档案：APPNOTE。TXT-。ZIP文件格式规范

版本：6.3.9

状态：最终版本-取代版本6.3.8

修订日期：2020年7月15日

版权所有（c）1989-2014、2018、2019、2020 PKWARE Inc.，保留所有权利。

1.0导言

---------------

1.1目的

-----------

1.1.1本规范旨在定义跨平台，

可互操作的文件存储和传输格式。自从

PKWARE，Inc.（“PKWARE”）于1989年首次出版，至今仍在使用

致力于确保系统的互操作性。压缩文件

通过定期发布和维护本文件

规格我们相信这一切。ZIP兼容的供应商和

使用这种格式并从中受益的应用程序开发人员

将分享并支持对互操作性的承诺。

1.2范围

---------

1.2.1 ZIP是使用最广泛的压缩文件格式之一。它是

通用于将文件聚合、压缩和加密到单个

可互操作的容器。没有特定的用途或应用需求

由该格式定义，不提供具体的实施指南

假如本文档提供了有关存储格式的详细信息

创建ZIP文件。记录和文件中提供了相关信息

描述ZIP文件是什么的字段。

1.3商标

--------------

1.3.1已注册PKWARE、PKZIP、Smartcrypt、SecureZIP和PKSFX

PKWARE，Inc.在美国和其他地方的商标。

PKPatchMaker、Deflate64和ZIP64是PKWARE，Inc.的商标。

本文件中引用的其他标记用于识别

仅用于其各自所有者的目的和财产。

1.4允许使用

-----------------

1.4.1本文件“APPNOTE.TXT-.ZIP文件格式规范”是

PKWARE的专有财产。使用本手册中包含的信息

文件仅允许用于创建产品，

以ZIP格式读取和写入文件的程序和进程

根据本协议的条款和条件。

1.4.2禁止在其他出版物中使用本文件的内容

仅允许通过参考本文件。任何复制品

或在未经事先通知的情况下全部或部分分发本文件

严禁获得PKWARE的书面许可。

1.4.3本文件中提供的某些技术组件为

PKWARE的专利专有技术，因此需要

与PKWARE单独签署的许可协议。可应用的

部件标有以下或类似的说明：

参考本文件中题为“合并”的章节

PKWARE专有技术进入您的产品“获取更多信息”。

1.5联系PKWARE

---------------------

1.5.1如果您对该格式、其使用或许可有疑问，或者

如需报告缺陷、请求更改或添加，请联系：

PKWARE公司。

匹兹堡大道东201号400室

威斯康星州密尔沃基53204

+1-414-289-9788

+1-414-289-9789传真

     zipformat@pkware.com

1.5.2关于本格式的信息和本文件的参考副本

可在以下网址公开获取：

     http://www.pkware.com/appnote

1.6免责声明

--------------

1.6.1尽管PKWARE将尝试提供电流和准确的

与文件格式、算法和

主题程序中，错误或遗漏的可能性不可忽略

被淘汰。因此，PKWARE明确否认任何担保

相关材料中包含的信息

主题程序和/或所创建文件的格式或

由主题程序和/或所用算法访问

主题程序或任何其他事项是最新的、正确的或有效的

交货时准确无误。任何可能导致损坏的风险

信息使用者假定信息不准确。

此外，与主题项目相关的信息

和/或受试者创建或访问的文件格式

主题程序使用的程序和/或算法是

如有变更，恕不另行通知。

2.0修订版

--------------

2.1文件状态

--------------------

2.1.1如果本文件的状态标记为草稿，则内容

定义了本规范的拟议修订，可能包括：

ZIP格式本身的更改，或者可能包含其他更改

此文档的内容更改。本文件的版本和

草案格式可能会在修改前进行修改

最终版本的发布状态。草案定期出版

向ZIP社区通知待定的更改，并

提供审查和评论的机会。

2.1.2本文件的最终版本为

视为该版本文件的最终形式

在新的更高版本发布之前，不会进行进一步更改

编号文件已发布。这种格式的更新版本

该规范旨在与所有先前版本保持互操作性

技术上可能的版本。

2.2更改日志

--------------

版本更改说明日期

------- ------------------ ----------

5.2-单密码对称加密2003年7月16日

存储

6.1.0-智能卡兼容性2004年1月20日

-关于证书存储的文档

6.2.0-中央目录介绍2004年4月26日

加密元数据的加密

-将OS X添加到由值生成的版本中

新增字段：2005年6月1日

使用ID 0x4690的POSZIP

-澄清了上的大小字段

“zip64中央目录记录的结尾”

6.2.2-最终功能规范文件化01/06/2006

用于强加密

-澄清和排版

更正

6.3.0-添加磁带定位存储2006年9月29日

参数

-支持的哈希算法的扩展列表

-支持压缩的扩展列表

算法

-支持的加密扩展列表

算法

-增加了Unicode文件名选项

存储

-一致使用的澄清

数据描述符记录的定义

-增加了额外的“额外字段”

定义

6.3.1-2007年11月4日修正的标准哈希值

SHA-256/384/512

6.3.2-新增压缩方法97 09/28/2007

-有文档记录的InfoZIP“额外字段”

UTF-8文件名和

文件注释存储

6.3.3-格式更改以支持2012年1月9日

更容易引用此APPNOTE

来自其他文件和标准

6.3.4-地址变更2014年1月10日

6.3.5-文件化压缩方法16 2018年11月31日

和99（4.4.5、4.6.1、5.11、5.17、，

附录E）

-更正了几处排版

错误（2.1.2、3.2、4.1.1、10.2）

-将遗留算法标记为否

更长时间适合使用（4.4.5.1）

-增加了MS DOS时间格式的清晰度

(4.4.6)

-为时间戳分配外部字段ID

(4.5.2)

-字段代码说明更正（A.2）

-更加一致地使用MAY/SHOULD/MUST

-扩展的0x0065记录属性代码（B.2）

-0x0022额外数据的初始信息

6.3.6-更正印刷错误2019年4月26日

(4.4.1.3)

6.3.7-增加了Z标准压缩方法ID

(4.4.5)

-更正了几个报告的打字错误

-标记通用位14的预期用途

-添加了数据流对齐额外数据信息

(4.6.11)

6.3.8-解决了Z标准压缩方法ID冲突

(4.4.5)

-添加了正在使用的其他压缩方法ID值

6.3.9-纠正了数据流对齐描述中的错误

(4.6.11)

3.0符号

-------------

3.1术语的使用必须或应表明所需的元素。

3.2不得或不得表明禁止使用元件。

3.3应表示推荐的元素。

3.4不应表示不建议使用的元件。

3.5可能表示可选元素。

4.0压缩文件

-------------

4.1什么是ZIP文件

----------------------

4.1.1 ZIP文件可通过标准识别。ZIP文件扩展名

尽管不需要使用文件扩展名。使用

扩大ZIPX也可以识别，可以用于ZIP文件。

使用ZIP格式的其他常见文件扩展名包括。罐子战争

.DOCX。XLSX。PPTX。ODT。ODS。ODP等。程序阅读或

编写ZIP文件应依赖所述的内部记录签名

在本文档中，以标识此格式的文件。

4.1.2 ZIP文件应至少包含一个文件，并且可能包含

多个文件。

4.1.3数据压缩可用于减小文件大小

放在ZIP文件中，但不是必需的。此格式支持

使用多种数据压缩算法。使用压缩时，

必须使用记录在案的压缩算法之一。实施者

被建议用他们的数据进行实验，以确定

可用的算法可根据需要提供最佳压缩。

压缩方法8（Deflate）是大多数人默认使用的方法

ZIP兼容的应用程序。

4.1.4数据加密可用于保护ZIP文件中的文件。

此格式中支持的加密密钥方法包括

密码和公钥/私钥。两者都可以单独使用

或者两者结合。加密可以应用于单个文件。

通过对ZIP文件进行加密，可以使用额外的安全性

存储在中央目录中的元数据。请参阅

信息的强加密规范。请参阅本节

本文件标题为“合并PKWARE专有技术”

