ET199超级多功能加密锁_加密锁篇

V1.1

北京坚石诚信科技有限公司

网址：www.jansh.com.cn

修订记录：

修订日期	版本	修订内容
2007 年 6 月	V1.0	第一版发布
2012 年 5 月	V1.1	去除有关外壳加密工具的介绍，初始化增加了修改 ATR 功能介绍，PIN 码管理增加了 PIN 码重试次数介绍，获取硬件信息增加了锁内时间的介绍

软件开发协议

北京坚石诚信科技有限公司（以下简称坚石）的所有产品，包括但不限于：开发工具包，磁盘，光盘，硬件设备和文档，以及未来的所有定单都受本协议的制约。如果您不愿接受这些条款，请在收到后的 7 天内将开发工具包寄回坚石，预付邮资和保险。我们会把货款退还给您，但要扣除运费和适当的手续费。

1. 许可使用

您可以将本软件合并、连接到您的计算机程序中，但其目的只是如开发指南中描述的那样保护该程序。您可以以存档为目的复制合理数量的拷贝。

2. 禁止使用

除在条款 1 中特别允许的之外，不得复制、反向工程、反汇编、反编译、修改、增加、改进软件、硬件和产品的其它部分。禁止对软件和产品的任何部分进行反向工程，或企图推导软件的源代码。禁止使用产品中的磁性或光学介质来传递、存储非本产品的原始程序或由坚石提供的产品升级的任何数据。禁止将软件放在服务器上传播。

3. 有限担保

坚石保证在自产品交给您之日起的 12 个月内，在正常的使用情况下，硬件和软件存储介质没有重大的工艺和材料上的缺陷。

4. 修理限度

当根据本协议提出索赔时，坚石唯一的责任就是根据坚石的选择，免费进行替换或维修。坚石对更换后的任何产品部件都享有所有权。

保修索赔单必须在担保期内写好，在发生故障 14 天内连同令人信服的证据交给坚石。当将产品返还给坚石或坚石的授权代理商时，须预付运费和保险。

除了在本协议中保证的担保之外，坚石不再提供特别的或隐含的担保，也不再对本协议中所描述的产品负责，包括它们的质量，性能和对某一特定目的的适应性。

5. 责任限度

不管因为什么原因，不管是因合同中的规定还是由于刑事的原因，包括疏忽的原因，而使您及任何一方受到了损失，由我方产品所造成的损失或该产品是起诉的原因或与起诉有间接关系，坚石对您及任何一方所承担的全部责任不超出您购买该产品所支付的货款。在任何情况下，坚石对于由于您不履行责任所导致的损失，或对于数据、利润、储蓄或其它的后续的和偶然的损失，即使坚石被建议有这种损失的可能性，或您根据第 3 方的索赔而提出的任何索赔均不负责任。

6. 协议终止

当您不能遵守本协议所规定的条款时，将终止您的许可和本协议。但条款 2、3、4、5 将继续有效。

章节目录

快速入门及注意

■ ET199 多功能锁出厂设置为：

（1）根目录开发商口令：“123456781234567812345678”。

（2）根目录用户口令：“12345678”。

（3）没有重试次数限制。如果您设置了根目录开发商口令的重试次数，那么当使用者连续输入错误的次数达到了最大限制的次数，根目录开发商口令被锁死，这时只能退回给我公司重新生产。

■ ET199 硬件有三种状态：空锁，加密锁（加密锁格式），身份认证锁（PKI 格式）。

（1）出厂默认为加密锁格式，当需要使用身份认功能时，需要使用 5.2.2 节中介绍的加密锁设置工具或者 API 接口（见 7.1.15 节中的说明），将 ET199 格式化为空 锁，然后再使用 PKI 设置工具进行 PKI 格式化（见《ET199 超级多功能锁用户手册－身份认证篇》中的 4.2.2 节）。

（2）将 PKI 格式的 ET199 恢复为加密锁时，需要先使用 PKI 设置工具格式化为空锁（见《ET199 超级多功能锁用户手册－身份认证篇》中的 4.2.1 节），然后再使用 5.2.2 节中介绍的加密锁设置工具或者 API 接口（见 7.1.15 节中的说明），将 ET199 格式化为加密锁。

■ 如果您需要使用 ET199 多功能锁进行开发或者测试请与我公司联系。

■ ET199 多功能锁是 USB 接口的 HID 设备,支持 Win98SE 以上的 Windows 操作系统,包括最 新的 Vista 系统，不需要安装额外的驱动程序。

■ ET199 多功能锁具有 64 位（8 个字节）全球唯一硬件序列号，及软件加密功能和身份 认证功能于一身，适用于软件保护和安全系统身份认证。

■ ET199 用户空间为 64K。硬件擦写次数 10 万次，读没有限制。工作温度：0℃－70℃。

■ 进行开发时请参见本手册的接口说明，以及开发包中的示例，特别是 Samples\CaseStudy 目录下的示例演示，可以直接运行 SampleBrowser.exe 工具进行查看。

■ 在使用 RSA 进行加密时，为了防止被加密的数据大于 n 的情况，请将原文数据的第一个字节设为 0。

第1章 ET199简介

1.1 关于ET199超级多功能锁

ET199 超级多功能锁（以下简称 ET199）是世界上第一款使用智能卡硬件，将软件保 护功能和身份认证功能合二为一的产品。ET199 采用无驱设计，功能强大，价格实在，能够使用在各种领域和众多的用途中，达到一锁多能的目的。

1.2 软件保护

在软件加密方面，ET199 采用了多种先进的关键加密技术，是目前软件保护领域中最安全的保护产品。

- 硬件上的安全性 很多软件被盗版都是由于硬件被复制，不能抵御破解攻击造成的。一个高强度的加密锁首先应有牢固的硬件基础。ET199 采用了安全强度最高的智能卡芯片，硬件不能被 复制，多重安全级别，并且集成了 16 位 CPU，8KRAM，64K 存储空间等模块。ET199 在具有如此强大和安全功能的同时，改进了生产工艺，极大降低了生产成本，从而使广大的软 件厂商不必再花费高额的加密锁成本就能够使用上智能卡型加密锁，从根本上提高了软件的加密强度。

- 硬件上的兼容性 ET199 采用无驱设计，使用高速 HID 协议，在 WIN98 二版以上的操作系统中不需要安装驱动程序，彻底解决了由于驱动安装而给软件开发商带来的各种各样的问题。无驱的 同时，ET199 使用高速协议，对应用软件没有任何速度上的影响。正因为 ET199 在硬件上卓越的兼容性，使用 ET199 几乎不需要任何维护，大大节省了维护成本。

- 先进的软件保护 ET199 就像一个小型计算机，能够完成众多以前只能在 PC 上完成的功能，如复杂的浮点运算。软件开发商可以将自己的程序代码转移到 ET199 中运行，计算机中没有任何 程序的痕迹，同时 ET199 内部的代码任何人也获取不到。软件开发商发行的是一个不完整的软件，只有与 ET199 结合，软件才能够正确执行。由于硬件上的安全性，可以认为 ET199 是不可破解的。

1.3 身份认证

在身份认证方面，ET199 可以做为数字证书的安全载体，敏感数据都被安全地保存 在 ET199 的安全存储区域中，未授权用户是无法接触到这些信息的。数据的签名和加密操作全部在 ET199 内部完成，私钥从生成的时刻起就一直保存其中，可有效的杜绝黑客程序的攻击。ET199 的安全性还在于其使用的加密算法都是被广泛公开，业界公认的， 经受了多年考验的标准算法。同时，一流的芯片封装工艺也保证了芯片内数据的安全性。

ET199 能够硬件产生 512，1024 和 2048 位的 RSA 密钥对，硬件实现 RSA 的各种运算。其采用 16 位的 CPU，结合 EnterSafe 中间件，使用高速无驱的通讯技术，能够十分迅速的完成各种 PKI 应用的操作。

ET199 提供符合业界广泛认可的PKCS#11 和Microsoft CryptoAPI 两种标准的接口。 任何兼容这两种接口的应用程序，都可以立即集成ET199 进行使用。同时，ET199 也针对多个第三方的软件产品进行了兼容性优化。此外，ET199 内置大容量的安全存储器，可以同时存储多个数字证书和用户私钥及其他数据。也就是说，多个 PKI 应用程序可以共用同一个ET199。

第2章 软件保护注意

在您使用ET199进行软件保护时，请务必认真阅读下面的注意，这些都是您在进行软件保护工作时将要了解和用到的。

2.1 ET199硬件上的优越性

- ET199 采用高强度的智能卡安全芯片，硬件不可复制。
- ET199 的用户空间为 64K。不少于 2K 的用户可使用 RAM 空间。能够充分保证在锁内运行的程序享有充足的卡片内存。
- 硬件擦写次数 10 万次，保存 10 年。
- 工作温度：0℃－70℃
- 64 位全球唯一硬件序列号。
- ET199 采用高速无驱设计，将高速和应用简便集成到一体。
- ET199 采用加密的 USB 端口通讯，加密算法在硬件中，保证了传输数据的安全性。

2.2 ET199口令

在使用 ET199 进行软件保护时，ET199 提供了两个级别的口令：开发商口令和用户口 令。不同的口令具有不同的安全权限。当硬件重新插拔或者断电时，安全权限会被重置 成没有认证过的状态。

注意：当硬件被关闭（例如调用 ETClose 函数）时，安全状态不会被重置。开发商口令和用户口令的初始值：

开发商口令：123456781234567812345678

用户口令：12345678

1、开发商口令（24 字节）：该口令是开发商在进行软件保护开发时使用到的， 其作用主要是对 ET199 硬件进行设置，如：创建文件/目录，删除文件/目录等。该口令为 24 个字节，初始值为：“123456781234567812345678”，十六进制表示为“0x310x320x33 0x34 0x35 0x36 0x37 0x38 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38”。虽然开发商口令只能向锁内写入和删除文件，而 不能获取锁内的任何数据，但还是建议开发商在创建完成一个目录后，将默认的开发商口令更改成自己的设定的独特值，以防止被非法者获取。

2、用户口令（8 字节）：该口令是用来调用 ET199 中的可执行文件的。在应用程 序中调用加密锁内可执

行文件时，需要先验证该用户口令。用户口令为 8 个字节，初始 值为：“12345678”，十六进制表示为“0x310x32 0x330x340x350x360x370x38”。

3、开发商口令和用户口令的相关事项：

（1）、开发商口令和用户口令都是针对加密锁中的目录来讲的。ET199 中可以创建 3 级 目录，每个目录都有各自的开发商口令和用户口令。每个目录的口令都是各自 独立的，如：修改了父目录的口令，子目录口令并不更改；取得了父目录的权 限，但不会取得子目录的相应权限。

（2）、在重新创建目录，或者清空目录后，目录的两级口令都会恢复为初始值。为了 安全考虑，这时请开发商一定要设置成自己的特定口令。

（3）、每个目录的开发商口令默认没有重试次数的限制，但为了防止暴力破解，我们 建议开发商自己设定这个口令的重试次数，可以设置为 1-254 次（见 7.1.11 节 中 ETChangePin 函数的说明），当设置为 0 或者 255（0xFF）次时，表明没有限 制。这时应注意：

在验证开发商口令时，当口令不正确，并且连续验证的次数超过了重试次数， 口令将被锁死。这时即使再输入正确的口令，该目录也不能被打开。当根目录被锁死后，没有任何办法可以恢复。为了最大限度的保护您的安全性，我们也无法进行解锁和重新设置，只能退回我公司重新生产。所以请开发商妥善保存好根目录的开发商口令。当子目录锁死后，开发商可以使用其父目录的口令， 清除父目录（见 7.1.9 节中 ETEraseDir 函数的说明）。如：父目录下有 A，B 两 个目录，当 A 目录锁死时，这时必须清除父目录，清除父目录后，父目录下的 所有目录和文件都被删除，父目录的开发商口令和用户口令恢复为默认值。开 发商口令：“123456781234567812345678”，用户口令：“12345678”。然后 再创建新的目录。您也可以使用加密锁设置工具重新建立根目录（见 5.2.2 节）。

（4）、当开发商口令被泄漏或者被盗取时，由于 ET199 设计上和硬件上的安全性，也 不会对软件带来任何安全上的影响，开发商完全可以不必担心。开发商口令验 证后只能写入和删除锁内的数据，而不能获取锁内的任何数据。那么非法拥有 该口令的人只能破坏加密锁中的数据，而不能读到锁内的任何内容，不能进行 软件或者硬件上的仿真,应用软件还是无法使用。必须拥有数据完好的加密锁， 软件才能正确执行。即便如此，我们还是建议开发商要妥善的保存好自己的口 令，增加一道安全措施。

（5）、用户口令与开发商口令一样,默认没有重试次数的限制,但为了防止暴力破解， 我们建议开发商自己设定这个口令的重试次数,可以设置为 1-254 次(见 7.1.11 节中 ETChangePin 函数的说明），当设置为 0 或者 255 次时，表明没有限制。如果用户口令锁死了，这时可以使用开发商口令清除该目录（见 7.1.9 节中 ETEraseDir 函数的说明），则目录的开发商口令和用户口令恢复为默认值。开发商口令：“123456781234567812345678”，用户口令：“12345678”。

2.3 ET199文件系统

ET199 中的文件系统与 Windows 的文件系统一样，是目录/文件结构，文件包括：可 执行文件，数据文件和密钥文件。每个文件和目录都有自己的 ID，这个 ID 为 2 个字节， 例如：0x1002，0x100A 等。

注意：同一级上，目录和文件的 ID 不能重复。

2.3.1 ET199目录特性

1、只有一个根目录，根目录下可以创建子目录。ET199 只支持三级目录结构（包 括根目录，如：\0001\0002，表示根目录（“\”，第一级）下 ID 为 0x0001 的 目录（“0001”，第二级）下的 ID 为 0x0002 的目录（“0002”，第三级））。 根目录占有 ET199 中所有的用户空间。

2、每个目录都有自己的开发商口令和用户口令，请注意不要将根目录的开发商口令锁死，具体内容参看上面的 2.2 节。

3、建立目录时，需要指定目录所占的空间大小。目录建立后，无法修改目录所占 有的空间尺寸。注意：子目录的空间不能超过其所在目录的可使用空间。

4、子目录不能被删除，只能将子目录中的内容（包括子目录下的目录和文件）清 空，在清空后，子目录的开发商口令和用户口令恢复为初始值，具体内容参看 上面的 2.2 节中的说明。如果要删除这个目录，只能将该目录的父目录进行清 空操作。如果目录为根目录，则删除根目录。再次使用时需要重新创建根目录。

2.3.2 ET199文件特性

1、可执行文件：可执行文件是由 C51 语言编写的，在加密锁内部运行的文件。您 的应用程序运行时，调用我们提供的各种语言的 API 接口，向该文件传入输入 数据，可执行文件在加密锁内部运行后，将结果返回给外部的应用程序。

（1）普通可执行文件：可以被其他可执行文件改写，开发商口令可以创建和写入， 用户口令只能运行。文件中的内容不能被任何人读取。当需要远程升级时，只能创建普 通可执行文件，这样可以通过程序改写锁内的可执行文件（见 3.2.8 节中的说明）。

（2）内部可执行文件：不能被其他可执行文件改写，开发商口令可以创建和写入，用户口令只能运行。文件中的内容不能被任何人读取。由于内部可执行文件不能被改写， 因此开发商需要远程升级时，不能创建这种类型的文件。

2、内部数据文件：存放数据信息的文件。该文件在开发商口令验证后，可以通过 API 接口写入。或者通过锁内可执行文件来读取和写入。即应用程序调用外部 的 API 接口，API 接口调用 ET199 内部的可执行文件，可执行文件（通过 C51 语言） 读写内部数据文件。

3、密钥文件：存储 RSA 密钥对（公钥和私钥）的文件。对于公钥文件开发商口令 可以写，可执行文件可以读写。对于私钥文件，开发商口令可以写，但文件中 的内容不能被任何人读取。在使用 ET199 加密锁设置工具（见 5.2 节中的说明） 产生 RSA 密钥对时会返回公钥和私钥的数据，因为私钥文件不能读取，只有在这时才能备份相关数据，因此开发商需要妥善保存，以备以后使用。

各种文件与口令的权限关系总结如下：

		开发商口令验证	用户口令验证	可执行文件
可执行文件	普通	写入	调用	写入
	内部	写入	调用	无
内部数据文件		写入	无	写入/读取
密钥文件	公钥	写入	无	写入/读取
	私钥	写入	无	写入

图表 1 文件与口令权限关系表

2.4 客户号，ATR文件和硬件序列号

客户号：客户号为4个字节，开发商可以通过API接口函数设置和得到这个客 户号（参见7.1.5节），来判断是否是自己的ET199。

ATR文件：ET199内置一个16字节的ATR文件，用户可以通过API接口来写入和读取这个文件中的内容。在写入ATR文件时，需要验证根目录开发商口令。 开发商可以设定ATR文件中的内容，然后根据这个内容来判断是否是自己所购买的锁。

硬件序列号：每一把 ET199 都有一个唯一的硬件序列号，8 个字节（64 位）， 开发商可以通过这个特性在软件中进行处理。

开发商可以通过ETControl接口来对客户号，ATR文件和硬件序列号进行操作，详 见7.1.5节。

2.5 ET199其他功能

- ET199 提供外壳加密工具。开发商可以将 API 函数调用与外壳加密结合起来，对软件进行保护。
- ET199 为硬件级通讯加密,在 USB 接口上传输的数据都是密文。即软件中的数据， 经过加密传送给 ET199，ET199 再在硬件内部进行解密，然后进行操作，反之亦 然。
- ET199 硬件实现 512、1024、2048 位的 RSA 运算功能，开发商可以使用 RSA 非对称 算法对数据进行加解密。同时 ET199 具有 DES 和 3DES 对称加解密功能。在 PKI（公 共密钥基础体系）领域中有数字信封的概念，即将对称密钥通过 RSA 非对称密 钥在双方进行交换，然后双方所有的交换数据都是经过对称密钥进行加解密 的。这种标准的过程也可以使用在软件保护中。
- ET199 提供了 MD5，SHA1 散列算法。散列算法是根据不同的输入，产生固定长度 的对应输出，相当于以前软件保护中经常提到的种子码算法。
- ET199 能够像 PC 一样完成各种复杂的单精度和双精度浮点运算。开发商可以将 应用软件中的公式

放入到加密锁中来完成。浮点数运算是在 ET199 硬件中完成 的，比在 C51 语言中效率高。

- ET199 的指示灯可以通过程序来设置为：开，关，闪烁三种状态。开发商可以 在程序中来进行设置，这样可以起到迷惑作用。如指示灯闪烁时并不一定在调 用锁内的可执行文件。

- ET199 具有远程升级功能。软件升级时，开发商不需要将加密锁回收，只需要 发给用户一个升级程序,就可以将锁内的文件进行升级(见 3.2.8 节中的说明)。

- ET199 中不同的可执行文件可以共享加密锁中的内存区。这样不同的可执行文 件在共享数据时，不必先写到文件中再读取出来，而是直接使用内存指针就可 以实现，减少了硬件的读写操作，增加了运行效率。

第3章 软件加密方法举例

上一章介绍了ET199超级多功能锁的特点，那么怎样才能将ET199与我们开发的软 件结合起来呢？本章将介绍一下软件保护中常见的方法和使用ET199来保护软件。这些 方法都是您在软件保护的书籍中或者相关的网站上可以找到的，在这里只是给大家做个 参考。真正好的软件加密方法还是智者见智，希望每个软件开发商都能够在本章的基础 上，使用ET199设计出自己独特的加密方案，真正达到您的软件不可破解。

3.1 一些简单的加密技巧

对于软件保护来说，有些技巧并不需要你掌握很多汇编或系统低层的知识也能够做 到。我们都知道，不论您的软件采用何种方式来加密，很多情况下总是会归结为某些条 件的判定，如果满足了这些条件，程序就会认为软件是允许执行下去的，否则就会报告一些错误并退出执行。不论你采用任何方法，这些代码是必然存在在你的程序中的，不 要费心思去考虑如何消除这些代码，这些代码必然以这样或那样的形式存在，我们考虑 的重点是如何有效的隐藏和保护这些代码。

3.1.1 有迷惑性的代码

通常来说，编程人员和解密者都具有很好的逻辑思维能力，他们都会认为如果某段 程序存在，那么就应该有这段程序存在的意义。编程人员为了代码的效率和空间会不断 的剔除无用的代码，解密者也是这么理解开发人员的，他通过分析程序中保护代码的每 一段指令来追索编程人员的保护思路。如果反其道而行之，故意添加大量的无用的代码， 把程序复杂化，毕竟开发者是在拿高级语言编写程序，而解密者是在看汇编代码，想搞 清楚哪些代码是有用的，哪些代码是无用的，并不是一件简单的事。

3.1.2 虚假的检查

故意去把返回结果和一些错误的答案进行比较，比较结果一定应该是错误的，如果比较结果发现是正确的，说明有人正在试图破解您的软件。反击的手段可以有多种选择。

3.1.3 验证时机的选择

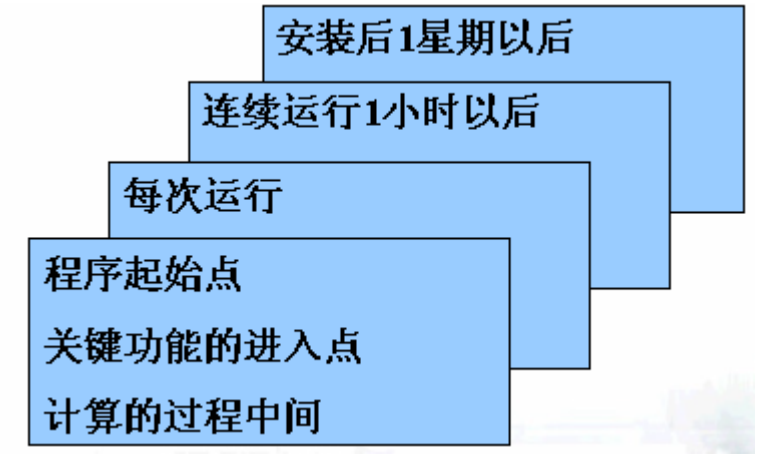

图 3-1 验证时机的选择

软件在运行时，验证时机可以是没有规律的，是散乱的，那么要查找到所有的加密点则不是一件容易的事情。

3.2 使用ET199进行加密

软件加密方法是每个软件开发商结合自己软件特点所设计出来的具有独特性的方 案，每个开发商应该有自己特别的方案，不要雷同，更要做好保密工作，防止破解者根 据经验进行破解。下面对几个方案进行分析，这些方案充分利用了ET199加密锁的硬件 不可复制性、智能卡的复杂运算能力和密钥保密性等特点。

3.2.1 ET199新的软件加密方法

传统的软件加密方法很多只是把加密锁当作存储设备，将简单的信息存储在里面，然后在软件运行时进行判断，也有许多开发商只是检测硬件是否存在，这些加密手段 在现在的破解中十分容易，根本就没有任何加密价值，软件非常容易被盗版。后来有的 加密锁中加入了自定义算法，但由于硬件的限制，算法也只有简单的加，减，左移，右移等，算法再怎么变化也就是这几种之间的组合，最主要的是这样的算法根本不能完成 软件需要的功能。

智能卡技术的介入，使加密锁可以完成以前只有在 PC 中才可以实现的功能，软件开发商可以通过这样的步骤来实现软件加密。

（1）使用高级语言开发软件

（2）将核心算法转换成 C51 语言，进行编译。转换成 ET199 的可执行文件的方式存储在加密锁中。

（3）应用程序运行过程中，遇到关键算法时，将需要运行的数据传给 ET199，ET199 在加密锁内调用可执行文件内完功能，将结果传给应用程序，程序继续执行。

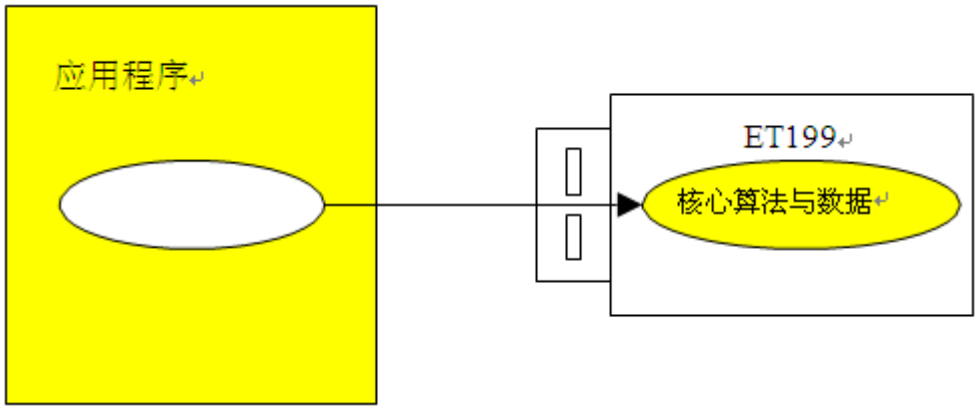

图 3-2 ET199保护软件示意图

很多商业软件，如CAD设计软件，工程预算软件，财务软件等里面包括很多数学计 算的公式，对这样的软件进行高强度的加密是非常容易的。由于ET199具有复杂的双精度浮点运算的能力，可以运算复杂的三角函数，因此能够满足这些软件的计算要求。那 么原来在计算机内存中完成的运算完全可以植入到加密锁中运行，这样破解者只能跟踪到程序中的输入和输出，而不清楚加密锁中的算法是什么，在没有加密锁的情况下，程 序中就没有计算的结果或结果不正确，从而软件不能运行。例如：我们可以将身份证15 位升级到18 位的算法放入ET199 中运行，没有加密锁就没有正确的结果，软件必须靠 加密锁来完成，从而不可破解。这种加密方法需要注意：

（1）算法要够复杂。算法如太简单，如只是一个加法，那么破解者完全可以通过输入和输出猜测到锁内的算法，使用软算法模拟，则程序被破解。

（2）ET199采用的是16位的智能卡芯片，浮点运算能力非常快。能满足多数运算要求。如某些应用软件要求运算速度，可以将复杂公式中的部分放到加密锁内运行。

（3）加密锁内的公式不能是已经公开的或者是泄漏的公式，这些不安全的公式可以被破解者软件模拟。

（4）计算的结果一定要是软件使用到的，而不是在软件中比较。如加密过程是比较的话，破解者很容易找到比较的逻辑判断点，进行屏蔽等修改，这样即使锁内的运算是多么复杂都是无效的。

上述的方法是软件保护中最有效的，ET199就是一个黑盒子，那么只要注意上面的四点说明，使用这样的方法进行保护的软件是不可能被破解的。

3.2.2 在ET199内进行逻辑判断

在对软件进行加密时，可以使用外部的API函数接口或者在锁内使用C51的接口来 获得硬件信息,如硬件ID号,客户号等。由于这些硬件信息是唯一的,因此可以在ET199 加密锁内部针对这些信息进行处理和判断，逻辑处理在加密锁内部，破解者是无法修改 的，从而达到软件不能被破解。在这种加密方发中加入随机数的成份会大大增加加密强 度。如:（1）软件运行时得到一个随机数和硬件ID，将随机数与这个ID进行处理，传 入到锁内的可执行文件中。(2)在锁内进行反向处理，得到硬件ID，这个ID与使用C51 语言得到的硬件 ID 进行比较，进行逻辑判断。这样破解者每次跟踪到的是随机内容，而逻辑的过程又是在加密锁内进行，软件不能被破解。

3.2.3 设置输入和输出迷宫

有的破解者使用USB协议分析仪，对PC与加密锁之间传输的数据进行分析。ET199 在USB接口传输的数据是密文。但为了更加安全，建议软件开发商在传输数据时做一些处理，即加解密的算法由每个软件开发商自己掌握，各不相同。除了ET199的通讯算法 外，再加上每个开发商自己的算法，一家的算法泄漏也不会影响其他的软件。加解密算 法可以开发商自己设计也可以使用DES或者3DES等流行的对称算法其过程如下：（1） 在外部API函数进行调用时，对要传输的数据进行处理。（2）在锁内的程序进行反向处 理，得到真正的输入，再进行下面的程序。

3.2.4 计时功能

开发商可以在第一次运行时，将系统时间记录下来，在锁内创建一个起始时间的内 部数据文件，然后把这个起始时间在加上一个时间（如30天，或使用分钟数等），再创 建一个结束时间的内部数据文件，那么就可以解决如何设定开始时间的问题。

程序第一次运行时，锁内没有这2个数据文件，返回打开失败，开始创建，后面再运行时，这2个文件打开成功，可以向开始时间文件写新读到的系统时间，这时要比较 新的时间是否比开始时间文件中的时间要早，防止用户修改系统时间，然后再与结束时 间比较，看是否在范围内。这样就可以完全实现外部时钟的功能，即节省了成本，又可以防止外部时钟失效的问题。

3.2.5 计次功能

开发商可以在ET199内创建一个内部数据文件用来记录软件的使用次数，软件每使用一次，就将这个文件的次数改写一次，当文件不存在或者记录的次数为0时，软件使 用到期。可以在ET199的可执行文件中进行逻辑处理，使用到期后核心的可执行文件无法运行。

3.2.6 外壳加密功能

外壳加密是保护您的软件最快捷的方法，使用外壳加密工具这种加密方案非常简单，对于没有源代码或没有很多时间编写代码加密的开发者是极为方便的。考虑到软件 的发展趋势，我们的外壳加密工具目前仅对 32 位应用程序有效。目前支持的文件格式有 Win32 PE 、DATA、.NET 和 JAVA。建议开发商在使用 API 调用完成加密工作后， 使用外壳工具再加一层保护，这样破解者要想破解软件必须先进行脱壳，大大增加了软件的加密强度。有关外壳加密工具的详细介绍可参照《ET199 外壳加密工具用户手册》。

3.2.7 DES/3DES，RSA与应用程序的综合运用

ET199加密锁采用的是16位的智能卡芯片，具备高性能的DES/3DES和RSA运算能力。DES/3DES，RSA这些对称和非对称的加解密算法可以使用在单机版软件，C/S结构和B/S结构中的软件中。下面就对C/S结构的软件举例说明：

