) ON AN "AS IS" BASIS,
      WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER
EXPRESS OR IMPLIED.</License>
    <Specification>FRAMEWORK_BUILD_PACKAGING_SPECIFICATION
0x00000052</Specification>
  </MsaHeader>
  <ModuleDefinitions>
    <SupportedArchitectures>IA32 X64 IPF
```

```
      EBC</SupportedArchitectures>
        <BinaryModule>false</BinaryModule>
        <OutputFileBasename>DxeTimerLibBbTest</OutputFileBasename>
      </ModuleDefinitions>
      <LibraryClassDefinitions>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>BaseLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>DebugLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>BaseMemoryLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>UefiBootServicesTableLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>UefiDriverEntryPoint</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>UefiTestLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>DxeBBTestCaseMainLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>TimerLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>IoLib</Keyword>
        </LibraryClass>
        <LibraryClass Usage="ALWAYS_CONSUMED">
          <Keyword>PrintLib</Keyword>
        </LibraryClass>
      </LibraryClassDefinitions>
      <SourceFiles>
        <!--
        TODO: Put the common source files and support files here
        -->
        <Filename>DxeBbTestSupport.c</Filename>
        <Filename>Common\TimerLibBbTestCommon.c</Filename>
        <Filename>Common\TimerLibBbTestFunction.c</Filename>
        <Filename>Common\Guid.c</Filename>
      </SourceFiles>
      <PackageDependencies>
        <Package
    PackageGuid="5e0e9358-46b6-4ae2-8218-4ab8b9bbdcec"/>
        <Package
```

```
    PackageGuid="E2BF700A-1704-424A-9185-602085099A9E"/>
      </PackageDependencies>
      <Protocols>
        <Protocol Usage="ALWAYS_CONSUMED">

    <ProtocolCName>gEfiLoadedImageProtocolGuid</ProtocolCName>
        </Protocol>
        <Protocol Usage="ALWAYS_CONSUMED">

    <ProtocolCName>gEfiStandardTestLibraryGuid</ProtocolCName>
        </Protocol>
        <Protocol Usage="ALWAYS_CONSUMED">
          <ProtocolCName>gEfiBbTestGuid</ProtocolCName>
        </Protocol>
        <Protocol Usage="ALWAYS_CONSUMED">
          <ProtocolCName>gEfiWbTestGuid</ProtocolCName>
        </Protocol>
      </Protocols>
      <Guids>
        <GuidCNames Usage="ALWAYS_CONSUMED">
          <GuidCName>gEfiDxeServicesTableGuid</GuidCName>
        </GuidCNames>
        <GuidCNames Usage="ALWAYS_CONSUMED">
          <GuidCName>gTimerLibTestGuid</GuidCName>
        </GuidCNames>
        <GuidCNames Usage="ALWAYS_CONSUMED">
          <GuidCName>gEfiNullGuid</GuidCName>
        </GuidCNames>
      </Guids>
      <Externs>
        <Specification>EFI_SPECIFICATION_VERSION
    0x00020000</Specification>
        <Specification>EDK_RELEASE_VERSION
    0x00090000</Specification>
        <Extern>
          <ModuleEntryPoint>DxeBBTestCaseMain</ModuleEntryPoint>
        </Extern>
        <Extern>