进入你的产品”以获取更多信息。

4.1.5必须使用CRC32为每个文件提供数据完整性。

4.1.6通过使用

数字签名。单个文件可以用一个或多个签名

数字签名。如果必须使用已签名的目录

单一签名。

4.1.7文件可以放在未压缩或存储的ZIP文件中。

本文件中使用的术语“存储”指文件

被复制到未压缩的ZIP文件中。

4.1.8放入ZIP文件的每个数据文件都可以被压缩、存储，

加密或数字签名，与数据库中的其他数据文件无关

相同的ZIP文件被存档。

4.1.9 ZIP文件可以流式传输，分为段（固定或固定）

可移动媒体）或“自解压”。自解压拉链

文件中必须包含目标平台的提取代码

压缩文件。

4.1.10针对特定平台或应用程序提供扩展性

需要通过额外的数据字段，这些字段可能是为自定义

目的。额外数据定义不得与现有数据冲突

记录定义。

4.1.11 ZIP的常见用途还包括使用清单文件。

清单文件在存储的文件中存储特定于应用程序的信息

在ZIP文件中。此清单文件应该是列表中的第一个文件

压缩文件。本规范不提供有关以下方面的任何信息或指导：

在ZIP文件中使用清单文件。请咨询应用程序开发人员

有关使用清单文件和任何其他配置文件的信息

有关在应用程序中使用ZIP的信息。

4.1.12压缩文件可以放在其他压缩文件中。

4.2压缩元数据

----------------

4.2.1 ZIP文件由定义的记录类型组成的元数据标识

包含维护文件所需的存储信息

放进一个ZIP文件中。每个记录类型都必须使用标题进行标识

标识记录类型的签名。签名值以

0x4b50的两字节常量标记，表示字符“PK”。

4.3文件的一般格式。压缩文件

---------------------------------

4.3.1 ZIP文件必须包含“中央目录记录的结尾”。拉链

仅包含“中央目录结束记录”的文件被视为

空的压缩文件。文件可以在ZIP文件中添加或替换，也可以删除。

ZIP文件必须只有一个“中央目录记录的结尾”。另外

本规范中定义的记录可根据需要用于支持

单个ZIP文件的存储要求。

4.3.2放入ZIP文件的每个文件前面必须有一个“本地”

该文件的“文件头”记录。每个“本地文件头”必须

随附相应的“中央目录头”记录

ZIP文件的中心目录部分。

4.3.3文件可以在ZIP文件中以任意顺序存储。拉链

文件可以跨越多个卷，也可以拆分为用户定义的卷

片段大小。所有值必须以小字节顺序存储，除非

本文件中对特定数据元素另有规定。

4.3.4压缩不得应用于“本地文件头”，即“加密”

标题”，或“中央目录记录的结尾”。单个“中央目录”

“目录记录”不能被压缩，而是所有中心记录的集合

目录记录可能会被压缩。

4.3.5文件数据后面可能有一个文件的“数据描述符”。数据

描述符用于促进ZIP文件流。

4.3.6总的来说。ZIP文件格式：

[本地文件头1]

[加密头1]

[文件数据1]

[数据描述符1]

.

.

.

[本地文件头n]

[加密头n]

[文件数据n]

[数据描述符n]

[存档解密头]

[存档额外数据记录]

[中央目录头1]

.

.

.

[中央目录头n]

[中央目录P64]

[zip64中央目录定位器的结尾]

[中央目录记录结束]

4.3.7本地文件头：

本地文件头签名4字节（0x04034b50）

版本需要提取2个字节

通用位标志2字节

压缩方法2字节

上次修改文件时间2字节

上次修改文件日期2字节

crc-32 4字节

压缩大小为4字节

未压缩大小4字节

文件名长度为2字节

额外字段长度2字节

文件名（可变大小）

额外字段（可变大小）

4.3.8文件数据

紧跟在文件的本地头之后

应放置压缩或存储的文件数据。

如果文件已加密，则文件的加密头

应该放在本地头之后和文件之前

数据[local file header][encryption header]系列

[file data][data descriptor]会对中的每个文件重复执行

.压缩档案。

零字节文件、目录和其他文件类型

“不包含内容”不能包含文件数据。

4.3.9数据描述符：

crc-32 4字节

压缩大小为4字节

未压缩大小4字节

4.3.9.1如果通用代码的第3位

设置目的位标志（见下文）。它是字节对齐的

紧接着压缩数据的最后一个字节。

此描述符仅应在无法使用时使用

在输出中寻找。ZIP文件，例如，输出时。压缩文件

是标准输出还是不可查找的设备。ZIP64（tm）格式

压缩和未压缩的大小各为8字节。

4.3.9.2压缩文件时，压缩和未压缩的大小

当

文件大小超过0xFFFFFF。但是ZIP64格式可能是

无论文件大小如何使用。提取时，如果

zip64扩展信息额外字段用于

压缩和未压缩的文件大小将为8

字节值。

4.3.9.3虽然最初没有指定签名，但

0x08074b50通常被用作签名值

用于数据描述符记录。实施者应该

请注意，使用或不使用此选项都可能会遇到ZIP文件

签名标记数据描述符，并应考虑

在读取ZIP文件以确保兼容性时，这两种情况都会发生。

4.3.9.4编写ZIP文件时，实施者应包括

标记数据描述符记录的签名值。什么时候

将使用当前定义的签名字段

数据描述符记录将立即跟随

签名

4.3.9.5可扩展数据描述符将在

此APPNOTE的未来版本。这项新记录旨在

解决与今后使用此记录的冲突，

并为流式文件处理提供更好的支持。

4.3.9.6当使用中央目录加密方法时，

数据描述符记录不是必需的，但可以使用。

如果存在，则通用位字段的位3设置为

指示其存在，数据描述符字段中的值

记录必须设置为二进制零。请参阅关于强者的部分

信息加密规范。请参阅中的章节

本文件标题为“合并PKWARE专有技术”

进入你的产品”以获取更多信息。

4.3.10存档解密头：

4.3.10.1 6.2版中引入了存档解密头

ZIP格式规范的一部分。此记录存在于支持中

作为的一部分实现的中央目录加密功能的

本文档中描述的强加密规范。

当中央目录结构被加密时，这种解密

标头必须位于加密数据段之前。

4.3.10.2加密数据段应包括档案

额外数据记录（如果存在）和加密的中央目录

结构数据。此数据记录的格式与

压缩文件数据之前的解密头记录。如果

中央目录结构是加密的，起始位置是

此数据记录是使用中心目录的开始来确定的

中央目录记录Zip64结尾的字段。看到了吗

关于信息的强加密规范部分

在存档解密标头记录中使用的字段上。

请参阅本文件中题为“合并”的章节

PKWARE专有技术进入您的产品”以获取更多信息。

4.3.11归档额外数据记录：

存档额外数据签名4字节（0x08064b50）

额外字段长度4字节

额外字段数据（可变大小）

4.3.11.1版本6.2中引入了归档额外数据记录

ZIP格式规范的一部分。此记录可用于支持

作为的一部分实现的中央目录加密功能的

本文档中描述的强加密规范。

如果存在，该记录必须立即位于中央记录之前

目录数据结构。

4.3.11.2该数据记录的大小应包括在

中心目录末尾的中心目录字段的大小

目录记录。如果中央目录结构被压缩，

但未加密，此数据记录的起始位置为

使用Zip64中的中心目录起始字段确定

中央目录记录结束。请参阅本文档中的章节

题为“将PKWARE专有技术融入您的

有关更多信息，请参阅“产品”。

4.3.12中央目录结构：

[中央目录头1]

.

.

.

[中央目录头n]

[数字签名]

文件头：

中心文件头签名4字节（0x02014b50）

版本由2个字节组成

版本需要提取2个字节

通用位标志2字节

压缩方法2字节

上次修改文件时间2字节

上次修改文件日期2字节

crc-32 4字节

压缩大小为4字节

未压缩大小4字节

文件名长度为2字节

额外字段长度2字节

文件注释长度为2字节

磁盘号开始2字节

内部文件属性2字节

外部文件属性4字节

本地头4字节的相对偏移量

文件名（可变大小）

额外字段（可变大小）

文件注释（可变大小）

4.3.13数字签名：

标题签名4字节（0x05054b50）

数据大小为2字节

签名数据（可变大小）

随着中央目录加密的引入

在本规范的6.2版中，中央

目录结构可以被压缩和加密存储。

虽然不是必需的，但在加密

中央目录结构，它将被压缩

以提高存储效率。关于

中央目录加密功能可在部分中找到

描述强加密规范。数字世界

签名记录将不会被压缩或加密。

4.3.14 Zip64中央目录记录的结尾

zip64中央方向的末端

签名4字节（0x06064b50）

zip64中心端的尺寸

目录记录8字节

版本由2个字节组成

版本需要提取2个字节

此磁盘的数量为4字节

带有

中央目录的开头为4字节

列表中的条目总数

此磁盘上的中央目录8字节

列表中的条目总数

中央目录8字节

中央目录的大小为8字节

中环起点偏移量

关于

起始磁盘号为8字节

zip64可扩展数据扇区（可变大小）

4.3.14.1存储在“zip64中央终端尺寸”中的值

目录记录”的大小应为剩余的

记录，不应包括前导的12个字节。

大小=SizeOfficedFields+SizeOfVariableData-12。

4.3.14.2上述记录结构定义了

zip64中央目录记录的结尾。第1版是

在本规范之前的版本中实现

6.2支持ZIP64大文件功能。这个

介绍中央目录加密功能

作为强加密的一部分在6.2版中实现

规范定义了该记录结构的版本2。

请参阅描述强加密的部分

第2版格式的详细说明

这张唱片。请参阅本文件中题为

“将PKWARE专有技术融入您的产品”