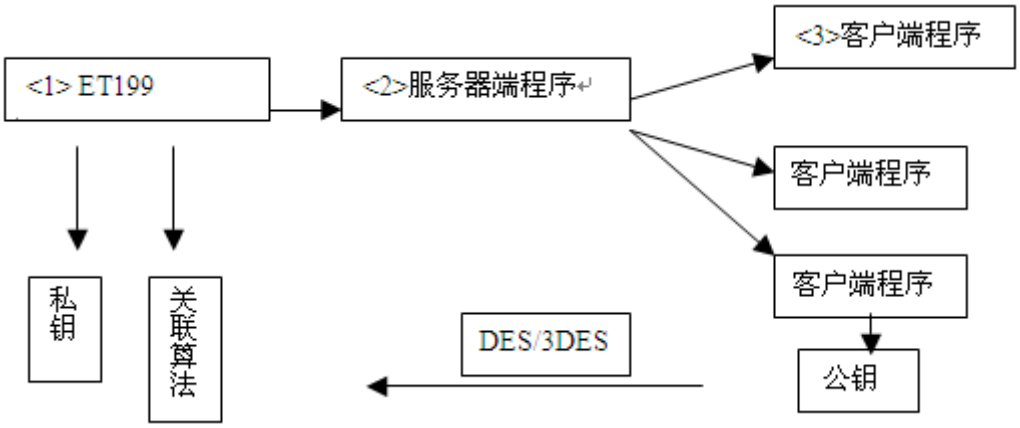

图 3-3 DES/3DES, RSA综合运用

<1>为插在服务器上运行的加密锁，锁内存储有RSA运算的私钥，以及开发商自己 编写的关联算法。

<2>服务器端程序，该程序主要是将客户端传来的数据进行处理，传输给加密锁，或者将加密锁处理的数据传给客户端。

<3>客户端程序，客户端需要有与服务器端加密锁中私钥对应的公钥。

应用程序运行时，客户端的程序产生随机数作为DES/3DES加解密的密钥，使用程 序中存在的公钥对随机数进行加密传给服务器端，服务器端在ET199锁内使用私钥进行解密，并将对称密钥存储成内部数据文件。这样程序运行时在网络上传输的数据都是 使用DES/3DES算法加密过的数据，保证了传输数据的保密性，被跟踪到没有关系。

为了防止解密者篡改客户端程序，即购买一把ET199加密锁，然后使用破解者自己的RSA密钥对，将客户端的公钥替换进行解密，开发商还应该在ET199中加入自己设计 的关联算法，这些算法以C51程序的形式在加密锁中，任何人无法得知其中的过程，那 么即使公私钥被替换,但开发商这些独特的算法破解者是不知道的,软件也无法被破解。 例如：如果要限制客户端的数目，当一个用户发出登录请求时，客户端将客户端的信息 经过软件的DES/3DES加密传给服务器端，服务器端软件将这些信息传入给ET199，加密 锁在锁内进行:(1) 将信息解密，与关联算法得到的信息进行比对，如一致则认为该客 户端有效。(2) 同时在锁内判断登陆数量是否已满。如这两点都满足，则表明客户端是 一个合法用户，将登陆加密的信息发给客户端，客户端使用软件的DES/3DES解密出客 户端需要的正确结果，完成连接。如条件不满足，返回错误的加密信息，虽然表面看起 来,与正确的数据差不多,但这时使用客户端的对称密钥解密后的数据是一堆无用乱码，连接失败。在这个示例中需要注意的就是:(1) 一定要有关联算法，否则很容易被重新生成RSA密钥对，而被破解。(2)各种检查和判断一定要在锁内进行，服务器端的程序只是起到一个传输的作用。

3.2.8 远程升级

软件开发商将加密后的软件卖给最终用户后，在软件进行升级时可以利用远程升级 功能而不必回收加密锁。升级过程如下：

(1)在软件发行时，ET199内部存有可执行文件和内部数据文件，另外还要有 一个用于升级的可执行文件，这里称为UP文件。升级时要保证传输的是密文，您可以使用RSA非对称加解密或者使用DES，3DES对称加解密。这里我们使用RSA非对称加解密，就需要ET199中存在一个私钥。例如 ET199内存在：A（可执行文件）B（内部

数据文件）PRI（私钥文件） 和UP(可执行文件)。

（2）当软件进行升级时开发商需要将新的A’和B’使用与已经发出的ET199 中私钥文件（PRI）对应的公钥文件（PUB）进行加密，将加密后的结果传输给最终用户。

（3）在最终用户处的ET199内部的UP可执行文件接收到A’和B’的密文， 使用锁内的私钥文件（PRI）将A’和B’解密，然后使用C51语言的文件操作接口（见6.3.1-6.3.6节中的说明），将锁内的A和B替换为A’和B’，升级完成。

还可以在传输的密文中加入硬件ID的特定信息，在UP可执行文件中加入硬件ID 的判断等，这样就能对指定ET199进行升级。

3.2.9 签名与验证签名

可以在 ET199 设备内部实现签名与验证签名的操作。ET199 会将 SHA1 散列的结果存 储在 COS 内部，直接在加密锁内进行验证，保证安全性。过程如下（以 1024 位 RSA 密钥对为例）：

（1）A 发送数据给 B，A 先使用_rsa_sign 接口（见 6.4.14 节）对原文（PlainText）进行签名操作，产生 128 字节的签名结果（A_SignText）。

（2）A 将原文（PlainText）和签名结果（A_SignText）传输给 B。B 使用 SHA1 散列算法对原文（PlainText）进行散列操作（见 6.4.5－6.4.7 节），得 到散列结果（B_SHA1Text），这个结果是存储在 COS 内部的，也可以导出。

（3）B 将 A 传来的签名结果（A_SignText）使用_rsa_verify 接口（见 6.4.15 节）进行验证操作，成功返回 0，失败返回非 0 值。

第4章 ET199入门

上一章讲述了一些加密手段和使用ET199的特点来对软件进行加密的方案。那么怎 样来使用ET199呢？本章通过一个简单的示例程序，进一步的阐明使用ET199对软件进 行保护。示例是一个身份证号由15位升级到18位的C语言程序。

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include "ET199_32.h"

unsigned char Wi[18] = {7,9,10,5,8,4,2,1,6,3,7,9,10,5,8,4,2,1};
char Ai[11]={'1','0','x','9','8','7','6','5','4','3','2'};
void ConvertID(char ID[15],char newID[18]) //转换算法
{
int i,j,s;
s=0;
memcpy(newID,ID,6);
newID[6]='1';
newID[7]='9';
memcpy(newID+8,ID+6,9);
for(i=0;i<17;i++)
{
j=(newID[i]-48)*Wi[i];
s+=j;
}
s%=11;
newID[17]=Ai[s];
}
```

```
void main(int argc, char* argv[])
{
char cOldID[16] = "110105720924001";//15 位身份证号
char cNewID[19] = {0}; //18 位身份证号 ConvertID(cOldID, cNewID);//进行转换
printf("%s\n", cNewID);
}
```

面我们就把上面的ConvertID算法转换成C51语言，转换后如下面所示：

```
#include "ET199.h"
#include <string.h>

unsigned char Wi[18] = {7,9,10,5,8,4,2,1,6,3,7,9,10,5,8,4,2,1};
char Ai[11]={'1','0','x','9','8','7','6','5','4','3','2'};
void main(void)
{
int i,j,s;
byte ID[15], newID[18];
s=0;

if(wInLen != 15)    //判断输入的数据是否是 15 个字符
_exit();

memcpy(ID, pbInBuff, 15);  //将外部程序输入的数据赋给 ID 数组
memcpy(newID,ID,6);
newID[6]='1';
newID[7]='9';
memcpy(newID+8,ID+6,9);
```

```
for(i=0;i<17;i++)
{
j=(newID[i]-48)*Wi[i];
s+=j;
}
s%=11;
newID[17]=Ai[s];
_set_response(18, newID);  // 输出新身份证号码
_exit(); //结束程序
}
```

在 C51 程序中有两个参数 wInLen 和 pbInBuff。wInLen 里面的存放的值是外部 PC 程序向C51程序输入数据的长度pbInBuff里面存放输入的数据见6.2.1节中的说明）

我们在KEIL编译器中对C51程序进行编译，得到一个二进制文件（.bin），这时可以把编译好的可执行文件下载到ET199中，有三种方法：

1、在 KEIL 编译器中将程序下载到 ET199 中（见 5.1 节中的说明）。

2、使用 ET199 加密锁设置工具（见 5.2 节中的说明）。

3、使用 ET199 提供的 API 接口（见 7.1.2 和 7.1.3 中的说明）。

假设我们将可执行文件导入到ET199的根目录下，文件ID设为0x0001。原来的程 序我们可以更改为：

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include "ET199_32.h"

void main(int argc, char* argv[])
{
char cOldID[16] = "110105720924001";
char cNewID[19] = {0};
```

```
DWORD dwRet = 0; DWORD dwCount=0;
ET_CONTEXT *pETContextList = NULL;
DWORD dwOut = 0;
BYTE* pbUserPIN = ET_DEFAULT_USER_PIN; //"12345678"

//第一次枚举ET199，得到机器上的ET199的个数
dwRet=ETEnum(NULL,&dwCount);
if(dwRet != ET_E_INSUFFICIENT_BUFFER && dwRet)
{
printf("ETEnum Error:0x%08x\n", dwRet);
return;
}

//分配空间
pETContextList = new ET_CONTEXT[dwCount];
memset(pETContextList,0,sizeof(ET_CONTEXT)*dwCount);

//获得枚举出的ET199的ET_CONTEXT结构
dwRet=ETEnum(pETContextList,&dwCount);
if(dwRet != ET_S_SUCCESS)
{
printf("ETEnum Error:0x%08x\n", dwRet);
return;
}

//打开第一把ET199
dwRet = ETOpen(&pETContextList[0]);
if(dwRet != ET_S_SUCCESS)
```

```
{
printf("ETOpen Error:0x%08x\n", dwRet);
return;
}

//验证根目录的用户口令
dwRet = ETVerifyPin(&pETContextList[0], pbUserPIN, 8, ET_USER_PIN);
if(dwRet != ET_S_SUCCESS)
{
printf("ETVerifyPin Error:0x%08x\n", dwRet);
return;
}

//运行ET199硬件内文件ID为0x0001的可执行文件
dwRet = ETExecute(&pETContextList[0], "0001",
cOldID, 15, cNewID, 19, &dwOut);
if(dwRet != ET_S_SUCCESS)
{
printf("ETExecute Error:0x%08x\n", dwRet);
return;
}

//关闭ET199
ETClose(&pETContextList[0]);

//释放分配的空间
if(pETContextList != NULL)
{
```

```
 delete [] pETContextList;
pETContextList = NULL;
}
//打印出结果
printf("%s\n", cNewID);
}
```

我们看到PC程序中没有任何算法的痕迹，所有算法的运行都是在ET199内部，破解者破解 PC 程序没有任何意义，只能跟踪到旧身份证号和新身份证号输入和输出，根本得不到运算过程，同时程序必须与ET199一起使用才能计算出正确的新身份证号码。 由于ET199采用的是安全强度最高的智能卡芯片，可执行文件的内容是任何人，包括我们都无法得到的。可以认为，只要算法够复杂并且没有外泄，使用ET199进行加密是无法被破解的。

第5章 ET199工具使用说明

上一章通过一个简单的示例演示了整个加密的过程。本章将介绍在使用 ET199 时要用到的工具，包括 KEIL 编译环境的配置，ET199 加密锁设置工具，ET199 虚拟文件系统管理器等。

5.1 KEIL集成环境

运行在ET199 硬件中的可执行文件是使用C51 语言编写的。C51 语言与C 语言的 语法基本一致，该语言经常用于硬件程序的开发上。在KEIL 环境下编译后，出来的是 一个后缀为bin 的二进制文件，这种文件导入到 ET199 中就可以通过外部的 API 函数（见 7.1.15）调用运行了。

KEIL 公司出品的 C51 语言的编译器为 uVision，您可以在 KEIL 公司的网站（http://www.keil.com）上下载到测试版，测试版与正式版相比有一些限制。本节中以 uVision2 的版本来进行说明。

5.1.1 KEIL环境设置工具

开始使用 KEIL 编译器进行开发时需要配置一些工程选项，为了方便配置，我们提供了一个配置工具。界面如下：

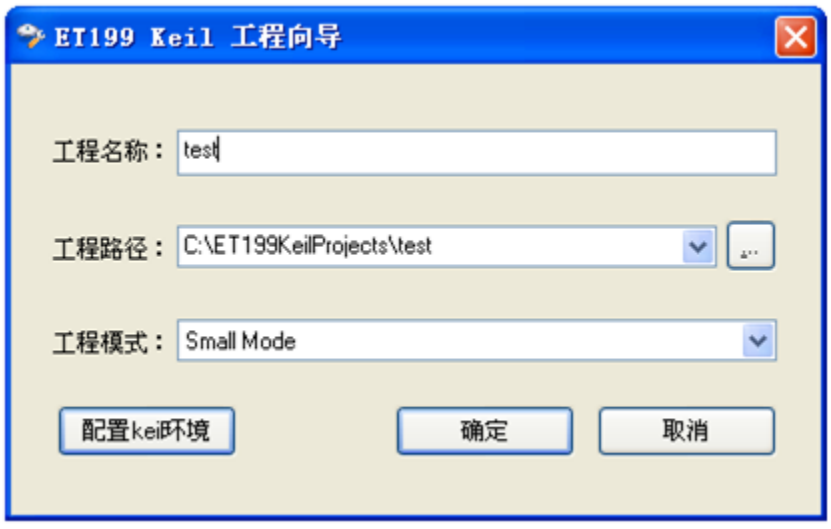

图 5-1 配置工具

在对话框中输入工程名称，工程路径和工程模式（建议使用Small Mode模式）鼠标左键单击“配置keil环境”按钮，弹出选择KEIL安装目录的对话框，见下图：

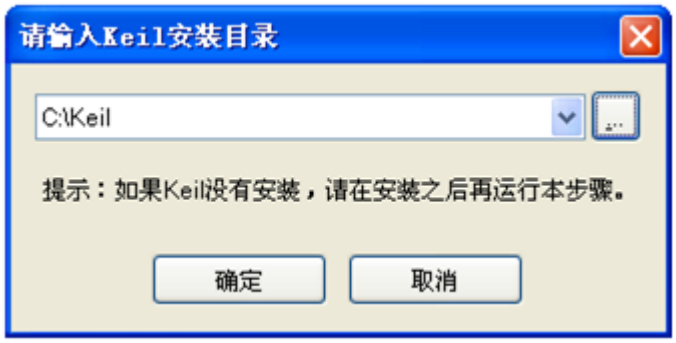

图 5-2 选择KEIL安装目录

鼠标左键单击  按钮，浏览KEIL 安装目录。按“确定”按钮后回到工程向导对话框，在此对话框上按确定按钮后，完成KEIL 环境的配置，并创建了一个新工程。

由于在配置KEIL 环境和创建工程的过程中需要拷贝库文件和C51 的头文件,因此 需要注意：在进行自动配置时要保证：TOOLS目录、LIBRARY目录和INCLUDE目录在一起 。即您可以将这三个目录拷贝到同一个目录下，然后运行 TOOLS 目录下的 KeilWizard.exe 工具。如果自动配置失败，您需要按照下面的过程进行手动配置。

5.1.2 配置KEIL集成环境

手动设置一些工程选项：

（1）将ET199Sim.dll（Tools目录下）复制到KEIL安装目录中的\C51\BIN目录下。

（2）修改KEIL安装目录下的TOOLS.INI文件，在C51段添加如下语句：TDRV4 = BIN\ET199Sim.dll ("ET199")，其中TDRV4是个序号，如果已被占用，则引用下一个数字。

5.1.3 创建工程

打开KEIL uVision2 的“Project”菜单中选择“New Project”菜单项建立新的工 程，在弹出的保存对话框中输入工程的名字进行保存。在保存工程文件名时显示 “Options for Target ‘Target1’”，选择 51 系列的CPU，如 Intel 80C51BH，对于 已有工程如没有选择51系列CPU的要重新选择，见下图所示：

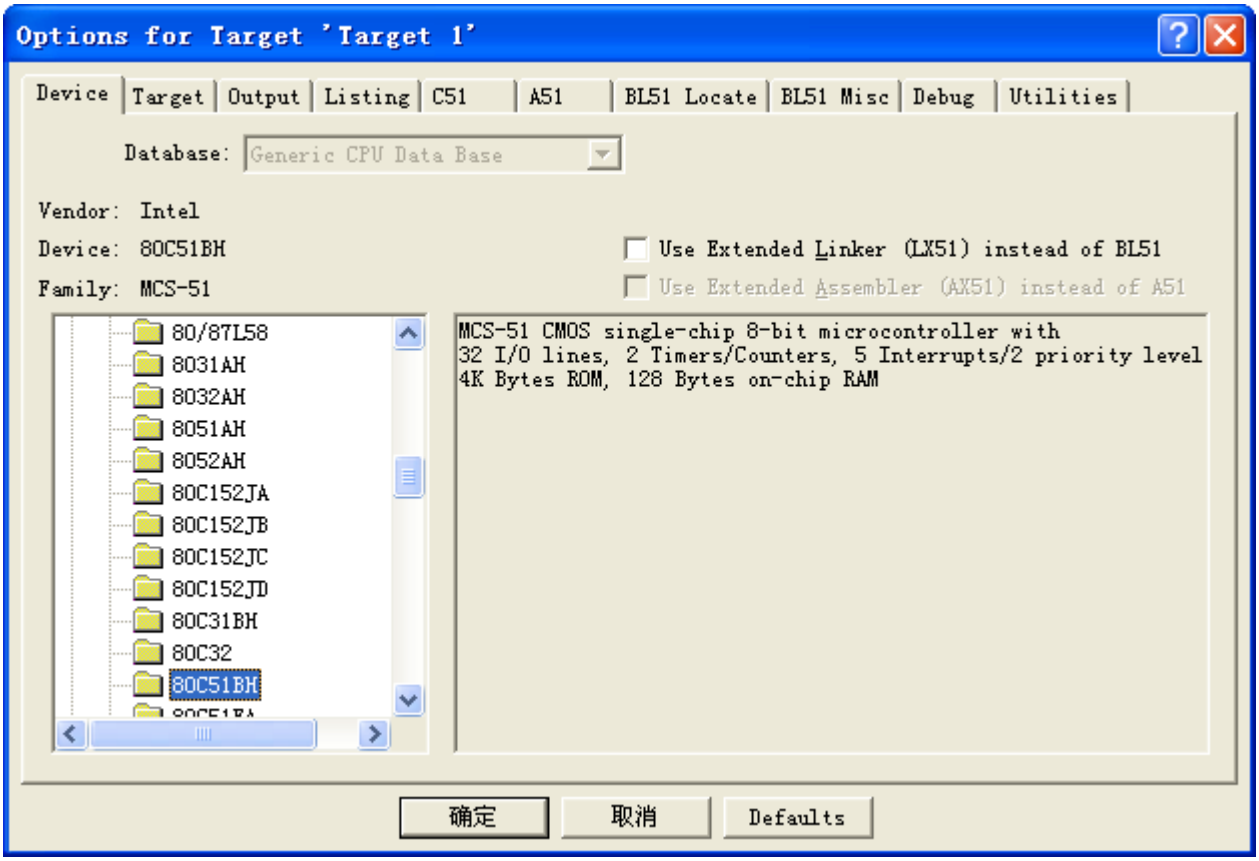

图 5-3 创建工程

5.1.4 设置工程选项

打开菜单项项“工程 Options for Target’Target1’”，对于 Target 页中 Memory Modul 可以选择默认的 Small 模式，建议用户使用这种模式，见下图所示：

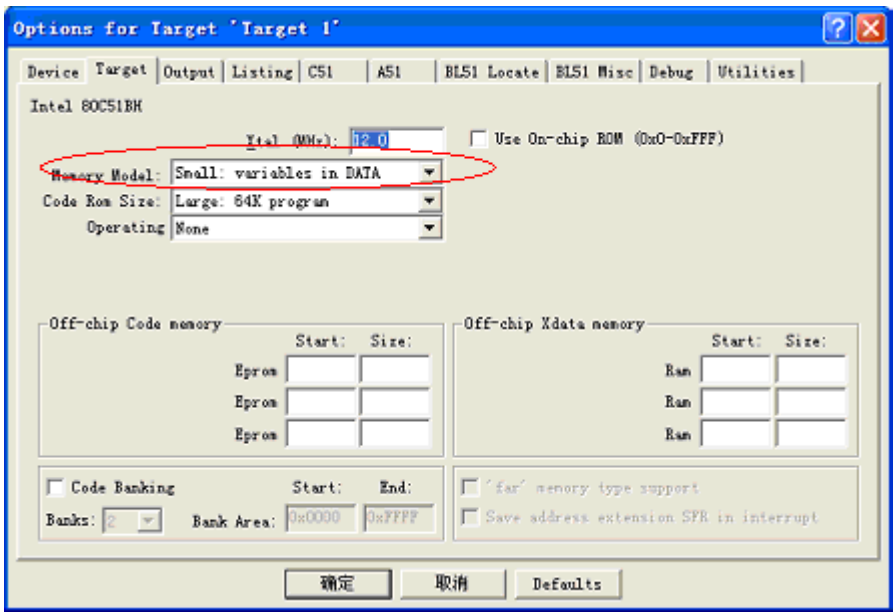

图 5-4

对于在 5.1.3 中打开的界面“工程 Options for Target’Target1’”，在 output 页中选中复选项“C

reat HEX File”,同时复选“Run User Program #1”输入“hexbin.exe test.hex test.bin”，其中 test.hex 和 test.bin 根据工程的名称不同而不同，使用前请将 hexbin.exe 文件（Tools 目录下）拷贝到您的工程目录下。hexbin.exe 工具的功能就是将 KEIL 产生的十六进制 HEX 文件转变成二进制 BIN 文件，该 BIN 文件可以导入到 ET199 中。见下图所示：

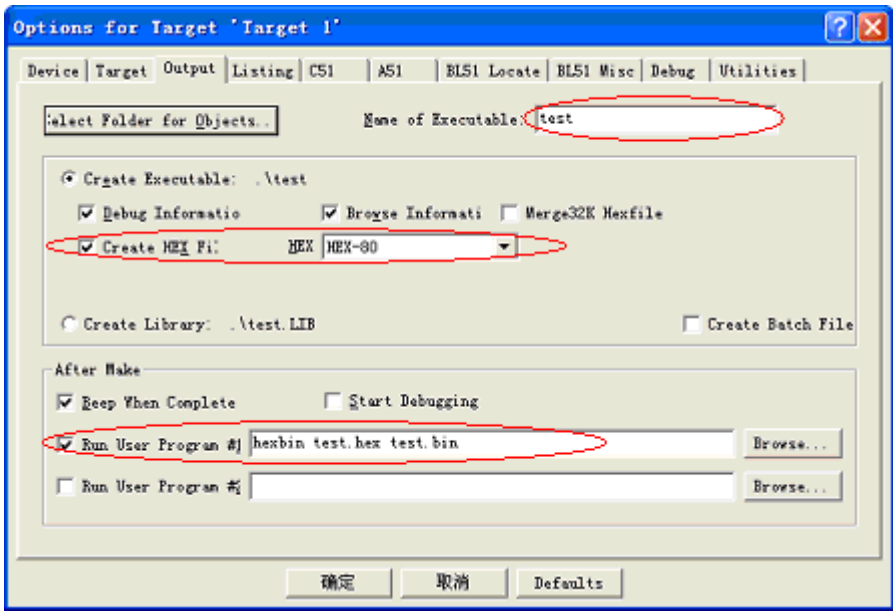

图 5-5

在 DEBUG 一栏中选择使用 ET199 的仿真器，即选中右上侧的“Use:”并在下拉框中选择 5.1.2 中讲到的 TOOL.INI配置文件中设好的ET199的仿真器，选择“Go till main”，在进行 ET199编程的时候如果需要进行调试时，需要设置该项，它的目的是跳过程序的 起始初始化代码，直接运行到main函数开始接受调试。见下图所示：

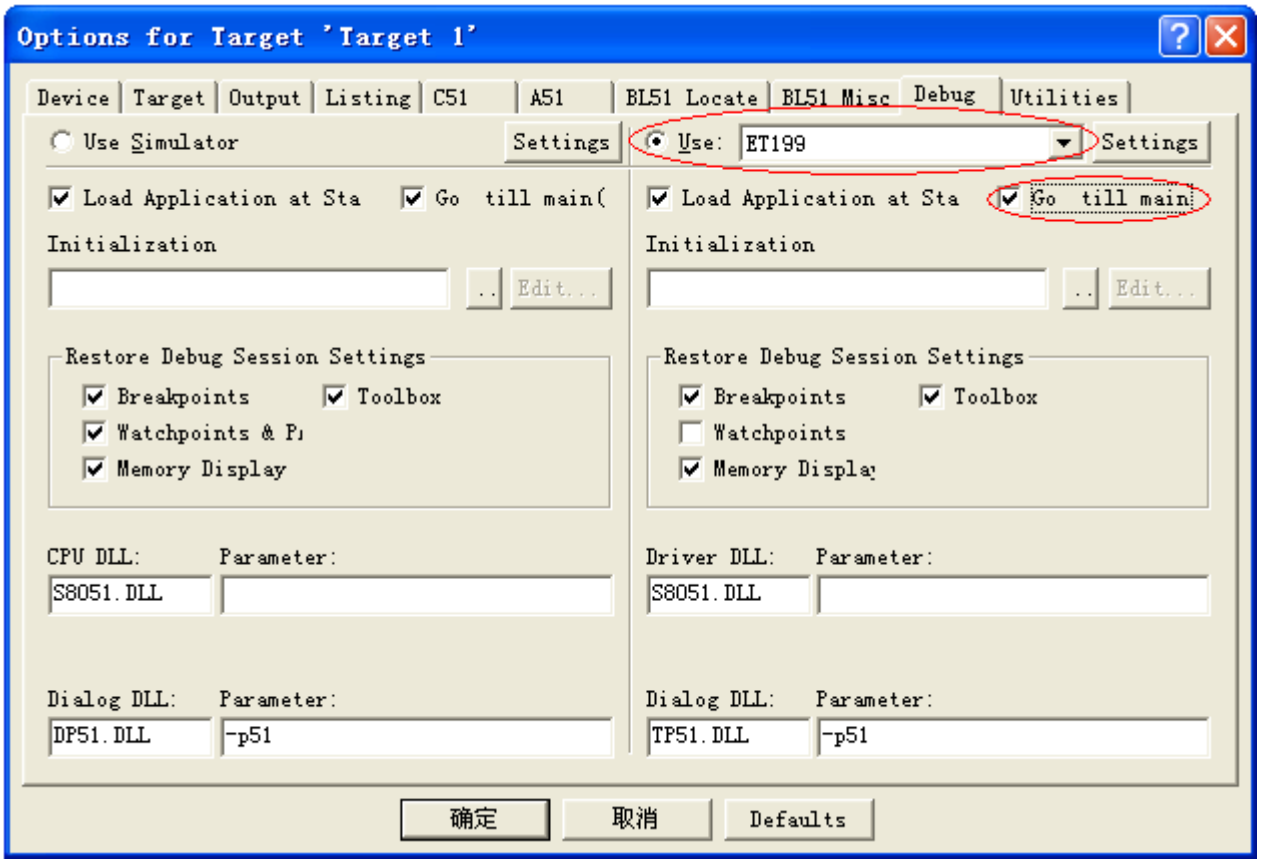

图 5-6

这时鼠标左键单击“Settings”按钮，进行虚拟文件系统的设置。虚拟文件系统是为了方便开发而设计的工具（见5.3节中的说明）。点击按钮后弹出界面如下图所示：

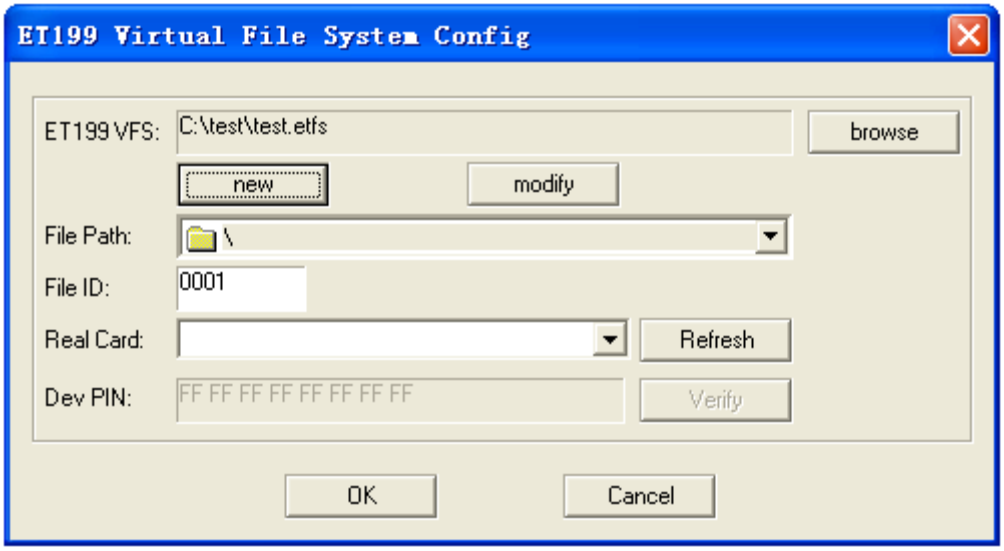

图 5-7

- 鼠标左键单击 new 按钮，打开保存对话框，保存一个新的虚拟文件系统。一个虚拟文件就相当于一把 ET199。
- 鼠标左键单击 modify 按钮，可以创建或者删除虚拟文件系统中的目录和文件（见 5.3 节中的说明）。

- File Path 用来选择可执行文件在虚拟文件系统中的保存路径。
- File ID 用来设置可执行文件的 ID。十六进制表示（0-9，A-F，不区分大小写）。
- Real Card 用来选择是使用虚拟环境还是使用真实设备。建议开发时在虚拟设 备中进行。在虚拟环境中，为了开发方便，是不需要验证开发商口令和用户口令的。
- Dev PIN 后面输入目录的开发商口令，这里只是在真实卡时有效。

最后在“Utilities”一栏中选择ET199调试器，配置了此项后，就会在KEIL主界 面中出现一个  按钮，这个按钮可以在您调试完程序后将C51程序下载到虚拟文件系统或者ET199中。见下图所示设置：