    <ModuleUnloadImage>DxeBBTestCaseUnload</ModuleUnloadImage>
        </Extern>
      </Externs>
    </ModuleSurfaceArea>
```

至此，我们完成了对模块开发环境 MdePkg 库的底层库函数的测试框架 MdeTestPkg 的搭建，以下需要完成的就是每一个库类的测试用例的编写。在开始具体测试用例的编写之前，以一小节的内容简单介绍一下 build .efi 驱动程序的机制。

3.3　Build .efi 驱动程序的机制

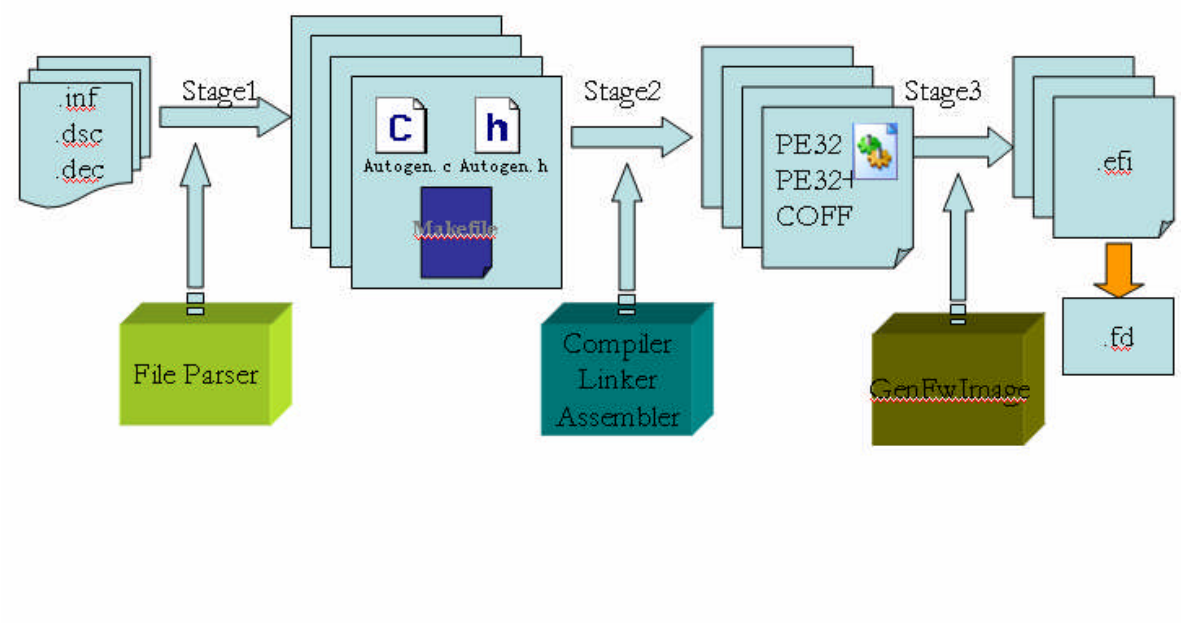

图 3.7 Build 过程的工作流程图

构建 .efi 驱动程序的机制[26]，如图 3.7 所示。

其中，　Stage 1 – 解析元数据文件；

Stage 2 – 处理源代码文件；

Stage 3 – 生成 EFI 格式化固件代码。

当我们要构建一个独立的模块时，需要在 Visual Studio 2005 中，输入如下命令：

> cd \edk2

> edksetup newbuild

> cd MdeTEstPkg/TestCase/ScsiLib/Dxe/UefiScsiLib

>build –p MdeTestPkg/MdeTestPkg.dsc –a IA32

然后开始 build 自动运行，在 build 结束后，将会成功构建出 .efi 驱动程序。

第 4 章　MDE 库在 DXE 阶段的测试用例的设计与编写

模块开发环境库，即 EDK II 框架下的 MdePkg 包包含着 44 个库类和 72 个库实例，每一个库类都对应着一个库的实例，库的实例是一个或多个库类的执行。库的执行可能会用到一个或多个其它的库类，并且也会产生一个构造和/或析构函数。库的实例会和一组模块类型相对应。大体上，库类会执行最小的错误检查，并且大多数库服务不会返回错误状态。库类可以执行的错误检查的最常见的形式是 ASSERT()。

4.1　MDE Library 库的分析与分类

以下总结了在 EDK II 开发环境中提供的库类。其中，包括了由包 MdePkg 产生的库类。每一个开发模块必须详细指定其在模块表面域文件中所用到的库类。这就使得在构建系统时需要在每一个模块表面域文件中为每一个库类包含其公共定义。同样地，库的执行，即库的实例，也会依赖于其他的库类，这同时使得构建系统需要在每一个库的源文件中为那些库类包含公共定义。

在这里，除非额外说明，一个库类可以被任何一个类型的模块所使用。如表 4.1 所示，其概括并详细总结了在模块开发环境包中可用的库类[5]。

表 4.1 模块开发环境包的库类

库名	描述
Base Library	提供字符串函数，链表函数，数学函数，同步函数，和 CPU 架构特定的函数。
Base Memory Library	提供拷贝内存，填充内存，空内存，和 GUID 函数。
Cache Maintenance Library	提供维持指令缓存和数据缓存服务。
CPU Library	提供在 Base Library 中由于对 PAL Library 的依赖而不能被定义的 CPU 架构特定函数。
Debug Library	提供打印调试和断言信息到调试输出设备的服务。

续表　　表 4.1 模块开发环境包的库类

库名	描述
Device Path Library	提供搭建和解析 UEFI 设备路径的库函数。
DXE Core Entry Point Library	Dxe core 阶段的模块进入点库。
DXE Services Library	提供简化 DXE 驱动开发的函数。这些函数能够帮助从 FFS 文件访问数据。
DXE Services Table Library	提供返回 DXE 服务表的指针的服务。只对 DXE 模块类型可用。
ExtractGuidedSection Library	提供处理不同章节数据的常规函数。
HOB Library	提供创建和解析 HOBs 的服务。只对 PEI 和 DXE 模块类型可用。
I/O Library	提供访问 I/O 端口和 MMIO 寄存器的服务。
Memory Allocation Library	提供指定和释放不同内存类型和标准的缓冲器的服务。
PAL Library	提供调用 PAL 的库服务。
PCD Library	提供取得和设置平台配置数据库实体的库服务。
PCI CF8 Library	提供利用 I/O 端口 OXcf8 和 OxCFC 访问 PCI 配置空间的服务。
PCI Express Library	提供利用 MMIO PCI 窗口访问 PCI 配置空间的服务。
PCI Library	提供访问 PCI 配置空间的服务。
PCI Segment Library	在具有多个 PCI 片段的平台上提供访问 PCI 配置空间的功能。
PE/COFF Entry Point Library	提供从 PE/COFF 固件代码获取 PE/COFF 访问点的服务。
PE/COFF Loader Library	提供加载和重定位 PE/COFF 固件代码的服务。
PEI Core Entry Point Library	PEI core 阶段的模块访问点库。

续表　　表 4.1 模块开发环境包的库类

库名	描述
PEI Services Table Pointer Library	提供返回 PEI 服务表指针的服务。
PEIM Entry Point Library	PEIM 的模块访问点库。
Performance Library	提供记录执行时间，然后返回的服务。
Post Code Library	提供发送进程/错误码到 POST 卡的服务。
Print Library	提供向缓存打印格式字符串的服务。支持所有的 Unicode 和 ASCII 串的组合。
Register Table Library (TBD)	提供建立和处理可能包含 I/O, MMIO, PCI 配置，内存和 MSR 操作的寄存器表的服务。
Report Status Code Library	提供记录状态码集的服务。
Resource Publication Library	提供发布所发现的系统资源的服务。
SAL Library	提供调用 SAL 的库服务。
Serial Port Library	提供三个公用串口 I/O 端口函数。
SMBUS Library	提供访问 SMBUS 设备的库函数。这个类型的库必须被导入到一个特定的 SMBUS 控制器中。
Timer Library	提供时钟延迟和性能计数服务。
UEFI Application Entry Point Library	对于 UEFI 程序的模块访问点库。
UEFI Boot Services Table Library	提供取得 EFI 启动服务表指针的服务。只在 DXE 和 UEFI 模块类型中可用。
UEFI Decompress Library	提供利用 UEFI 解压缩算法解压缓存器的服务。
UEFI Driver Entry Point Library	UEFI 驱动， DXE 驱动， DXE Runtime 驱动，和 DXE SMM 驱动的模块访问点库。
UEFI Library	提供公共 UEFI 操作的库函数。只在 DXE 和 UEFI 模块类可用。

续表　　表 4.1 模块开发环境包的库类

库名	描述
UEFI Runtime Library	提供每一 UEFI Runtime 服务的库函数。只在 DXE 和 UEFI 模块类型中可用。
UEFI Runtime Services Table Library	提供返回 EFI Runtime 服务表指针的服务。只在 DXE 和 UEFI 模块类型中可用。
UEFI SCSI Library	提供提交定义在 SCSI-2 规范说明书中用于 Scsi 设备的 Scsi 命令的函数。
UEFI USB Library	提供支持定义在 Usb Hid 1.1 的规范说明书中的隐藏请求和定义在 Usb 1.1 规范说明书中的标准请求的大多数 Usb 应用程序接口。

4.2　DXE 阶段测试 MDE 库的环境搭建

根据本文第三章的设计思想，本章对 EDK II 库在 DXE 阶段测试环境框架如图 4.1 所示：

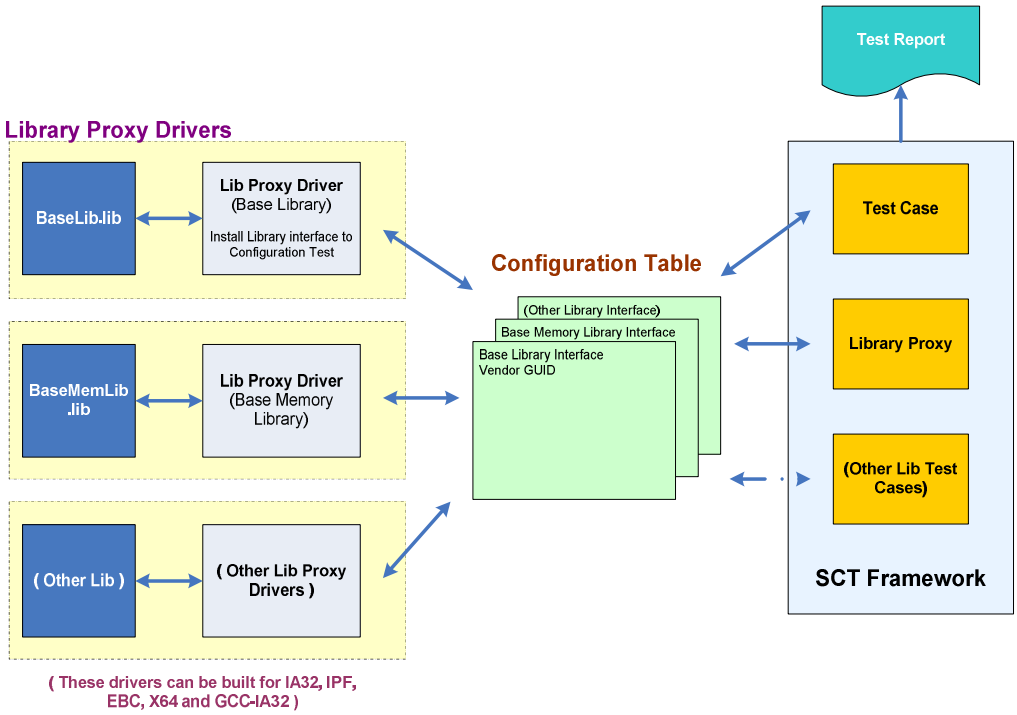

图 4.1 EDK II 库测试框架（DXE）

在以上的库类中，以库类 PcdLib 为例，基于 NT32 平台，设计采取以下步骤测试在 DXE 阶段对于模块开发环境库中的类的测试：

1. 把库类 PcdLib 所必需的 PCD 数据加进 NtS2Pkg.dsc 文件中：把在 MdeTestPkg.dsc 中用到的所有 [Pcds*] 拷贝到 NtS2Pkg.dsc 文件中；

2. 把 Proxy 模块和测试实例模块加进 Nt32Pkg.dsc 中：把以下内容拷贝到 Nt32Pkg.dsc 文件的 [Components.IA32] 段中，如下[19]：