有关适用于使用此版本2的更多信息

记录

4.3.14.3特殊用途数据可存放在zip64中

此版本的V1或V2之后的数据扇区字段

记录确保识别该特殊用途数据

它必须包括一个识别头块，由

以下内容：

标题ID-2字节

数据大小-4字节

Header ID字段指示数据中的数据类型

后面的数据块。

Data Size（数据大小）标识此操作后面的字节数

数据块类型。

4.3.14.4可能存在多个专用数据块。

每个字段前面必须有标题ID和数据大小字段。现在的

此字段中支持的头ID值的映射如下

定义见附录C。

4.3.15 Zip64中央目录结束定位器

zip64中央方向定位器末端

签名4字节（0x07064b50）

带有

zip64的开始

中央目录4字节

zip64的相对偏移量

中央目录记录8字节结束

磁盘总数4字节

4.3.16中央目录记录的结尾：

中央目录签名结束4字节（0x06054b50）

此磁盘的数量为2字节

带有

中央目录的开头为2字节

列表中的条目总数

此磁盘上的中央目录2字节

中的条目总数

中央目录为2字节

中央目录的大小为4字节

中环起点偏移量

关于

起始磁盘号为4字节

.ZIP文件注释长度为2字节

.ZIP文件注释（可变大小）

4.4字段说明

--------------------------

4.4.1关于字段的一般说明

4.4.1.1除非另有说明，否则所有字段均未签名并存储

在英特尔中，低字节：高字节，低字：高字顺序。

4.4.1.2字符串字段不以null结尾，因为长度

是明确给出的。

4.4.1.3中央目录中的条目不一定

以相同的顺序出现的文件。压缩文件。

4.4.1.4如果中央目录末尾的某个字段

记录太小，无法保存所需数据，该字段应为

设置为-1（0xFFFF或0xFFFFFF）和ZIP64格式记录

应该创建。

4.4.1.5中央目录记录的结尾和Zip64结尾

中央目录定位器记录的位置必须位于同一位置

在分割或跨越存档时使用磁盘。

4.4.2版本制作人（2字节）

4.4.2.1高位字节表示文件的兼容性

属性信息。如果外部文件属性

与MS-DOS兼容，可由PKZIP读取

DOS版本2.04g则该值为零。如果这些

属性不兼容，则此值将

确定属性所在的主机系统

可共用的软件可以使用这些信息来确定

文本文件等的行记录格式。

4.4.2.2当前映射为：

0-MS-DOS和OS/2（FAT/VFAT/FAT32文件系统）

1-Amiga 2-OpenVMS

3-UNIX 4-VM/CMS

5-Atari ST 6-OS/2 H.P.F.S。

系统7-Macintosh

9-CP/M 10-Windows NTFS

11-MVS（OS/390-Z/OS）12-VSE

13-Acorn Risc 14-VFAT

15-备用MVS 16-BeOS

17-串联18-OS/400

19-OS X（达尔文）20-255-未使用

4.4.2.3下字节表示ZIP规范版本

（本文档的版本）由软件支持

用于对文件进行编码。值/10表示主修科目

版本号，值mod 10为次要版本

数字

4.4.3需要提取的版本（2字节）

4.4.3.1所需的最低支持ZIP规范版本

要提取文件，请如上所述进行映射。该值基于

ZIP程序必须支持的特定格式功能

能够提取文件。如果需要多个功能

应用于文件时，最低版本必须设置为

具有最高价值的特征。新特性或新特性

影响已发布格式规范的更改将被删除

使用比上一个版本号更高的版本号实现

发布值以避免冲突。

4.4.3.2当前的最低功能版本定义如下：

1.0-默认值

1.1-文件是卷标

2.0-文件是一个文件夹（目录）

2.0-使用Deflate压缩对文件进行压缩

2.0-使用传统PKWARE加密对文件进行加密

2.1-使用Deflate64（tm）压缩文件

2.5-使用PKWARE DCL内爆压缩文件

2.7-文件是补丁数据集

4.5-文件使用ZIP64格式扩展名

4.6-使用BZIP2压缩对文件进行压缩\*

5.0-文件使用DES加密

5.0-文件使用3DES加密

5.0-使用原始RC2加密对文件进行加密

5.0-使用RC4加密对文件进行加密

5.1-使用AES加密对文件进行加密

5.1-使用正确的RC2加密对文件进行加密\*\*

5.2-使用更正的RC2-64加密对文件进行加密\*\*

6.1-使用非OAEP密钥包装对文件进行加密\*\*\*

6.2-中央目录加密

6.3-使用LZMA压缩文件

6.3-使用PPMd压缩文件+

6.3-使用Blowfish对文件进行加密

6.3-使用Twofish对文件进行加密

4.4.3.3需要提取的版本说明

\*7月初。PKZIP的x（7.2之前）版本错误地设置了

BZIP2压缩为50需要提取的版本

本来应该是46岁。

\*\*请参阅强加密规范一节

有关RC2修正的更多信息。

\*\*\*使用非OAEP密钥包装的证书加密是

从6.1开始的所有版本的预期操作模式。

对OAEP密钥包装的支持只能用于

发送要由打开的ZIP文件时的向后兼容性

PKZIP的版本早于6.1（5.0或6.0）。

+使用PPMd压缩的文件必须设置版本

但是，需要将字段提取到6.3，而不是全部压缩

程序强制执行此操作，可能无法解压缩

如果设置了此值，则使用PPMd压缩的数据文件。

当使用ZIP64扩展时

还必须设置zip64中央目录结束记录。

应适当设置此字段，以指示

正在使用版本1或版本2格式。

4.4.4通用位标志：（2字节）

位0：如果设置，则表示文件已加密。

（对于方法6-内爆）

第1位：如果使用的压缩方法是6型，

内爆，则此位（如果设置）表示

使用8K滑动字典。如果清楚的话，

然后使用4K滑动字典。

第2位：如果使用的压缩方法是6型，

内爆，则此位（如果设置）表示

3棵香农-法诺树被用来编码

滑动字典输出。如果清除，则为2

使用香农法诺树。

（对于方法8和9-放气）

第2位第1位

使用了0正常（-en）压缩选项。

0 1使用了最大（-exx/-ex）压缩选项。

1使用了10快速（-ef）压缩选项。

1使用了1个超快（-es）压缩选项。

（对于方法14-LZMA）

第1位：如果使用的压缩方法是14型，

LZMA，那么这个位，如果设置，表示

流结束（EOS）标记用于

标记压缩数据流的结束。

如果清除，则不存在EOS标记

压缩数据的大小必须是已知的

提取。

注：如果压缩

方法是任何其他方法。

第3位：如果设置了该位，则字段crc-32将被压缩

“大小”和“未压缩大小”在中设置为零

本地标题。将正确的值放入

紧跟在压缩后的数据描述符

数据（注：PKZIP版本2.04g仅适用于DOS）

识别此位用于方法8压缩，更新

PKZIP的版本可以识别任何

压缩方法。）

第4位：保留用于方法8，用于增强

泄气。

第5位：如果设置了该位，则表示文件已被删除

压缩补丁数据。（注意：需要PKZIP）

2.70或更高版本）

第6位：强加密。如果设置了此位，则必须

将提取值所需的版本设置为至少

50，还必须设置位0。如果AES加密

使用时，提取值所需的版本必须

至少51岁。请参见描述强项的部分

有关详细信息，请参阅加密规范。参考

本文件中题为“合并PKWARE”的章节

将专有技术应用到您的产品中“获取更多信息”