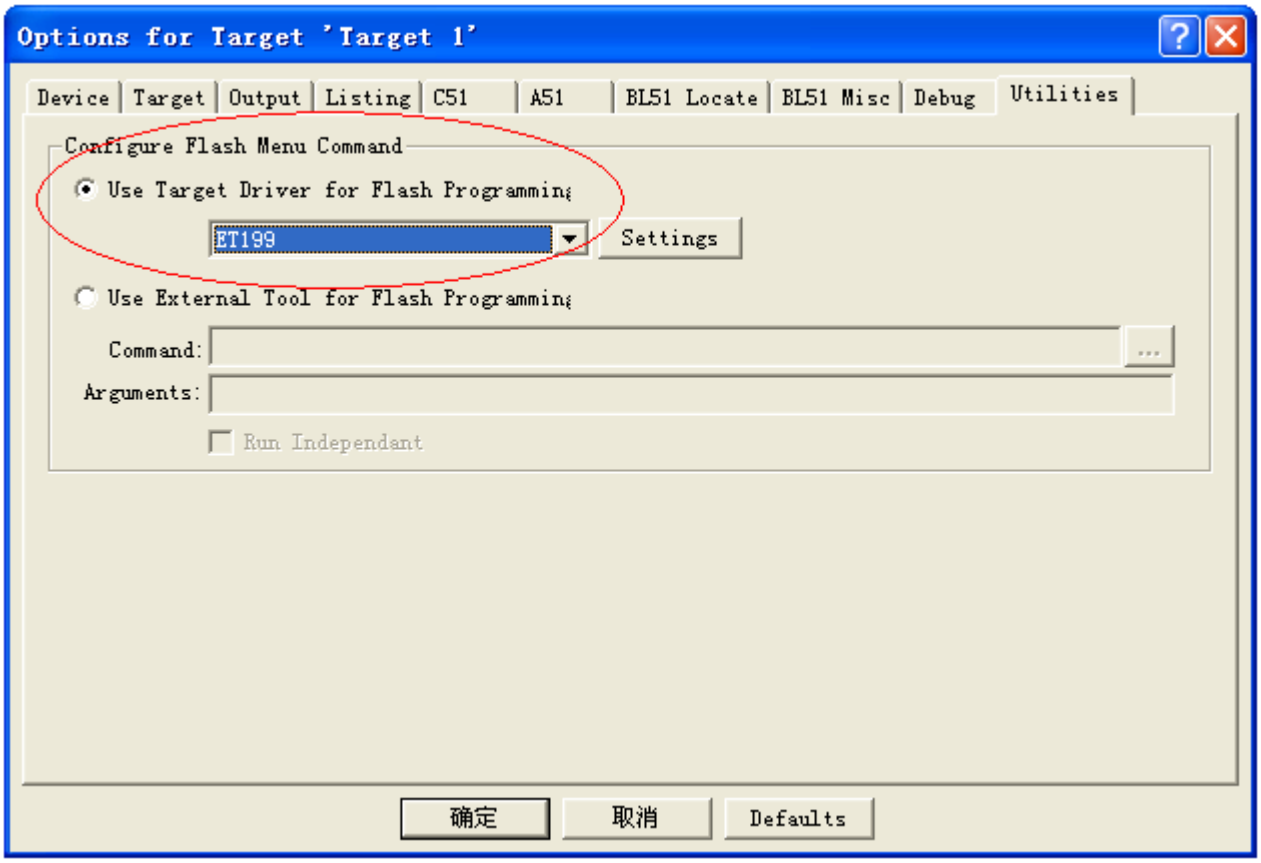

图 5-8

完成上面的工程配置后就可以向工程中添加文件了。选中菜单项“View ->Project Window”,在右边显示树列表，一般默认都已经打开。选中“SourceGroup1”点击右键 选中 Add files to Group ‘Source Group1’将 et199.h 和 small_mode.LIB 添加到工程 中，然后加入用户的 C51 文件。测试时可以加入我们提供的 C51 示例。加入方法见下图所示：

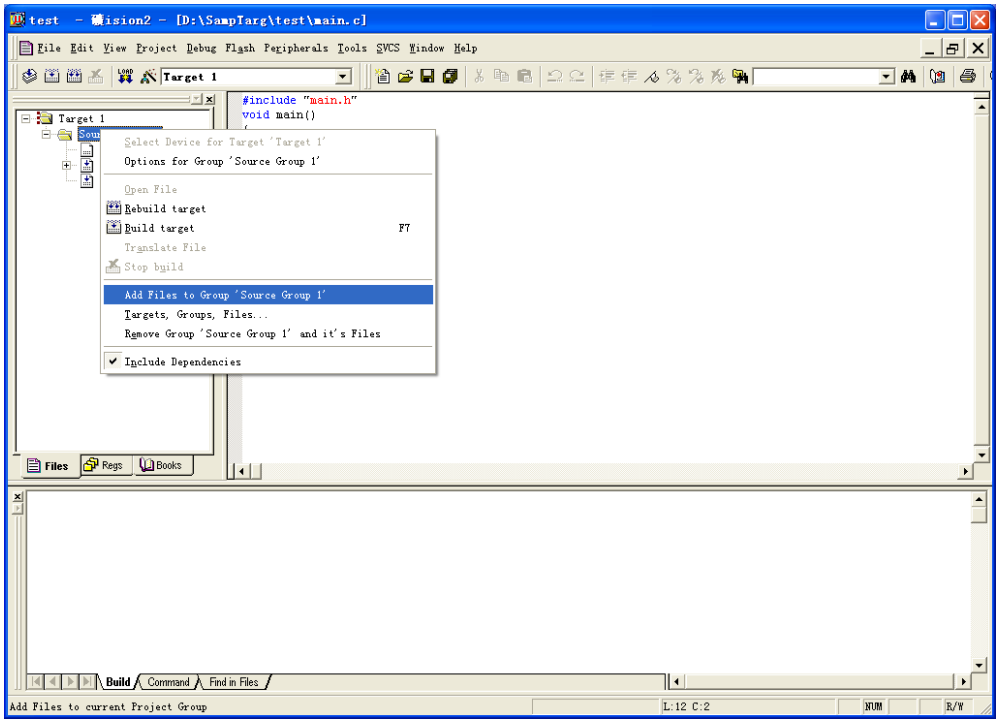

图 5-9

5.1.5 调试

完成上述过程，程序进入调试状态。菜单项“View->ProjectWindow”窗口中显示 的是 C51 的寄存器。关于 Keil uVision2 的使用和调试，请参考其自身帮助说明。

5.1.6 退出

为了让 C51 程序停止执行并结束调试，程序末尾必须调用_exit()函数（见 6.1.1 中的说明）。ET199 仿真器在遇到_exit()时，立即使 Keil 退出调试状态，与用户选中菜单 项“DebugÆStart/Stop Debug Session”功能相同。当然，用户可在程序调试的任何 阶段，选中“DebugÆStart/StopDebug Session”菜单项，结束调试。

5.1.7 写入C51程序

在配置完 Utilities 页中的“Using Targer Driver for Flash programming” 项后就可以在 keil 环境中直接将 C51 程序写入到虚拟卡中或者真实卡片中（根据 5.1.4 节，图 5-8 的设置），在 keil 中选择 Flash 菜单中 Download 菜单项，即可完成写卡。

5.1.8 测试范例

通过上面的过程配置完成后可以使用一个简单的C51程序来检查配置的是否成功。 如将输入的“HELLO”输出，程序如下：

```
#include "ET199.h"
```

```
#include <string.h>


void main(void)

{

byte xdata p[10];

if(wInLen != 5) //判断输入的数据是否是5个字符

_exit();

memcpy(p, pbInBuff, wInLen);//将输入的"HELLO"赋给p

_set_response(5, p);    //输出p

_exit();

}
```

编译通过后，可以在 KEIL 中进行调试。选择 KEIL 编译环境菜单上的“Debug”菜 单，在下拉菜单上鼠标左键单击“Start/Stop Debug Session”调试程序，程序运行停止时，这时可以按 F5 键让程序继续，这时会弹出虚拟文件系统的输入数据对话框，如下图所示：

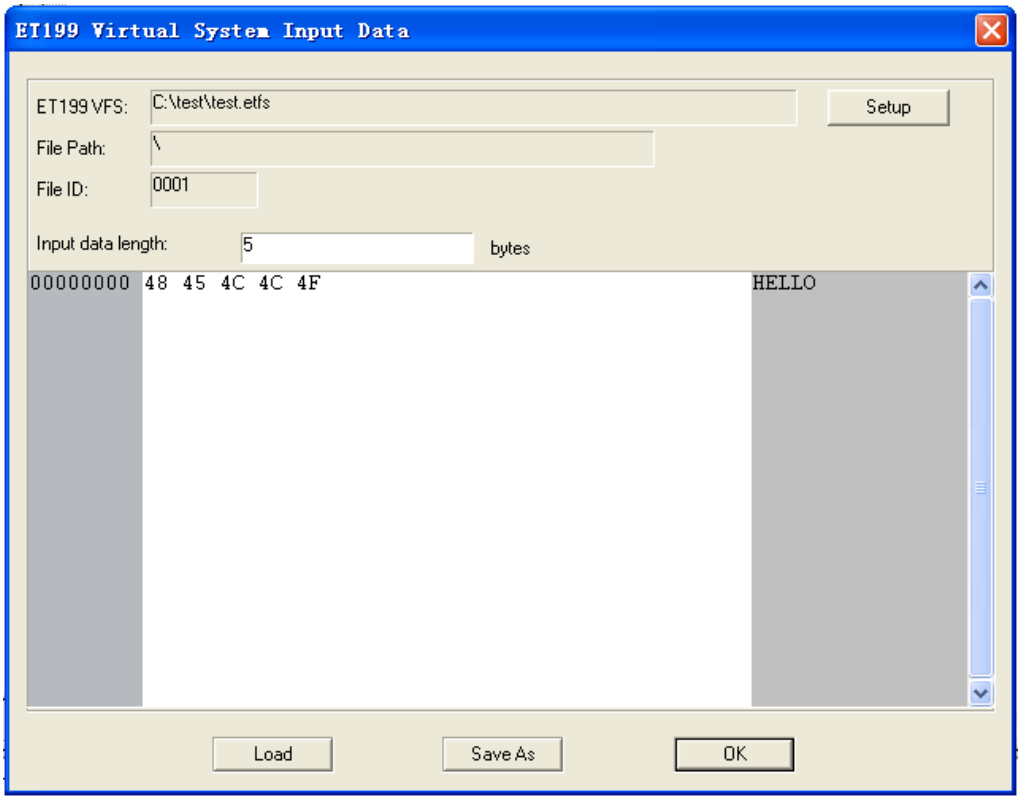

图 5-10

在 Input Data Length 后面的输入框中输入 5，然后在下面输入 HELLO，可以在中 间白色的十六进制框中输

入十六进制，或者在右边的灰色框中输入ASCII码。输入完成 后鼠标左键单击“OK”按钮，产生输出结果。要结束调试可以，按“Debug”菜单下的 “Start/Stop Debug Session”菜单或者按Ctrl+F5键。

5.2 ET199加密锁设置工具

ET199硬件有三种状态：空锁，加密锁（格式化成加密锁格式）和身份认证锁（格 式化成PKI格式）。ET199出厂时就已经是加密锁格式，可以直接用于对软件加密。当使 用过一段时间后，如果您需要对加密锁重新设置，如使用PKI身份认证功能，重新设置 根目录等，这时就需要使用到本章所介绍的工具。这个工具的主要功能为：

- 重新格式化加密锁。将空锁格式化成加密锁格式，或者将已经是加密锁格式的 ET199 重新格式化，即重建根目录。
- 将加密锁格式的 ET199 格式化为空锁。
- 将虚拟文件导入到 ET199 中。
- 设置外壳种子码
- 将磁盘上的文件下载到 ET199 中。
- 运行 ET199 中的可执行文件。
- 开发商口令和用户口令管理。
- RSA 密钥对管理。
- 获取硬件信息。

5.2.1 启动设置工具

设置工具只支持一把ET199。运行该工具前只能在计算机的USB接口上插入一把硬件。如果没有插上硬件，将弹出如下图所示的提示，退出程序。

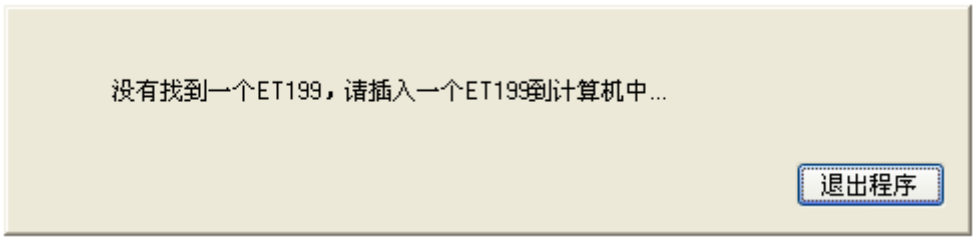

图 5-11 没有插入硬件

如果插上两把以上的ET199，则弹出如下图所示的提示，退出程序。

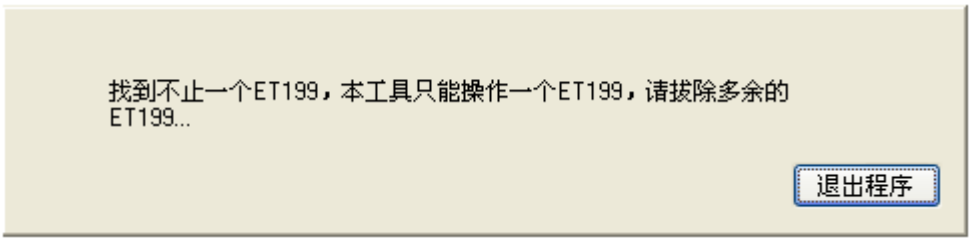

图 5-12 插入两个硬件

如果将ET199从机器上拔出，该工具也会弹出如图5-11所示的提示，退出程序。

5.2.2 格式化

启动设置工具后，使用鼠标左键单击左边的“初始化”图标，显示的界面如下图所示：

图 5-13 格式化界面

格式化加密锁：

选择“清空卡片”框中的“格式化加密锁（重建根目录）”选项，点击右下方的初始化按钮，弹出如下图所示“验证开发商密码”对话框，输入正确的开发商口令，点击“确定”按钮，则将ET199格式化为加密锁格式。

注意：开发商口令默认没有重试次数限制，如果您设置了重试次数，请输入正确的根目录开发商口令，否则连续输入错误口令的次数超过重试次数，根目录开发商口令将被锁死，这时只能将锁退回给我公司，重新生产，参见2.2节的说明。

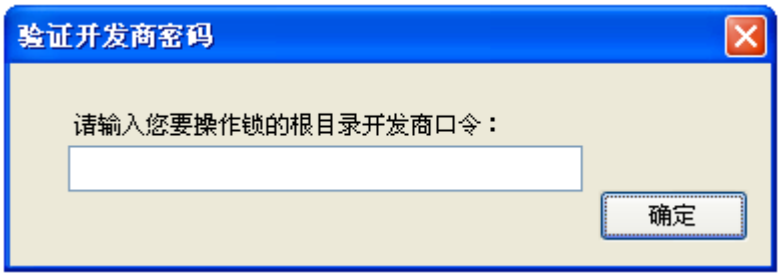

图 5-14 验证根目录开发商口令

格式化为空锁：

如果您需要使用PKI身份认证功能，需要先将ET199格式化成PKI格式。ET199在出厂时是加密锁格式，这时您需要使用设置工具的格式化为空锁的功能，先将ET199格式化为空锁，然后再使用PKI格式化工具（见《ET199超级多功能锁用户手册—身份认证篇》中的 4.2.2 节），把 ET199 格式化成 PKI 格式。 选择“清空卡片”框中的“加密锁初始化为空锁（可以重新选择加密锁和 PKI 功能）”选项，点击右下方的初始化按钮，则将 ET199 格式化为空锁。

将虚拟文件导入到 ET199 中：

在使用ET199对软件进行加密时，可以在虚拟文件系统中进行，一个虚拟文件就相当于一把ET199（见5.3节中的说明），加密工作结束后，虚拟文件中包括我们创建的目 录和文件，这时就需要使用设置工具，将这些目录和文件导入到真实的ET199设备中。

选择“清空卡片”框中的“使用虚拟文件系统复制加密锁”选项，在下面的“下载虚拟文件系统”框中点击“浏览”按钮，选择虚拟文件，点击右下方的初始化按钮，则将虚拟文件中的内容下载到 ET199 中。

设置外壳种子码：

ET199 提供了外壳加密工具（见 5.4 节中的说明），在使用外壳加密工具进行加密时，必须先要设置外壳种子码。使用 ET199 外壳工具加密后的程序运行时，需要在计算机上 插上 ET199 硬件，且这把 ET199 中的外壳种子码与加密时用到的 ET199 的外壳种子码要一致，否则加密后的程序不能运行。 选择“清空卡片”框中的“外壳种子码”选项，在下面的“设置外壳种子码”框中输入自定义的外壳种子码，不能超过 8 个字符，点击右下方的初始化按钮，则设置了 ET199 的外壳种子码。

烧制客户号：

用户可以在这里通过输入自定义的种子来设置 ET199 中的客户号（客户号参见 2.4 节），不同的种子对应不同的客户号。种子可以是任意内容。

注意：在选择格式化为空锁时，该项无效

修改 ATR：

ET199内置一个16字节的ATR文件，用户可以通过API接口来写入和读取这个文件中的内容。用户可以在这里通过输入自定义的 ATR 来设置ET199中 ATR 文件内容。

批量初始化：

界面最下面的“选项”框中，在“执行自动批量初始化，灯闪烁表示工作进行，灯熄灭后可以换锁”前面打上

勾，这时可以对ET199进行批量操作。在处理第一把锁时，您需要手动输入并验证你的开发商口令，点击 记住此次输入的开发商口令 选项和 自动验证 选项后，点击 确定 ，之后，您只需要插拔硬件，就可以按照设置对 ET199 进行批量处理。

5.2.3 下载文件

使用设置工具可以将磁盘上的文件下载到 ET199 硬件中。用鼠标左键单击左边的“下载文件”图标，显示的界面如下图所示：

图 5-15 下载文件界面

1、鼠标左键单击“浏览”按钮，浏览磁盘上的文件。

2、在“下载文件在卡片上的路径”下面的输入框中输入 ET199 中的路径，出于安全性的考虑，ET199 不提供列目录的功能，因此您必须在这里手动输入一个路径。如：“\”表示在 ET199 的根目录下创建，“\0008”表示在根目录下的 ID 为 0x0008 的目录下创建文件。需要注意：ET199 只支持三级目录结构（包括 根目录，见 2.3 中的说明），这里最多只能填入两个目录 ID，如“\0008\0009”， 表示根目录（“\”，第一级）下的 ID 为 0x0008 目录（“0008”,第二级）下的 ID 为 0x0009 目录（“0009”，第三级）下创建文件。

3、文件名称，即文件 ID。

4、文件大小。您在选择完磁盘上的文件后，这里会自动填入该文件的大小（字节）。 当手动进行输入时，如果选择的磁盘文件的大小比这里输入的小时，则写入的文件会在尾部填 0，如果选择的磁盘文件的大小比这里输入的大时，则只导入这里输入长度的数据。需要注意：导入文件时，文件的大小不能超过所在目录的可用空间。

5、选择文件类型。文件类型参见 2.3 节中的说明。

6、将“同名时覆盖原文件”前面的勾打上，则会替换 ET199 内已经存在的相同文件名的文件。设置好后，鼠标左键单击右下方的“下载”按钮，将磁盘上的文件导入到 ET199 中。

5.2.4 运行文件

可以使用设置工具中的运行文件的功能，运行 ET199 硬件内的可执行文件来查看一下运行结果，从而检查C51程序的正确性。用鼠标左键单击左边的“运行文件”图标，显示的界面如下图所示：

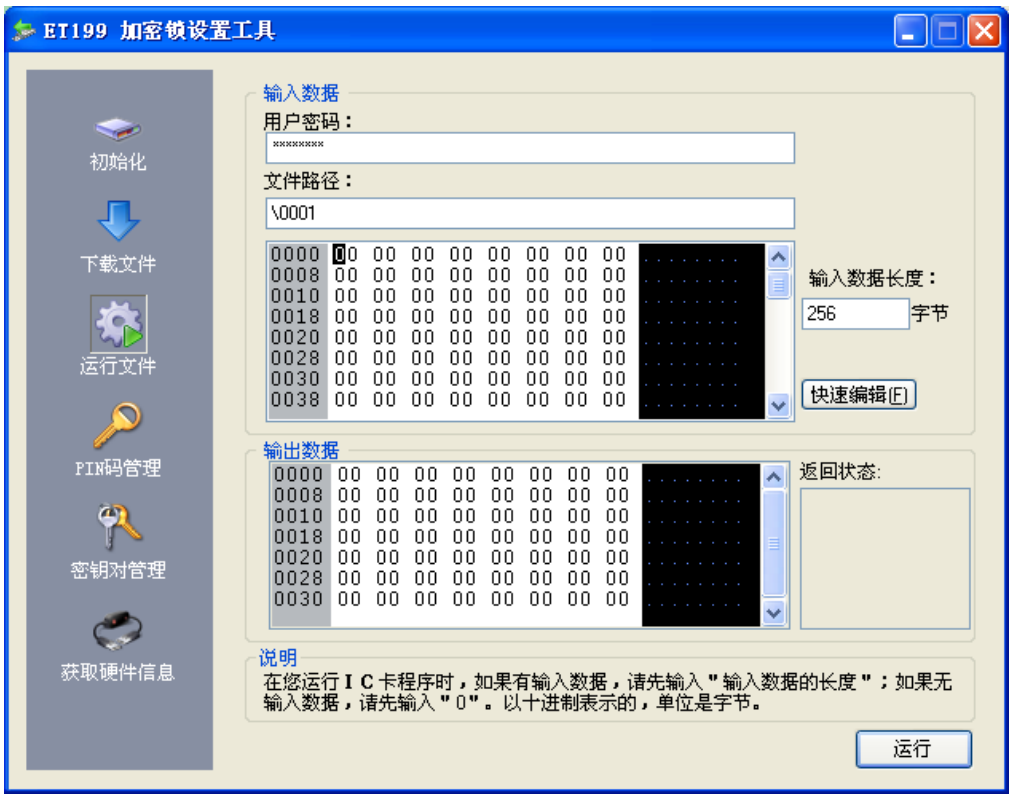

图 5-16 运行文件界面

1、在“用户密码”输入框中输入所要运行的可执行文件所在目录的用户密码。

2、在“文件路径”输入框中，输入可执行文件在 ET199 中的路径。如“\0001” 表示要运行根目录下的文件ID为0x0001的可执行文件。

3、如果可执行文件有输入数据，在“输入数据长度”的输入框中输入长度（十进制）。同时要在左边的输入栏中输入数据。您可以在中间白色的十六进制栏中 输入十六进制数，或者在右边黑色的 ASCII 码输入栏中直接输入字符。如输入 8个字节（HELLO，1，2，3）见下图所示：

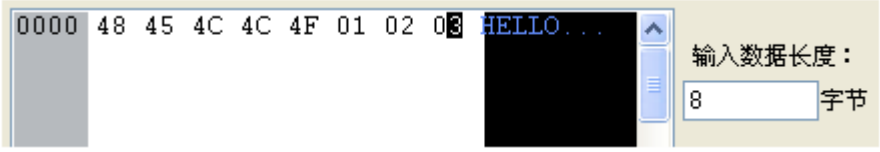

图 5-17 输入数据

您还可以使用鼠标选中编辑位置，然后左键单击“快速编辑”按钮，对输入的数据进行快速编辑。见下图所

示：

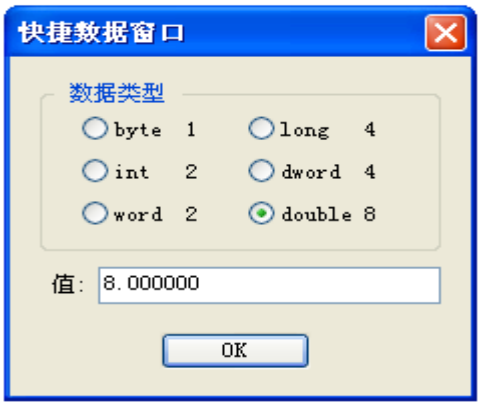

图 5-18 快速编辑

由于可执行文件是使用C51程序编译的，在C51语言中，int，long，float，double 等类型与 PC 中使用的语言（VC，VB，Delphi 等）的表示方法是相反的（见 6.2 中的说明）如：long 类型的 8 在 PC 中的内存中表示为：0x08 0x00 0x00 0x00（高字节在后），在C51程序中为：0x00 0x00 0x00 0x08（高字节在前），这时输入就应该以C51语言的方式输入，使用快速编辑功能会将这个顺序自动颠倒。另外C51语言中int类型为2个字节，而在PC上使用的语言中int类型多为4个字节，这就需要按C51语言的规定正确输入，使用快速编辑功能也能够解决这个问题。

设置好后，点击右下角的“运行”按钮运行锁内的可执行文件，运行结果显示在“输出数据”栏和返回状态中。

5.2.5 PIN码管理

设置工具可以更改ET199硬件中每个目录的开发商口令和用户口令。用鼠标左键单击左边的“PIN码管理”图标，显示的界面如下图所示：

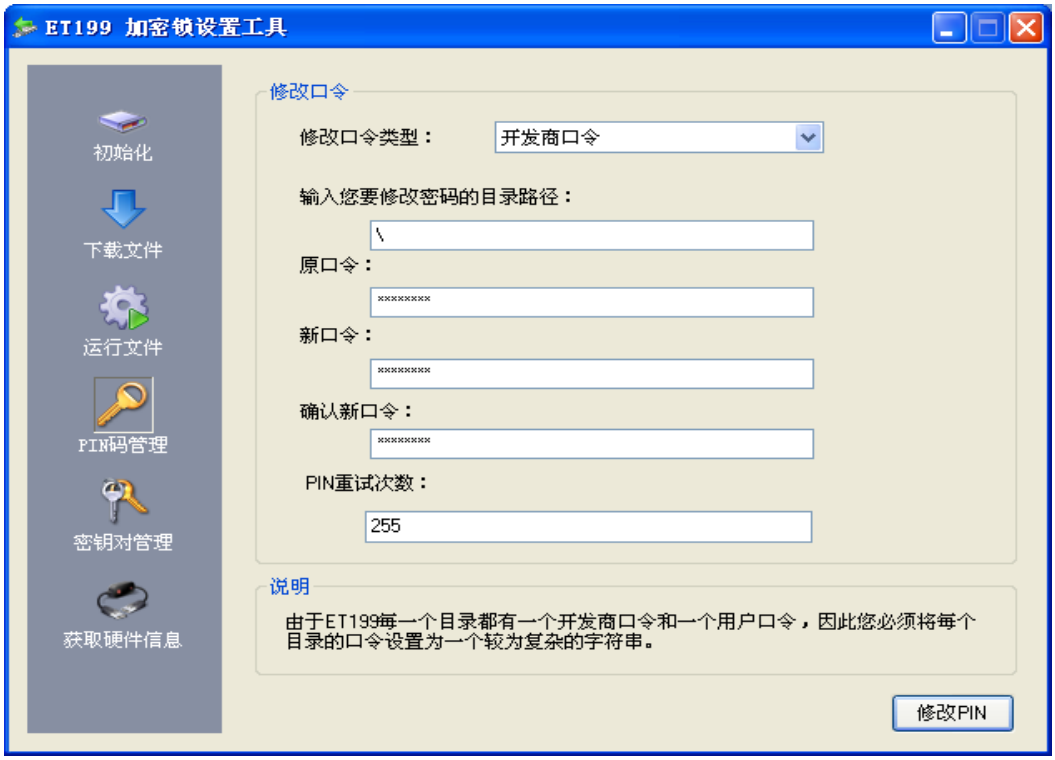

图 5-19 PIN 码管理界面

输入口令类型、目录路径、原口令、新口令、确认新口令以及 PIN 重试次数后，鼠标左键单击右下角的“修改PIN”按钮来修改口令。

注意：ET199只支持三级目录结构（见5.2.3中 的说明），因此目录路径中最多只能有两个目录ID。开发商口令为24个字符，用户口令为8个字符。特别注意：当您的输入不够24个字符或者8个字符时，工具会自动以0（0x00）值补齐，而不是字符“0”（0x30）。

5.2.6 密钥对管理

设置工具可以生成 RSA 密钥对，并将密钥导出备份。如果您在加密过程中使用到 RSA 功能，就需要先产生 RSA 密钥对。可以使用 C51 程序产生 RSA 密钥对，但这样产生的密 钥对中的私钥无法导出，那么每把 ET199 中的私钥都不一样。如果您需要密钥对相同的 ET199 硬件，这时就需要使用设置工具或者 API 接口（见 7.1.17 中的说明）来产生 RSA 密钥对，并将密钥对导出备份，然后通过本工具下载文件的功能，虚拟文件管理器或者 API 接口将密钥对导入到其他的 ET199 中。用鼠标左键单击左边的“密钥对管理”图标， 显示的界面如下图所示：

图 5-20 密钥对管理界面

在产生RSA密钥对时需要先填入：RSA密钥位数，E值（RSA密钥中一个属性值，四个字节，一般选用65537，即0x000x010x000x01），密钥文件路径，密钥文件ID，导出路径。填写好后，使用鼠标左键单击右下角的“创建密钥对”按钮，创建RSA密钥。

注意：ET199 只支持三级目录结构（见 5.2.3 中的说明），因此路径中最多只能有两个目录 ID。另外如果没有填写“PC 文件”栏中的磁盘的路径，则密钥对不备份到磁盘上。

5.2.7 获取硬件信息

使用鼠标左键单击左边的“获取硬件信息”图标，显示的界面如下图所示：

图 5-21 获取硬件信息界面

鼠标左键单击“获取信息”按钮，获取ET199的各种硬件信息。

5.3 ET199虚拟文件系统管理器

软件开发商在进行ET199内部程序（C51语言）开发时，可以结合KEIL编译器和本章所介绍的工具一起使用。ET199虚拟文件系统管理器相当于一个真实卡的模拟环境，用户在开发时可以不需要ET199硬件，只需在这个工具中完成所有内部程序（C51语言）的开发工作。本工具可以完成全部ET199内部系统函数（第六章）的功能。一个虚拟文件就相当于一把ET199。