```
[Components.IA32]
  …
  MdeTestPkg/Proxy/PcdLibProxy/Dxe/DxePcdLib/DxePcdLibProxy.inf
  MdeTestPkg/TestCase/PcdLib/Dxe/PcdLibBbTest.inf
```

3. 用修改过的 Nt32Pkg.dsc 文件，在 Microsoft Visual Studio 2005 中通过如下命令构建出 NT32 固件代码：

```
Build –p Nt32Pkg/Nt32Pkg.dsc –a IA32
```

4. 把 build 出来的 DxePcdLibProxy.efi 和 DxePcdLibBbTest.efi 从以下路径分别拷贝到 SCT 目录下的 Proxy 和 Test 目录下，然后像测试其他库一样如下运行 SCT 程序：

```
Build\NT32\DEBUG_MYTOOLS\IA32\MdeTestPkg\TestCase\PcdLib\Dxe\PcdLi
bBbTest\DEBUG↵
Build\NT32\DEBUG_MYTOOLS\IA32\MdeTestPkg\Proxy\PcdLibProxy\Dxe\Dxe
PcdLib\DxePcdLibProxy\DEBUG↵
```

注意：

如果库类 PcdLib 被成功构建出来，那么在如下路径的 AutoGen.c 将会包含在 MdeTestPkg.dec 中所定义的 gEfiMdeTestPkgTokenSpaceGuid 和 GUID_EX 的 GUID 码：

```
C:\edk2\Build\NT32\DEBUG_MYTOOLS\IA32\MdeModulePkg\Universal\PCD\D
xe\Pcd\DEBUG\AutoGen.c↵
```

其中在 gDXEPcdDbInit 中，将会包含 gEfiMdeTestPkgTokenSpaceGuid 和 GUID_EX：

```
/* GuidTable */↵
{↵
    {0x71c3d618, 0x51b4, 0x45a2, {0xba, 0x20, 0x52, 0x6c, 0xb9, 0xd7, 0x37, 0xdf}} ,
    {0xe958edfc, 0xc21b, 0x405b, {0xbb, 0x8d, 0x0b, 0x4a, 0x44, 0x9c, 0xa8, 0xc4}}
```

如果两个 GUID 码没有在 AutoGen.c 文件中，那么所有的 *EX* 操作和在库类 PcdLib 中的 *CallOnSet 将会在执行的过程中停住。在这种情况下，需要耐心细致地

重新温习下 DSC 文件，以找出错误之处。

如果想在 LakeportX64 平台上以同样的方式测试 PcdLib 库类，那么处理的过程将和 NT32 平台一样。

4.3　构建新的 Proxy 和测试实例

当所有的源文件和模块表面域文件被创建以后，构建过程对于我们来说就相当简单了。只需要以下两步：

- 把相关的模块（ Proxy 和 Test Case）利用 FrameworkWizard 工具加入到 MdeTestPkg_xxx.fpd；
- 编译： 在 MdeTestPkg 目录下运行‘ Build ’命令，如果没有错误发生，这个 Proxy 和 Test Case 的 .efi 文件将会在 MdeTestPkg\Build 目录下被创建。

注释： Framework Wizard 工具是一款能够帮助测试工程师和开发工程师配置和更改特定框架构建的选项和设置的可视化图形用户界面。

4.4　运行 DXE 测试实例

在 DXE 阶段的测试实例在 EFI SCT（Self-Certification Test，即自我验证测试）中运行。设计以下步骤来进行 DXE 阶段的测试：

1. 从 www.tianocore.org. 上取得最新的 EFI SCT 二进制包；
2. 把新构建的 Proxy .efi 文件拷贝到 <SCT>/Proxy 目录下；
3. 把新构建的 Test Case .efi 文件拷贝到 <SCT>/Test 目录下；
4. 把如下的入口访问点加入到 <SCT>/Data/Category.ini 文件的最下面；