信息

第7位：当前未使用。

第8位：当前未使用。

第9位：当前未使用。

第10位：当前未使用。

第11位：语言编码标志（EFS）。如果设置了这个位，

此文件的文件名和注释字段

必须使用UTF-8编码。（见附录D）

第12位：PKWARE保留用于增强压缩。

位13：加密中央目录时设置为

本地标头中的选定数据值被屏蔽为

隐藏它们的实际值。请参见描述

有关详细信息，请参阅强加密规范。参考

本文件中题为“合并”的章节

将PKWARE专有技术应用到您的产品中

更多信息。

位14：PKWARE为备用流保留。

第15位：由PKWARE保留。

4.4.5压缩方法：（2字节）

0-文件已存储（无压缩）

1-文件已缩小

2-文件压缩系数为1

3-文件压缩系数为2

4-文件压缩系数为3

5-文件压缩系数为4

6-文件内爆

7-保留用于标记化压缩算法

8-文件被压缩

9-使用Deflate64（tm）增强的放气功能

10-PKWARE数据压缩库内爆（旧IBM简写版）

11-由PKWARE保留

12.使用BZIP2算法压缩文件

13-由PKWARE保留

14-LZMA

15-由PKWARE保留

16-IBM z/OS CMPSC压缩

17-由PKWARE保留

18-使用IBM TERSE（新）压缩文件

19-IBM LZ77 z体系结构

20-已弃用（对zstd使用方法93）

93-标准（zstd）压缩

94-MP3压缩

95-XZ压缩

96-JPEG变体

97-WavPack压缩数据

98-PPMd第一版，第1版

99-AE-x加密标记（见附录E）

4.4.5.1方法1-6是传统算法，不再适用

建议在压缩文件时使用。

4.4.6日期和时间字段：（每个2字节）

日期和时间以标准MS-DOS格式编码。

如果输入来自标准输入，则日期和时间为

为该数据启动压缩的时间。

如果加密中央目录和通用位

设置标志13以指示掩蔽，即存储在

本地标头将为零。MS-DOS时间格式不同

来自更常用的计算机时间格式，例如

UTC。例如，MS-DOS使用相对于1980年的年份值

2秒精度。

4.4.7 CRC-32:（4字节）

CRC-32算法是由

大卫·施瓦德勒（David Schwaderer）和他的优秀作品

《C程序员NetBIOS指南》一书由

Howard W.Sams&Co.Inc.的“神奇数字”

CRC是0xdebb20e3。正确的CRC前置和后置

使用条件化，这意味着CRC寄存器

已预先设定所有值（起始值

0xffffffff）的值，该值由

取CRC残差的1的补码。

如果设置了通用标志的第3位，则

字段在本地标头中设置为零，并且

值被放入数据描述符和中央

目录加密中央目录时，如果

本地标头不是ZIP64格式和通用

位标志13被设置为指示掩蔽，即存储的值

本地头中的值将为零。

4.4.8压缩大小：（4字节）

4.4.9未压缩大小：（4字节）

压缩（4.4.8）和未压缩文件的大小，

分别为（4.4.9）。当存在解密头时

将数据和值放在文件的前面

压缩文件大小将包括解密的字节数

头球。如果设置了通用位标志的位3，

这些字段在本地标头和

将正确的值放入数据描述符中，然后

在中央目录中。如果档案是ZIP64格式的

该字段中的值为0xFFFFFF，大小为

在相应的8字节ZIP64扩展信息中

额外的领域。加密中央目录时，如果

本地标头不是ZIP64格式和通用位

设置标志13以指示掩蔽，即为

本地标头中的未压缩大小将为零。

4.4.10文件名长度：（2字节）

4.4.11额外字段长度：（2字节）

4.4.12文件注释长度：（2字节）

文件名、额外字段和注释的长度

字段。任意长度的总和

目录记录和这三个字段不应

通常超过65535字节。如果输入来自标准

输入时，文件名长度设置为零。

4.4.13磁盘编号开始：（2字节）

此文件开始的磁盘的编号。如果

存档为ZIP64格式，此字段中的值为

0xFFFF，大小将在相应的4字节zip64中

扩展信息额外字段。

4.4.14内部文件属性：（2字节）

位1和2保留供PKWARE使用。

4.4.14.1如果设置，该字段的最低位表示：，

文件显然是ASCII或文本文件。如果不是

设置，该文件显然包含二进制数据。

剩余的位在1.0版中未使用。

4.4.14.2如果设置，该字段的0x0002位表示

每个字段前面都有一个4字节的可变记录长度控制字段

指示记录长度的逻辑记录。这个

记录长度控制字段存储在小端字节中

顺序此标志独立于文本控制字符，

如果与文本数据一起使用，则包括

记录总长度中的控制字符。这

为大型机数据传输支持提供了价值。

4.4.15外部文件属性：（4字节）

外部属性的映射是

取决于主机系统（请参阅“版本由”部分）。对于

MS-DOS，低位字节是MS-DOS目录

属性字节。如果输入来自标准输入，则

字段设置为零。

4.4.16本地头的相对偏移量：（4字节）

这是从磁盘上第一个磁盘开始的偏移量

该文件显示在哪个位置，本地头应该显示在哪个位置

找到了。如果存档为ZIP64格式，则

此字段为0xFFFFFF，大小将以

对应的8字节zip64扩展信息额外字段。

4.4.17文件名：（变量）

4.4.17.1文件名，带有可选的相对路径。

存储的路径不得包含驱动器或驱动器

设备字母，或前导斜杠。所有的斜线

必须是前斜杠“/”而不是

向后斜杠“\”与Amiga兼容

和UNIX文件系统等，如果输入来自标准

输入时，没有文件名字段。

4.4.17.2如果使用中央目录加密功能

设置通用位标志13，指示屏蔽文件

存储在本地头中的名称将不是实际的文件名。

由唯一的十六进制值组成的屏蔽值将

被储存。该值将按顺序递增

档案中的文件。请参阅强加密部分

有关检索加密文件名的详细信息的规范。

请参阅本文件中题为“合并PKWARE”的章节

“将专有技术引入您的产品”以获取更多信息。

4.4.18文件注释：（可变）

此文件的注释。

4.4.19该磁盘的数量：（2字节）

此磁盘的编号，其中包含中央磁盘

目录结束记录。如果档案是ZIP64格式的

该字段中的值为0xFFFF，大小为

位于中央处理器的相应4字节zip64端

目录字段。

4.4.20中央处理器启动时的磁盘编号

目录：（2字节）

中央处理器所在磁盘的编号

目录启动。如果档案是ZIP64格式的

该字段中的值为0xFFFF，大小为

位于中央处理器的相应4字节zip64端

目录字段。

4.4.21中央目录中的条目总数

这个磁盘：（2字节）

此磁盘上的中央目录项数。

如果存档是ZIP64格式，并且

此字段为0xFFFF，大小将以

相应的8字节zip64中央端口结束

目录字段。

4.4.22中央目录中的条目总数：（2字节）

中的文件总数。压缩文件。如果

存档为ZIP64格式，此字段中的值

是0xFFFF，大小将在相应的8字节中

zip64中央目录字段的结尾。

4.4.23中央目录的大小：（4字节）

整个中心目录的大小（字节）。

如果存档是ZIP64格式，并且

此字段为0xFFFFFF，大小将以

相应的8字节zip64中央端口结束

目录字段。

4.4.24相对于

起始磁盘号：（4字节）

上的中心目录开始的偏移量

中央目录启动的磁盘。如果

存档文件采用ZIP64格式，其中的值

字段为0xFFFFFF，大小将在

相应的8字节zip64中央端口结束

目录字段。

4.4.25 .ZIP文件注释长度：（2字节）

这篇评论的长度。压缩文件。

4.4.26 .ZIP文件注释：（变量）

对此的评论。压缩文件。ZIP文件注释数据

存储时没有安全保护。没有加密或数据认证

此时应用于该区域。机密信息

不应存储在本节中。

4.4.27 zip64可扩展数据扇区（可变大小）

（目前保留供PKWARE使用）

4.4.28额外字段：（变量）

这应该用于存储扩展。如果是额外的

信息需要存储在ZIP文件中，以便进行特殊处理

应用程序或平台需要时，应将其存储在此处。

支持本规范早期版本的程序可以

然后安全地跳过该文件，并找到下一个文件或头。

在1.0版中，此字段的长度为0。

现有的额外字段在本节中定义

下面是可扩展的数据字段。

4.5可扩展数据字段

--------------------------

4.5.1为了允许不同的程序和不同的类型

要存储在中的“额外”字段中的信息。拉链

所有文件都必须使用以下结构

在该字段中存储数据的程序：

header1+data1+header2+data2。

每个标题必须包括：

标题ID-2字节

数据大小-2字节

注意：所有字段均以英特尔低字节/高字节顺序存储。

Header ID字段指示当前存储的数据类型

下面是数据块。

0到31的标头ID保留供PKWARE使用。

其余的ID可供第三方供应商用于

专有用法。

4.5.2 PKWARE定义的当前标头ID映射为：

0x0001 Zip64扩展信息额外字段

0x0007 AV信息

0x0008保留用于扩展语言编码数据（PFS）

（见附录D）

0x0009 OS/2

0x000a NTFS

0x000c OpenVMS

0x000d UNIX

0x000e为文件流和fork描述符保留

0x000f补丁描述符

0x0014 PKCS#7存储X.509证书

0x0015 X.509的证书ID和签名

个人档案

0x0016 X.509中央目录的证书ID

0x0017强加密头

0x0018记录管理控制

0x0019 PKCS#7加密收件人证书列表

0x0020为时间戳记录保留

0x0021策略解密密钥记录

0x0022智能加密密钥提供程序记录

0x0023 Smartcrypt策略密钥数据记录

0x0065 IBM S/390（Z390）、AS/400（I400）属性

-未压缩

0x0066预留给IBM S/390（Z390）、AS/400（I400）

属性-压缩

0x4690 POSZIP 4690（保留）

4.5.3-Zip64扩展信息额外字段（0x0001）：

以下是zip64扩展的布局

信息“额外”块。如果其中一个

本地或中心目录中的偏移量字段

记录太小，无法保存所需数据，

创建Zip64扩展信息记录。

zip64中字段的顺序已扩展

信息记录是固定的，但字段必须

仅当相应的本地或中央

目录记录字段设置为0xFFFF或0xFFFFFF。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（ZIP64）0x0001此“额外”块类型的2字节标记