下面我们通过一些具体的操作过程来熟悉这个工具。

5.3.1 工具界面

ET199虚拟文件系统管理器界面简单，包括：菜单栏，工具栏，列表栏和信息栏。如下图所示：

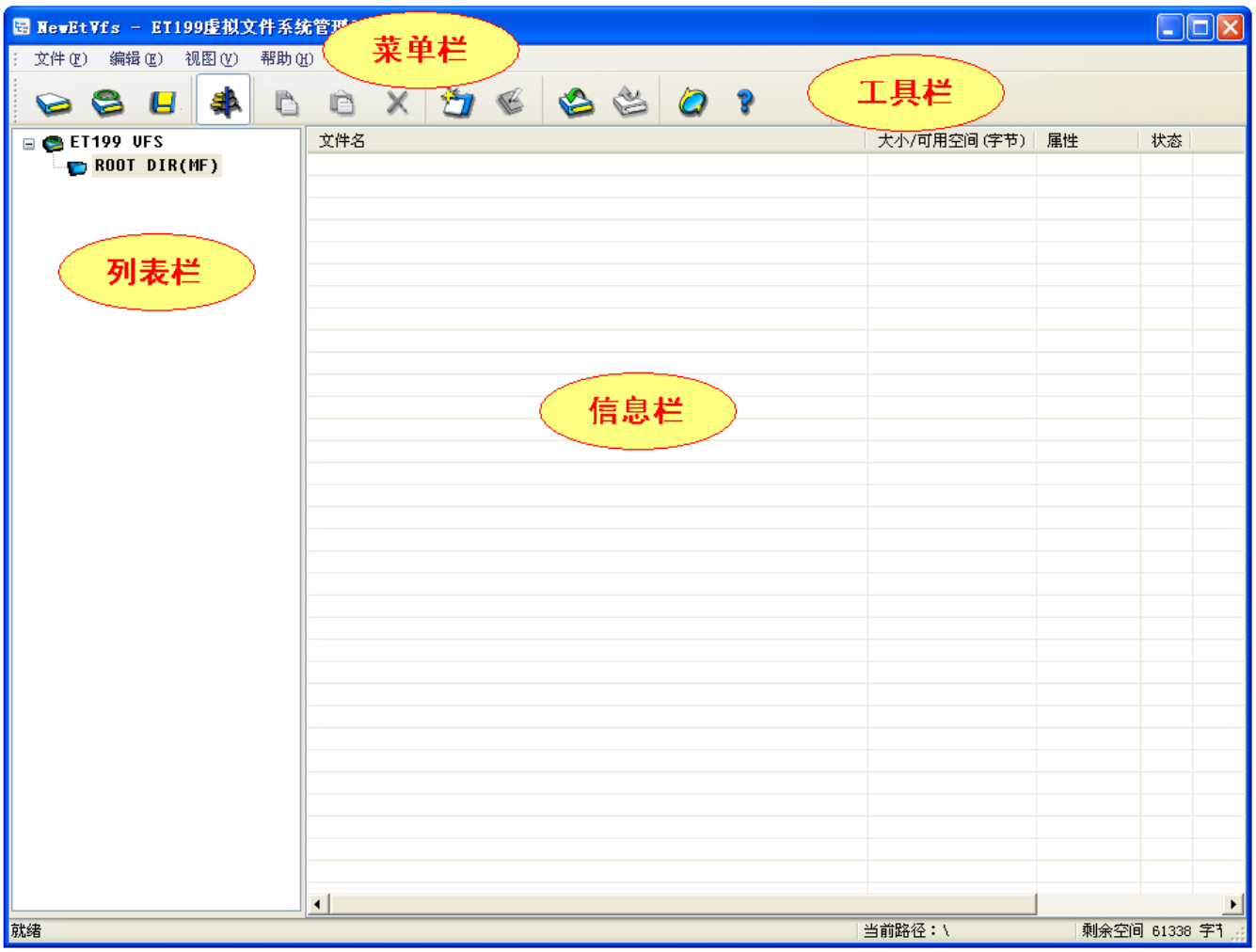

图 5-22 ET199虚拟文件系统管理器界面

菜单栏：包括了各种功能的菜单选项。工具栏:与菜单栏上的提供的功能一样,用户可以方便的点击图标来完成各种操作。列表栏：列出虚拟文件系统中的目录结构。信息栏：列出虚拟文件系统中的目录/文件的详细信息。

5.3.2 新建、保存和打开虚拟文件系统

1、首先我们要新建一个新的虚拟文件，有两种方法来实现：

（1）点击菜单栏上的“文件（F）”菜单，选择弹出的下拉菜单上的“新建（N）”选项，如图 5-23 所示：

（2）点击工具栏上的新建按钮  （位于工具栏左起第一个按钮），来新建虚拟文件。

图 5-23 新建虚拟文件

注意：创建完虚拟文件后，虚拟文件中已经存在了一个根目录，下面的操作都 是在这个根目录下完成的。

2、保存已经存在的虚拟文件，有两种方法实现：

（1）点击菜单栏上的“文件（F）”菜单，选择弹出的下拉菜单上的“保存（S）”选项，如下图所示：

图 5-24 保存虚拟文件

选择“保存（S）”选项后会弹出保存文件的对话框，选择路径后，将虚拟文件保存在硬盘上。见下图所示：

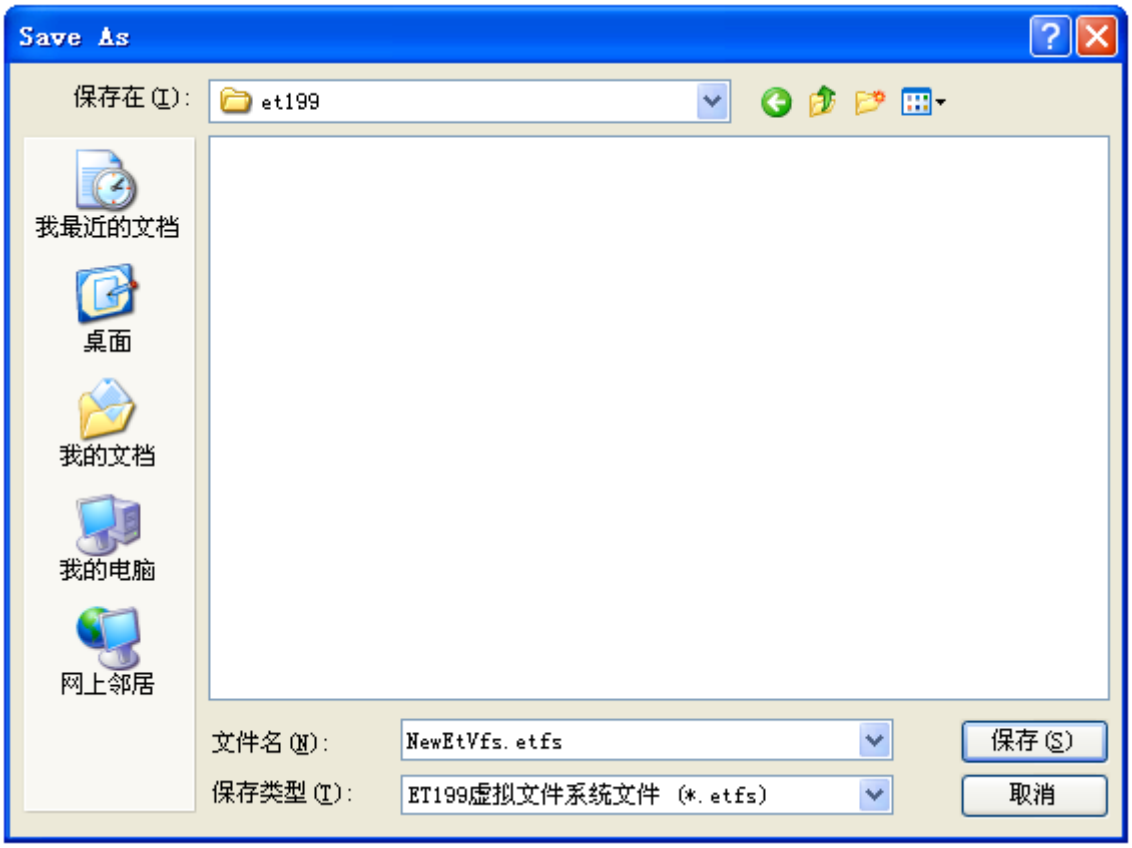

图 5-25 保存文件对话框

（2）点击工具栏上的保存按钮  （位于工具栏左起第三个按钮），来保存虚拟文件。点击完后会弹出如图 5-11 所示的保存文件对话框，选择路径后，将虚拟文件保存在硬盘上。

3、打开已经存在的虚拟文件，有两种方法实现：

（1）点击菜单栏上的“文件（F）”菜单，选择弹出的下拉菜单上的“打开（O）”选 项，如下图所示：

图 5-26 打开虚拟文件

选择“打开（O）”选项后会弹出打开文件的对话框，选择路径后，打开已经存储在磁盘上的虚拟文件，如下图所示：

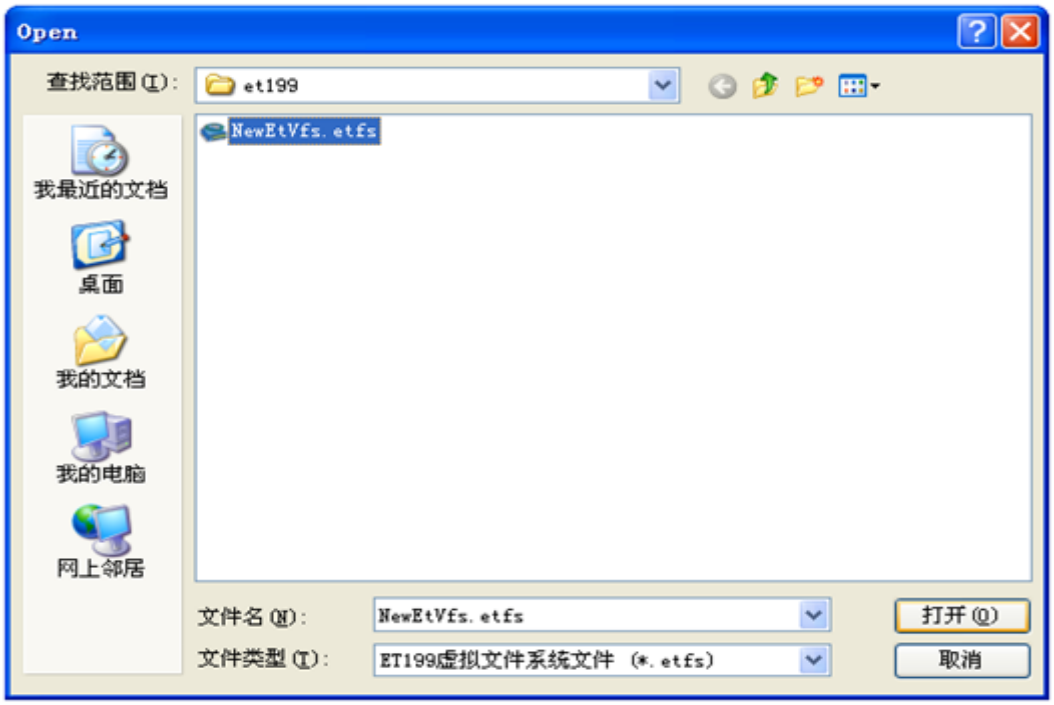

图 5-27 打开文件对话框

（2）点击工具栏上的打开按钮  （位于工具栏左起第二个按钮），打开已经存在的虚拟文件。点击完后会弹出如图5-13所示的打开文件对话框，选择路径后，打开已经存储在磁盘上的虚拟文件。

5.3.3 创建目录

在上一节中，我们创建了一个新的虚拟文件，这个虚拟文件就相当于一个模拟的 ET199，我们在这个虚拟文件中可以进行各种目录和文件的操作，与在真实的 ET199 设备中完全一样。关于目录和文件的相关内容，请参见2.3节中的说明。

创建目录，有三种方法实现：

（1）点击菜单栏的”编辑（E）”菜单，选择弹出的下拉菜单上的“创建目录（R）”选项，如下图所示：

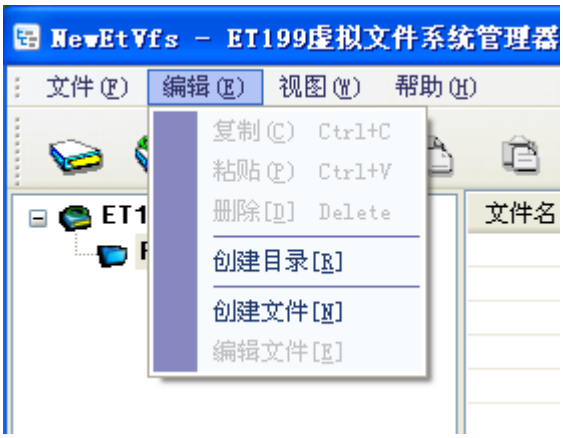

图 5-28 创建目录

选择“创建目录（E）”选项后，弹出创建目录的对话框，如下图所示：

图 5-29 创建目录信息对话框

开发商需要填写要创建目录的目录ID，目录大小，开发商PIN和用户PIN。填写完 成后，按“确定”按钮，创建新目录。这时可以在列表栏和信息栏中看到多了一个 ID 为“0001”的目录。如下图所示：

图 5-30 创建目录完成

我们看到右边的信息栏中列出了目录的信息，包括文件名（即目录 ID），大小/可用空间、属性。需要注意：由于目录的信息需要占用一些空间，虽然创建的目录的大小是 10240字节，但真正能够使用的空间是10208字节。另外ET199中的文件系统中目录结构只允许有三级（包括根目录），见 2.3 节中的说明。

（2）点击工具栏上的创建目录的按钮  （位于工具栏左起第八个按钮），创建新的目录。点击完后会弹出如图 5-29 所示的创建目录信息对话框，填写完相关的信息后，按“确定”按钮，创建新目录。

（3）在信息栏单击右键，在弹出的上下文菜单上选择“新建目录”选项，来创建新的目录。如下图所示：

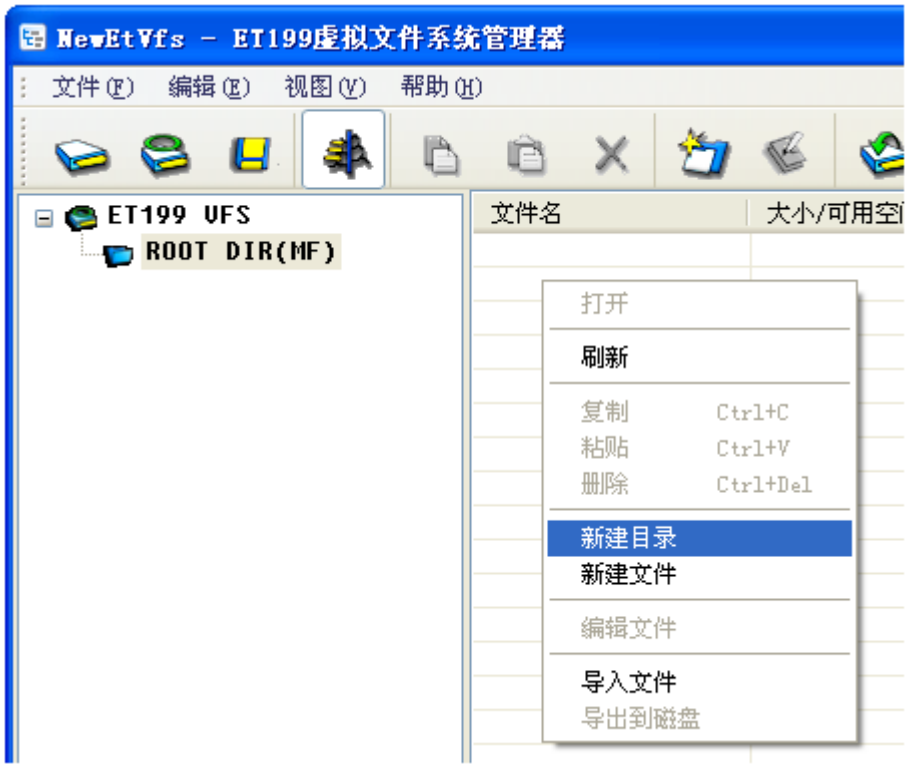

图 5-31 上下文菜单创建目录

5.3.4 创建文件

文件可以创建在根目录下，也可以创建其他目录下。有两种方法实现：

（1）点击菜单栏的“编辑（E)”菜单，选择弹出的下拉菜单上的“创建文件（N)” 选项，如下图所示：

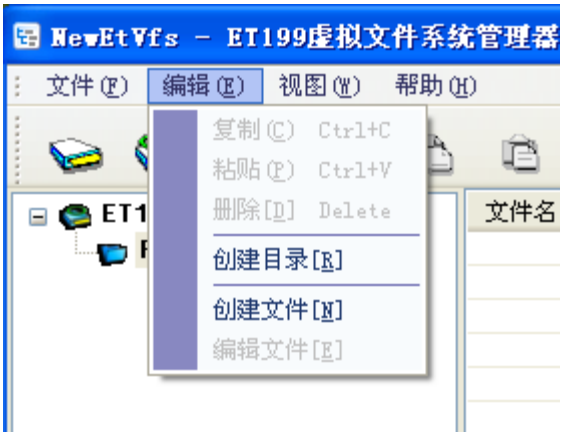

图 5-32 创建文件

选择“创建文件（N）”选项后，弹出创建文件对话框，见下图所示：

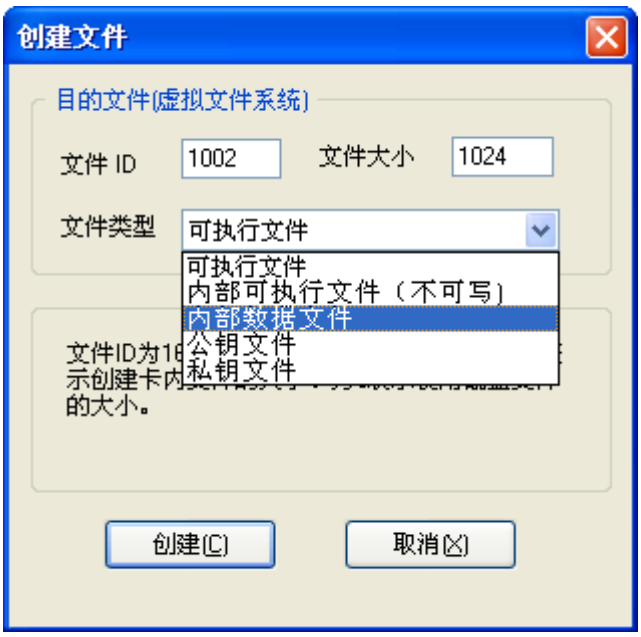

图 5-33 创建文件对话框

创建文件时需要填写文件 ID，文件大小和文件类型。创建文件后，可以在信息栏中看到多了一个 ID 为“1002”的内部数据文件。见下图所示：

图 5-34 创建文件完成

注意：文件ID为2个字 节，以十六进制数表示的范围：0x0001-0xFFFF，文件大小不能超过其所在目录的可使用空间。

（2）在信息栏单击鼠标右键，在弹出的上下文菜单上选择“新建文件”选项，来创建新的文件。如下图所示：

图 5-35 上下文菜单创建文件

5.3.5 导入文件

除了在虚拟文件系统中创建文件外，我们还可以将存在磁盘上的文件导入到虚拟环境中来。有两种方法实现：

（1）点击工具栏上的导入文件的按钮（位于工具栏右起第四个按钮），来导入文件。说明中是将经过KEIL编译器编译好的 C51 程序，即将编译好的二进制文件（.bin 文件）导入到虚拟系统中。点击按钮后会弹出如图 5-21 所示的导入文件对话框，填写完相关的信息后，按“创建”按钮，导入文件。

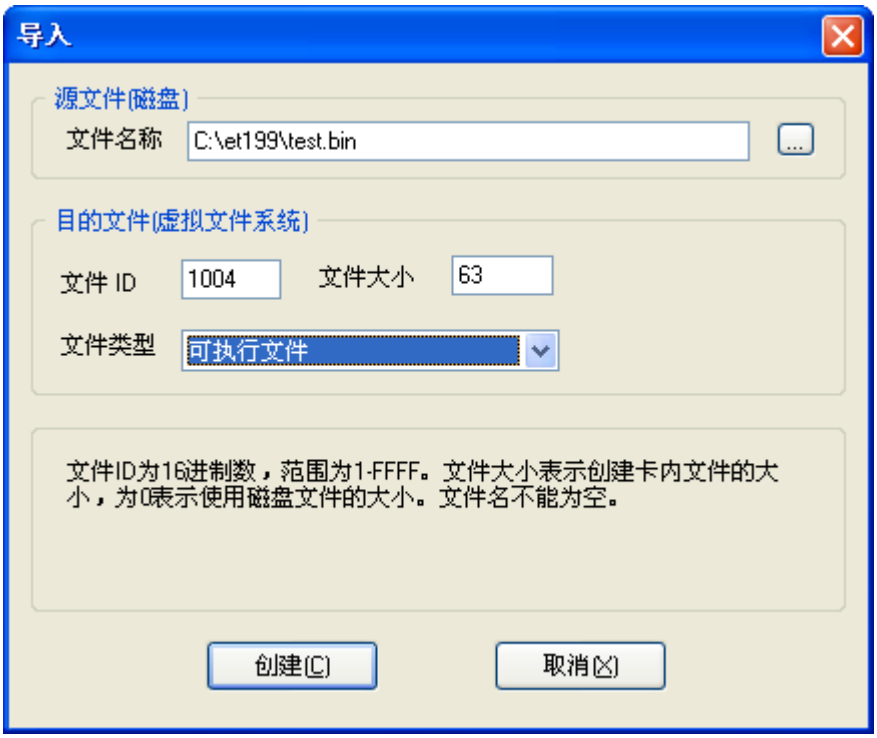

图 5-36 导入文件

“源文件（磁盘）”框中的  按钮用于浏览磁盘上已经存在的文件。“目的文件（虚拟文件系统）”框中的文件大小会自动填入要导入文件的大小。

当导入文件大小手工输入时。如果输入值大于磁盘上文件的实际大小，则导入后，多余的填充“0”；如果反之，则截取输入的长度进行导入。

（2）在信息栏单击鼠标右键，在弹出的上下文菜单上选择“导入文件”选项，来导入新的文件。如下图所示：

图 5-37 上下文菜单导入文件

5.3.6 打开目录和文件

打开目录和文件，有两种方法实现：

（1）双击信息栏中列出的文件和目录，就可以打开相应的文件和目录。如果我们要进入图5-23所示的ID为0001的目录时，只需要鼠标左键双击信息栏中文件名为0001 的那行就可以打开该目录，如下图所示：

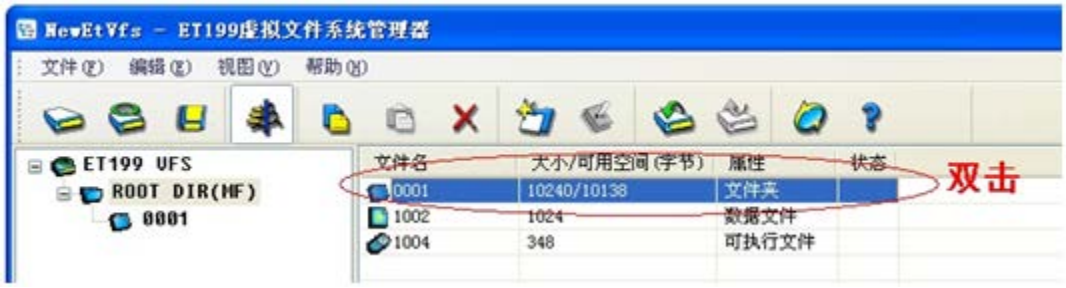

图 5-38 双击打开目录

打开目录后左边列表栏中的目录图标由关闭变为打开，右边信息栏显示已打开目录中的文件结构。说明中，由于ID为0001的目录下面没有创建任何目录和文件，所以显示为空的，见下图所示：

图 5-39 打开目录后

如果打开文件，会弹出编辑文件的对话框，见下面的编辑文件的说明。

（2）在信息栏中将鼠标移到目录或者文件上，单击鼠标右键，在弹出的上下文菜 单上选择“打开”选项，来打开目录或者文件。见下图所示：

图 5-40 上下文菜单打开文件

5.3.7 编辑文件

在虚拟文件系统中可以对所有文件进行编辑操作，考虑到开发时的方便，在操作时 不需要验证开发商口令和用户口令。编辑文件，有三种方法实现：

（1）在信息栏中选择文件后（鼠标左键单击），点击菜单栏上的“编辑（E）”菜单， 选择弹出的下拉菜单上的“编辑文件（E）”选项，如下图所示：

图 5-41 编辑文件

选择“编辑文件（E)”选项后，弹出编辑文件对话框，见下图所示：

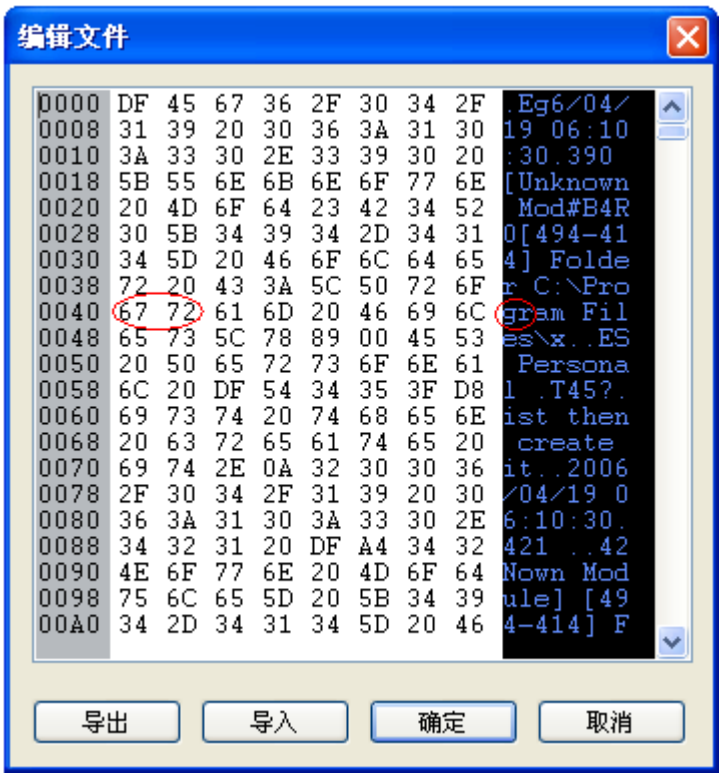

图 5-42 编辑文件对话框

我们可以直接在对话框中中间白色的十六进制显示区中更改，也可以在右边的 ASCII字符显示区中更改文件中的内容，如上图的红圈处。“导入“按钮可以将存在磁盘上文件导入到正在编辑的文件中。“导出”按钮可以将正在编辑的文件导出到磁盘上。

注意：如果导入的文件大小不足所要编辑的文件，那么导入进来后，后面的部分保持原来的内容。如果导入的文件大小超过所要编辑的文件，则将会把导入的文件进行截取。总之，正在编 辑的文件的大小是不会变的。

（2）选择文件后，点击工具栏上的编辑文件的按钮  （位于工具栏右起第五个按钮），来编辑文件。

（3）在信息栏中将鼠标移到文件上，单击鼠标右键，在弹出的上下文菜单上选择 “编辑文件”选项，来编辑文件。见下图所示：

图 5-43 上下文菜单编辑文件

5.3.8 导出文件

可以将虚拟文件系统中的文件导出到磁盘中，有三种方法实现：

（1）在信息栏中将鼠标移到文件上，单击鼠标右键，在弹出的上下文菜单上选择“导出到磁盘”选项，来导出文件。见下图所示：

图 5-44 上下文菜单导出文件

（2）点击工具栏上的导出文件的按钮  （位于工具栏右起第三个按钮），来导出文件。

（3）使用编辑文件对话框来导出文件。见 5.3.7 中的说明，弹出编辑文件对话框中后，点击“导出”按钮，来导出文件。见下图所示：

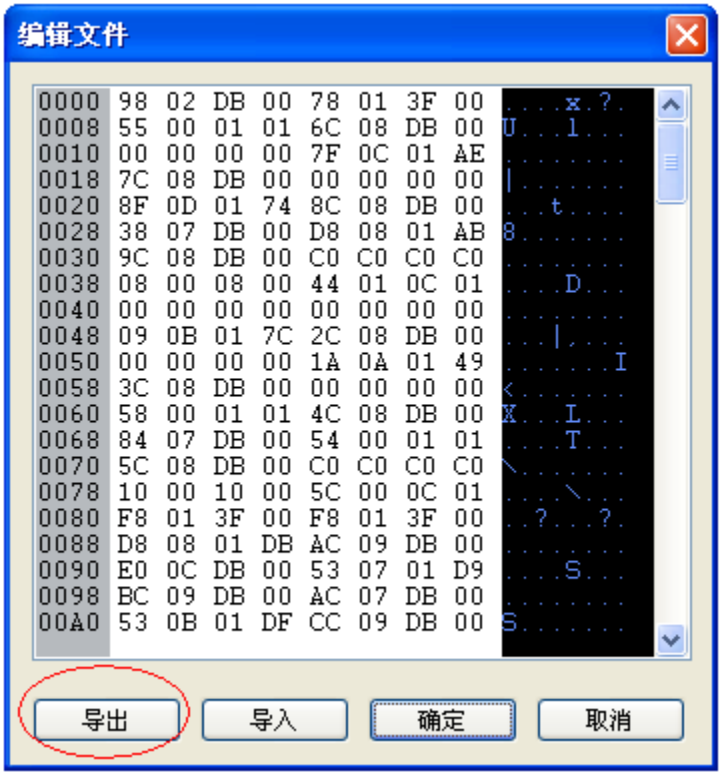

图 5-45 导出文件

5.3.9 复制，粘贴和删除

我们可以对在虚拟文件系统中的文件和目录进行复制，粘贴和删除操作。与我们在 Windows操作系统上的操作一致。可以通过三种办法来完成这些操作。

（1）使用菜单栏进行操作。先在信息栏中选择要进行操作的目录或者文件后（鼠标左键单击），点击菜单栏上的“编辑（E）”菜单，选择弹出的下拉菜单上的“复制（C）”、 “粘贴（P）”或者“删除（D）”选项，如下图所示：

图 5-46 菜单栏操作

（2）使用工具栏进行操作。先在信息栏中选择要进行操作的目录或者文件后（鼠标左键单击），点击工具栏上的复制按钮 （位于工具栏左起第五个按钮）、粘贴按钮 （位于工具栏左起第六个按钮）或者删除按钮 （位于工具栏左起第七个按钮）。

（3）使用信息栏中的上下文菜单进行操作。先在信息栏中选择要进行操作的目录 或者文件后，单击鼠标右键，在弹出的上下文菜单上选择“复制”，“粘贴”或者“删除” 选项。见下图所示：

图 5-47 信息栏上下文菜单操作

5.3.10 附属功能

（1）修改目录的开发商口令和用户口令。在列表栏中的目录图标上，单击鼠标右键在弹出上下文菜单上选择“修改密码”菜单，弹出修改对话框，如下图所示：

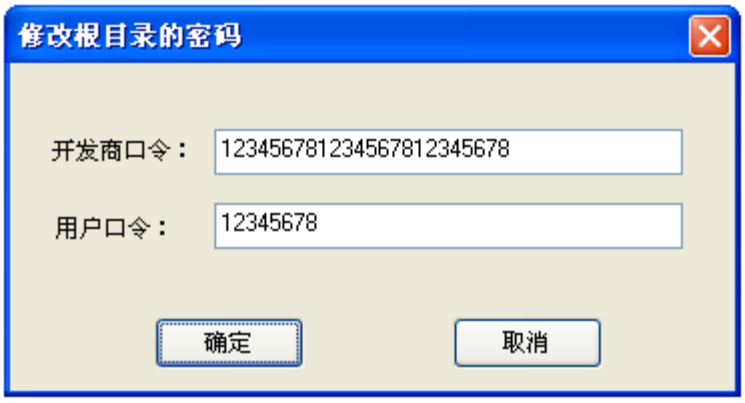

图 5-48 列表栏上下文菜单操作

（2）打开最近打开过的虚拟文件系统。点击菜单栏上的“文件（F）”菜单，在弹出的下拉菜单上有最近曾经打开的虚拟文件系统，鼠标单击要打开的虚拟文件，就可以方便的打开，不需要再通过文件选择框浏览磁盘上的文件。如下图所示：