```
[Category Data]
Revision      = 0x00010000
CategoryGuid  = E4C4D568-9866-4f31-8215-E1F4BA106D1F
InterfaceGuid = 32436d97-f5b1-4a84-889b-40c9e4ed5916
Name          = Timer Library
Description   = Timer Library Test
```

Revision——指明了这个类型数据（即 SCT 包）的结构，在本次 SCT 的发布版本中，应该是：**0x00010000**；

CategoryGuid——指明了测试的 GUID （不是指单一的测试实例），这里应该和之前

我们介绍的在测试源代码中的相同；

上面简单的入口点是用于 TimerLib 的测试实例，所以 **CategoryGuid** 应该用 TIMER_LIB_BB_TEST_GUID，这个 GUID 是在 TimerLibBbTestCommon.h 文件中被声明的，并且引用了 TimerLibBbTestCommon.c 文件中的 gBbTestProtocolField；

InterfaceGuid——指明了 Proxy。这个 GUID 的定义可以在 MdeTestPkg.spd 文件中找到并且被用于相关的模块表面域文件中，例如在 TimerLib 的测试实例的 .MSA 文件中：

```
<GuidCNames Usage="ALWAYS_CONSUMED">
  <GuidCName>gTimerLibTestGuid</GuidCName>
</GuidCNames>
```

gTimerLibTestGuid——在 MdeTestPkg.spd 中被声明并且它的 C 代码在构建时被构建工具自动产生。它的声明如下所示：

```
<Entry Name="TimerLibTest">
  <C_Name>gTimerLibTestGuid</C_Name>

<GuidValue>32436D97-F5B1-4A84-889B-40C9E4ED5916</GuidValue>
  <HelpText/>
</Entry>
```

在把测试实例拷贝和注册到 SCT 运行环境中后，在 EFI Shell 下用命令“SCT –U”运行 EFI SCT 的图形界面[25]。如下图 4.2 所示， TimerLib 库的测试将被集成到测试菜单中。我们选择需要测试的测试实例，然后按 F9 开始执行。

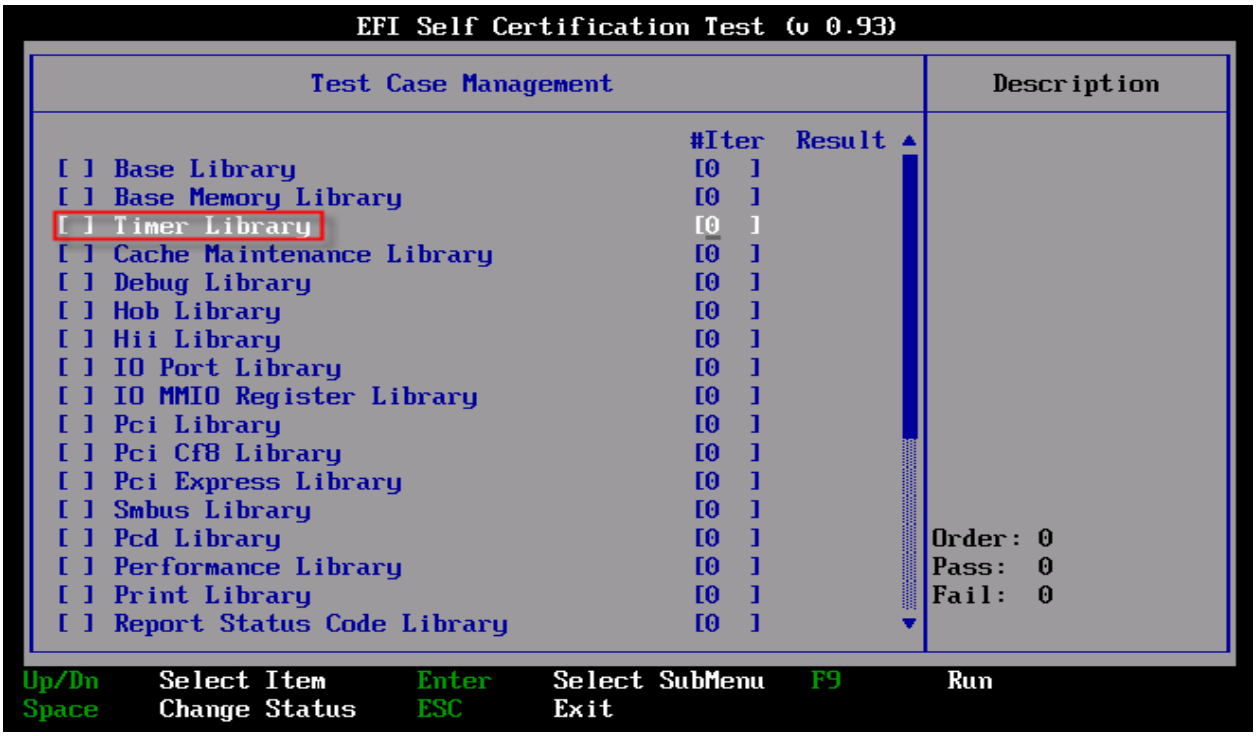

图 4.2 Timer Library 测试图形界面

4.5　如何查看测试结果

1. 启动个人 PC 计算机平台进入 NT32 模拟平台的 EFI Shell 环境；

2. 在 EFI SHELL 环境下，进入 SCT 包中，运行命令“SCT –U”进入到运行 SCT 包的界面环境，然后选中库类 TimerLib 中的所有 Testcase， 点击“run”按钮开始库类 TimerLib 的测试[12]。

3. 当所有测试实例都执行完毕，会在 SCT 图形界面返回所有测试实例的执行结果。返回的结果会有以下三种，分别是： PASS、 FAIL、 NO RESULT。我们需要对返回时 FAIL 和 NO RESULT 的测试实例重新查看我们编写的测试实例，以确定我们设计的测试实例是否正确[9]。

第 5 章　测试结果的统计与总结

5.1　设计一个具体的测试实例

对于在 MdePkg 中的每一个库，在 MdeTestPkg\TestCase 目录下都有一个相应的测试库；同样地，在 MdeTestPkg\Proxy 目录下都有一个相应的 Proxy 库。因此，在 MdeTestPkg 包中正确设计一个测试实例需要同时修改 MdeTestPkg\TestCase 目录下的文件和 MdeTestPkg\Proxy 目录下的文件。设计原则如下：

一. 在 MdeTestPkg\TestCase 目录下：

- 命名规范：

对于 function 测试： XXBbTestEntry()

对于 conformance 测试： XXBBConformanceTestEntry()

- 输入变量：

```
XXBbTestEntry (
                    IN VOID                     *this,
                    IN VOID                     *ClientInterface,
                    IN EFI_TEST_LEVEL TestLevel,
                    IN EFI_HANDLE             SupportHandle
                  )
```

对于每一在 MdeTestPkg\TestCase 目录下的测试实例，在 gBbTestEntryField[] 中加入一个 EFI_BB_TEST_ENTRY 访问点；

在这里，以 StrStrBbTestEntry 为例，需要把以下内容加入到:

```
BaseLibTestCommon.c\gBbTestEntryFiled[]= {
  {
    STRSTR_BB_TEST_ENTRY_GUID,
    L"StrStr",
    L"StrStr BB Test",
    EFI_TEST_LEVEL_DEFALUT,
    gSupportProtocolGuild,
    EFI_TEST_CASE_AUTO,
    StrStrBbTestEntry
```

```
    }
  }
```

然后，把测试实例的 GUID 号加入到 **BbLIbCommon.h 文件中。（此处，我们利用位于 "C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools" 目录下的 GuidGen.exe 来产生 GUID 号）例如测试实例“StrStrBbTestEntry”,把产生的以下 GUID 号加入到 BaseLibBbTestCommon.h 文件中：

```
# define STRSTR_BB_TEST_ENTRY_GUID
{0x75ce41ce, 0xd3db, 0x44b2, 0xa3,{ 0x13, 0x98, 0x5, 0xb9, 0x9d, 0x43, 0x4f}}
```

二. 在 MdeTestPkg\Proxy 目录下：

- 为相应的 Proxy 接口定义函数访问点。必须保证所下的定义与在 MDE 规范说明书中定义的库函数功能相同。例如“StrStrBbTestEntry”，把如下内容加入到 BaseLibProxy.h 中：

```
Typedef
UINTN
(EFIAPI *BASE_LIB_STR_STR) (
  IN CONST CHAR16 *String,
  IN CONST CHAR16 *SearchString
)
```

- 在相关的库接口结构中加入一项（与库的 Proxy 相对应）。以“StrStrBbTestEntry”为例，在 BaseLibProxy.h/BASE_LIBRARY_INTEFACE 结构体中加入如下的声明：

```
BASE_LIB_STR_STR                    StrStr;
```

- 把模块开发环境库函数赋值给定义在 BASE_LIBRARY_INTERFACE 结构体中 Proxy 函数指针。同样以“StrStrBbTestEntry”，把如下的的声明加入到 initializeBaseLibinterface.c\intializedProxInterface() 中：

```
Interface->StrStr =   StrStr;
```

至此，我们完成了一个测试实例的设计与编写。

5.2　经验方法总结

1. 在 DXE 阶段遇到编译问题的调试方法

在设计编写 DXE 阶段测试 MDE 库类的测试实例时，对于所编写的测试实例的调试是不可缺少的一个关键步骤。就 EDKII 来说，在测试框架的搭建和测试用例的设计过程中，总结了调试的方法和技巧，如下：

在 NT32 模拟器中，通过软件 Microsoft VisualStudio2005 Enterprise 中进行调试，这个方法针对 IA32 架构的测试实例程序，比较简单易掌握。其做法具体如下：

- 把如下部分的代码加入到测试实例的 *.msa 文件中：

```
<ModuleBuildOptions>
  <Options>
   <Option BuildTargets="DEBUG" ToolChainFamily="MSFT" ToolCode="CC" SupArchList="IA32">/Od</Option>
   <Option BuildTargets="DEBUG" ToolChainFamily="MSFT" ToolCode="PCH" SupArchList="IA32">/Od</Option>
  </Options>
</ModuleBuildOptions>
```

- 把如下部分的代码加入到 *.fpd 文件的 <BuildOptions> 一项中：