大小2字节此“额外”块的大小

起初的

大小8字节原始未压缩文件大小

压缩的

压缩数据大小为8字节

相对标题

偏移量8字节本地头记录的偏移量

磁盘启动

数字4字节在其上运行的磁盘的编号

这个文件开始了

本地标题中的此条目必须包含两个原始条目

和压缩文件大小字段。如果加密

中央目录和通用位的第13位

设置的标志指示屏蔽，即存储在

原始文件大小的本地头将为零。

4.5.4-OS/2额外字段（0x0009）：

以下是OS/2属性“额外”的布局

块（最新版本09/05/95）

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

对于此类型的“额外操作系统”标记“0x0002字节”

t为以下数据块设置2字节大小

BSize 4字节未压缩块大小

CType 2字节压缩类型

解压块的EACRC 4字节CRC值

（var）可变压缩块

OS/2扩展属性结构（FEA2LIST）是

压缩，然后全部存储在

结构在这个系统中，只有一个“数据块”

瓦菲尔德[]。

4.5.5-NTFS额外字段（0x000a）：

以下是NTFS属性的布局

“额外”块。（注意：此时的时间为Mtime，Atime）

Ctime值可以在任何WIN32系统上使用。）

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（NTFS）0x000a此“额外”块类型的2字节标记

t将总“额外”块的大小调整为2字节

保留4个字节以备将来使用

Tag1 2字节NTFS属性标记值#1

大小12字节属性#1的大小，以字节为单位

（var）Size1属性#1数据

.

.

.

TagN 2字节NTFS属性标记值#N

SizeN 2字节属性的大小#N，以字节为单位

（var）SizeN属性#N数据

对于NTFS，Tag1到TagN的值如下：

（目前仅为NTFS定义了一组属性）

标签大小说明

----- ---- -----------

0x0001属性#1的2字节标记

大小12字节属性#1的大小，以字节为单位

Mtime 8字节文件上次修改时间

Atime 8字节文件上次访问时间

Ctime 8字节文件创建时间

4.5.6-OpenVMS额外字段（0x000c）：

以下是OpenVMS属性的布局

“额外”块。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（VMS）0x000c此“额外”块类型的2字节标记

t将总“额外”块的大小调整为2字节

CRC 4字节块剩余部分的32位CRC

Tag1 2字节OpenVMS属性标记值#1

大小12字节属性#1的大小，以字节为单位

（var）Size1属性#1数据

.

.

.

OpenN值标记35n字节数

SizeN 2字节属性的大小#N，以字节为单位

（var）SizeN属性#N数据

OpenVMS额外字段规则：

4.5.6.1.将存在一个或多个属性，这些属性

将以上述TagX和SizeX值开头。

这些值与ATR$C\_XXXX和ATR$S\_XXXX相同

ATR中定义的常数。H在OpenVMS C下。两者都不是

这些值中的任何一个都将为零。

4.5.6.2.不执行单词对齐或填充。

4.5.6.3.表现良好的PKZIP/OpenVMS程序不应生成

具有相同TagX值的多个子块。还有，必须

在特定情况下，不能有多个0x000c类型的“额外”块

目录记录。

4.5.7-UNIX额外字段（0x000d）：

以下是UNIX“extra”块的布局。

注意：所有字段都存储在英特尔低字节/高字节中

顺序

值大小描述

----- ---- -----------

（UNIX）0x000d此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

Atime 4字节文件上次访问时间

Mtime 4字节文件上次修改时间

Uid 2字节文件用户ID

Gid 2字节文件组ID

（var）可变长度数据字段

可变长度数据字段将包含文件类型

具体数据。目前唯一允许的值是

硬文件或符号文件的原始“链接到”文件名

链接，以及的主要和次要设备节点号

字符和块设备节点。自设备节点

不能是符号链接或硬链接，只能是一组

存储可变长度的数据。链接文件将具有

存储的原始文件的名称。这个名字不是空的

结束。它的大小可以通过检查TSize来确定-

12设备条目将有八个字节存储为两个4字节

字节条目（小端格式）。第一条

将是主要设备编号，第二个是次要设备编号

设备号。

4.5.8-补丁描述符额外字段（0x000f）：

4.5.8.1以下是补丁描述符的布局

“额外”块。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（补丁）此“额外”块类型的0x000f 2字节标记

t将总“额外”块的大小调整为2字节

版本2的描述符字节数

标志4字节的动作和反应（见下文）

OldSize将要修补的文件的大小为4字节

OldCRC 4字节32位要修补的文件的CRC

NewSize结果文件的大小为4字节

NewCRC结果文件的4字节32位CRC

4.5.8.2作用和反应

位描述

---- ----------------

0用于自动检测

1.视为自我修补

2-3保留

4-5行动（见下文）

6-7保留

8-9对缺席文件的反应（见下文）

10-11对更新文件的反应（见下文）

12-13对未知文件的反应（见下文）

14-15保留

16-31保留

4.5.8.2.1措施

行动价值

------ -----

没有0

加1

删除2

补丁3

4.5.8.2.2反应

反应值

-------- -----

问0

跳过1

忽略2

失败3

4.5.8.3补丁支持由PKPatchMaker（tm）技术提供

并包含在美国专利和正在申请的专利中。使用或

在产品中实现某些技术方面的设置

在当前的APPNOTE中，包括

强加密或修补需要PKWARE的许可证。

请参阅本文件中题为“合并”的章节

将PKWARE专有技术引入您的产品”了解更多信息

信息

4.5.9-PKCS#7存储X.509证书（0x0014）：

此字段必须包含有关每个证书的信息

文件可以与签名。当中央目录加密

ZIP文件的功能已启用，此记录将显示在

存档额外数据记录，否则它将出现在

第一个中央目录记录，并将在任何

其他记录。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（存储）0x0014此“额外”块类型的2字节标记

t存储数据的大小为2字节

t数据t调整有关商店的数据

4.5.10-X.509个人文件的证书ID和签名（0x0015）：

此字段包含有关使用哪个证书的信息

PKCS#7存储用于签署特定文件。它也

包含签名数据。此字段可以显示多个

次，但每个证书只能出现一次。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（CID）此“额外”块类型的0x0015 2字节标记

t将下面的数据大小调整为2字节

TData TSize签名数据

4.5.11-X.509中央目录的证书ID和签名（0x0016）：

此字段包含有关使用哪个证书的信息

PKCS#7存储用于对中央目录结构进行签名。

当为某个应用程序启用中央目录加密功能时

ZIP文件，该记录将出现在存档额外数据记录中，

否则它将出现在第一个中央目录记录中。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（CDID）0x0016此“额外”块类型的2字节标记

t将下面的数据大小调整为2字节

TData TSize数据

4.5.12-强加密头（0x0017）：

值大小描述

----- ---- -----------

0x0017此“额外”块类型的2字节标记

t将下面的数据大小调整为2字节

格式2字节此记录的格式定义

AlgID 2字节加密算法标识符

Bitlen 2字节加密密钥的位长度

标志2字节处理标志

CertData TSize-8证书解密额外字段数据

（有关数据，请参阅说明。）

在描述

下的证书处理方法

强加密规范）

请参阅描述强加密规范的部分

详细信息。请参阅本文件中题为

“将PKWARE专有技术融入您的产品”

了解更多信息。

4.5.13-记录管理控制（0x0018）：

值大小描述

----- ---- -----------

（Rec CTL）0x0018此“额外”块类型的2字节标记

CSize额外块数据总量的2字节大小

Tag1 2字节记录控制属性1

大小12字节属性1的大小，以字节为单位

数据1大小1属性1数据

.

.

.