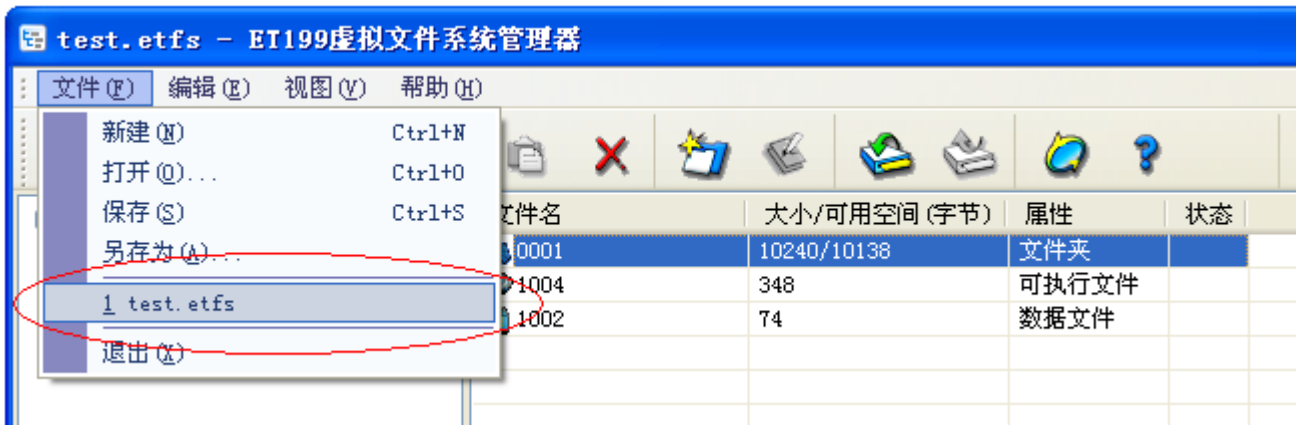

图 5-49 打开最近的文件

（3）隐藏列表栏。通过工具栏上的隐藏列表栏按钮  （位于工具栏左起第四个按钮），来隐藏列表栏。

（4）视图菜单。可以隐藏工具栏，状态栏和设备目录树（列表栏）。见下图所示：

图 5-50 视图菜单

（5）刷新功能。当虚拟文件系统中的目录或者文件发生改变时，有时需要刷新一下，以便能够显示最新的状态。使用工具栏上的刷新按钮  （位于工具栏右起第二个按钮），刷新当前状态。或者使用信息栏中上下文菜单上的刷新功能，见下图所示：

图 5-51 上下文菜单刷新

5.3.11 菜单栏和工具栏

新建	新建虚拟文件
打开	打开已经存在的虚拟文件
保存	将虚拟文件保存到磁盘
另存为	将虚拟文件保存到磁盘
退出	退出虚拟文件系统管理工具

图表 2 文件菜单

复制	目录/文件复制操作
粘贴	目录/文件粘贴操作
删除	目录/文件删除操作
创建目录	创建目录
创建文件	创建文件
编辑文件	编辑文件

图表 3 编辑菜单

工具栏	隐藏/显示工具栏
状态栏	隐藏/显示状态栏
设备目录树	隐藏/显示列表栏

图表 4 视图菜单

	新建虚拟文件系统
	打开虚拟文件系统
	保存虚拟文件系统
	隐藏/显示列表栏（设备目录树）
	目录/文件复制操作
	目录/文件粘贴操作
	目录/文件删除操作
	创建目录
	编辑文件
	导入文件
	导出文件
	刷新
	关于信息

图表 5 工具栏

5.4 双精度浮点数与二进制数组转换工具

当使用C51语言编程涉及到双精度浮点数时，会经常在双精度浮点数与二进制数组间进行转换。在高级语言中，一个双精度浮点数是使用一个8字节的数组来表示的，如：1.0表示为：0x00 0x00 0x00 0x00 0x00 0x00 0xF0 0x3F。使用该工具可以方便的进行转换。工具界面如下图所示：

图 5-52 转换工具

上面的按钮“E”和“PI”表示数学中的 e 和 π，可以方便的转换为二进制数组。

第6章 ET199内部系统函数（C51语言）

前面已经讲述了 ET199 的产品特点，工具说明和加密过程。本章将要详细介绍 ET199 内部系统函数的接口。运行在 ET199 内部的程序是使用 C51 语言进行开发的，C51 语言基本与C 语言一致，可以认为是C 语言的一个子集，它是专门用于硬件开发的 语言。很多做硬件程序的朋友对C51 语言应该是很熟悉的。C51 语言自带有丰富的函数 库，包括字符串处理，内存处理等，在编写 ET199 的 C51 程序时都可以方便的调用。同时我们也提供了相当多的额外接口，这些接口将在本章中进行说明，包括：输入输出、文件操作、数学运算、密码学算法等。

在使用 C51 语 言加入 ET199 的 C51 库 时，有 三 种模式 ： 微型模式 （compact_mode.LIB），小模式（small_mode.LIB）和大模式（large_mode.LIB）。建议使用小模式和大模式。不建议使用微型模式，因为其寻址范围有限，不能完成某些功能，如 2048 位的 RSA 运算。当选用不同的库时，注意 C51 工程的设置中（见 5.1.4 节）选择对应的选项，只更改 Memory Model 项就可以了。

内存分为：128 字节的data 区，128 字节idata 区和2k＋256 字节的xdata 区。其中 data 和xdata 中存放C51 程序的变量和函数的参数等，idata 区是用来给内部寄存器传递 参数使用的，所以不要将C51 程序的变量声明在idata 区中。

（1）小模式（small_mode.LIB）：默认的变量声明在 data 区中，所以在编程中声明变量时要加上 xdata,将变量声明在xdata 区中。一般情况下建议使用这种模式。

（2）大模式（large_mode.LIB）：默认的变量声明在xdata 区中，当程序中的函数较多，并存在多次调用时，建议使用这种模式。

在声明变量时要加上xdata 标志（例如：int xdata i = 0; ）。如果没有xdata 标志，当在小模式下，声明的变量都在 128 字节的 data 区中，存在不能分配超过 128 字节数组的情况。加上 xdata 标志后，变量是分配在 2K 的 xdata 空间中。

当您赋值常量时，建议您放到code 区即加上 code 标志（例如：const int code I = 8; ）这样是占用可执行文件的区域，而不占用内存。

6.1 退出程序

6.1.1 exit

void _exit()

说明：

退出在ET199中运行的使用C51编写的程序。在C51程序中一旦遇到exit()， 则程序终止运行。

参数：无

返回值： 无

6.2 输入输出

ET199 内的可执行文件（C51 语言）接收外部程序的输入数据或者向外部程序输出结果的最大长度为256个字节。

6.2.1 pbInBuff和wInLen

```
#define pbInBuff    ((BYTE xdata *)INPUT_DATA_OFFSET + 2 )
#define wInLen  (*(WORD xdata *)INPUT_DATA_OFFSET )
```

说明：

输入数据。外部接口（即PC上的程序）向在ET199内运行的C51程序传入数据，C51程序接收到数据后在ET199内进行处理。

pbInBuff 是一个地址，指向输入的数据。

wInLen 是输入数据的长度。

参数： 无

返回值： 无

举例：

1、C51 程序中，在给 int，long，word，dword，float，double 等类型赋值时是反序：即高字节在前，低字节在后。而 PC 上的程序为低字节在前，高字节在后，因此在输入时需要先做一下倒序。倒序可以在外部的高级语 言中使用函数完成,或者在C51语言中使用6.2.3节中的swap接口完成。示例如下：例如要向C51程序输入一个数字8，外部为C语言。

C语言中：

```
////////倒序函数////////
void FlipBuffer(unsigned char* pBuf, unsigned long ulLen)
{
  unsigned char ucTemp;
  for(unsigned long i = 0; i < ulLen >> 1; ++i)
  {
    ucTemp = pBuf[i];
```

```
      pBuf[i] = pBuf[ulLen - i - 1];
      pBuf[ulLen - i - 1] = ucTemp;
   }
}

int main(int argc, char* argv[])
{
   short sCData = 8;    //内存中为：0x08 0x00
   long lCData = 8; //内存中为：0x08 0x00 0x00 0x00
   float fCData = 8.0; //内存中为：0x00 0x00 0x00 0x41
   double dCData = 8.0;//内存中为：0x00 0x00 0x00 0x00 0x00 0x00 0x20 0x40
   BYTE pbInData[18]={0};//输入short，long，float和double，一共18个字节

   FlipBuffer((unsigned char*)&sCData, 2);//内存中为：0x00 0x08
   FlipBuffer((unsigned char*)&lCData, 4);//内存中为：0x00 0x00 0x00 0x08
   FlipBuffer((unsigned char*)&fCData, 4);//内存中为：0x41 0x00 0x00 0x00
   FlipBuffer((unsigned char*)&dCData, 8);//内存中为：0x40 0x20 0x00 0x00 0x00 0x00
0x00 0x00

   memcpy(pbInData, &sCData, 2);
   memcpy(pbInData+2, &lCData, 4);
   memcpy(pbInData+6, &fCData, 4);
   memcpy(pbInData+10, &dCData, 8);
   //将pbInData输入到ET199中，输入18个字节
}
```

C51语言中：

```
if(wInLen != 18)    //判断输入的数据长度是否是10个字节
_exit();

//////C51的int类型为2个字节，因此在C语言中声明为short////////
int xdata iData;
//数值8赋值给iData，pbInBuff中前2个字节，序号为0，1
memcpy(&iData, pbInBuff, 2);

//////C51的long类型与C语言一样，为4个字节////////
long xdata lData;
//数值8赋值给lData，pbInBuff中序号为2到5的4个字节
memcpy(&lData, pbInBuff + 2, 4);

//////C51的foat类型与C语言一样，为4个字节////////
float xdata fData;
//数值8.0赋值给fData
//pbInBuff中序号为6到9的4个字节
memcpy (&fData, pbInBuff + 6, 4);

//////C51中Double类型为8字节数组////////
DOUBLE xdata dData;
//数值8.0赋值给dData
//pbInBuff中序号为10到17的8个字节
memcpy (&dData, pbInBuff + 10, 8);
```

2、赋值给字符串类型时不需要倒序。

C语言中：

```
int main(int argc, char* argv[])
{
   //包括字符串结尾0x48 0x45 0x4C 0x4C 0x4F 0x00
   char cBuffer[6] = “HELLO”;
}
```

C51语言中：

```
char xdata buffer[250];
memcpy(buffer, pbInBuff, 6);
//将6个字节（“HELLO”, 0x48 0x45 0x4C 0x4C 0x4F 0x00）赋值给buffer
```

3、给多个变量赋值时，例如：将 long，char 数组和 double 传给 C51 程序。

C语言中：

```
int main(int argc, char* argv[])
{
   BYTE pbInData[18] = {0};
   long lCData = 8;
   FlipBuffer((unsigned char*)&lCData, 4);
   char cBuffer[6] = “HELLO”;
   double dCData = 8.0;
   FlipBuffer((unsigned char*)&dCData, 8);

   memcpy(pbInData, &lCData, 4);
   memcpy(pbInData+4, cBuffer, 6);
   memcpy(pbInData+10, &dCData, 8);
   //将pbInData输入到ET199中，输入18个字节
}
```

C51语言中：

```
long xdata lData;
char xdata buffer[250];
double xdata dData;

if(wInLen != 18)    //判断输入的数据长度是否是18个字节
_exit();

memcpy(&lData, pbInBuff, 4);
//数值8（0x00 0x00 0x00 0x08）赋值给lData，long为4个字节，
//pbInBuff中序号为0到3的4个字节

memcpy(buffer, pbInBuff + 4, 6);
//“HELLO”（0x48 0x45 0x4C 0x4C 0x4F 0x00）赋给buffer，
//pbInBuff中序号为4到9的6个字节

memcpy(&dData, pbInBuff + 10, 8);
//8.0（0x40 0x20 0x00 0x00 0x00 0x00 0x00 0x00）赋值给dData，
//pbInBuff中序号为10到17的8个字节
```

6.2.2 set_response

*BYTE _set_response(WORD wLen, void *pvData)*

说明：

输出数据。C51程序在ET199内运行完成后，调用这个函数将结果返回给外部程序（即高级语言的程序）。注意：输出数据的最大长度为 256 字节。

参数：

wLen　[in]输出数据的长度。

*pvdata [in]输出数据的所在的地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

1、倒序输出：当输出数据为int，long，word，dword，float，double 等时，由于C51 程序与外部的程序有一个顺序颠倒的问题，因此需要将输出倒序后，再继续进行后面的处理。倒序可以在外部的高级语言中使用函数完成，或者在 C51语言中使用6.2.3节中的swap接口完成。

C51语言中：

```
int xdata a;
a = 1+2;
_set_response(2,&a); //内存中为：0x00 0x03
```

C语言中：

```
////////倒序函数////////
void FlipBuffer(unsigned char* pBuf, unsigned long ulLen)
{
  unsigned char ucTemp;
  for(unsigned long i = 0; i < ulLen >> 1; ++i)
  {
    ucTemp = pBuf[i];
    pBuf[i] = pBuf[ulLen - i - 1];
    pBuf[ulLen - i - 1] = ucTemp;
  }
}

int main(int argc, char* argv[])
{
  int i = 0;
  ……
```

```
  //ET199中运行结果传出给i，这时i的内存为：0x00 0x03


  FlipBuffer((unsigned char*)&i, 2); //倒序后为：

  0x03 0x00 printf("%d\n",i); //这时打印的结果是正确的，为3。

}
```

2、输出字符串时不需要倒序。

C51语言中：

```
char xdata buffer[] = "HELLO";

_set_response(6, buffer); //将"HELLO"传出
```

C语言中：

```
int main(int argc, char* argv[])

{

  char cBuffer[1024];

  ……

  //ET199中运行结果传出给cBuffer，

  //这时cBuffer内存为：0x48 0x45 0x4C 0x4C 0x4F 0x00


  printf("%s\n", cBuffer); //这时打印的结果是"HELLO"

}
```

6.2.3 swap

void _swap(void* pvData,unsigned short wLen)

说明：

颠倒数据的顺序。在导入int，long，word，dword，float，double等数据时需要颠倒。

参数：

*pvData [in]要颠倒顺序的数据的存放地址。

wLen [in]数据的长度。

返回值： 无

举例：

```
char xdata buffer[] = “HELLO”;

_swap(buffer, 5); //颠倒后的buffer 为：“OLLEH”
```

6.3 文件操作

6.3.1 create

*BYTE _create(WORD wFileID, WORD wSize,BYTE bFileType, BYTE bFlag,HANDLE *pHandle)*

说明：

创建文件。可以在ET199中创建数据文件，可执行文件和密钥文件。C51程序不能创建目录，目录是通过外部的接口来创建的。当使用这个接口创建可执行文件时，请不要创建 FILE_TYPE_INTERNAL_EXE 类型的可执行文件，因为这样的文件即使创建了，也无法写入数据，是一个没有内容的废文件。

参数：

wFileID [in]文件的标识符，即文件的ID，2个字节。

wSize [in]文件的大小，2个字节。

bFileType [in]文件类型字节，1个字节，参看表6-1。

bFlag [in]文件标志字节，1个字节，参看表6-2。

*pHandle [out]返回的文件句柄。如果文件已打开，则返回文件的句柄。

文件类型		
名称	值	含义
FILE_TYPE_EXE	0x00	普通可执行文件（可写）
FILE_TYPE_DATA	0x01	内部数据文件（可读写）
FILE_TYPE_RSA_PUB	0x02	RSA 公钥文件（可读写）
FILE_TYPE_RSA_SEC	0x03	RSA 私钥文件（可写）
FILE_TYPE_INTERNAL_EXE	0x04	内部可执行文件（不可读写）

图表 6 文件类型

关于文件类型见2.3.2中的说明。在C51程序（即ET199内部的可执行文件）中，对这些文件的读写属性说明如下：

普通可执行文件：可以被其他可执行文件改写，任何情况下都不能读取。

内部可执行文件：不能被其他可执行文件读写。

内部数据文件：可以被可执行文件读写。

RSA公钥文件：可以被可执行文件读写。

RSA 私钥文件：可以被其他可执行文件改写，任何情况下都不能读取。

创建文件标志		
CREATE_OPEN_ALWAYS	0x00	如果文件已存在，打开该文件；反之则创建新文件并打开
CREATE_FILE_NEW	0x01	创建并打开新文件，如果文件已经存在则返回错误
CREATE_OPEN_EXISTING	0x02	打开已存在文件，功能同_open()函数

图表 7 文件标志

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
HANDLE xdata hFile = 0;

BYTE xdata bRes = 0;

//创建一个大小为16（0x10）个字节，文件ID为0x1008的数据文件
bRes=_create(0x1008,0x10,FILE_TYPE_DATA,CREATE_OPEN_ALWAYS,&hFile);

if(bRes != 0)

{

  _set_response(1,&bRes);

  _exit();

}
```

6.3.2 open

BYTE _open(WORD wFileID, HANDLE *pHandle)

说明：

打开文件。打开ET199中的文件，

（1）当打开：数据文件，公钥文件时，可对这些文件进行读写，

（2）当打开：私钥文件，普通可执行文件（有可写属性）时，只能改写，不能 读取。

（3）内部可执行文件也可以打开，但这种文件不能读写。 参数：

wFileID [in]文件ID，2个字节。

*pHandle[out]返回的文件句柄。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
word xdata wFid = 0x1008;
HANDLE xdata hFile = 0;
BYTE xdata bRes = 0;
bRes = _open(wFid, &hFile); //打开一个文件ID为0x1008的文件
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}
```

6.3.3 close

BYTE _close(HANDLE handle)

说明： 关闭文件。对文件操作完成后，请调用该函数关闭文件。调用该函数前必须先要打开文件（open）。

参数：

handle [in]文件句柄。使用open函数时得到的句柄。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
word xdata wFid = 0x1008;
HANDLE xdata hFile = 0;
_open(wFid, &hFile); //打开一个文件ID为0x1008的文件
……
```

```
_close(hFile);  //关闭文件ID为0x1008的文件
```

6.3.4 read

*BYTE _read(HANDLE handle, WORD wOffset, BYTE bLen, void *pvData)*

说明： 读取文件。该函数只能使用在：数据文件和公钥文件。该函数不能使用在可执行文件上，可执行文件的内容是不能被读出的。调用该函数前必须先要打开文件（open）

参数：

handle [in]文件句柄。使用open函数时得到的句柄。

wOffset [in]偏移值。即从文件的第几个字节开始读取数据。

bLen [in]读取数据的长度。

*pvData [out]从文件中读出数据所要放入的地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
word xdata wFid = 0x1008;
HANDLE xdata hFile = 0;
char xdata buffer[250];
BYTE xdata bRes = 0;

bRes = _open(wFid, &hFile); //打开一个文件ID为0x1008的文件
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}

bRes = _read(hFile, 0, 10, buffer);
//从文件ID为0x1008的文件中 第0个字节开始读取10个字节到buffer中
if(bRes != 0)
```

```
{
  _set_response(1,&bRes);
  _exit();
}
_close(hFile);  //关闭文件ID为0x1008的文件
```

6.3.5 write

*BYTE _write(HANDLE handle, WORD wOffset, BYTE bLen,const void *pvData)*

说明： 写入文件。将数据写入到文件中。该函数使用在：数据文件，公钥文件和普通 可执行文件（有可写属性）。调用该函数前必须先要打开文件（open）。

参数：

handle [in]文件句柄。使用open函数时得到的句柄。

wOffset [in]偏移值。即从文件的第几个字节开始写入数据。

bLen [in]写入数据的长度。

*pvData [in]存放要写入数据的地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
word xdata wFid = 0x1008; HANDLE xdata hFile = 0;
char xdata buffer[] = “HELLO”;
BYTE xdata bRes = 0;

bRes = _open(wFid, &hFile); //打开一个文件ID为0x1008的文件
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}
```

```
bRes = _write(hFile, 0, sizeof(buffer), buffer);

//将“HELLO”写入到ID为0x1008的文件中，从该文件的第0字节开始写入

if(bRes != 0)

{

_set_response(1,&bRes);

_exit();

}


_close(hFile);  //关闭文件ID为0x1008的文件
```

6.3.6 get_file_infor

BYTE _get_file_infor(PEFINFO pFileInfo)

说明： 获取文件的属性信息。

参数：

pFileInfo [out]文件信息结构的指针，包括：文件 ID，文件类型和文件大小。

EFINFO文件信息结构定义如下：

```
typedef struct _FILE_INFO
{
    WORD wFileID;     //文件ID
    BYTE bFileType;   //文件类型，见前面的表6-1
    WORD wFileSize;   //文件大小
}
EFINFO,*PEFINFO;
```

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
EFINFO xdata fInfo; BYTE xdata bRes = 0;


fInfo.wFileID = 0x1008;
```

```
bRes = _get_file_infor(&fInfo); //取得ID为0x1008文件的信息

if(bRes != 0)

{

   _set_response(1,&bRes);

   _exit();

}
```

6.4 密码学算法

本节涉及到很多密码学中的概念，在这里做一个简单的介绍。

对称加解密算法： 对称加解密算法的密钥是相同的，即加密和解密使用的密钥是同一个。密钥的长度也是固定的，如DES：8个字节，3DES：16个字节。加密的数据的长度也要是固定的，一般为8的倍数。目前流行的对称加解密算法有DES，3DES等。

ECB和CBC： ECB和CBC是加解密中常用的两种模式。在对称加解密中，数据是按块来进行的，DES和3DES都是8个字节为一块。ECB模式下，块与块之间没有关系，是各自独立的。CBC模式下，块与块之间是有关系的，前块的加密结果与后块做异或，然后再进行加密，以此类推。

补齐方式： 在对称加解密时，数据是分块进行的，DES 和 3DES 都是 8 个字节为一块。这时 如果加密的数据不是 8 的倍数，就需要进行补齐。补齐的规则可以由开发商自己来设定。常见的方式为，少几个字节就补上该数字。如：原数据为 0x11 0x22 0x33 三个字节，这时缺少 5 个字节，那么后面都补数字 5，即：0x11 0x22 0x33 0x05 0x050x05 0x05 0x05，这时解密后读取最后一个字节，然后去掉相应的 5 个字节，即得 到原文。如果原文正好是 8 的倍数，那么就多补 8 个字节，都赋值为 0x08，这样解密后，最后一个字节是 0x08，去掉 8 个字节后就是原文。

散列算法： 散列算法是根据不同的输入，产生与之对应的固定长度的输出。目前流行的散列算法有SHA1，MD5等。SHA1的散列结果长度为20字节，MD5的散列结果长度为16字节。

非对称加解密算法： 非对称加解密的加密密钥和解密密钥是不同的，分为公钥和私钥。公钥是公开的密钥，可以散发给其他人；私钥是秘密的，需要拥有者本人自己掌握，不能透露给任何人。公钥和私钥是匹配的，通常称为公私钥对。一般使用公钥进行加密，即私钥的主人将与之匹配的公钥散发给其他人，其他人使用这个公钥进行加密，将加密后的数据发送给私钥主人，这时只有与这个公钥匹配的私钥才能正确解密数据。

目前流行的非对称加解密算法有RSA算法等。RSA密钥对通常分为512位，1024位 和2048位，位数越高，加密强度越强，但运算速度也越慢。

ET199 采用高性能智能卡芯片，在硬件中支持 512 位，1024 位和 2048 位 RSA 运算。

数字签名

数字签名的功能是防止数据在传输中被篡改。整个功能是使用 RSA 密钥对运算 来实现的。数字签名的

过程为：

（1）先将数据使用散列算法进行散列，当然也可以直接使用明文数据， 但如果明文数据比较大，RSA运算将会比较慢。一般都采用先将明文数据散列的方法，得到一个固定长度的数据。

（2）将（1）中的散列后的数据使用私钥进行签名操作，将原文数据和签名的结果发送给接收者。

（3）接收者使用与（1）中相同的散列算法，将原文数据进行散列，然后使用与私钥匹配的公钥，把散列的结果与签名后的结果进行验签，如果验证签名成功表示数据在传输过程中没有被篡改。

6.4.1 des_enc

*BYTE _des_enc(const void *pvKey, BYTE bLen, void *pvData)*

说明：

使用DES对称算法加密数据。请注意：DES密钥为8个字节，要加密的数据长度一定要是8的倍数。数据长度不为8的倍数时，请自行补齐，常见的补齐原则参看本节开始的说明。这里的加密为ECB模式。要完成CBC模式或者其他模式，请在此基础上自行处理。

参数：

*pvKey [in]DES加密密钥的地址。密钥长度为8个字节。

bLen [in]加密数据长度。一定要是8的倍数。

*pvData [in/out]输入为要加密的明文数据，输出为加密后的密文数据。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;

//要加密的明文，长度为8的倍数

char xdata text[8] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};

//DES密钥，长度8字节

BYTE xdata deskey[8] = {0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88};


bRes = _des_enc(deskey, 8, text); //text为双向缓冲区

if(bRes != 0)

{

  _set_response(1,&bRes);
```

```
   _exit();
}


//加密后的text为：0x9A 0xB9 0xB1 0x88 0xB1 0x6A 0x62 0x40
_set_response(8,text);
_exit();
```

6.4.2 des_dec

*BYTE _des_dec(const void *pvKey, BYTE bLen, void *pvData)*

说明：

使用DES对称算法解密数据。请注意：DES密钥为8个字节，要解密的数据长 度一定要是 8 的倍数。解密后的数据可以按照本节开始的补齐原则来得到原文。这里的解密为ECB模式。如要完成CBC模式或者其他模式，请在此基础上自行处理。

参数：

*pvKey [in]DES解密密钥的地址。密钥长度为8个字节。

bLen [in]解密数据长度。一定要是8的倍数。

*pvData [in/out]输入为要解密的密文数据，输出为解密后的明文数据。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
//要解密的密文，长度为8的倍数
char xdata text[8] = {0x9A, 0xB9, 0xB1, 0x88, 0xB1, 0x6A, 0x62, 0x40,};
//DES密钥，长度8字节
BYTE xdata deskey[8] = {0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88};


bRes = _des_dec(deskey, 8, text);
if(bRes != 0)
{
```

```
  _set_response(1,&bRes); //text为双向缓冲区
  _exit();
}

//解密后的text为：0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
_set_response(8,text);
_exit();
```

6.4.3 tdes_enc

*BYTE _tdes_enc(const void *pvKey, BYTE bLen, void *pvData)*

说明：

使用3DES对称算法加密数据。请注意：3DES密钥为16个字节，要加密的数据长度一定要是8的倍数。数据长度不为8的倍数时，请自行补齐，常见的补齐原则参看本节开始的说明。这里的加密为ECB模式。如要完成CBC模式或者 其他模式，请在此基础上自行处理。

参数：

*pvKey [in]3DES加密密钥的地址。密钥长度16字节。

bLen [in]加密数据长度。一定要是8的倍数。

*pvData [in/out]输入为要加密的明文数据，输出为加密后的密文数据。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
//要加密的明文，长度为8的倍数
char xdata text[8] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
//3DES密钥，长度16字节
BYTE xdata tdeskey[16] = {0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,
0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF};

bRes = _tdes_enc(tdeskey, 8, text); //text为双向缓冲区
```

```
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}
//加密后的text为：0x5D 0x99 0x07 0x87 0xB0 0x67 0x37 0x87
_set_response(8,text);
_exit();
```

6.4.4 tdes_dec

*BYTE _tdes_dec(const void *pvKey, BYTE bLen, void *pvData)*

说明：

使用3DES对称算法解密数据。请注意：3DES密钥为16个字节，要解密的数 据长度一定要是8的倍数。解密后的数据可以按照本节开始的补齐原则来得到原文。这里的解密为ECB模式。如要完成CBC模式或者其他模式，请在此基础 上自行处理。

参数：

*pvKey [in]3DES解密密钥的地址。密钥长度16字节。

bLen [in]解密数据长度。一定要是8的倍数。

*pvData [in/out]输入为要解密的密文数据，输出为解密后的明文数据。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
//要解密的密文，长度为8的倍数
char text[8] = {0x5D, 0x99, 0x07, 0x87, 0xB0, 0x67, 0x37, 0x87};
//3DES密钥，长度16字节
BYTE xdata tdeskey[16] = {0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,
0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF};
```

```
bRes = _tdes_dec(tdeskey, 8, text); //text为双向缓冲区
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}

//解密后的text为：0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
_set_response(8,text);
_exit();
```

6.4.5 sha1_init

BYTE _sha1_init(PSHA_CONTEXT pCtx)

说明：

使用SHA1散列算法前，先要进行初始化。

参数：

pCtx [in]上下文环境结构的地址。

SHA_CONTEXT上下文环境结构定义如下：

```
typedef struct _tagSHA_CONTEXT
{
    DWORD   h[5];
    DWORD   dwTotalLength; BYTEbRemainLength;
    BYTEpbRemainBuf[ET199_SHA_CBLOCK];
}SHA_CONTEXT,*PSHA_CONTEXT;
```

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见sha1_final的示例。

6.4.6 sha1_update

*BYTE _sha1_update(PSHA_CONTEXT pCtx,const void *pvData, BYTE bLen)*

说明：

使用SHA1散列算法进行运算。

参数：

pCtx　[in]上下文环境结构的地址。

*pvData [in]进行散列的数据。

bLen　[in]进行散列数据的长度。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见sha1_final的示例。

6.4.7 sha1_final

*BYTE _sha1_final(PSHA_CONTEXT pCtx, void *pvResult)*

说明：

得到SHA1散列算法运算的结果。SHA1的散列结果长度为20字节。

参数：

pCtx [in]上下文环境结构的地址。

*pvResult[out]散列结果存放的地址。

注意：

散列结果也会在COS内部保留（见3.2.9），COS 内部存在一个 Buffer 会 保存上一次散列计算的结果。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;

char xdata result[20];

BYTE xdata text[] = "HELLO";
```

```
SHA_CONTEXT xdata sha1ctx;


//进行散列前要先初始化上下文环境
bRes = _sha1_init(&sha1ctx);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}


//进行散列计算
bRes = _sha1_update(&sha1ctx, text, 5);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}


//得到散列的结果，SHA1的散列结果长度为20个字节
bRes = _sha1_final(&sha1ctx, result);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}


_set_response(20,result);
_exit();
```

6.4.8 md5_init

BYTE _md5_init(PMD5_CONTEXT pCtx)

说明：

使用MD5散列算法前，先要进行初始化。

参数：

pCtx　[in]上下文环境结构的地址。

MD5_CONTEXT上下文环境结构定义如下：