```
<Options>
   <Option BuildTargets="DEBUG" ToolChainFamily="MSFT" SupArchList="IA32" ToolCode="DLINK">/EXPORT:InitializeDriver=_ModuleEntryPoint /ALIGN:4096 /SUBSYSTEM:CONSOLE</Option>
   <Option BuildTargets="RELEASE" ToolChainFamily="MSFT" SupArchList="IA32" ToolCode="DLINK">/ALIGN:4096</Option>
</Options>
```

- 在要调试的代码中加入 “_asm int 3” 来设置断点。
- 然后重新编译测试用例源程序代码，并重新在 NT32 模拟器中执行。当源程序执行到设置断点的位置，将会触发一个异常的应用程序，此时我们通过选择“取消”和“Visual Studio 2005 的新实例”两个选项，程序就进入了调试环境。如下，图 5.1 所示：

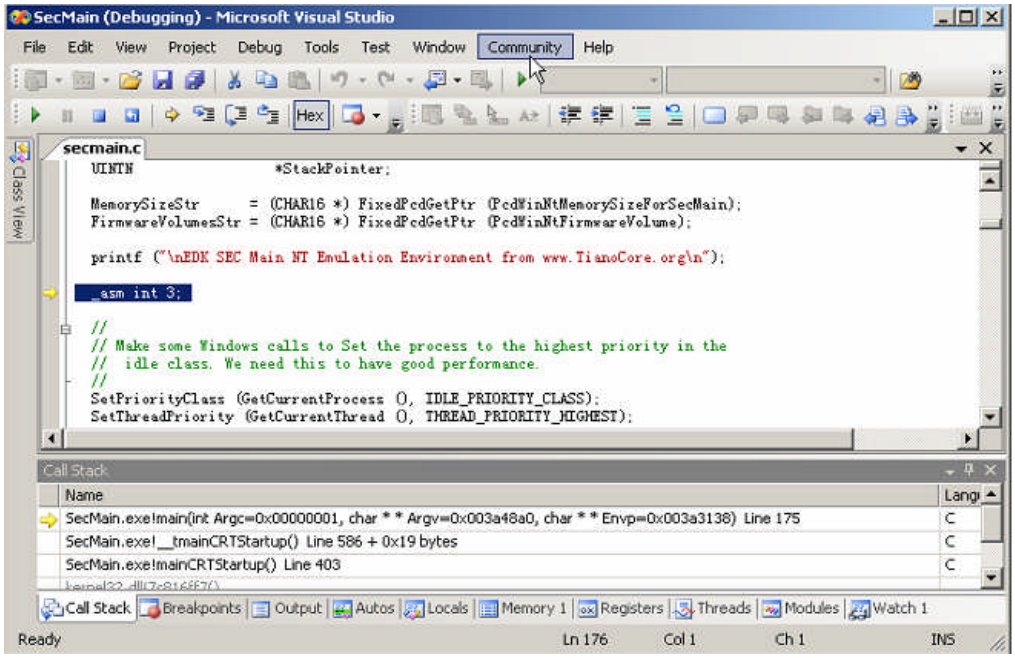

图 5.1　NT32 模拟器的调试环境

2. 错误信息及其解决办法

经常会像下面这样发生典型的错误信息：C:\edk2\MdeTestPkg\TestCase\BaseLib\Dxe\Common not found. 有的时候构建过程中还会提示在相应的目录下没有 C 源文件。出现这个错误的原因是：没有在 Dxe 和 Pei 目录的 Common 目录下建立软链接。在构建过程中有两种方法在 Windows 平台上建立软链接：

1) 运行 Fixup.bat 来链接在 MdeTestPkg 下的所有 Common 目录；

2) 运用整合工具。我们可以在 MdeTestPkg/Application 目录下找到它。这里以 BaseLib 为例，在 MdeTestPkg\TestCase\BaseLib 下有三个目录，分别是：Common， Dxe 和 Pei。其中 Dxe 和 Pei 阶段可以共享公共目录。因此，可以采取以下步骤将其链接到 Dxe 和 Pei 目录下：

   I. 创建一个名字为“Common”的新目录；

   II. 在 Dxe 和 Pei 目录下运行命令：

   >%WORKSPACE%\MdeTestPkg\Application\junctionCommon ..\Common

3. 异常错误信息及其解决办法

在 SCT 运行测试实例的过程中， Windows 会抛出一个“OS exception” 的错误。出现这种情况有以下三种可能的原因：

- 没有访问权限：任何超越用户地址空间去访问内存地址的行为都将会导致程序崩溃。可能在 MdePkg 库的应用程序接口或测试实例代码中存在一个小错误。解决方法：记录下发生这个异常的上下文场景和断点，然后仔细检查测试实例代码；
- 赋给了一个空指针：赋给一个空指针是一个严重的错误，这将导致系统运行时崩溃。这和在编译的过程中系统崩溃不同，原因是程序可以在没有警告的情况下顺利地编译通过。在这种情况下，应该检查代码以确保没有空指针；
- 特权指令： Nt32 平台可能会阻止一些特权指令（如：改变 CPU 状态等）执行特殊的动作。在这种情况下，需要移除调用特权指令的相应模块，然后重新构建包。

参考文献

[1] 洪蕾. UEFI 的颠覆之旅[J]. 中国计算机报. 2007.

[2] 倪光南. UEFI BIOS 是软件业的蓝海[C]. UEFI 技术大会. 2007.

[3] Intel Corporation. Intel Platform Innovation Framework for EFI Human Interface Infrastructure Specification[J]. 2006: 12-23.

[4] Intel Corporation. UEFI 2.1 Porting Guide [J]. Version 2.1. 2007: 18-24.

[5] Intel Corporation. MDE Library Spec [J]. Version 0.2. 2006: 10-30, 41-43.

[6] UEFI Forum. Unified Extensible Firmware Interface Specification. Version 2.3[J]. 2009: 109-122.

[7] Framework Open Source Community. EFI Shell Developer Guide [J]. Ver0.91. 2005: 13-15.

[8] Framework Open Source Community. Edk Getting Started Guide [J]. Ver0.41. 2005: 11-15.

[9] Intel Corporation. UEFI Self-Certification Test Getting Started Guide [J]. Version 2.1. 2009: 21-22.

[10] Framework Open Source Community. EFI_Shell_getting_Started_Guide [J]. Ver0.31. 2005: 10-11.

[11] Framework Open Source Community. Shell_UserGuideRHelp7thOutputFinal2 [J]. 2005: 13-17.

[12] Framework Open Source Community. ShellCommandManual [J]. Ver1.0. 2007: 7-15.

[13] Framework Open Source Community. Pre_EFI Initialization Core Interface Specification (PEI CIS) [J]. 2004: 169-170.
http://www.intel.com/technology/framework/download.htm

[14] Framework Open Source Community. System Management Mode Core Interface (SMM CIS) [J]. 2006: 135-138.

[15] Framework Open Source Community. Driver Execution Environment Core Interface Specification (DXE CIS) [J]. Version 0.91. 2004: 17-24.
http://www.intel.com/technology/framework/download.htm

[16] Intel Corporation. EDK II C Coding Standard [J]. Revision0.51. 2006: 8-21.

[17] Intel Corporation. EDK II Module Development Environment Library Test Infrastructure [J]. Revision 0.02. 2006: 2-13, 114-122.

[18] Intel Corporation. EDKFlash Description File (FDF) Specification [J]. Revision 1.1.

2008: 35-95.

[19]Intel Corporation. EDK II Extended INF File Specification [J]. Revision 1.1. 2008: 11-15.

[20]Distributed Management Task Force (DMTF). System Management BIOS Reference Specification [J]. Version 2.5. 2006: 17-40.
http://www.intel.com/technology/framework/download.htm

[21]Intel Corporation. EDKII user manual [J]. Revision 0.5. 2008: 10-12.

[22]Vincent Zimmer. Beyond BIOS [M]. Intel Corporation. 2006: 17-32, 143-146.

[23]Intel Corporation. Driver Writer's Guide for UEFI 2.0[J]. Revision 0.95. 2009: 9-51.

[24]Intel Corporation. Build and Packaging Architecture Specification [J]. Revision 0.53. 2006: 45-46.

[25]Intel Corporation. EFI Self-Certification Test Case Writer’s Guide [J]. Revision 0.93B. 2005: 43-54.

[26]Intel Corporation. EDK Build Specification [J]. Revision 1.1. 2008: 115-122.

[27]Intel Corporation. EDK II Platform Configuration Datebase Infrastructure Description [J]. Revision 0.55. 2009: 11-13.

致　谢

三年的研究生生活即将结束，心中有着许多的感动与不舍，在论文的最后，怀着感恩的心，再次对师长、亲友表示最诚挚的谢意。

首先尤其要感谢的是我的校内导师李雄飞教授，读研三年期间，每一步都是在李老师的指导下完成的。李老师严谨的治学作风和和蔼的学者气质是我以后的工作和生活中做人的榜样，并鼓舞我不断前进。每当学习，工作中有困难时，李老师都会耐心地提出非常中肯的指导性意见，让我学会很多做人做事的道理。在本论文撰写的过程中，李老师也提出了一系列的修改意见。以后走入工作岗位，希望还能继续不断地向李老师学习，因为李老师是我人生成长过程中的恩师!

还有我的企业导师金晶博士，在我实习期间从学习、工作、做人、做事方面给予我很多指引，而且从论文的选题、实验、到写作的过程中都是在金晶博士的悉心指导下完成的。同时金晶博士对理论知识的准确把握，指导论文时的清晰逻辑，也使我对所研究领域迅速有了概括性的认识，少走了许多弯路。

其次要感谢实习所在公司的领导和同事们在该课题的研究方面提供的巨大的支持，特别感谢 Gu，Alex 和王岩。没有他们在专业培训、研究设备以及工作时间上提供的支持和帮助，该论文将无法完成。

感谢实验室的众多朋友：刘素丽，刘江、倪兴荣、桂坤、徐梅、刘硕；谢谢你们在过去一年里给我的帮助，我们一起度过了一段快乐的时光!

感谢张海龙和孙涛师兄和我的同门郭建芳，以及朋友张东娜在学习和生活中给我的无私帮助，感谢我的室友王晓宇，魏唯和王俊华以及各位研究生同学，日常生活中的点点滴滴都将成为美好的回忆。

最后，深情感谢我的爸爸，妈妈，妹妹和所有其他亲人，是你们在我人生成长的每一步都提出了很多指导意见，尤其是杜小明给与我的支持与鼓励。在新的生活即将来临的时候，写下以上文字，纪念我的研究生生活。

摘　要

基于 UEFI 的底层 API 的性能分析及其功能测试的研究与设计

UEFI 是统一可扩展固件接口，即新一代 BIOS 平台的体系规范，它为平台的底层固件和操作系统之间定义了一个新的接口模型，这个接口不但包括平台的相关信息，而且能够启动操作系统和加载应用程序，并为其提供实时服务的请求。