TagN 2字节记录控制属性N

SizeN 2字节属性N的大小，以字节为单位

DataN SizeN属性N数据

4.5.14-PKCS#7加密接收方证书列表（0x0019）：

此字段可能包含有关每个证书的信息

用于加密处理，可以用来识别谁是

允许解密加密文件。此字段只应出现

在存档额外数据记录中。此字段不是必需的，并且

仅通过保存公共文件来帮助档案修改

加密密钥数据。个人安全要求可能会决定

为了防止信息泄露，应该省略这些数据。

注意：所有字段均以英特尔低字节/高字节顺序存储。

值大小描述

----- ---- -----------

（CStore）0x0019此“额外”块类型的2字节标记

t存储数据的大小为2字节

t数据t调整有关商店的数据

TData:

值大小描述

----- ---- -----------

版本2字节格式版本号-此时必须为0x0001

CStore（var）PKCS#7数据块

请参阅描述强加密规范的部分

详细信息。请参阅本文件中题为

“将PKWARE专有技术融入您的产品”

了解更多信息。

4.5.15-MVS额外字段（0x0065）：

以下是MVS“额外”块的布局。

注意：有些字段是以大端格式存储的。

除非另有规定，所有文本均为EBCDIC格式。

值大小描述

----- ---- -----------

（MVS）0x0065此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

ID 4字节EBCDIC“Z390”0xE9F3F9F0或

目标四的“T4MV”

（var）TSize-4属性数据（见附录B）

4.5.16-OS/400额外字段（0x0065）：

下面是OS/400“额外”块的布局。

注意：有些字段是以大端格式存储的。

除非另有规定，所有文本均为EBCDIC格式。

值大小描述

----- ---- -----------

（OS400）0x0065此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

ID 4字节EBCDIC“I400”0xC9F4F0或

目标四的“T4MV”

（var）TSize-4属性数据（见附录A）

4.5.17-策略解密密钥记录额外字段（0x0021）：

以下是策略解密密钥“extra”块的布局。

TData是一个长度可变、内容可变的字段。它坚持住了

有关加密和/或加密密钥源的信息。

有关当前TData结构的信息，请联系PKWARE。

此“额外”块中的信息可能会以本地方式放置

在注释字段中。请参阅本文档中的章节

题为“将PKWARE专有技术融入您的

有关更多信息，请参阅“产品”。

值大小描述

----- ---- -----------

0x0021此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

t数据t调整有关密钥的数据

4.5.18-密钥提供程序记录额外字段（0x0022）：

以下是密钥提供程序“extra”块的布局。

TData是一个长度可变、内容可变的字段。它坚持住了

有关加密和/或加密密钥源的信息。

有关当前TData结构的信息，请联系PKWARE。

此“额外”块中的信息可能会以本地方式放置

在注释字段中。请参阅本文档中的章节

题为“将PKWARE专有技术融入您的

有关更多信息，请参阅“产品”。

值大小描述

----- ---- -----------

0x0022此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

t数据t调整有关密钥的数据

4.5.19-策略密钥数据记录额外字段（0x0023）：

以下是策略密钥数据“额外”块的布局。

TData是一个长度可变、内容可变的字段。它坚持住了

有关加密和/或加密密钥源的信息。

有关当前TData结构的信息，请联系PKWARE。

此“额外”块中的信息可能会以本地方式放置

在注释字段中。请参阅本文档中的章节

题为“将PKWARE专有技术融入您的

有关更多信息，请参阅“产品”。

值大小描述

----- ---- -----------

0x0023此“额外”块类型的2字节标记

t为以下数据块设置2字节大小

t数据t调整有关密钥的数据

4.6第三方映射

------------------------

4.6.1常用的第三方映射包括：

0x07c8 Macintosh

0x2605 ZipIt Macintosh

0x2705 ZipIt Macintosh 1.3.5+

0x2805 ZipIt Macintosh 1.3.5+

0x334d信息压缩Macintosh

0x4341橡子/SparkFS

0x4453 Windows NT安全描述符（二进制ACL）

0x4704 VM/CMS

0x470f MVS

0x4b46 FWKCS MD5（见下文）

0x4c41 OS/2访问控制列表（文本ACL）

0x4d49信息压缩OpenVMS

0x4f4c Xceed原始位置额外字段

0x5356 AOS/VS（ACL）

0x5455扩展时间戳

0x554e Xceed unicode额外字段

0x5855 Info ZIP UNIX（原版、OS/2、NT等）

0x6375信息压缩Unicode注释额外字段

0x6542 BeOS/BeBox

0x7075信息压缩Unicode路径额外字段

0x756e ASi UNIX

0x7855信息压缩UNIX（新）

0xa11e数据流对齐（Apache Commons压缩）

0xa220微软开放式包装增长提示

0xfd4a SMS/QDOS

0x9901 AE-x加密结构（见附录E）

0x9902未知

第三方定义的额外字段的详细说明

参与方映射将作为以下信息记录：

PKWARE可以使用这些数据结构。

PKWARE不保证任何已发布信息的准确性

第三方数据。

4.6.2第三方额外字段必须包括使用

本文件第节中定义的格式

标题为可扩展数据字段（第4.5节）。

数据大小字段指示以下数据的大小：

数据块。程序可以使用此值跳到

下一个头块，传递所有

不感兴趣。

注：如上所述，整个系统的大小。压缩文件

标题，包括文件名、注释和额外信息

字段大小不应超过64K。

4.6.3如果两个不同的程序适用于相同的

标题ID值，强烈建议

程序应在文件中放置至少两个字节的唯一签名

开始时的大小（最好是4字节或更大）

每个数据区。每个程序都应该验证其

除了标头ID之外，还存在唯一的签名

值是正确的，然后再假设它是一个

已知类型。

第三方映射：

4.6.4-ZipIt Macintosh额外字段（长）（0x2605）：

以下是ZipIt额外区块的布局

对于Macintosh。本地标头和中心标头版本

都是一样的。如果文件被删除，则必须存在此块

存储的MacBinary已编码，如果文件

未存储为二进制编码。

值大小描述

----- ---- -----------

（Mac2）此额外块类型的0x2605短标记

t缩小此块的总数据大小

“ZPIT”属于额外字段签名

FnLen文件名的字节长度

文件名变量完整Macintosh文件名

文件类型字节[4]四字节Mac文件类型字符串

Creator字节[4]四字节Mac Creator字符串

4.6.5-ZipIt Macintosh额外字段（文件缩写）（0x2705）：

以下是一个缩短版本的布局

用于Macintosh的ZipIt额外块（无“全名”条目）。

ZipIt 1.3.5和更新版本使用此变体来输入

没有MacBinary编码的文件（不是目录）

文件本地标头和中心标头版本相同。

值大小描述

----- ---- -----------

（Mac2b）此额外块类型的0x2705短标记

t此数据块的总数据大小（12）

“ZPIT”属于额外字段签名

文件类型字节[4]四字节Mac文件类型字符串

Creator字节[4]四字节Mac Creator字符串

fdFlags从FInfo中删除短属性。法国国旗，

可以省略

保留，可以省略0x0000

4.6.6-ZipIt Macintosh额外字段（简称目录）（0x2805）：

以下是一个缩短版本的布局

用于Macintosh的ZipIt额外块仅用于目录

条目。ZipIt 1.3.5及更新版本使用了此变体

保存一些可选的Mac特定目录信息。

本地标头和中心标头版本相同。

值大小描述

----- ---- -----------

（Mac2c）0x2805此额外块类型的短标记

t此数据块的总数据大小（12）

“ZPIT”属于额外字段签名

frFlags是来自DInfo的短属性。五月十日

省略

View beShort ZipIt View标志，可以省略

视图字段指定ZipIt内部设置，如下所示：

一些旗帜：

位0如果设置，文件夹显示为展开（打开）

在ZipIt中查看存档内容时。

保留位1-15，零；

4.6.7-FWKCS MD5额外字段（0x4b46）：

FWKCS Content\_签名系统，用于

自动识别独立于文件名的文件，

可选地添加并使用一个额外字段来支持

快速创建增强的内容签名：

标题ID=0x4b46

数据大小=0x0013

前言='M'，'D'，'5'

后跟16个字节，其中包含未压缩文件的

128\_位MD5哈希（1），低字节优先。

当FWKCS修改。要添加的ZIP文件中心目录

这个文件的额外字段还替换

该文件未压缩的中心目录项

具有测量值的文件长度。

如果需要，FWKCS提供了一个选项来剥离这个额外字段

礼物，从一个。压缩文件中心目录。加上

FWKCS保留了这个额外的字段。压缩文件真实性

验证；如果剥离这个额外字段，FWKCS

保留所有版本的AV到PKZIP版本2.04g。

FWKCS和FWKCS内容\_签名系统

弗雷德里克·W·坎特的商标。

（1） R.Rivest，RFC1321。麻省理工学院计算机实验室

科学与RSA数据安全公司，1992年4月。

陆上通信线。76-77:“MD5算法被放置在

公共领域，以供审查和作为

标准"