```
typedef struct _tagMD5_CONTEXT
{
  DWORD h[4];
  DWORD dwTotalLength;
  BYTE  bRemainLength;
  BYTE  pbRemainBuf[ET199_MD5_CBLOCK];
}MD5_CONTEXT,*PMD5_CONTEXT;
```

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

md5_final的示例。

6.4.9 md5_update

*BYTE _md5_update(PMD5_CONTEXT pCtx,const void *pvData, BYTE bLen)*

说明：

使用MD5散列算法进行运算。

参数：

pCtx　[in]上下文环境结构的地址。

*pvData　[in]进行散列的数据。

bLen　[in]进行散列数据的长度。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见md5_final的示例。

6.4.10 md5_final

*BYTE _md5_final(PMD5_CONTEXT pCtx, void *pvResult)*

说明：

得到MD5散列算法运算的结果。MD5的散列结果长度为16字节。

参数：

pCtx [in]上下文环境结构的地址。

*pvResult [in] 散列结果存放的地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
char xdata result[16];
BYTE xdata text[] = "HELLO";
MD5_CONTEXT xdata md5ctx;

//进行散列前要先初始化上下文环境
bRes = _md5_init(&md5ctx);
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}

//进行散列计算
bRes = _md5_update(&md5ctx, text, 5);
if(bRes != 0)
{
```

```
   _set_response(1,&bRes);
   _exit();
}

//得到散列的结果，MD5的散列结果长度为16个字节
bRes = _md5_final(&md5ctx, result);
if(bRes != 0)
{
   _set_response(1,&bRes);
   _exit();
}

_set_response(16,result);
_exit();
```

6.4.11 rsa_enc

*BYTE _rsa_enc(BYTE bMode, WORD wFileID, WORD wLen, void *pvData)*

说明：

使用RSA公钥进行加密。

参数：

bMode　[in]RSA加密模式，见表6-3。默认为1024位加密。

wFileID　[in]公钥文件ID。

wLen　[in]要加密数据的长度。见下面的注意。

*pvData　[in/out]输入为要加密的明文数据，输出为加密后的密文数据。

注意：

（1）bMode：RSA加密模式。RSA运算模式分为RSA_CALC_NORMAL和RSA_CALC_PKCS。每种模式都要与RSA密钥的位数结合使用（如：RSA_CALC_NORMAL | RSA_CALC_BIT_1024），位数默认为1024。

RSA加密模式		
名称	值	含义
RSA_CALC_NORMAL	0x00	直接运算，无编码
RSA_CALC_PKCS	0x02	PKCS#1标准加密
RSA密钥位数		
RSA_CALC_BIT_512	0x10	使用512位RSA密钥加密
RSA_CALC_BIT_1024（默认）	0x00	使用1024位RSA密钥加密
RSA_CALC_BIT_2048	0x20	使用2048位RSA密钥加密

图表 8 RSA运算模式

（2）当为RSA_CALC_NORMAL模式时，这时输入的是固定长度，根据RSA密钥的长度不同而不同，这时第3个参数应根据下表的规则给相应的值。如果原文数据超出或者不足这个固定的字节长度时，需要先进行分块和补齐操作，可以参看 6.4 节开始的补齐原则。如果不进行补齐操作，接口会自动在前面填充0。

RSA_CALC_NORMAL模式明文长度		
模式	值	明文长度（字节）
RSA_CALC_BIT_512	0x10	64
RSA_CALC_BIT_1024（默认）	0x00	128
RSA_CALC_BIT_2048	0x20	256

图表 9 RSA_CALC_NORMAL模式明文长度

（3）当为 RSA_CALC_PKCS模式时,输入的数据长度根据RSA密钥的长度不同而不同。这时第3个参数应根据下表的规则给相应的值。

RSA_CALC_NORMAL 模式明文长度		
模式	值	明文长度（字节）
RSA_CALC_BIT_512	0x10	1-53
RSA_CALC_BIT_1024（默认）	0x00	1-117
RSA_CALC_BIT_2048	0x20	1-245

图表 10 RSA_CALC_PKCS模式明文长度

（4）RSA加密后的数据根据RSA密钥的长度不同而不同，这时要注意分配给第 4 个参数*pvData缓冲区的空间大小。

加密后密文数据长度		
名称	值	加密后密文数据长度（字节）
RSA_CALC_BIT_512	0x10	64
RSA_CALC_BIT_1024（默认）	0x00	128
RSA_CALC_BIT_2048	0x20	256

图表 11 加密后密文数据长度

（5）在使用 RSA 进行加密时，为了防止被加密的数据大于 n 的情况，请将原文数据的第一个字节设为0。*pvData参数所指的第一个字节为0。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见rsa_gen_key的示例。

6.4.12 rsa_dec

*BYTE _rsa_dec(BYTE bMode, WORD wFileID, WORD wLen, void *pvData)*

说明：

使用RSA私钥进行解密。

参数：

bMode [in]RSA运算模式见表6-3。默认为1024位加密。

wFileID [in]私钥文件ID。

wLen [in]要解密数据的长度。见下面的注意。

*pvData [in/out]输入为要解密的密文数据，输出为解密后的明文数据。

注意：

（1）bMode：RSA 加密模式。见6.4.11 rsa_enc中的说明

（2）wLen：输入数据的长度根据 RSA 密钥的长度不同而不同。由于是加密后的数据,该长度为一固定值。如在加密时对原文数据进行过分块和补齐操作，解密后需要进行相应的反向操作得到明文数据。

密文数据长度		
名称	值	加密后密文数据长度（字节）
RSA_CALC_BIT_512	0x10	64
RSA_CALC_BIT_1024（默认）	0x00	128

RSA_CALC_BIT_2048	0x20	256

图表 12 密文数据长度

（3）当以RSA_CALC_PKCS解密后的数据的第一个字节表示的是原文的长度，后跟原文。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见rsa_gen_key的示例。

6.4.13 rsa_gen_key

BYTE _rsa_gen_key(WORD wPubID, WORD wPriID,WORD wKeyBitLen)

说明：

产生RSA密钥对。

参数：

wPubID　　[in]公钥文件ID。

wPriID　　[in]私钥文件ID。

wKeyBitLen　[in]RSA密钥长度。为：512/1024/2048中的一个。

注意：

（1）考虑到加密强度和运算速度，一般采用的RSA密钥长度为1024位。不同长度的RSA密钥在加解密时对应不同长度的数据，参见6.4.11和6.4.12 中的注意。

（2）wPubID和wPriID必须为当前目录下没有被使用的ID，该函数会创建 2 个新文件。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
char xdata plaintext[] = "HELLO";

BYTE xdata ciphertext[128];

BYTE xdata bRes = 0;

WORD xdata wPubKeyFileID = 0x1002;    //公钥文件ID

WORD xdata wPriKeyFileID = 0x1004;    //私钥文件ID

WORD xdata wKeyLen = 1024; //RSA密钥长度
```

```
//产生1024位RSA密钥对
bRes = _rsa_gen_key(wPubKeyFileID, wPriKeyFileID, wKeyLen);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}

memset(ciphertext,0,128);  //将ciphertext赋值为0
//将“HELLO”拷贝到ciphertext中
//为了防止被加密数据大于n值 第一个字节为0 所以从ciphertext+1开始
memcpy(ciphertext+1, plaintext, 6);
//RSA加密，加密结果128字节在ciphertext中
bRes = _rsa_enc(RSA_CALC_PKCS | RSA_CALC_BIT_1024, wPubKeyFileID, 6, ciphertext);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}

//RSA解密，输入长度为128字节，解密结果在ciphertext中
bRes = _rsa_dec(RSA_CALC_PKCS |RSA_CALC_BIT_1024, wPriKeyFileID, 128, ciphertext);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
```

```
}
_set_response(128, ciphertext);
_exit();
```

6.4.14 rsa_sign

*BYTE _rsa_sign(BYTE bMode, WORD wFileID, WORD wLen, void *pvData)*

说明：

使用RSA私钥进行签名。

参数：

bMode [in]签名模式，见表6-8。

wFileID [in]私钥文件ID。

wLen [in]要签名的数据长度，见表6-8。

*pvData [in/out]输入为要签名的数据。输出为签名后的数据。

注意;

（1）在进行签名时，一般先要进行HASH散列运算，参见6.4开始的数字签名中的说明。本接口使用的散列算法为SHA1算法，散列后的数据长度为20 字节。标准的签名一般要遵循PKCS#1的标准，该标准规定，在经过SHA1 散列算法散列后的HASH值，再在前面填充15个字节（0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x1a, 0x05, 0x00, 0x04, 0x14），即一共 35 个字节进行签名。如果您的签名要与其他的标准应用交 互，那么要使用 RSA_CALC_PKCS 的模式。在进行签名时有四种模式，每种模式都要与 RSA 密钥的位数结合使用 （如：RSA_CALC_HASH | RSA_CALC_BIT_1024），位数默认为 1024。请参看下表中的说明：

RSA签名模式			
名称	值	含义	签名数据长度（字节）
RSA_CALC_NORMAL	0x00	直接对 SHA1 散列后的HASH值进行私钥签名。您需要先在外部做一下SHA1散列运算	20
RSA_CALC_HASH	0x01	接口内部对输入数据进行 SHA1 散列运算， 再将 HASH 结果进行私钥签名	因为是先做 HASH 运算，因此长度没有限 制但要小于卡片的内存2K
RSA_CALC_PKCS	0x02	直接对 SHA1 散列后的HASH值按照PKCS#1的 标准进行私钥签名，接口内部做了填充操作。您需要先在外部做一下SHA1散列运算	20

RSA_CALC_HASH|RSA_CALC_PKCS	0x03	接口内部对输入数据进行SHA1散列运算，再将HASH结果按照PKCS#1的标准进行私钥签名	因为是先做HASH运算，因此长度没有限制，但要小于卡片的内存2K
RSA密钥位数			
RSA_CALC_BIT_512	0x10	使用512位RSA密钥签名	
RSA_CALC_BIT_1024（默认）	0x00	使用1024位RSA密钥签名	
RSA_CALC_BIT_2048	0x20	使用2048位RSA密钥签名	

图表 13 RSA 签名模式

（2）输出的签名后的数据根据RSA密钥的长度不同而不同,这时要注意分配给 第4个参数*pvData缓冲区的空间大小。具体长度见下表：

RSA签名后结果长度	
RSA密钥	签名后结果长度（字节）
512	64
1024	128
2048	256

图表 14 RSA签名后结果长度

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

见rsa_verify的示例。

6.4.15 rsa_verify

BYTE _rsa_verify(BYTE bMode, WORD wFileID, WORD wLen, void *pvData)

说明：

使用RSA公钥进行验证签名。

参数：

bMode [in]签名模式，见表6-8和下面的注意。

wFileID [in]公钥文件ID。

wLen [in]签名的数据长度。为：64/128/256的其中一个

*pvData [in]签名后的数据。

注意：

（1）在使用 rsa_sign 签名函数签名时如果选择的签名模式为 RSA_CALC_NORMAL 和 RSA_CALC_HASH 时，这两种模式都是对 SHA1 散列后的 HASH 值直接签名，那么验证签名的模式（第 1 个参数）应为 RSA_CALC_NORMAL。

（2）在使用 rsa_sign 签名函数签名时如果选择的签名模式为 RSA_CALC_PKCS 和 RSA_CALC_HASH|RSA_CALC_PKCS 时，这两种模式都是对 SHA1 散列后的 HASH 值按照 PKCS#1 的标准进行填充后再进行签名，那么验证签名的模式（第 1 个参数）应为 RSA_CALC_PKCS。

（3）在验证签名时，本函数会使用到 COS 内部的一个保存散列结果的 Buffer，该 Buffer 会保存上一次散列计算的结果（见 3.2.9）。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
char xdata text[] = "HELLO";
BYTE xdata signtext[128];
BYTE xdata bRes = 0;
WORD xdata wPubKeyFileID = 0x1002; //公钥文件ID
WORD xdata wPriKeyFileID = 0x1004; //私钥文件ID

memset(signtext,0,128);
memcpy(signtext,text,6);

//使用RSA私钥进行签名，先进行SHA1散列，
//再按PKCS#1标准进行签名运算。返回128字节的签名结果
bRes=_rsa_sign(RSA_CALC_HASH|RSA_CALC_PKCS, wPriKeyFileID,6,signtext);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}
```

```
//使用RSA公钥验证签名
bRes = _rsa_verify(RSA_CALC_PKCS, wPubKeyFileID, 128, signtext);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
}

//返回签名后的结果
_set_response(128, signtext);
_exit();
```

6.5 系统功能

6.5.1 rand

*BYTE _rand(void *pvData, BYTE bLen)*

说明： 取随机数。

参数：

*pvData [out]取得的随机数存放的地址。

bLen [in]随机数的长度。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0; BYTE xdata bRand[8];

memset(bRand, 0, 8); bRes = _rand(bRand,8); if(bRes != 0)
{
_set_response(1,&bRes);
```

```
_exit();
}

_set_response(8, bRand);
_exit();
```

6.5.2 get_version

*BYTE _get_version(BYTE bFlag,void *pvData,BYTE bLen)*

说明： 获取硬件信息。

参数：

bFlag [in]标识，取得标识对应的相关信息。见下表的说明

*pvData [out]存放信息的缓冲区的地址。

bLen[in]缓冲区的大小。

硬件信息Flag标识	
GLOBAL_SERIAL_NUMBER	取8 字节序列号
GLOBAL_CLIENT_NUMBER	取4 字节客户号
GLOBAL_COS_VERSION	取2 字节COS版本

图表 15 硬件信息Flag标识说明

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
BYTE xdata bSN[8];
memset(bSN, 0, 8);
bRes = _get_version(GLOBAL_SERIAL_NUMBER,bSN,8);
if(bRes != 0)
{
_set_response(1,&bRes);
_exit();
```

```
}
_set_response(8, bSN);
_exit();
```

6.6 双精度浮点运算

C51语言本身并不支持双精度浮点运算，是我们在ET199中实现的。在使用双精度浮点数运算时，不能像整数和单精度浮点数运算一样，直接使用+，-，×，÷等符号， 而要使用ET199提供的接口。C51中双精度浮点数在表示时与PC中的高级语言不同，有 一个倒序的问题。例如：双精度浮点数1.0在PC高级语言中（Debug调试时可以查看双精度浮点数的地址）表示为：0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F。在 C51语言中双精度浮点数1.0为：0x3F, 0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00， 因此由外部输入到C51程序时，需要倒序，参见6.2.1中的说明。

我们提供了一个工具（Double2Byte.exe），可以方便的在双精度将浮点数与 8 字节数组之间进行自动转换。本节所讲述的双精度数学运算的接口中声明为 DOUBLE 类型的，都是一个 8 个字节的数组。调用方法参见 add 后的示例，其他接口的调用方法类似。

6.6.1 add

BYTE _add(DOUBLE result, DOUBLE x, DOUBLE y)

说明： 双精度浮点数加法运算。

参数：

result [out]相加的结果。

x [in]被加数。

y [in]加数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
DOUBLE xdata dx, dy, dResult;
//dx为 1.0 ,( 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F )
DOUBLE_INIT(dx, 1.0);
//dy为2.0 ,( 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40 )
```

```
DOUBLE_INIT(dy, 2.0);
bRes = _add(dResult, dx, dy);
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}
//返回为双精度浮点数 3.0 ,( 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x40 )
_set_response(8, &dResult);
_exit();
```

6.6.2 sub

BYTE _sub(DOUBLE result, DOUBLE x, DOUBLE y)

说明： 双精度浮点数减法运算。

参数：

result [out]相减的结果。

x [in]被减数。

y [in]减数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.3 mul

BYTE _mul(DOUBLE result, DOUBLE x, DOUBLE y)

说明： 双精度浮点数乘法运算。

参数：

result [out]相乘的结果。

x [in]被乘数。

y [in]乘数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.4 div

BYTE _div(DOUBLE result, DOUBLE x, DOUBLE y)

说明：双精度浮点数除法运算。

参数：

result　[out]相除的结果。

X　　[in]被除数。

Y　　[in]除数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.5 atan2

BYTE _atan2(DOUBLE result, DOUBLE x, DOUBLE y)

说明：

双精度浮点数商的反正切运算。

参数：

result　[out]商的反正切运算的结果。

x　　[in]被除数。

y　　[in]除数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.6 mod

BYTE _mod(DOUBLE result, DOUBLE x, DOUBLE pMod)

说明：　双精度浮点数求模运算。

参数：

result　[out]求模运算的结果。

x　　[in]被除数。

pMod　[in]除数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.7 pow

BYTE _pow(DOUBLE result, DOUBLE x, DOUBLE pExp)

说明： 双精度浮点数求幂运算。

参数：

result [out]求幂运算的结果。

x [in]底数。

pExp [in]指数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.8 modf

BYTE _modf(DOUBLE remain, DOUBLE x, DOUBLE intpart)

说明：

将一个双精度浮点数分割为整数部分和小数部分。

参数：

remain [out]分割后小数部分。

x [in]被分割的数。

intpart [out]分割后整数部分。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.9 frexp

*BYTE _frexp(DOUBLE result, DOUBLE x, WORD *pExp)*

说明：

把一个双精度浮点数拆成一个小数乘2的幂。

参数：

result [out]分拆后小数部分。

x [in]被分拆的数。

*pExp [out]指数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.10 ldexp

BYTE _ldexp(DOUBLE result, DOUBLE x, WORD exp)

说明：

求双精度浮点数乘2的幂。

参数：

result　[out]运算结果。

x　　[in]参与计算的双精度浮点数。

exp　　[in]2的指数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.11 sin

BYTE _sin(DOUBLE result, DOUBLE x)

说明：　正弦运算。

参数：

result　[out]运算结果。

x　　[in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.12 cos

BYTE _cos(DOUBLE result, DOUBLE x)

说明：　余弦运算。

参数：

result　[out]运算结果。

x　　[in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.13 tan

BYTE _tan(DOUBLE result, DOUBLE x)

说明：

正切运算。

参数：

result　[out]运算结果。

x　[in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.14 asin

BYTE _asin(DOUBLE result, DOUBLE x)

说明：

反正弦运算。

参数：

result　[out]运算结果。

x　[in]正弦值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.15 acos

BYTE _acos(DOUBLE result, DOUBLE x)

说明：

反余弦运算。

参数：

result　[out]运算结果。

x　[in]余弦值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.16 atan

BYTE _atan(DOUBLE result, DOUBLE x)

说明： 反正切运算。

参数：

result [out]运算结果。

x [in]正切值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.17 sinh

BYTE _sinh(DOUBLE result, DOUBLE x)

说明： 双曲正弦运算。

参数：

result [out]运算结果。

x [in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.18 cosh

BYTE _cosh(DOUBLE result, DOUBLE x)

说明： 双曲余弦运算。

参数：

result [out]运算结果。

x [in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.19 tanh

BYTE _tanh(DOUBLE result, DOUBLE x)

说明： 双曲正切运算。

参数：

result [out]运算结果。

x [in]弧度值。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.20 ceil

BYTE _ceil(DOUBLE result, DOUBLE x)

说明： 不小于该双精度浮点数的最小整数。

参数：

result [out]运算结果。

x [in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.21 floor

BYTE _floor(DOUBLE result, DOUBLE x)

说明： 不大于该双精度浮点数的最大整数。

参数：

result [out]运算结果。

x [in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.22 abs

BYTE _abs(DOUBLE result, DOUBLE x)

说明： 双精度浮点数的绝对值。

参数：

result [out]运算结果。

x [in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.23 exp

BYTE _exp(DOUBLE result, DOUBLE x)

说明：求以e为底，双精度浮点数为指数的幂。

参数：

result　[out]运算结果。

x　[in]指数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.24 log

BYTE _log(DOUBLE result, DOUBLE x)

说明：　求双精度浮点数的自然对数。

参数：

result　[out]运算结果。

x　[in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.25 log10

BYTE _log10(DOUBLE result, DOUBLE x)

说明：　求双精度浮点数的常用对数。

参数：

result　[out]运算结果。

x　[in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.26 sqrt

BYTE _sqrt(DOUBLE result, DOUBLE x)

说明：　求双精度浮点数的平方根。

参数：

result [out]运算结果。

x [in] 双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.6.27 cmp

char _cmp(DOUBLE x, DOUBLE y)

说明：比较两个双精度浮点数的大小。

参数：

x [in]双精度浮点数。

y [in]双精度浮点数。

返回值：

返回0表示相等；返回-1表示x<y；返回1表示x>y。

举例：

```
char xdata cRes = 0; DOUBLE xdata dx,dy;

//dx为1.0｛0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F｝ DOUBLE_INIT(dx, 1.0);

//dy为2.0｛0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40｝

DOUBLE_INIT(dy, 2.0);

cRes = _cmp(dx, dy);

_set_response(1, &cRes);//返回为0xFF = -1

_exit();
```

6.7 单精度浮点运算

C51语言中单精度浮点数的表示和PC机上的高级语言一样,都是4个字节,但顺序相反，需要进行倒序处理，请参看6.2.1中的说明。例如：单精度浮点数3.0，高级语言中（Debug调试时可以查看单精度浮点数的地址）表示为：0x00,0x00,0x40,0x40） 而在C51中，表示为：0x400x400x000x00四个字节，需要进行倒序，考虑到PC的性能远远高于智能卡，建议在高级语言中做倒序处理。调用方法参见 addf 后的示例，其他接口的调用方法类似。

6.7.1 addf

float _addf(float x, float y)

说明： 单精度浮点数加法运算。

参数：

x [in]被加数。

y [in]加数。

返回值： 返回运算结果。

举例：

```
float xdata fx;
float xdata fy;
float xdata fResult;

fx = 1.0;
fy = 2.0;
fResult = _addf(fx, fy);

//结果返回0x40 0x40 0x00 0x00四个字节，与高级语言中的顺序相反。
_set_response(4, &fResult);
_exit();
```

6.7.2 subf

float _subf(float x, float y)

说明：

单精度浮点数减法运算。

参数：

x [in]被减数。

y [in]减数。

返回值： 返回运算结果。

6.7.3 mulf

float _mulf(float x, float y)

说明： 单精度浮点数乘法运算。

参数：

x [in]被乘数。

y [in]乘数。

返回值： 返回运算结果。

6.7.4 divf

float _divf(float x, float y)

说明： 单精度浮点数除法运算。

参数：

x [in]被除数。

y [in]除数。

返回值： 返回运算结果。

6.7.5 atan2f

float _atan2f(float x, float y)

说明： 单精度浮点数商的反正切运算。

参数：

x [in]被除数。

y [in]除数。

返回值： 返回运算结果。

6.7.6 fmodf

float _fmodf(float x, float y)

说明： 单精度浮点数求模运算。

参数：

x [in]被除数。

y [in]除数。

返回值： 返回运算结果。

6.7.7 powf

float _powf(float x, float y)

说明： 单精度浮点数求模运算。

参数：

x [in]底数。

y [in]指数。

返回值： 返回运算结果。

6.7.8 cmpf

char _cmpf(float x, float y)

说明： 比较两个单精度浮点数的大小。

参数：

x [in]单精度浮点数。

y [in]单精度浮点数。

返回值：

返回0表示相等；返回-1表示x<y；返回1表示x>y。

举例：

```
float xdata fx;
float xdata fy;
char xdata cRes;

fx = 1.0;
fy = 2.0;
cRes = _cmpf(fx, fy);

_set_response(1, &cRes); //返回为0xFF = -1
_exit();
```

6.7.9 sinf

float _sinf(float x)

说明： 正弦运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.10 cosf

float _cosf(float x)

说明： 余弦运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.11 tanf

float _tanf(float x)

说明： 正切运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.12 asinf

float _asinf(float x)

说明： 反正弦运算。

参数：

x [in]正弦值。

返回值： 返回运算结果。

6.7.13 acosf

float _acosf(float x)

说明： 反余弦运算。

参数：

x [in]余弦值。

返回值： 返回运算结果。

6.7.14 atanf

float _atanf(float x)

说明： 反正切运算。

参数：

x [in]正弦值。

返回值： 返回运算结果。

6.7.15 sinhf

float _sinhf(float x)

说明： 双曲正弦运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.16 coshf

float _coshf(float x)

说明： 双曲余弦运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.17 tanhf

float _tanhf(float x)

说明： 双曲正切运算。

参数：

x [in]弧度值。

返回值： 返回运算结果。

6.7.18 ceilf

float _ceilf(float x)

说明： 不小于该单精度浮点数的最小整数。

参数：

x [in]单精度浮点数。

返回值： 返回不小于该单精度浮点数的最小整数。

6.7.19 floorf

float _floorf(float x)

说明： 不大于该单精度浮点数的最大整数。

参数：

x [in]单精度浮点数。

返回值： 返回不大于该单精度浮点数的最大整数。

6.7.20 absf

float _absf(float x)

说明： 单精度浮点数的绝对值。

参数：

x [in]单精度浮点数。

返回值： 返回该单精度浮点数的绝对值。

6.7.21 expf

float _expf(float x)

说明：

求以e为底，单精度浮点数为指数的幂。

参数：

x [in]指数。

返回值： 返回运算结果。

6.7.22 logf

float _logf(float x)

说明： 求单精度浮点数的自然对数。

参数：

x [in]单精度浮点数。

返回值： 返回运算结果。

6.7.23 log10f

float _log10f(float x)

说明： 求单精度浮点数的常用对数。

参数：

x [in]单精度浮点数。

返回值： 返回运算结果。

6.7.24 sqrtf

float _sqrtf(float x)

说明： 求单精度浮点数的平方根。

参数：

x [in]单精度浮点数。

返回值： 返回运算结果。

6.8 类型转换

C51 语言中使用不同数据类型进行计算时，有时需要相互之间进行转换，这就需要调用本节所提供的接口。

在C51语言中，int，long，float，double等类型的表示与在高级语言中是相反的， 参见6.2.1中的说明。例如：单精度浮点数1.0在高级语言中（Debug调试时可以查看单精度浮点数的地址）表示为：0x00,0x00,0x80,0x3F。这与C51语言中0x3F，0x80，0x00，0x00是相反的。要注意倒序的问题。

在C51语言中DOUBLE为一个8字节的数组，其表示也是与PC中高级语言相反的。 例如:双精度浮点数1.0在PC的高级语言中（Debug调试时可以查看双精度浮点数的地址）表示为：0x00,0x00,0x00,0x00,0x00,0x00,0xF0,0x3F。但在C51中为：0x3F,0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00。

本节中的6.8.1-6.8.4的调用方法参见dbltof后的示例。

本节中的6.8.5-6.8.8中使用的是宏定义，调用时给宏的名称。参见dtof后的示例。

6.8.1 dbltof

*BYTE _dbltof(float *result, DOUBLE x)*

说明： 双精度浮点数转换为单精度浮点数。

参数：

*result [out]单精度浮点数的地址。

x　　[in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;
float xdata fRes;
DOUBLE xdata dx;
//dx为1.0 ,( 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F )
DOUBLE_INIT(dx, 1.0);
bRes = _dbltof(&fRes, dx);
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}
_set_response(4, &fRes); //输出：0x3F 0x80 0x00 0x00，与高级语言相反
_exit();
```

6.8.2 ftodbl

BYTE _ftodbl(DOUBLE result, float x)

说明：　单精度浮点数转换为双精度浮点数。

参数：

result　　[out]双精度浮点数。

x　　[in]单精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.8.3 dbltol

*BYTE _dbltol(long *result, DOUBLE x)*

说明：

双精度浮点数转换为32位有符号整数。

参数：

*result　[out]32位有符号整数的地址。

x　　[in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.8.4 ltodbl

BYTE _ltodbl(DOUBLE result, long x)

说明：

32位有符号整数转换为双精度浮点数。

参数：

result　[out]双精度浮点数。

x　　[in] 32位有符号整数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.8.5 dtof

BYTE _dtof(float presult, DOUBLE px)*

说明：　双精度浮点数转换为单精度浮点数。

参数：

*presult　[out]单精度浮点数地址。

px　　[in] 双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

举例：

```
BYTE xdata bRes = 0;

float xdata fRes; DOUBLE xdata dx;

//dx为1.0（0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F）

DOUBLE_INIT(dx, 1.0);
```

```
bRes = _dtof(&fRes, dx);
if(bRes != 0)
{
  _set_response(1,&bRes);
  _exit();
}
_set_response(4, &fRes); //输出：0x3F 0x80 0x00 0x00，与高级语言相反
_exit();
```

6.8.6 ftod

BYTE _ftod(DOUBLE presult, float* px)

说明： 单精度浮点数转换为双精度浮点数。

参数：

presult [out]双精度浮点数。

*px [in]单精度浮点数地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.8.7 dtol

BYTE _dtol(long* presult, DOUBLE px)

说明：

双精度浮点数转换为32位有符号整数。

参数：

*presult [out] 32位有符号整数地址。

px [in]双精度浮点数。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.8.8 ltod

BYTE _ltod (DOUBLE presult, long* px)

说明：

32位有符号整数转换为双精度浮点数。

参数：

presult　[out]双精度浮点数。

*px　[in] 32位有符号整数地址。

返回值：

返回0表示成功。返回非0值表示执行该函数失败，参看6.10错误编码。

6.9 结构和常量

在使用 C51 语言进行编程时为了方便，定义了一些结构，常量和宏。这些都可以在 ET199.h 头文件中查找到。见下面的说明。

6.9.1 数据类型

为了与程序员熟悉的多数高级语言一样，我们在C51中声明了如下类型：

```
typedef unsigned char        BOOL;     //布尔类型
typedef unsigned short       WORD;     //无符号短整形，16位（2字节）
typedef unsigned short       WORD;     //无符号短整形，16位（2字节）
typedef unsigned long        dword;    //无符号长整形，32位（4字节）
typedef unsigned long        DWORD;    //无符号长整形，32位（4字节）
typedef unsigned char        byte;     //无符号字符，1字节
typedef unsigned char        BYTE;     //无符号字符，1字节
typedef BYTE              DOUBLE[8];   //双精度浮点数，8字节数组
typedef BYTE                 HANDLE;   //句柄，1字节
```

6.9.2 宏定义

```
#define SHA_DIGEST_LENGTH     20      //SHA1散列值长度
#define MD5_DIGEST_LENGTH     16      //MD5散列值长度
#define DES_KEY_LENGTH        8       //DES密钥长度
#define TDES_KEY_LENGTH       16      //3DES密钥长度
#define ET199_SHA_CBLOCK      64      //SHA1散列块长度
#define ET199_MD5_CBLOCK      64      //MD5散列块长度
#define RSA_CRYPT_NOPKCS      RSA_CALC_NORMAL     //直接加密
#define RSA_CRYPT_PKCS        RSA_CALC_PKCS       //按标准#PKCS1模式加密
```

```
#define RSA_SIGN_NOPH        RSA_CALC_NORMAL      //对散列后的数据签名
#define RSA_SIGN_HASH        RSA_CALC_HASH        //先进行SHA1散列，再签名
#define RSA_SIGN_PKCS        RSA_CALC_PKCS        //按#PKCS1标准，对散列后的数据签名

//按#PKCS1标准，先进行SHA1散列，再签名
#define RSA_SIGN_PH     (RSA_CALC_HASH|RSA_CALC_PKCS)
#define RSA_VERI_NOPKCS      RSA_CALC_NORMAL      //直接对数据验证签名
#define RSA_VERI_PKCS        RSA_CALC_PKCS        //按#PKCS1标准验证签名

#define RSA_CALC_BIT_512       0x10               //512 位加密
#define RSA_CALC_BIT_1024      0x00               //1024 位加密
#define RSA_CALC_BIT_2048      0x10               //2048 位加密

//双精度浮点数转换成单精度浮点数
#define _dtof(result,x) _dbltof(result,x)
//双精度浮点数转换为32位有符号整数
#define _dtol(result,x) _dbltol(result,x)
//单精度浮点数转换成双精度浮点数
#define _ftod(result,x) _ftodbl(result,*(x))
//32位有符号整数转换为双精度浮点数
#define _ltod(result,x) _ltodbl(result,*(x))

//短整形（short，int）倒序
#define _swap_u16(pvData)_swap(pvData,2)
//长整形（long），单精度浮点数（float）倒序
#define _swap_u32(pvData)_swap(pvData,4)
//将单精度浮点数y赋值给双精度浮点数x
#define DOUBLE_INIT(x,y) _ftodbl(x,y)
```