在 UEFI 开源社区(www.tianocore.org)中，有四个与 UEFI BIOS 相关的开源项目，它们是 EFI Dev Kit（EDK）、 EDKII、 EFI Shell 和 EFI ToolKit。其中 EDKII（EFI Development Kit） 是一个开源的 EFI BIOS 的发布框架，其中包含一系列的开发示例和大量基本的底层库函数，它提供了一个类库，即模块开发环境库 MdePkg，库中包含了所有底层 API，即应用程序接口。因此，对于 MDE 即模块开发环境的底层库函数的分析与测试能够在最大程度上保证开发的稳定性和质量。此外，对于整个分析和测试的设计过程中，能够充分体现出 UEFI 在从事程序设计相对于传统 BIOS 环境下的优势。随着 UEFI BIOS 的进一步推广，越来越多的 OEM 厂商，甚至个人，都会使用这个类库中提供的底层库函数所提供的功能，去开发其他一些基于底层硬件或者操作系统层的新功能。然而，对于当前硬件平台的底层硬件， MdePkg 提供的功能是否稳定并且正确，这就需要我们搭建一个测试框架，从整体上对其进行测试。

MDE(Module Development Environment) 即模块开发环境库，以 EDKII 开发框架作为其依托。在 MDE 库的包里， MdePkg 由开发者设计编写， MdePkg 使得开发者能够在 EDK II 框架下开发自己的模块，同时提供一个非常稳定的并且能够和 UEFI 和 PI 充分兼容的平台，从而使得开发者能够开发 UEFI 驱动， UEFI 程序和 UEFI 操作系统装载器， UEFI 诊断，和 UEFI 平台初始化兼容的硅模块等等。因此，测试 MdePkg 的稳定性与正确性是非常重要的，它决定着开发的质量。

本论文所实现的是模块开发环境库中的 44 个库类在 DXE 阶段的功能分析与测试，如：执行过程是否稳定，其所提供的功能是否正确，并且在所需要的平台架构上是否能够稳定执行。并且由于类库的互通性，使得所要测试的类库能够在不同的平台架构（如： IA32， X64 和 IA64 等）上成功运行，具有很好的稳定性和健壮性。

但在本论文中，最终在 UEFI BIOS 的 IA-32 架构硬件平台上，实现了 MDE 库在 NT32 架构平台上的测试框架的搭建以及对于 MDE 库中的所有库类的测试实例的设计，编写和测试。

具体来说，本论文取得了以下的研究成果：

第一、 对于 EDKII 开发框架以及模块开发环境库所提供的库类的功能的研究

本论文的研究重心在于测试 MDE 库所提供的底层库函数，而这些底层库函数是存在于 EDK II 包中，具体来讲就是 MDE 包以 EDK II 开发框架为依托，因此，对于 EDK II 开发框架的整体架构的研究是论文进展的第一步。

EDKII（EFI Development Kit） 是一个开源的 EFI BIOS 的发布框架，负责为 UEFI 和 PI 的开发提供一个良好的构建和版本跟踪环境。其中包含一系列的开发示例和 44 个基本的底层库函数类，实质上是一个用于装载框架的基本源代码和示例驱动程序的容器。本论文所实现的是模块开发环境库中的 44 个库类在 DXE 阶段的功能分析与测试，如：执行过程是否稳定，其所提供的功能是否正确，并且在所需要的平台架构上是否能够稳定执行。并且由于类库的互通性，使得所要测试的类库能够在不同的平台架构（如： IA32， X64 和 IA64 等）上成功运行，具有很好的稳定性和健壮性。但在本论文中，最终在 UEFI BIOS 的 IA-32 架构硬件平台上，实现了 MDE 库在 NT32 架构平台上的测试框架的搭建以及对于 MDE 库中的所有库类的测试实例的设计，编写和测试。

第二、测试 MDE 库的开发框架 MdeTestPkg 的整体框架架构。

在熟悉了 EDK II 的框架后，我们就要搭建测试 MDE 库的测试框架， MDE 库的测试框架就是 MdeTestPkg 包，其作用就是通过 Porxy 和 TestCase 包所提供的接口去测试 MdePkg 包中所提供的库函数的功能。

MDE（Module Development Environment）是附加在 EDK 和 EDKII 中的一个类库。因此，对于 MDE 即模块开发环境的底层库函数的分析与测试能够在最大程度上保证开发的稳定性和质量。此外，对于整个分析和测试的设计过程中，能够充分体现出 UEFI 在从事程序设计相对于传统 BIOS 环境下的优势。随着 UEFI BIOS 的进一步推广，越来越多的 OEM 厂商，甚至个人，都会使用这个类库提供的底层库函数所提供的功能，去开发其他一些基于底层硬件或者操作系统层的新功能。然而，对于当前硬件平台的底层硬件， MDE 库提供的功能是否稳定并且正确，这就需要我们搭建一个测试框架，从整体上对其进行测试。

第三、编写 MdeTestPkg 下的测试实例，实现对 MDE 库的分析与测试。

EDK II 库的测试架构被设计在包 MdeTestPkg 中实现。库实例是一个或多个库类的执行。库的执行可能有选择性地用到一个或多个其他的库类，并且也可能有选择地产生一个构造函数和（或者）一个析构函数。对于库实例的构造函数和析构函数的调用语法会根据模块类型的不同而不同。同样地库实例也被设计成与一个特定集合的模块类型相连接。如果库实例被声明成一个基模块类型，那么它可以和任何一种类型的模块相连接。

对于在 MdePkg 中的每一个库，在 MdeTestPkg\TestCase 目录下都有一个相应的测试库；同样地，在 MdeTestPkg\Proxy 目录下都有一个相应的 Proxy 库。因此，在 MdeTestPkg 包中正确设计一个测试实例需要同时修改 MdeTestPkg\TestCase 目录下的文件和 MdeTestPkg\Proxy 目录下的文件。

总的来说，本论文研究的依据主要涉及以下几个方面的内容： UEFI BIOS 概括的分析和理解， UEFI 技术的开发框架 EDKII 整体结构分析，重点在于设计搭建 MDE 库的开发框架 MdeTestPkg，从而实现在开发框架 MdeTestPkg 下设计编写在 DXE 阶段下的测试 MDE 库的测试实例。

除此之外，本文还介绍了该项技术在 PEI、 Runtime 等其它阶段和 X64 架构平台上的测试框架 MdeTestPkg 的设计原理以及 MdeTestPkg 下设计测试实例所依据的原理，从而实现对 MDE 库的全面分析与测试。

关键词：

统一可扩展固件接口，模块开发环境，Proxy 驱动，测试实例，EDK II，树状结构图，EFI Shell

Abstract

Research and Design on Performance Analyzing and Function Testing of Firmware API Based on UEFI

UEFI, which is Unified Extensible Firmware Interface， is the system specification of new generation BIOS platform. It defines a new interface module between the firmware and operating system, and the interface not only include the platform related information, but also can boot an operating system and load applications, and supplies runtime services.

In the UEFI open-source community（www.tianocore.org）, there are four open-source projects related to the UEFI BIOS. They are EFI Dev Kit (EDK), EDKII, EFI Shell and EFI ToolKit. EDK II (EFI Development Kit) is an open-source released framework of EFI BIOS, including a series of developing examples and lots of basic API functions, it supplies an library class, that is Module Development Environment ‘MdePkg’, this library class includes all of the APIs, which is called application interface. And so, analyzing and testing on the functions of API Library functions of Module Development Environment can guarantee the stability and quality of the development to its largest extent. Besides, in the process of designing the analyzing and testing flow, we can see that UEFI have an advantage over BIOS within the field of Programing. With the further spreading of UEFI BIOS, more and more OEM, as well as individual, will use the API functions supplied by library classes, to develop some other new functions based on UEFI BIOS or operating system. However, in our present UEFI, whether the functions supplied by MdePkg are correct and stable needs us to set up a framework, to testing it on the whole.

MDE, the shortage of ‘Module Development Environment’, embedded in EDK II developing environment. In the package of MDE, MdePkg is designed by developers, the function of which is to help users develop their own modules in EDK II infrastructure. And at the same time, it can supply a very stable platform that can compatible with UEFI and PI Specification, to help developers develop UEFI drivers, UEFI applications and UEFI operating system loader, UEFI diagnose, and also silicon modules that are compatible with UEFI Platform Initalization. So, the task of testing the stability and correctness of MdePkg is quite important, for it determines the quality of development.

In this paper, I finished the functional analyzing and testing of 44 library classes in