4.6.8-信息压缩Unicode注释额外字段（0x6375）：

将文件注释的UTF-8版本存储在

中央目录头。（最新版20070912）

值大小描述

----- ---- -----------

（UCom）0x6375此额外块类型（“uc”）的短标记

t缩小此块的总数据大小

该额外字段的1字节版本，当前为1

ComCRC32 4字节注释字段CRC32校验和

UnicodeCom变量UTF-8版本的条目注释

当前版本设置为数字1。如果有需要

要更改此字段，版本将递增。变化

可能不向后兼容，因此不应使用此额外字段

在无法识别版本时使用。

ComCRC32是文件注释的标准zip CRC32校验和

中央目录头中的字段。这是用来验证

自Unicode注释额外字段创建以来，注释字段没有更改

它被创造出来了。如果实用程序更改了文件注释，就会发生这种情况

字段，但不更新UTF-8注释额外字段。如果CRC

检查失败，应忽略此Unicode注释额外字段，并

应改为使用标题中的文件注释字段。

UnicodeCom字段是文件注释字段的UTF-8版本

在标题中。由于UnicodeCom被定义为UTF-8，因此没有UTF-8字节

使用订单标记（BOM）。此字段的长度由

从TSize中减去前面字段的大小。如果两个

文件名和注释字段是新的通用位UTF-8

标志位11（语言编码标志（EFS））可用于指示

文件名和注释中的两个字段都是UTF-8

在这种情况下，Unicode路径和Unicode注释额外字段不适用

需要而不应该创建。请注意，对于后退

兼容性，仅当本机字符集

正在压缩的路径和注释中的一个已经在UTF-8中。它是

预期相同的文件注释存储方法，或者通用

目的位11或额外字段，用于本地和中央

文件的目录头。

4.6.9-信息压缩Unicode路径额外字段（0x7075）：

将文件名字段的UTF-8版本存储在

本地头和中央目录头。（最新版20070912）

值大小描述

----- ---- -----------

（UPath）0x7075此额外块类型的短标记（“up”）

         TSize         Short       total data size for this block

         Version       1 byte      version of this extra field, currently 1

         NameCRC32     4 bytes     File Name Field CRC32 Checksum

         UnicodeName   Variable    UTF-8 version of the entry File Name

      Currently Version is set to the number 1.  If there is a need

      to change this field, the version will be incremented.  Changes

      MAY NOT be backward compatible so this extra field SHOULD NOT be

      used if the version is not recognized.

      The NameCRC32 is the standard zip CRC32 checksum of the File Name

      field in the header.  This is used to verify that the header

      File Name field has not changed since the Unicode Path extra field

      was created.  This can happen if a utility renames the File Name but

      does not update the UTF-8 path extra field.  If the CRC check fails,

      this UTF-8 Path Extra Field SHOULD be ignored and the File Name field

      in the header SHOULD be used instead.

      The UnicodeName is the UTF-8 version of the contents of the File Name

      field in the header.  As UnicodeName is defined to be UTF-8, no UTF-8

      byte order mark (BOM) is used.  The length of this field is determined

      by subtracting the size of the previous fields from TSize.  If both

      the File Name and Comment fields are UTF-8, the new General Purpose

      Bit Flag, bit 11 (Language encoding flag (EFS)), can be used to

      indicate that both the header File Name and Comment fields are UTF-8

      and, in this case, the Unicode Path and Unicode Comment extra fields

      are not needed and SHOULD NOT be created.  Note that, for backward

      compatibility, bit 11 SHOULD only be used if the native character set

      of the paths and comments being zipped up are already in UTF-8. It is

      expected that the same file name storage method, either general

      purpose bit 11 or extra fields, be used in both the Local and Central

      Directory Header for a file.

   4.6.10 -Microsoft Open Packaging Growth Hint (0xa220):

          Value         Size        Description

----- ---- -----------

          0xa220        Short       tag for this extra block type

          TSize         Short       size of Sig + PadVal + Padding

          Sig           Short       verification signature (A028)

          PadVal        Short       Initial padding value

          Padding       variable    filled with NULL characters

    4.6.11 -Data Stream Alignment (Apache Commons-Compress) (0xa11e):

       (per Zbynek Vyskovsky) Defines alignment of data stream of this

       entry within the zip archive.  Additionally, indicates whether the

       compression method should be kept when re-compressing the zip file.

       The purpose of this extra field is to align specific resources to

       word or page boundaries so they can be easily mapped into memory.

         Value         Size        Description

----- ---- -----------

         0xa11e        Short       tag for this extra block type

         TSize         Short       total data size for this block (2+padding)

         alignment     Short       required alignment and indicator

         0x00          Variable    padding

       The alignment field (lower 15 bits) defines the minimal alignment

       required by the data stream.   Bit 15 of alignment field indicates

       whether the compression method of this entry can be changed when

       recompressing the zip file.  The value 0 means the compression method

       should not be changed.  The value 1 indicates  the compression method

       may be changed. The padding field contains padding to ensure the correct

       alignment.  It can be changed at any time when the offset or required

       alignment changes. (see https://issues.apache.org/jira/browse/COMPRESS-391)

4.7 Manifest Files

------------------

    4.7.1 Applications using ZIP files MAY have a need for additional

    information that MUST be included with the files placed into

    a ZIP file. Application specific information that cannot be

    stored using the defined ZIP storage records SHOULD be stored

    using the extensible Extra Field convention defined in this

    document.  However, some applications MAY use a manifest

    file as a means for storing additional information.  One

    example is the META-INF/MANIFEST.MF file used in ZIP formatted

    files having the .JAR extension (JAR files).

    4.7.2 A manifest file is a file created for the application process

    that requires this information.  A manifest file MAY be of any

    file type required by the defining application process.  It is

    placed within the same ZIP file as files to which this information

    applies. By convention, this file is typically the first file placed

    into the ZIP file and it MAY include a defined directory path.

    4.7.3 Manifest files MAY be compressed or encrypted as needed for

    application processing of the files inside the ZIP files.

    Manifest files are outside of the scope of this specification.

5.0 Explanation of compression methods

--------------------------------------

5.1 UnShrinking - Method 1

--------------------------

    5.1.1 Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm

    with partial clearing.  The initial code size is 9 bits, and the

    maximum code size is 13 bits.  Shrinking differs from conventional

    Dynamic Ziv-Lempel-Welch implementations in several respects:

    5.1.2 The code size is controlled by the compressor, and is

    not automatically increased when codes larger than the current

    code size are created (but not necessarily used).  When

    the decompressor encounters the code sequence 256

    (decimal) followed by 1, it SHOULD increase the code size

    read from the input stream to the next bit size.  No

    blocking of the codes is performed, so the next code at

    the increased size SHOULD be read from the input stream

    immediately after where the previous code at the smaller

    bit size was read.  Again, the decompressor SHOULD NOT

    increase the code size used until the sequence 256,1 is

    encountered.

    5.1.3 When the table becomes full, total clearing is not

    performed.  Rather, when the compressor emits the code

    sequence 256,2 (decimal), the decompressor SHOULD clear

    all leaf nodes from the Ziv-Lempel tree, and continue to

    use the current code size.  The nodes that are cleared

    from the Ziv-Lempel tree are then re-used, with the lowest

    code value re-used first, and the highest code value

    re-used last.  The compressor can emit the sequence 256,2

    at any time.

5.2 Expanding - Methods 2-5

---------------------------

    5.2.1 The Reducing algorithm is actually a combination of two

    distinct algorithms.  The first algorithm compresses repeated

    byte sequences, and the second algorithm takes the compressed

    stream from the first algorithm and applies a probabilistic

    compression method.

    5.2.2 The probabilistic compression stores an array of 'follower

    sets' S(j), for j=0 to 255, corresponding to each possible

    ASCII character.  Each set contains between 0 and 32

    characters, to be denoted as S(j)[0],...,S(j)[m], where m<32.

    The sets are stored at the beginning of the data area for a

    Reduced file, in reverse order, with S(255) first, and S(0)

    last.

    5.2.3 The sets are encoded as { N(j), S(j)[0],...,S(j)[N(j)-1] },

    where N(j) is the size of set S(j).  N(j) can be 0, in which

    case the follower set for S(j) is empty.  Each N(j) value is

    encoded in 6 bits, followed by N(j) eight bit character values

    corresponding to S(j)[0] to S(j)[N(j)-1] respectively.  If

    N(j) is 0, then no values for S(j) are stored, and the value

    for N(j-1) immediately follows.

    5.2.4 Immediately after the follower sets, is the compressed data

    stream.  The compressed data stream can be interpreted for the

    probabilistic decompression as follows:

    let Last-Character <- 0.

    loop until done

        if the follower set S(Last-Character) is empty then

            read 8 bits from the input stream, and copy this

            value to the output stream.

        otherwise if the follower set S(Last-Character) is non-empty then

            read 1 bit from the input stream.

            if this bit is not zero then

                read 8 bits from the input stream, and copy this

                value to the output stream.

            otherwise if this bit is zero then

                read B(N(Last-Character)) bits from the input

                stream, and assign this value to I.