6.9.3 结构和枚举

SHA1 散列算法上下文结构

```
typedef struct _tagSHA_CONTEXT { DWORD    h[5];
     DWORD   dwTotalLength;
     BYTEbRemainLength;
```

```
    BYTEpbRemainBuf[ET199_SHA_CBLOCK];
}SHA_CONTEXT,*PSHA_CONTEXT;

typedef SHA_CONTEXT ShaContext; //SHA_CONTEXT结构类型

typedef ShaContext* PShaContext;//指向SHA_CONTEXT结构的指针类型
```

MD5 散列算法上下文结构

```
typedef struct _tagMD5_CONTEXT { DWORD h[4];
  DWORD dwTotalLength; BYTE   bRemainLength;
  BYTE  pbRemainBuf[ET199_MD5_CBLOCK];
}MD5_CONTEXT,*PMD5_CONTEXT;

typedef MD5_CONTEXT Md5Context; //MD5_CONTEXT结构类型

typedef Md5Context* PMd5Context;//指向MD5_CONTEXT结构的指针类型
```

文件结构

```
typedef struct _FILE_INFO {
  WORD  wFileID;   //文件ID
  BYTE  bFileType; //文件类型
  WORD  wFileSize; //文件大小
}EFINFO,*PEFINFO;
```

文件类型

```
enum
{
    FILE_TYPE_EXE = 0x00,  //可执行文件
    FILE_TYPE_DATA, //数据文件
    FILE_TYPE_RSA_PUBLIC,  //RSA 公钥文件
    FILE_TYPE_RSA_PRIVATE, //RSA 私钥文件
    FILE_TYPE_INTERNAL_EXE //可执行文件（不可读写）
```

```
};
```

创建文件

```
enum
{
//如果文件已存在，打开该文件；反之则创建新文件并打开
CREATE_OPEN_ALWAYS = 0x00,
//创建并打开新文件，如果文件已经存在则返回错误
CREATE_FILE_NEW,
//打开已存在文件，功能同_open()函数
CREATE_OPEN_EXISTING
};
```

硬件信息

```
enum
{
GLOBAL_SERIAL_NUMBER = 0x00,  //取8 字节序列号
GLOBAL_CLIENT_NUMBER,//取4 字节客户号
GLOBAL_COS_VERSION  //取2 字节COS版本
};
```

加解密和签名

```
enum{
RSA_CALC_NORMAL = 0x00, //直接运算，无编码
RSA_CALC_HASH,  //对输入数据进行HASH 运算，再将结果进行私钥加密
RSA_CALC_PKCS,  //PKCS#1标准加密
};
```

6.10 错误编码

错误编码定义如下：

```
#define ET_SUCCESS                  0x00    //  成功
#define ET_UNKNOWN                  0x01    //  未知错误
#define ET_INVALID_PARAMETER        0x02    //  无效的参数
#define ET_INVALID_ADDRESS          0x03    //  无效的地址,虚拟机地址越界
#define ET_INVALID_SIZE             0x04    //  无效的长度
#define ET_FILE_NOT_FOUND           0x05    //  文件没找到
#define ET_ACCESS_DENIED            0x06    //  访问文件失败
#define ET_FILE_SELECT              0x07    //  文件打开个数已达上限
#define ET_INVALID_HANDLE           0x08    //  无效的文件句柄
#define ET_FILE_OUT_OF_RANGE        0x09    //  文件读写越界
#define ET_FILE_TYPE_MISMATCH       0x0A    //  文件存在但类型不匹配
#define ET_FILE_SIZE_MISMATCH       0x0B    //  文件存在但长度不匹配
#define ET_NO_SPACE                 0x0C    //  文件夹空间不足
#define ET_FILE_EXIST               0x0D    //  文件已存在
#define ET_INVALID_KEY_FORMAT       0x0E    //  无效的RSA密钥文件格式
#define ET_KEY_LEN_MISMATCH         0x0F    //  传入的密钥长度与实际长度不匹配
#define ET_RSA_INVALID_KEY_FILE     0x10    //  文件类型不符合要求
#define ET_RSA_ENC_DEC_FAILED       0x11    //  RSA加密解密失败
#define ET_RSA_SIGN_VERI_FAILED     0x12    //  RSA签名验证失败
#define ET_SHA1                     0x13    //  SHA1计算错误
#define ET_MD5                      0x14    //  MD5计算错误
#define ET_INVALID_ADDRESS          0x15    //  无效的内存指针
#define ET_EEPROM                   0x16     //  写 EEPROM 错误
```

第7章 ET199外部API函数（PC语言）

7.1 API接口函数

7.1.1 ETEnum

*DWORD WINAPI ETEnum(ET_CONTEXT *pETContextList, DWORD *pdwDeviceCount)*

说明：

枚举插在计算机上所有能正常打开的ET199设备,将获取的ET199设备的个数放在*pdwDeviceCount指针所指的变量中。

参数：

*pETContextList [out]指向ET_CONTEXT结构体数组的指针。

*pdwDeviceCount [in/out]指向存放ET199设备个数的DWORD指针。

注意：

（1）当不知道机器上插了几只 ET199 时，这时可先将*pETContextList 参数赋值为 NULL，然后得到计算机上 ET199 的个数。分配空间后再次调用 ETEnum 函数，获得插在机器上的每个 ET199 的信息。

（2）在调用其他函数进行操作前先要调用 ETEnum 函数枚举设备，每个设备的信息都存放在 ET_CONTEXT 结构中，该结构定义如下：

```
typedef struct{
DWORD    dwIndex;    //设备索引，从0开始
DWORD    dwVersion; //硬件版本号
HANDLE hLock;    //设备句柄
BYTE reserve[12]; //系统保留
DWORD    dwCustomer;    //客户号
BYTE bAtr[MAX_ATR_LEN];//ATR信息
BYTE ID[MAX_ID_LEN];    //设备ID号。
DWORD    dwAtrLen;  //ATR长度
```

```
}ET_CONTEXT,*PET_CONTEXT;

MAX_ATR_LEN 16 //ATR的最大长度

MAX_ID_LEN  8   //ID的最大长度
```

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例： 见 ETClose 的示例。

7.1.2 ETOpen

DWORD WINAPI ETOpen(ET_CONTEXT *pETCtx)

说明：

打开ET199设备。调用该函数前需要先调用ETEnum函数。

参数：

*pETCtx [in/out]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

注意：

（1）当要打开指定的加密锁时，可以对 ATR 文件和客户号进行辨别，来打开相应的 ET199。ATR 文件是通过 ETEnum 时得到的。客户号是通过 ETControl 函数得到的，每个开发商的 ID 都是不同的。

（2）该函数打开时是共享模式，即允许其他进程也可以打开 ET199。在这种模式下是支持多进程的。

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看 7.2.1 错误编码。

举例：见 ETClose 的示例。

7.1.3 ETOpenEx

DWORD WINAPI ETOpenEx(ET_CONTEXT *pETCtx, ET_OPENINFO *pETOpenInfo)

说明：

按设定模式打开ET199设备。调用该函数前需要先调用ETEnum函数。

参数：

*pETCtx [in/out]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

*pETOpenInfo [in]指向ET_OPENINFO结构的指针，该结构描述打开的方式。

注意：

（1）使用该函数可以按照指定的模式打开 ET199，模式分为共享和独占两种。共享模式时允许本进程或者其他进程再次打开 ET199 设备。独占模式时再次打开 ET199 设备，会找不到加密锁。所以当您使用独

占模式时要特别小心，一定在打开完成后进行关闭，否则会造成其他的打开尝试全部失败，从而软件运行异常。

（2）ET_OPENINFO 结构定义如下

```
typedef struct _ET_OPENINFO {

   DWORD dwOpenInfoSize;    // 结构体大小

   DWORD dwShareMode;   // 模式

} ET_OPENINFO,*PET_OPENINFO;

模式分为两种：

#define ET_EXCLUSIVE_MODE  0   /** 独占模式*/

#define ET_SHARE_MODE  1   /** 共享模式*/
```

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看 7.2.1 错误编码。

举例： 见 ETClose 的示例。

7.1.4 ETClose

*DWORD WINAPI ETClose(ET_CONTEXT *pETCtx)*

说明：

关闭ET199设备。调用该函数前需要先打开设备（ETOpen或者ETOpenEx）

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针，该指针由ETEnum函数返回。

注意：

调用该函数关闭设备后，并不清除 ET199 设备的安全状态。如果要清除安全状态，需要在关闭前调用 ETControl 函数。

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看 7.2.1 错误编码。

举例：

```
DWORD dwRet = 0;

DWORD dwET199Count = 0;

ET_CONTEXT* pETContextList = NULL;
```

```
ET_OPENINFO etOpenInfo = {0};
DWORD i = 0;
//枚举插在机器上的 ET199 设备
dwRet=ETEnum(NULL,&dwET199Count);
if(dwRet != ET_E_INSUFFICIENT_BUFFER && dwRet)
{
   printf("ETEnum Error:0x%08x\n", dwRet);
   return;
}

//分配空间
pETContextList = new ET_CONTEXT[dwET199Count];
memset(pETContextList,0,sizeof(ET_CONTEXT)*dwET199Count);

//再次枚举，将每个 ET199 设备的信息放到 ET_CONTEXT 结构中
dwRet=ETEnum(pETContextList,&dwET199Count);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETEnum Error:0x%08x\n", dwRet);
   goto Clean;
}

//循环对不同的锁进行操作
for(i = 0; i<dwET199Count; ++i)
{
    //打开 ET199 设备
    dwRet = ETOpen(&pETContextList[i]);
```

```
    if(dwRet != ET_S_SUCCESS)
    {
       printf("ETOpen Error:0x%08x\n", dwRet);
       goto Clean;
    }

    /*
    //按独占方式打开 ET199 设备
    etOpenInfo.dwOpenInfoSize = sizeof(ET_OPENINFO);
    etOpenInfo.dwShareMode = ET_EXCLUSIVE_MODE;
    dwRet = ETOpenEx(&pETContextList[i], &etOpenInfo);
    if(dwRet != ET_S_SUCCESS)
    {
       printf("ETOpenEx Error:0x%08x\n", dwRet);
       goto Clean;
    }
    */

    //......
    //关闭 ET199 设备
    dwRet = ETClose(&pETContextList[i]);
    if(dwRet != ET_S_SUCCESS)
    {
        printf("ETClose Error:0x%08x\n", dwRet);
        goto Clean;
    }
}
Clean:
```

```
//释放分配的内存
if(pETContextList != NULL)
{
   delete [] pETContextList;
   pETContextList = NULL;
}
```

7.1.5 ETControl

*DWORD WINAPI ETControl(CONST ET_CONTEXT *pETCtx,*

DWORD dwCtlCode,

*CONST VOID *pInBuffer,*

DWORD dwInBufferLen,

*VOID *pOutBuffer,*

DWORD dwOutBufferLen,

*DWORD *pdwBytesReturned)*

说明：

发送控制指令给 ET199。调用该函数前需要先打开设备（ETOpen 或者ETOpenEx）

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

dwCtlCode [in]控制指令。

*pInBuffer [in]输入缓冲区指针，存放控制指令的额外数据。

dwInBufferLen [in]输入缓冲区长度。

*pOutBuffer [out]输出缓冲区指针,存放发送控制指令后返回的信息。

dwOutBufferLen [in]输出缓冲区长度。

*pdwBytesReturned [out]实际返回的数据长度。

注意：

控制指令dwCtlCode见下表：

宏	值	功能	输入	输出
ET_LED_UP	0x0000000	指示灯亮	无	无

ET_LED_DOWN	0x0000000	指示灯灭	无	无
ET_LED_WINK	0x00000003	指示灯闪烁	频率数（DWORD），如为 5，表示 0.5 秒闪烁一	无
T_GET_DEVICE_TYPE	0x00000011	获取设备类型	无	1 字节数据
ET_GET_SERIAL_NUMBER	0x00000012	获取 ET199 硬件唯一序列号 （参见 2.4 节）	无	8 字节硬件唯一序列号
ET_GET_DEVICE_USABLE_SPACE	0x00000013	获取 ET199 设备总的可用存储空间	无	4 字节无符号整数（DWORD），表示总的可用存储空间的大小
ET_GET_DEVICE_ATR	0x00000014	获取 ATR 信息（参见 2.4 节）	无	16 字节 ATR 信息
ET_GET_CUSTOMER_NAME	0x00000015	获取当前设备的客户号(参见 2.4 节）	无	4 字节客户号
ET_GET_MANUFACTURE_DATE	0x00000016	获取当前 设 备的 生产日期	无	ET_MANUFACTURE_DATE 结构，见下面的定 义
ET_GET_DF_AVAILABLE_SPACE	0x00000017	获取当前目录剩余可用空间	无	2 字节无符号整数（WORD)），表示当前 目录剩余可 用空间的大小
ET_GET_EF_INFO	0x00000018	获取当前目录下特定文件的属性信息	文件 ID，以字符串的 方式传入，如：“0001”	EFINFO 结构，见下面定义
ET_GET_COS_VERSION	0x00000019	获取 COS 版本信息	无	2 字节 COS 版本信息
ET_GET_CURRENT_TIME	0x00000020	得到硬件时钟芯片时间，获取的为标准 UTC 时间（仅支持 ET 金刚锁）	无	tm 结构，见 C 语言的 time.h 文件
ET_SET_DEVICE_ATR	0x00000021	设置 ATR 信息（参见 2.4 节),这时需要验证 根 目录开发商口令	16 字节自定义值	无

ET_SET_DEVICE_TYPE	0x00000022	设置卡片类型，设置成空锁 前 需要 先删除根目录。设置成加密锁 或者 PKI 格式前需要先设置成空锁	卡片类型，见下面的宏定义	无
ET_SET_SHELL_KEY	0x00000023	设置外壳种子码	8 字节外壳	无
ET_SET_CUSTOMER_NAME	0x00000024	设置当前 设 备的客户号（参见 2.4 节）	4 字节客户号	
ET_RESET_DEVICE	0x00000031	重置设备，清除设备当前的安全状态	无	无

图表 16 控制指令

```
//ET199类型宏定义

#define  ET_DEVICE_TYPE_PKI 0x01        /*身份验证锁类型*/

#define  ET_DEVICE_TYPE_DONGL  0x02        /*加密锁类型*/

#define  ET_DEVICE_TYPE_EMPTY  0x04        /*空锁类型*/


//ET_MANUFACTURE_DATE结构定义

typedef struct

{

BYTE byYear; /** 年*/

BYTE byMonth;    /** 月*/

BYTE byDay;  /** 日*/

BYTE byHour; /** 时*/

BYTE byMinute;   /** 分*/

BYTE bySecond;   /** 秒*/

}ET_MANUFACTURE_DATE,*PET_MANUFACTURE_DATE;
```

```
//EFINFO结构定义
typedef struct
{
WORD wFileID;   /** 文件ID*/
BYTE bFileType; /** 文件类型*/
WORD wFileSize; /** 文件大小*/
}EFINFO,*PEFINFO;
```

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
DWORD dwRet = 0;
DWORD dwET199Count = 0;
ET_CONTEXT* pETContextList = NULL;
DWORD dwOut = 0;
//枚举插在机器上的 ET199 设备
dwRet=ETEnum(NULL,&dwET199Count);
if(dwRet != ET_E_INSUFFICIENT_BUFFER && dwRet)
{
printf("ETEnum Error:0x%08x\n", dwRet);
return;
}

//分配空间
pETContextList = new ET_CONTEXT[dwET199Count];
memset(pETContextList,0,sizeof(ET_CONTEXT)*dwET199Count);
```

```
//再次枚举，将每个 ET199 设备的信息放到 ET_CONTEXT 结构中
dwRet=ETEnum(pETContextList,&dwET199Count);
if(dwRet != ET_S_SUCCESS)
{
printf("ETEnum Error:0x%08x\n", dwRet);
goto Clean;
}

//打开找到的第一把 ET199
dwRet = ETOpen(&pETContextList[0]);
if(dwRet != ET_S_SUCCESS)
{
printf("ETOpen Error:0x%08x\n", dwRet);
goto Clean;
}
//闪烁频率，每隔 0.5 秒闪烁，ET_LED_WINK
DWORD Frq = 5;
dwRet = ETControl(&pETContextList[0],ET_LED_WINK,
&Frq,sizeof(Frq),NULL,0,NULL);
if(dwRet != ET_S_SUCCESS)
{
printf("ETControl ET_LED_WINK Error:0x%08x\n", dwRet);
goto Clean;
}

//获取 ET199 设备类型，ET_GET_DEVICE_TYPE
DWORD dwDeviceType = 0;
dwRet = ETControl(&pETContextList[0],ET_GET_DEVICE_TYPE, NULL,
```

```
0,&dwDeviceType,sizeof(dwDeviceType),&dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_GET_DEVICE_TYPE Error:0x%08x\n", dwRet);
   goto Clean;
}

//获取硬件序列号（8 字节）ET_GET_SERIAL_NUMBER
BYTE bSN[8] = {0};
dwRet = ETControl(&pETContextList[0],
ET_GET_SERIAL_NUMBER, NULL, 0, bSN, 8, &dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_GET_SERIAL_NUMBER Error:0x%08x\n", dwRet);
   goto Clean;
}

//获取 ET199 设备总的可用存储空间，ET_GET_DEVICE_USABLE_SPACE
DWORD dwTotalSpace = 0;
dwRet = ETControl(&pETContextList[0],ET_GET_DEVICE_USABLE_SPACE,
NULL,0,&dwTotalSpace,sizeof(dwTotalSpace),&dwOut);
if(dwRet != ET_S_SUCCESS)
{
printf("ETControl ET_GET_DEVICE_USABLE_SPACE Error:0x%08x\n", dwRet);
goto Clean;
}

//获取 ET199 设备的 ATR（16 字节）,ET_GET_DEVICE_ATR
```

```
BYTE pbATR[16] = {0};
dwRet = ETControl(&pETContextList[0],ET_GET_DEVICE_ATR,
NULL,0,pbATR,16,&dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_GET_DEVICE_ATR Error:0x%08x\n", dwRet);
   goto Clean;
}

//获取 ET199 设备的客户号（4 字节）,ET_GET_CUSTOMER_NAME
BYTE pbCustomerName[4] = {0};
dwRet = ETControl(&pETContextList[0],ET_GET_CUSTOMER_NAME,
NULL,0,pbCustomerName,4,&dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_GET_CUSTOMER_NAME Error:0x%08x\n", dwRet);
   goto Clean;
}

//获取 ET199 设备的生产日期，ET_GET_MANUFACTURE_DATE
ET_MANUFACTURE_DATE md = {0};
dwRet = ETControl(&pETContextList[0],ET_GET_MANUFACTURE_DATE,
NULL,0,&md,sizeof(ET_MANUFACTURE_DATE),&dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_GET_MANUFACTURE_DATE Error:0x%08x\n", dwRet);
   goto Clean;
}
```

```
//获取当前目录下的剩余空间，ET_GET_DF_AVAILABLE_SPACE

DWORD dwFreeSpace = 0;

dwRet = ETControl(&pETContextList[0], ET_GET_DF_AVAILABLE_SPACE,
NULL,0,&dwFreeSpace,sizeof(dwFreeSpace),&dwOut);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETControl ET_GET_DF_AVAILABLE_SPACE Error:0x%08x\n", dwRet);

   goto Clean;

}



//获取文件 ID 为 0x0001 的文件信息 EFINFO ef = {0};

dwRet = ETControl(&pETContextList[0], ET_GET_EF_INFO,

"0001",4,&ef,sizeof(EFINFO),&dwOut);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETControl ET_GET_DF_AVAILABLE_SPACE Error:0x%08x\n", dwRet);

   goto Clean;

}



//获取 ET199 设备的 COS 版本信息，ET_GET_COS_VERSION

DWORD wCosVersion = 0;

dwRet = ETControl(&pETContextList[0], ET_GET_COS_VERSION,
NULL,0,&wCosVersion,sizeof(wCosVersion),&dwOut);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETControl ET_GET_COS_VERSION Error:0x%08x\n", dwRet);

   goto Clean;

}
```

```
//验证根目录开发商密码

dwRet = ETVerifyPin(&pETContextList[0],ET_DEFAULT_DEV_PIN,
ET_DEV_PIN_LEN,ET_DEV_PIN);

if(dwRet)

{

   printf("ETVerifyPin Error:0x%08x\n", dwRet);

   goto Clean;

}



//设置 ET199 设备的 ATR（16 字节）信息，ET_SET_DEVICE_ATR

//设置前需要先验证根目录的开发商口令

BYTE pbSetATR[17] = "RockeyET199ET199";

dwRet = ETControl(&pETContextList[0], ET_SET_DEVICE_ATR,
pbSetATR,16,NULL,0,&dwOut);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETControl ET_SET_DEVICE_ATR Error:0x%08x\n", dwRet);

   goto Clean;

}



//删除根目录

dwRet = ETEraseDir(&pETContextList[0],NULL);

if(dwRet)

{

   printf("ETEraseDir Error:0x%08x\n", dwRet);

   goto Clean;

}
```

```
//设置 ET199 设备的类型，ET_SET_DEVICE_TYPE

//设置成空锁前需要先删除根目录

//设置成空锁后，无法再继续任何加密锁功能的操作

BYTE bDeviceType = ET_DEVICE_TYPE_EMPTY;

dwRet = ETControl(&pETContextList[0],ET_SET_DEVICE_TYPE,

&bDeviceType,1,NULL,0,&dwOut);

if(dwRet)

{

   printf("ETControl ET_SET_DEVICE_TYPE Error:0x%08x\n", dwRet);

   goto Clean;

}


//设置外壳种子码，ET_SET_SHELL_KEY

BYTE pbSeed[8] = {0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8};

dwRet = ETControl(&pETContextList[0], ET_SET_SHELL_KEY, pbSeed,8,NULL,0,&dwOut);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETControl ET_SET_SHELL_KEY Error:0x%08x\n", dwRet);

   goto Clean;

}


//设置 ET199 设备的客户号（4 字节），ET_SET_CUSTOMER_NAME

//设置前需要先验证根目录的开发商口令

BYTE pbSetCustomerName[4] = {1,2,3,4};

dwRet = ETControl(&pETContextList[0],ET_SET_CUSTOMER_NAME,

pbSetCustomerName,4,NULL,0,&dwOut);

if(dwRet != ET_S_SUCCESS)
```

```
{
   printf("ETControl ET_SET_CUSTOMER_NAME Error:0x%08x\n", dwRet);
   goto Clean;
}

//清除设备当前的安全状态
dwRet = ETControl(&pETContextList[0], ET_RESET_DEVICE, NULL,0,NULL,0,&dwOut);
if(dwRet != ET_S_SUCCESS)
{
   printf("ETControl ET_RESET_DEVICE Error:0x%08x\n", dwRet);
   goto Clean;
}
Clean:
dwRet = ETClose(&pETContextList[0]);

//释放分配的内存
if(pETContextList != NULL)
{
   delete [] pETContextList;
   pETContextList = NULL;
}
```

7.1.6 ETCreateDir

DWORD WINAPI ETCreateDir(

*CONST ET_CONTEXT *pETCtx,*

LPCSTR lpszDirID,

DWORD dwDirSize,

DWORD dwFlags)

说明： 创建目录，如创建成功，新创建的目录为当前目录。调用该函数前需要先打开设备（ETOpen或者

ETOpenEx），创建子目录时要验证开发商口令。

参数：

*pETCtx　　[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

lpszDirID　[in]目录ID。为0-9，a-f或者A-F中的4个字符。见注意。

dwDirSize　[in]目录大小。不能大于当前目录下的剩余空间，见注意。

dwFlags　　[in]创建目录时的标志。见注意。

注意：

（1）pszDirID 参数：该参数为目录 ID，在 ET199 中，目录 ID 为 2 个字节， 通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符(0-9,a-f 或者 A-F),因此第二个参数输入 4 个字符的字符串，如：“1002”表示目 录 ID 为 0x1002，“00A8”表示目录 ID 为 0x00A8。另外，字符串要 4 个字符，不能省略，如“9B”，这样会造成混淆，应为“009B”，表示目录 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）dwDirSize参数：这个参数为要创建的目录的大小，不能大于当前目录的剩余空间。另外在创建目录时，目录的信息也会占用一部分空间，也需要考虑进去。目录信息要占用102字节，如果建立的目录大小为1000字节，那么可用空间为1000-102=898字节。

（3）dwFlags参数：创建目录时的标志。

i)、ET_CREATE_ROOT_DIR　0x00000000 根目录标志。在创建根目录时，目录ID参数lpszDirID只能输入NULL或者空字符串（“”），目录大小参数 dwDirSize只能为0。

ii)、ET_CREATE_SUB_DIR　0x00000001 子目录标志，创建子目录。创建后，需要先验证该目录的用户口令，才能 获得权限。

（4）创建根目录时，直接创建，根目录会占用全部的用户空间(64K)。创建当前工作目录下的子目录时，需要先验证当前工作目录的开发商口令。

返回值： 返回 ET_S_SUCCESS(0x00000000)表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

见ETCreateDirEx的示例。

7.1.7 ETCreateDirEx

```
DWORD WINAPI ETCreateDirEx(
                    CONST ET_CONTEXT      *pETCtx,
                    LPCSTR    lpszDirID,
                    DWORD   dwDirSize,
```

DWORD dwFlags,

*CONST ET_CREATEDIRINFO *pCreateDirInfo)*

说明：

创建目录，如创建成功，新创建的目录为当前目录。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx），创建子目录时要验证开发商口令。

参数：

*pETCtx	[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。
lpszDirID	[in]目录ID。为0-9，a-f或者A-F中的4个字符。见注意。
dwDirSize	[in]目录大小。不能大于当前目录下的剩余空间，见注意。
dwFlags	[in]创建目录时的标志。见注意。
*pCreateDirInfo	[in]指向ET_CREATEDIRINFO结构的指针。见注意。

注意：

（1）lpszDirID 参数：该参数为目录 ID，在 ET199 中，目录 ID 为 2 个字节， 通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如："1002"表示目 录 ID 为 0x1002，"00A8"表示目录 ID 为 0x00A8。另外，字符串要 4 个字符，不能省略，如"9B"，这样会造成混淆，应为"009B"，表示目录 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）dwDirSize 参数：这个参数为要创建的目录的大小，不能大于当前目录的 剩余空间。另外在创建目录时，目录的信息也会占用一部分空间，也需要考虑进去。目录信息要占用 102 字节，如果建立的目录大小为1000字节，那么可用空间为1000-102=898字节。

（3）dwFlags参数：创建目录时的标志。

i)、 ET_CREATE_ROOT_DIR 0x00000000 根目录标志。在创建根目录时，目录ID参数lpszDirID只能输入NULL或 者空字符串（""），目录大小参数dwDirSize只能为0。

ii)、ET_CREATE_SUB_DIR 0x00000001 子目录标志，创建子目录。创建后，需要先验证该目录的用户口令，才能获得权限。

（4）*pCreateDirInfo参数：该参数为指向ET_CREATEDIRINFO结构的指针。

i)、当使用该函数创建根目录时，必须填写该结构。创建时把该结构中的 16 字节数组中的数据写入到ATR文件中（ATR文件参见2.4的说明）。 在使用ETEnum函数枚举ET199时，会得到每个设备的ATR文件的内容，从而进行判断。创建根目录后，将直接获得根目录的开发级权限。这时开发商口令和用户口令都将被设置成默认值，即开发商口令： "12345678123456781234567812345678"，用户口令："12345678"。请开发商将口令设置成自己保密的独特值，见2.2节的说明。

ii)、当使用该函数创建子目录时，与ETCreateDir函数一样。创建后，需要先验证该目录的用户口令，

才能获得权限。

```
typedef struct _ET_CREATEDIRINFO{

//结构体大小，必须为sizeof(CREATEDIRINFO)

DWORD dwCreateDirInfoSize;

//16字节ATR文件内容，用户自定义

BYTE szAtr[MAX_ATR_LEN]; // MAX_ATR_LEN为16

} ET_CREATEDIRINFO,*PET_CREATEDIRINFO;
```

（5）创建根目录时，直接创建，根目录会占用全部的用户空间（64K）创建当 前工作目录下的子目录时，需要先验证当前工作目录的开发商口令。

返回值：

返回ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看7.2.1错误编码。

举例：

```
//使用 ETCreateDirEx 函数创建根目录

ET_CREATEDIRINFO cDirInfo = {0};

cDirInfo.dwCreateDirInfoSize = sizeof(ET_CREATEDIRINFO);

memcpy(cDirInfo.szAtr,"et199199et199199",16);


dwRet = ETCreateDirEx(&pETContextList[0], NULL,

0, ET_CREATE_ROOT_DIR, &cDirInfo);

if(dwRet != ET_S_SUCCESS)

{

    printf("ETCreateDirEx Error:0x%08x\n", dwRet);

}

//使用ETCreateDir函数创建一个目录ID为0x1008，

//大小为10000字节的子目录

dwRet = ETCreateDir(&pETContextList[0], "1008",10000, ET_CREATE_SUB_DIR);

if(dwRet != ET_S_SUCCESS)
```

```
{
    printf("ETCreateDir Error:0x%08x\n", dwRet);
}
```

7.1.8 ETChangeDir

*DWORD WINAPI ETChangeDir(CONST ET_CONTEXT *pETCtx,LPCSTR lpszPath)*

说明：

改变当前工作目录。调用该函数前需要先打开设备（ETOpen或者ETOpenEx）

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针，该指针由 ETEnum函数返回。

lpszPath [in]新的工作目录的路径。见注意。

注意：

lpszPath参数：新的工作目录的路径。绝对路径或者相对路径：

i)、绝对路径：以根目录为参考，包目录括路径的全部。如:"\"（选择 根目录);"\1008\1002"（选择根目录下的ID为0x1008目录下的ID 为 0x1002 的目录）在编程中要注意转义字符，有"\"的输入应该 为"\\"。

ii)、相对路径：以当前目录为参考。如:"1006"（选择当前目录下的 ID 为0x1006的目录）

返回值：

返回ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看7.2.1错误编码。

举例：