MDE on DXE phase, such as: whether the executing process is stable, whether the functions supplied by MDE is correct, and also whether can be executed correctly on the appointed platform. Due to the interoperability between different platforms, the library classes that need testing can run successfully on different platform (IA32, X64, and IA64 etc.), so possess good stability and robustness. But in my paper, I finally finish the setting up of testing framework of MDE library on NT 32-based platform, and the designing, writing and testing of the test cases of all library classes in MDE, on IA32 platform based on UEFI BIOS.

Specifically speking, this paper achieves the following research outputs:

Firstly, Research on EDK II development framework, as well as the function of MDE Package.

This paper is focus on supplying an environment to test the firmware API of MDE library, and this APIs exist in EDK II package. That is to say, MDE package is embedded in EDK II developing framework. So, setting up the testing framework of EDK II MdePkg is the first step in my paper.

EDKII (EFI Development Kit) is an open sourced releasing-framework based on UEFI BIOS, it is responsible to supply with a good building and edition-tracing environment for developing UEFI and PI. In EDK II, a series of developing cases and 44 firmware APIs are included. And essentially, it is a container that is used to load source codes and samples of drivers. In this paper, the intent is to finish the performance analyzing and function testing of 44 library classes in MDE package during the phase of DXE based, such as: whether the process of executing is stable, whether the functions supplied by MDE Package can work properly, and whether the functions can be executed stably on the needed platform architecture. And due to the interoperability between different platforms, the library classes that need testing can run successfully on different platform (IA32, X64, and IA64 etc.), so possess good stability and robustness. But in my paper, I finally finish the setting up of testing framework of MDE library on NT 32-based platform, and the designing, writing and testing of the test cases of all library classes in MDE, on IA32 platform based on UEFI BIOS.

Secondly, the whole Architecture Framework ‘MdeTestPkg’ of testing the developing framework ’MdePkg’ in EDKII

After getting familiar with EDKII framework, I need to set up the testing framework of MDE library MdePkg. The testing framework of MDE library is MdeTestPkg, the function of which is testing the function of library functions in MdePkg through the

interface between Proxy and Testcase.

MDE (Module Development Environment) is a package in EDK and EDK II, the name of which is MdePkg. So, analyzing and testing the firmware APIs of MDE can ensure the stability and quality of the development extremely. Besides, the whole designing process of analyzing and testing, we can see the advantage of UEFI to Legacy BIOS in the program designing. And with the further spreading of UEFI BIOS, more and more OEM manufacturer, enen individual, will use the function of this firmware APIs supplied by the MDE library classes, to develop other new function based on the firmware hardware or operating system. But, for firmware of the current platform, whether the function of MDE library is stable and correct needs us to set up a testing framework, to test it on the whole.

Thirdly, designing test cases in MdeTestPkg, to analyze and test the function of MDE library.

The testing framewok of EDK II MdePkg is designed in MdeTesPkg. A library instance is an implementation of one or more library classes. A library implementation may optionally consume one or more other library classes and may also optionally produce a constructor and/or a destructor function. The call semantics for the library instance constructors and destructors vary with module type. A library instance is also designed to link with a specific set of Module Types. If a library instance is declared to be a Module Type of Base, then it can be linked with a module of any type.

For each library in MdePkg, there is a corresponding testing library in the directory of MdeTestPkg\TestCase; Likewise, there is a corresponding Proxy library in the directory of MdeTestPkg\Proxy. So, in order to design a test case correctly in MdeTestPkg, we need to alter the files under MdeTestPkg\TestCase and MdeTestPkg\Proxy at the same time.

And generally speaking, the basis that this paper rely on can be summed up from the following aspects: the analyzing and understanding of UEFI BIOS, the researching of global framework of EDK II based on UEFI, and the focus is setting up the testing framework of MDE library, and designing the testcases of MDE library in MdeTestPkg during the phase of DXE.

Besides, the paper introduce the testing framework MdeTestPkg and test cases under “MdeTestPkg” directory designed at the stage of PEI and Runtime,and based on X64-architecture platform to analyze and test the MDE library.

Keywords:

UEFI BIOS, Module Development Environment, Proxy Drivers, Test Case, EDK II, Tree Organization, EFI Shell