```
//改变工作目录为根目下的ID为0x1008目录下的ID为0x1002的目录
dwRet = ETChangeDir(&pETContextList[0], "\\1008\\1002");
if(dwRet != ET_S_SUCCESS)
{
    printf("ETChangeDir Error:0x%08x\n", dwRet);
}
```

7.1.9 ETEraseDir

*DWORD WINAPI ETEraseDir(CONST ET_CONTEXT *pETCtx,LPCSTR lpszDirID)*

说明： 清除当前工作目录下所有内容。清除后当前工作目录的安全属性被重置。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx），并验证开发商口令。

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

lpszDirID [in]保留值。必须为 NULL 或者空字符串（“”）。

注意：

（1）如果当前工作目录是子目录，则删除当前目录下的所有内容，包括文件和次级子目录，但当前目录不被删除。删除完成后，当前目录的开发商口令和用户口令重置为默认值。开发商口令：“123456781234567812345678”，用户口令：“12345678”。

（2）如果当前目录为根目录，则直接删除根目录，加密锁恢复为空锁状态。开发商再次使用时，需要重新创建根目录。

（3）在清除当前工作目录时，要先验证当前工作目录的开发商口令。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//将工作目录转换成根目录
dwRet = ETChangeDir(&pETContextList[0], "\\");
if(dwRet == ET_S_SUCCESS)
{
    //使用默认开发商口令验证，如果不是默认口令，
    //这里需要替换成您自己的口令
    dwRet = ETVerifyPin(&pETContextList[0],ET_DEFAULT_DEV_PIN, ET_DEV_PIN_LEN,
    ET_DEV_PIN);
    if(dwRet)
    {
        printf("ETVerifyPin Error:0x%08x\n", dwRet);
    }
    //删除根目录
    dwRet = ETEraseDir(&pETContextList[0],NULL);
    if(dwRet)
```

```
    {
        printf("ETEraseDir Error:0x%08x\n", dwRet);
    }
}
```

7.1.10 ETVerifyPin

DWORD WINAPI ETVerifyPin(

*CONST ET_CONTEXT *pETCtx,*

*CONST BYTE *pbPin,*

DWORD dwPinLen,

DWORD dwPinType)

说明：

验证当前工作目录的口令（开发商口令或者用户口令），以取得相应的权限，参见2.2和2.3节。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx）。

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

*pbPin [in]口令的指针。

dwPinLen [in]口令长度。开发商口令为 24 字节，用户口令为 8 字节。

dwPinType [in]口令类型。见注意

注意：

（1）dwPinType 参数：口令类型分为开发商口令（ET_DEV_PIN）和用户口令（ET_USER_PIN）。

#define ET_USER_PIN 0x00000000 /** 用户 PIN*/

#define ET_DEV_PIN 0x00000001 /** 开发商 PIN*/

（2）开发商口令和用户口令默认没有重试次数的限制，可以无限次重试。如果开发商设置了重试次数，那么当连续输错的次数超过了设定的次数，则口令被锁死。如果该目录的开发商口令被锁死，需要清除这个目录的上一级目录。如果用户口令被锁死，需要使用开发商口令清除这个目录，恢复为默认值。开发商口令：“123456781234567812345678”，用户口令： “12345678”。参见 2.2 节（3）中的说明。

（3）如果验证口令失败，会返回一个错误值（0xF0000C00）。其中最后一个字 节表示目前的重试次数。如：0xF0000C0D，表明还有 13 次重试次数；0xF0000C00，表示已经没有重试次数，已经锁死了；0xF0000CFF，表示没有重试次数限制。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//验证当前目录的用户口令
BYTE pbUserPin[8] = {0};
memcpy(pbUserPin,"12345678",8);
dwRet = ETVerifyPin(&pETContextList[0], pbUserPin, 8, ET_USER_PIN);
if(dwRet != ET_S_SUCCESS)
{
    printf("ETVerifyPin Error:0x%08x\n", dwRet);
}
```

7.1.11 ETChangePin

```
DWORD WINAPI ETChangePin(
        CONST ET_CONTEXT    *pETCtx,
        CONST BYTE          *pbOldPin,
        DWORD               dwOldPinLen,
        CONST BYTE          *pbNewPin,
        DWORD               dwNewPinLen,
        DWORD               dwPinType,
        BYTE                byPinTryCount)
```

说明： 修改当前工作目录的口令（开发商口令或者用户口令），修改成功后取得相应的权限，参见 2.2 和 2.3 节。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx）

参数：

*pETCtx　[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

*pbOldPin　[in]旧口令的指针。

dwOldPinLen [in]旧口令长度。开发商口令为 24 字节，用户口令为 8 字节。

*pbNewPin　[in]新口令的指针。

dwNewPinLen [in]新口令长度。开发商口令为 24 字节，用户口令为 8 字节。

dwPinType　[in]口令类型。见注意

byPinTryCount　[in]口令重试次数。见注意

注意：

（1）dwPinType 参数：口令类型分为开发商口令（ET_DEV_PIN）和用户口令（ET_USER_PIN）

```
#define    ET_USER_PIN 0x00000000   /** 用户 PIN*/
#define    ET_DEV_PIN   0x00000001   /** 开发商 PIN*/
```

（2）修改口令的接口中包括了验证口令的功能，因此涉及到口令锁死的问题， 请参见 ETVerifyPin 接口中的注意和 2.2 节（3）中的说明。

（3）当设为 1—254 时为重试次数限制，当设置为 0 和 255（0xFF）时，为无重试次数限制。见 2.2 节中的说明。

（4）关于口令的宏定义：

```
#define  ET_DEFAULT_TRY          0xff        /**默认重试次数（无限次）*/
#define  ET_DEFAULT_DEV_PIN      (BYTE *)    "123456781234567812345678"
#define  ET_DEFAULT_USER_PIN     (BYTE *)    "12345678"
#define  ET_DEV_PIN_LEN          24
#define ET_USER_PIN_LEN          8
```

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//改变当前目录的用户口令
BYTE pbOldUserPin[8] = {0};
BYTE pbNewUserPin[8] = {0};
memcpy(pbOldUserPin,"12345678",8);
memcpy(pbNewUserPin,"88888888",8);
dwRet = ETChangePin(&pETContextList[0],
pbOldUserPin, 8, pbNewUserPin, 8, ET_USER_PIN,15);
if(dwRet != ET_S_SUCCESS)
{
printf("ETChangePin Error:0x%08x\n", dwRet);
}
```

7.1.12 ETCreateFile

```
DWORD WINAPI ETCreateFile(
        CONST ET_CONTEXT      *pETCtx,
```

LPCSTR lpszFileID,
DWORD dwFileSize,
BYTE bFileType)

说明：

在当前目录下创建文件。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx），并验证开发商口令。

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

lpszFileID [in]文件 ID。见注意。

dwFileSize [in]要创建的文件大小。见注意。

bFileType [in]文件类型。见注意。

注意：

（1）lpszFileID 参数：该参数为文件 ID，在 ET199 中，文件 ID 为 2 个字节，通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省 略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）dwFileSize 参数：为要创建的文件的大小。注意：文件本身也有文件属性 信息（16 字节），如：创建 100 字节的文件时（dwFileSize=100），实际占 用的空间为 100+16=116 字节。

（3）bFileType 参数：该参数为文件类型。见下表的说明：

FILE_TYPE_EXE	创建普通可执行文件。普通可执行文件开发 商口令验证后可以写入，或者通过 ET199 内部 的其他可执行文件写入，但任何情况下都不 能读取可执行文件中的内容。见 2.3.2 节中的说明。
FILE_TYPE_INTERNAL_EXE	创建内部可执行文件。内部可执行文件开发 商口令验证后可以写入，但不能被其他可执行文件改写，任何情况下都不能读取可执行 文件中的内容。见 2.3.2 节中的说明。
FILE_TYPE_DATA	创建内部数据文件。内部数据文件在开发商口令验证后可以写入，或者可以通过 ET199 内 部的可执行文件进行读写。见 2.3.2 节中的说明
FILE_TYPE_RSA_PUBLIC	创建 RSA 公钥文件。公钥文件在开发商口令验证后可以写入，或者可以通过 ET199 内部的可 执行文件进行读写。见 2.3.2 节中的说明。
FILE_TYPE_RSA_PRIVATE	创建 RSA 私钥文件。私钥文件在开发商口令验证后可以写入，或者可以通过 ET199 内部的可 执行文件进行写入，但任何情况下都不能读取私钥文件中的数据。见 2.3.2 节中的说明。

图表 17 文件类型

```
#define   FILE_TYPE_EXE              0    /** 可执行文件*/
#define   FILE_TYPE_DATA             1    /** 数据文件*/
#define   FILE_TYPE_RSA_PUBLIC       2    /** RSA 公钥文件*/
#define   FILE_TYPE_RSA_PRIVATE      3    /** RSA 私钥文件*/
#define   FILE_TYPE_INTERNAL_EXE     4    /** 可执行文件（不可读写）*/
```

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//验证根目录开发商口令

dwRet = ETVerifyPin(&pETContextList[0], ET_DEFAULT_DEV_PIN,
ET_DEV_PIN_LEN,ET_DEV_PIN);

if(dwRet != ET_S_SUCCESS)

{

   printf("ETVerifyPin Error:0x%08x\n", dwRet);

}


//在当前工作目录下创建文件ID为0x1008的内部数据文件，

//文件大小为：10000字节

dwRet = ETCreateFile(&pETContextList[0], "1008",10000, FILE_TYPE_DATA);

if(dwRet != ET_S_SUCCESS)

{

    printf("ETCreateFile Error:0x%08x\n", dwRet);

}
```

7.1.13 ETWriteFile

```
DWORD WINAPI ETWriteFile(
                CONST ET_CONTEXT    *pETCtx,
                LPCSTR              lpszFileID,
                DWORD               dwOffset,
```

*CONST VOID *pBuffer,*

DWORD dwBufferSize)

说明：

在当前工作目录下写入文件。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx），并验证开发商口令。

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

lpszFileID [in]文件 ID。见注意。

dwOffset [in]偏移值。即从文件哪里开始写入。

*pBuffer [in]写入数据的指针。

dwBufferSize [in]写入数据的长度。

注意：

（1）lpszFileID 参数：该参数为文件 ID，在 ET199 中，文件 ID 为 2 个字节，通过 字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省 略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//向当前工作目录中ID为0x1008的文件中，从第0个字节开始写入“HELLO”

BYTE pbData[] = "HELLO";

dwRet = ETWriteFile(&pETContextList[0],"1008", 0, pbData, lstrlen((const
char*)pbData));

if(dwRet != ET_S_SUCCESS)

{

    printf("ETWriteFile Error:0x%08x\n", dwRet);

}
```

7.1.14 ETWriteFileEx

DWORD WINAPI ETWriteFileEx(

*CONST ET_CONTEXT *pETCtx,*

```
LPCSTR          lpszFileID,
DWORD           dwOffset,
CONST VOID      *pBuffer,
DWORD           dwBufferSize,
DWORD           dwFileSize,
DWORD           *pdwBytesWritten,
DWORD           dwFlags,
BYTE            bFileType)
```

说明： 在当前工作目录下创建文件，写入文件。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx），并验证开发商口令。

参数：

*pETCtx [in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。

lpszFileID [in]文件 ID。见注意。

dwOffset [in]偏移值。即从文件哪里开始写入，见注意。

*pBuffer [in]写入数据的指针。

dwBufferSize [in]写入数据的长度。

dwFileSize [in]要创建的文件大小。

*pdwBytesWritten [out]实际写入的数据长度。

dwFlags [in]标志项。见注意。

bFileType [in]文件类型。见注意。

注意：

（1）建议用户使用 ETCreateFile 函数和 ETWriteFile 函数来完成文件的创建和写入。而不使用本函数。由于本函数将创建数据和写入数据集成到一起，需要在参数上注意很多的细节，容易混乱。

（2）lpszFileID 参数：该参数为文件 ID，在 ET199 中，文件 ID 为 2 个字节，通过 字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F）因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省 略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（3）dwOffset 参数：偏移值，即从文件哪里开始写入数据。只对内部数据文件有效，当文件为其他类型时，dwOffset 参数必须为 0。

（4）dwFlags 参数：标志项，见下表。

ET_CREATE_NEW	创建新文件。当创建可执行文件时，如果没有指定，创建的可执行文件为普通可执行文件，即可以被其他可执行文件改写。见 2.3.2 节中的说明。

ET_UPDATE_FILE	对已经存在的文件进行更新。

图表 18 标志项

（5）bFileType 参数：文件类型，表 7-2

返回值： 返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看 7.2.1 错误编码。

举例：

```
//向当前工作目录中ID为0x1008的文件中，从第0个字节开始写入“HELLO”
BYTE pbData[] = "HELLO";
dwRet = ETWriteFileEx(&pETContextList[0], "1008", 0,
pbData, lstrlen((const char*)pbData), 0,
&dwOut, ET_UPDATE_FILE, FILE_TYPE_DATA);
if(dwRet != ET_S_SUCCESS)
{
    printf("ETWriteFileEx Error:0x%08x\n", dwRet);
}
```

7.1.15 ETExecute

```
DWORD WINAPI ETExecute(
        CONST ET_CONTEXT    *pETCtx,
        LPCSTR              lpszFileID,
        CONST VOID          *pInBuffer,
        DWORD               dwInbufferSize,
        VOID                *pOutBuffer,
        DWORD               dwOutBufferSize,
        DWORD               *pdwBytesReturned)
```

说明： 在当前工作目录下，执行指定的可执行文件。调用该函数前需要先打开设备（ETOpen 或者 ETOpenEx）并验证用户口令。

参数：

*pETCtx	[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。
lpszFileID	[in]文件 ID。见注意。
*pInBuffer	[in]输入缓冲区，存放传给可执行文件（C51 程序）的数据。

dwInbufferSize [in]传入数据的长度。

*pOutBuffer [out]输出缓冲区，存放由可执行文件（C51 程序）传出的数据。

dwOutBufferSize [in]输出缓冲区的长度。见注意。

*pdwBytesReturned [out]可执行文件（C51 程序）返回的数据长度。

注意：

（1）lpszFileID 参数：该参数为文件 ID，在 ET199 中，文件 ID 为 2 个字节，通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）dwOutBufferSize 参数：为*pOutBuffer 指针所指向的输出缓冲区的长度，当该长度小于 ET199 内部的可执行文件（C51 程序）返回的长度时，该接口将返回错误。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
//C51程序，一个简单的加法。

//先使用KEIL编译成.bin文件，然后导入到ET199中，见5.2.3的说明

//导入成可执行文件，文件名称：1006

#include "ET199.h"

#include <string.h>


void main(void)

{

  int xdata i,j;


  memcpy(&i, pbInBuff, 2);

  memcpy(&j, pbInBuff + 2, 2);


  _swap( &i , sizeof(int) ) ;

  _swap( &j , sizeof(int) ) ;
```

```
   i = i+j;
   _set_response(2,&i);
   _exit();
}

//VC程序
//倒序程序
void FlipBuffer(unsigned char* pBuf, unsigned long ulLen)
{
   unsigned char ucTemp;
   for(unsigned long i = 0; i < ulLen >> 1; ++i)
   {
       ucTemp = pBuf[i];
       pBuf[i] = pBuf[ulLen - i - 1];
       pBuf[ulLen - i - 1] = ucTemp;
   }
}
void main(int argc, char* argv[])
{
   //打开锁
   ……
   short a,b;
   BYTE pbInput[4] = {0};  //输入缓冲区 BYTE
   pbOutput[4] = {0}; //输出缓冲区
   DWORD dwOut = 0;
   a = 8;
```

```
    b = 6;

    //C51 程序中 int 类型为 2 个字节，所以这里声明为 short 类型

    //C51 程序中 int 为倒序，应先倒序再传入 C51 程序中，见 6.2.1 节的说明

    //由于 PC 的性能远远高于智能卡，建议倒序在 PC 中完成，增加运行效率

    //可以在 C51 语言中使用_swap 函数做倒序处理，或者在这里做处理

    //FlipBuffer((unsigned char*)&a, 2);

    //FlipBuffer((unsigned char*)&b, 2);

    memcpy(pbInput, &a, 2);

    memcpy(pbInput+2, &b, 2);


    //调用可执行文件前需要先验证用户口令

    dwRet = ETVerifyPin(&pETContextList[0], ET_DEFAULT_USER_PIN, ET_USER_PIN_LEN,
ET_USER_PIN);

    if(dwRet != ET_S_SUCCESS)

    {

        printf("ETVerifyPin Error:0x%08x\n", dwRet);

    }


    //调用 ET199 内 ID 为 0x1006 的可执行文件

    //传入 2 个 short 类型数据，共 4 个字节

    //可执行文件将输入的数据进行处理，返回一个 long 类型，4 个字节

    dwRet = ETExecute(&pETContextList[0], "1006", pbInput, 4, pbOutput, 4, &dwOut);

    if(dwRet != ET_S_SUCCESS)

    {

         printf("ETExecute Error:0x%08x\n", dwRet);

    }

    //C51 返回是一个 short 类型，需要颠倒顺序
```

```
    FlipBuffer((unsigned char*)pbOutput, 2);
    printf("Result: %d\n", *pbOutput);
    ……
    //关闭锁
}
```

7.1.16 PETWriteFile

DWORD WINAPI PETWriteFile(

CONST ET_CONTEXT	**pETCtx,*
LPCSTR	*lpszFileID,*
LPCSTR	*lpszPCFilePath,*
DWORD	**pdwFileSize,*
DWORD	*dwFlags,*
BYTE	*bFileType,*
DWORD	**pdwBytesWritten)*

说明：

本接口可以直接将磁盘上的文件数据写入到 ET199 中。调用该函数前需要先打 开设备（ETOpen 或者 ETOpenEx），并验证开发商口令。

参数：

*pETCtx	[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。
lpszFileID	[in]文件 ID。见注意。
lpszPCFilePath	[in]磁盘文件路径。见注意。
*pdwFileSize	[in/out]要创建的文件大小。见注意。
dwFlags	[in]标志项。见注意。
dwFileType	[in]文件类型。
*pdwBytesWritten	[out]实际写入数据的长度。

注意：

（1）lpszFileID 参数：该参数为文件 ID，在 ET199 中，文件 ID 为 2 个字节，通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）lpszPCFilePath 参数：该参数为磁盘上文件的路径。当为 NULL 时，只在 ET199 内创建新文件。

（3）*pdwFileSize 参数：创建文件或者对 ET199 内的文件进行更新时，该参数 为 0 或者 NULL。其他情况下，为 NULL 时，不返回文件的大小。

（4）dwFlags 参数：标志项。见表 7.1.14 节中的表 7-3。

（5）bFileType 参数：文件类型。见 7.1.12 节中的表 7-2。

（6）*pdwBytesWritten 参数为 NULL 时，将不返回写入数据的长度。

（7）使用磁盘上的文件创建加密锁内文件时，如果*pdwFileSize 参数为 0 或者 NULL 时，创建的文件大小就是磁盘上文件的实际大小。但当磁盘文件过大（大于 0xFFFF=65535）时，取磁盘文件的前 0xFFFF（65535）字节写入到加密锁内。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败， 参看 7.2.1 错误编码。

举例：

```
//在ET199内创建ID为0x1008的内部数据文件

//内容为C盘根目录下text.txt文件中的内容

dwRet = PETWriteFile(&pETContextList[0], "1008", "c:\\test.txt",0,
ET_CREATE_NEW,FILE_TYPE_DATA,&dwOut);

if(dwRet != ET_S_SUCCESS)

{

    printf("PETWriteFile Error:0x%08x\n", dwRet);

}
```

7.1.17 ETGenRsaKey

```
DWORD WINAPI ETGenRsaKey(
        CONST ET_CONTEXT        *pETCtx,
        WORD                    wKeySize,
        DWORD                   dwE,
        LPCSTR                  lpszPubFileID,
        LPCSTR                  lpszPriFileID,
        OUT PVOID               pbPubKeyData,
        DWORD                   *dwPubKeyDataSize,
        PVOID                   pbPriKeyData,
        DWORD                   *dwPriKeyDataSize)
```

说明：

产生RSA密钥对。调用该函数前需要先打开设备（ETOpen或者ETOpenEx），并验证开发商口令。

参数：

*pETCtx	[in]指向ET_CONTEXT结构体的指针该指针由ETEnum函数返回。
wKeySize	[in]RSA 密钥位数。512，1024 或者 2048。
dwE	[in]RSA 密钥对的 E 值。一般为 65537。
lpszPubFileID	[in]公钥文件 ID。见注意。
lpszPriFileID	[in]私钥文件 ID。见注意。
pbPubKeyData	[out]公钥数据。
*dwPubKeyDataSize	[in/out]输入为缓冲区长度，输出为公钥数据长度。
pbPriKeyData	[out]私钥数据。
*dwPriKeyDataSize	[in/out]输入为缓冲区长度，输出为私钥数据长度。

注意：

（1）lpszPubFileID 参数和 lpszPriFileID 参数：该参数为文件 ID，在 ET199 中， 文件 ID 为 2 个字节，通过字符串的形式来输入。2 个字节表示为十六进制是 4 个字符（0-9,a-f 或者 A-F），因此第二个参数输入 4 个字符的字符串，如：“1002”表示文件 ID 为 0x1002，“00A8”表示文件 ID 为 0x00A8。另外，字符串要 4 个字符，不能省略，如“9B”，这样会造成混淆，应为“009B”，表示文件 ID 为 0x009B。ET199 中有一些保留 ID，不能使用，见附录 B。

（2）lpszPubFileID 参数和 lpszPriFileID 参数必须为当前目录下没有被使用的 ID，该函数会创建 2 个新文件。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

举例：

```
COS_RSA_PUBLIC_KEY_1024 pubKeyData;
COS_RSA_PRIVATE_KEY_1024 priKeyData;
DWORD pubSize = sizeof(COS_RSA_PUBLIC_KEY_1024);
DWORD priSize = sizeof(COS_RSA_PRIVATE_KEY_1024);
//产生 1024 位的 RSA 密钥对
dwRet = ETGenRsaKey(&pETContextList[0],1024,65537,"1003","1004",
&pubKeyData,&pubSize,&priKeyData,&priSize);
if(dwRet != ET_S_SUCCESS)
```

```
{
    printf("ETGenRsaKey Error:0x%08x\n", dwRet);
}
```

7.1.18 ETFormatErrorMessage

DWORD WINAPI ETFormatErrorMessage(

DWORD dwRet,

LPSTR lpszMessage,

DWORD dwMsgBufLen)

说明：

获取错误信息说明。

参数：

dwRet	[in]错误号。见7.2.1错误编码
lpszMessage	[out]返回错误号对应的错误信息。
dwMsgBufLen	[in]错误信息缓冲区的长度。

返回值：

返回 ET_S_SUCCESS（0x00000000）表示成功。返回其它值表示执行该函数失败，参看 7.2.1 错误编码。

7.2 错误编码

7.2.1 错误编码

```
ET_S_SUCCESS                    0x00000000    /** 操作成功*/
ET_E_KEY_REMOVED                0xF0000001    /** 设备未连接，或者被移除*/
ET_E_INVALID_PARAMETER          0xF0000002    /** 参数错误*/
ET_E_COMM_ERROR                 0xF0000003    /** 通讯错误，例如数据传输超时*/
ET_E_INSUFFICIENT_BUFFER        0xF0000004    /** 缓冲区空间不足*/
ET_E_NO_LIST                    0xF0000005    /** 没有找到设备列表*/
ET_E_DEVPIN_NOT_CHECK           0xF0000006    /** 开发商口令没有验证*/
ET_E_USERPIN_NOT_CHECK          0xF0000007    /** 用户口令没有验证*/
ET_E_RSA_FILE_FORMAT_ERROR      0xF0000008    /** RSA 文件格式错误*/
ET_E_DIR_NOT_FOUND              0xF0000009    /** 目录没有找到*/
ET_E_ACCESS_DENIED              0xF000000A    /** 访问被拒绝*/
```

ET_E_ALREADY_INITIALIZED	0xF000000B	/** 产品已经初始化*/
ET_E_INCORRECT_PIN	0xF0000C00	/** 密码不正确*/
ET_E_DF_SIZE	0xF000000D	/** 指定的目录空间大小不够*/
ET_E_FILE_EXIST	0xF000000E	/** 文件已存在*/
ET_E_UNSUPPORTED	0xF000000F	/** 功能不支持或尚未建立文件系统*/
ET_E_FILE_NOT_FOUND	0xF0000010	/** 未找到指定的文件*/
ET_E_ALREADY_OPENED	0xF0000011	/** 卡已经被打开*/
ET_E_DIRECTORY_EXIST	0xF0000012	/** 目录已存在*/
ET_E_CODE_RANGE	0xF0000013	/** 虚拟机内存地址溢出*/
ET_E_INVALID_POINTER	0xF0000014	/** 虚拟机错误的指针 */
ET_E_GENERAL_FILESYSTEM	0xF0000015	/** 常规文件系统错误 */
ET_E_OFFSET_BEYOND	0xF0000016	/** 文件偏移量超出文件大小*/
ET_E_FILE_TYPE_MISMATCH	0xF0000017	/** 文件类型不匹配*/
ET_E_PIN_BLOCKED	0xF0000018	/** PIN 码锁死*/
ET_E_INVALID_CONTEXT	0xF0000019	/** ETContext 参数错误*/
ET_E_ERROR_UNKNOWN	0xFFFFFFFF	/** 未知的错误*/
ET_E_LOAD_FILE_FAILED	0xF0001001	/** 下载文件失败*/

附录A 函数接口速查

C51语言		
C51函数	章节	功能
退出程序		
exit	6.1.1	退出程序
输入输出		
pbInBuff和wInLen	6.2.1	输入数据的内容和输入数据的长度
set_response	6.2.2	输出数据
swap	6.2.3	颠倒数据的顺序
文件操作		
create	6.3.1	创建文件
open	6.3.2	打开文件
close	6.3.3	关闭文件
read	6.3.4	读取文件
write	6.3.5	写入文件
get_file_infor	6.3.6	获取文件的属性信息
密码学算法		
des_enc	6.4.1	DES对称算法加密数据
des_dec	6.4.2	DES对称算法解密数据
tdes_enc	6.4.3	3DES对称算法加密数据
tdes_dec	6.4.4	3DES对称算法解密数据
sha1_init	6.4.5	SHA1散列初始化
sha1_update	6.4.6	SHA1散列运算
sha1_final	6.4.7	SHA1散列运算结果
md5_init	6.4.8	MD5散列初始化
md5_update	6.4.9	MD5散列运算

md5_final	6.4.10	MD5散列运算结果
rsa_enc	6.4.11	RSA公钥加密
rsa_dec	6.4.12	RSA私钥解密
rsa_gen_key	6.4.13	产生RSA密钥对
rsa_sign	6.4.14	RSA私钥进行签名
rsa_verify	6.4.15	RSA公钥进行验证签名
系统功能		
rand	6.5.1	随机数
get_version	6.5.2	硬件信息
双精度浮点数运算		
add	6.6.1	双精度浮点数加法
sub	6.6.2	双精度浮点数减法
mul	6.6.3	双精度浮点数乘法
div	6.6.4	双精度浮点数除法
atan2	6.6.5	双精度浮点数商的反正切
mod	6.6.6	双精度浮点数求模
pow	6.6.7	双精度浮点数求幂
modf	6.6.8	将一个双精度浮点数分割为整数部分和小数部分
frexp	6.6.9	把一个双精度浮点数拆成一个小数乘2的幂
ldexp	6.6.10	双精度浮点数乘2的幂
sin	6.6.11	双精度正弦
cos	6.6.12	双精度余弦
tan	6.6.13	双精度正切
asin	6.6.14	双精度反正弦
acos	6.6.15	双精度反余弦
atan	6.6.16	双精度反正切
sinh	6.6.17	双精度双曲正弦

cosh	6.6.18	双精度双曲余弦
tanh	6.6.19	双精度双曲正切
ceil	6.6.20	不小于该双精度浮点数的最小整数
floor	6.6.21	不大于该双精度浮点数的最大整数
abs	6.6.22	双精度浮点数的绝对值
exp	6.6.23	以e为底，双精度浮点数为指数的幂
log	6.2.24	双精度浮点数的自然对数
log10	6.2.25	双精度浮点数的常用对数
sqrt	6.2.26	双精度浮点数的平方根
cmp	6.2.27	比较两个双精度浮点数的大小
单精度浮点运算		
addf	6.7.1	单精度浮点数加法
subf	6.7.2	单精度浮点数减法
mulf	6.7.3	单精度浮点数乘法
divf	6.7.4	单精度浮点数除法
atan2f	6.7.5	单精度浮点数商的反正切
fmodf	6.7.6	单精度浮点数求模
powf	6.7.7	单精度浮点数求模
cmpf	6.7.8	比较两个单精度浮点数的大小
sinf	6.7.9	单精度正弦
cosf	6.7.10	单精度余弦
tanf	6.7.11	单精度正切
asinf	6.7.12	单精度反正弦
acosf	6.7.13	单精度反余弦
atanf	6.7.14	单精度反正切
sinhf	6.7.15	单精度双曲正弦
coshf	6.7.16	单精度双曲余弦

tanhf	6.7.17	单精度双曲正切
ceilf	6.7.18	不小于该单精度浮点数的最小整数
floorf	6.7.19	不大于该单精度浮点数的最大整数
absf	6.7.20	单精度浮点数的绝对值
expf	6.7.21	求以e为底，单精度浮点数为指数的幂
logf	6.7.22	求单精度浮点数的自然对数
log10f	6.7.23	求单精度浮点数的常用对数
sqrtf	6.7.24	求单精度浮点数的平方根
类型转换		
dbltof	6.8.1	双精度浮点数转换为单精度浮点数
ftodbl	6.8.2	单精度浮点数转换为双精度浮点数
dbltol	6.8.3	双精度浮点数转换为32位有符号整数
ltodbl	6.8.4	32位有符号整数转换为双精度浮点数
dtof	6.8.5	双精度浮点数转换为单精度浮点数
ftod	6.8.6	单精度浮点数转换为双精度浮点数
dtol	6.8.7	双精度浮点数转换为32位有符号整数
ltod	6.8.8	32位有符号整数转换为双精度浮点数
API接口		
API函数	章节	功能
ETEnum	7.1.1	枚举插在计算机上所有能正常打开的ET199设备
ETOpen	7.1.2	打开ET199设备
ETOpenEx	7.1.3	按设定模式打开ET199设备
ETClose	7.1.4	关闭ET199设备
ETControl	7.1.5	功能函数
ETCreateDir	7.1.6	创建目录
ETCreateDirEx	7.1.7	创建目录
ETChangeDir	7.1.8	改变当前工作目录

ETEraseDir	7.1.9	清除当前工作目录下所有内容
ETVerifyPin	7.1.10	验证当前工作目录的口令
ETChangePin	7.1.11	修改当前工作目录的口令
ETCreateFile	7.1.12	创建文件
ETWriteFile	7.1.13	写入文件
ETWriteFileEx	7.1.14	写入文件
ETExecute	7.1.15	执行可执行文件
PETWriteFile	7.1.16	磁盘文件写入ET199中的文件
ETGenRsaKey	7.1.17	产生RSA密钥对
ETFormatErrorMessage	7.1.18	获取错误信息说明

附录B ET199保留ID

ET199 中保留了一些 ID，这些 ID 不能分配给文件和目录。 保留 ID 为：0x0000，0x3F00，0x3F01

图目录

表格目录