                Copy the value of S(Last-Character)[I] to the

                output stream.

        assign the last value placed on the output stream to

        Last-Character.

    end loop

    B(N(j)) is defined as the minimal number of bits required to

    encode the value N(j)-1.

    5.2.5 The decompressed stream from above can then be expanded to

    re-create the original file as follows:

    let State <- 0.

    loop until done

        read 8 bits from the input stream into C.

        case State of

            0:  if C is not equal to DLE (144 decimal) then

                   copy C to the output stream.

                 otherwise if C is equal to DLE then

                   let State <- 1.

            1:  if C is non-zero then

                   let V <- C.

                   let Len <- L(V)

                   let State <- F(Len).

                 otherwise if C is zero then

                   copy the value 144 (decimal) to the output stream.

                   let State <- 0

            2:  let Len <- Len + C

                    let State <- 3.

            3:  move backwards D(V,C) bytes in the output stream

                    (if this position is before the start of the output

                    stream, then assume that all the data before the

                    start of the output stream is filled with zeros).

                    copy Len+3 bytes from this position to the output stream.

                    let State <- 0.

          end case

    end loop

    The functions F,L, and D are dependent on the 'compression

    factor', 1 through 4, and are defined as follows:

    For compression factor 1:

        L(X) equals the lower 7 bits of X.

        F(X) equals 2 if X equals 127 otherwise F(X) equals 3.

        D(X,Y) equals the (upper 1 bit of X) \* 256 + Y + 1.

    For compression factor 2:

        L(X) equals the lower 6 bits of X.

        F(X) equals 2 if X equals 63 otherwise F(X) equals 3.

        D(X,Y) equals the (upper 2 bits of X) \* 256 + Y + 1.

    For compression factor 3:

        L(X) equals the lower 5 bits of X.

        F(X) equals 2 if X equals 31 otherwise F(X) equals 3.

        D(X,Y) equals the (upper 3 bits of X) \* 256 + Y + 1.

    For compression factor 4:

        L(X) equals the lower 4 bits of X.

        F(X) equals 2 if X equals 15 otherwise F(X) equals 3.

        D(X,Y) equals the (upper 4 bits of X) \* 256 + Y + 1.

5.3 Imploding - Method 6

------------------------

    5.3.1 The Imploding algorithm is actually a combination of two

    distinct algorithms.  The first algorithm compresses repeated byte

    sequences using a sliding dictionary.  The second algorithm is

    used to compress the encoding of the sliding dictionary output,

    using multiple Shannon-Fano trees.

    5.3.2 The Imploding algorithm can use a 4K or 8K sliding dictionary

    size. The dictionary size used can be determined by bit 1 in the

    general purpose flag word; a 0 bit indicates a 4K dictionary

    while a 1 bit indicates an 8K dictionary.

    5.3.3 The Shannon-Fano trees are stored at the start of the

    compressed file. The number of trees stored is defined by bit 2 in

    the general purpose flag word; a 0 bit indicates two trees stored,

    a 1 bit indicates three trees are stored.  If 3 trees are stored,

    the first Shannon-Fano tree represents the encoding of the

    Literal characters, the second tree represents the encoding of

    the Length information, the third represents the encoding of the

    Distance information.  When 2 Shannon-Fano trees are stored, the

    Length tree is stored first, followed by the Distance tree.

    5.3.4 The Literal Shannon-Fano tree, if present is used to represent

    the entire ASCII character set, and contains 256 values.  This

    tree is used to compress any data not compressed by the sliding

    dictionary algorithm.  When this tree is present, the Minimum

    Match Length for the sliding dictionary is 3.  If this tree is

    not present, the Minimum Match Length is 2.

    5.3.5 The Length Shannon-Fano tree is used to compress the Length

    part of the (length,distance) pairs from the sliding dictionary

    output.  The Length tree contains 64 values, ranging from the

    Minimum Match Length, to 63 plus the Minimum Match Length.

    5.3.6 The Distance Shannon-Fano tree is used to compress the Distance

    part of the (length,distance) pairs from the sliding dictionary

    output. The Distance tree contains 64 values, ranging from 0 to

    63, representing the upper 6 bits of the distance value.  The

    distance values themselves will be between 0 and the sliding

    dictionary size, either 4K or 8K.

    5.3.7 The Shannon-Fano trees themselves are stored in a compressed

    format. The first byte of the tree data represents the number of

    bytes of data representing the (compressed) Shannon-Fano tree

    minus 1.  The remaining bytes represent the Shannon-Fano tree

    data encoded as:

        High 4 bits: Number of values at this bit length + 1. (1 - 16)

        Low  4 bits: Bit Length needed to represent value + 1. (1 - 16)

    5.3.8 The Shannon-Fano codes can be constructed from the bit lengths

    using the following algorithm:

    1)  Sort the Bit Lengths in ascending order, while retaining the

        order of the original lengths stored in the file.

    2)  Generate the Shannon-Fano trees:

        Code <- 0

        CodeIncrement <- 0

        LastBitLength <- 0

        i <- number of Shannon-Fano codes - 1   (either 255 or 63)

        loop while i >= 0

            Code = Code + CodeIncrement

            if BitLength(i) <> LastBitLength then

                LastBitLength=BitLength(i)

                CodeIncrement = 1 shifted left (16 - LastBitLength)

            ShannonCode(i) = Code

            i <- i - 1

        end loop

    3)  Reverse the order of all the bits in the above ShannonCode()

        vector, so that the most significant bit becomes the least

        significant bit.  For example, the value 0x1234 (hex) would

        become 0x2C48 (hex).

    4)  Restore the order of Shannon-Fano codes as originally stored

        within the file.

    Example:

        This example will show the encoding of a Shannon-Fano tree

        of size 8.  Notice that the actual Shannon-Fano trees used

        for Imploding are either 64 or 256 entries in size.

    Example:   0x02, 0x42, 0x01, 0x13

        The first byte indicates 3 values in this table.  Decoding the

        bytes:

                0x42 = 5 codes of 3 bits long

                0x01 = 1 code  of 2 bits long

                0x13 = 2 codes of 4 bits long

        This would generate the original bit length array of:

        (3, 3, 3, 3, 3, 2, 4, 4)

        There are 8 codes in this table for the values 0 thru 7.  Using

        the algorithm to obtain the Shannon-Fano codes produces:

                                      Reversed     Order     Original

    Val  Sorted   Constructed Code      Value     Restored    Length

--- ------ ----------------- -------- -------- ------

    0:     2      1100000000000000        11       101          3

    1:     3      1010000000000000       101       001          3

    2:     3      1000000000000000       001       110          3

    3:     3      0110000000000000       110       010          3

    4:     3      0100000000000000       010       100          3

    5:     3      0010000000000000       100        11          2

    6:     4      0001000000000000      1000      1000          4

    7:     4      0000000000000000      0000      0000          4

    The values in the Val, Order Restored and Original Length columns

    now represent the Shannon-Fano encoding tree that can be used for

    decoding the Shannon-Fano encoded data.  How to parse the

    variable length Shannon-Fano values from the data stream is beyond

    the scope of this document.  (See the references listed at the end of

    this document for more information.)  However, traditional decoding

    schemes used for Huffman variable length decoding, such as the

    Greenlaw algorithm, can be successfully applied.

    5.3.9 The compressed data stream begins immediately after the

    compressed Shannon-Fano data.  The compressed data stream can be

    interpreted as follows:

    loop until done

        read 1 bit from input stream.

        if this bit is non-zero then       (encoded data is literal data)

            if Literal Shannon-Fano tree is present

                read and decode character using Literal Shannon-Fano tree.

            otherwise

                read 8 bits from input stream.

            copy character to the output stream.

        otherwise              (encoded data is sliding dictionary match)

            if 8K dictionary size

                read 7 bits for offset Distance (lower 7 bits of offset).

            otherwise

                read 6 bits for offset Distance (lower 6 bits of offset).

            using the Distance Shannon-Fano tree, read and decode the

              upper 6 bits of the Distance value.

            using the Length Shannon-Fano tree, read and decode

              the Length value.

            Length <- Length + Minimum Match Length

            if Length = 63 + Minimum Match Length

                read 8 bits from the input stream,

                add this value to Length.

            move backwards Distance+1 bytes in the output stream, and

            copy Length characters from this position to the output

            stream.  (if this position is before the start of the output

            stream, then assume that all the data before the start of

            the output stream is filled with zeros).

    end loop

5.4 Tokenizing - Method 7

-------------------------

    5.4.1 This method is not used by PKZIP.

5.5 Deflating - Method 8

------------------------

    5.5.1 The Deflate algorithm is similar to the Implode algorithm using

    a sliding dictionary of up to 32K with secondary compression

    from Huffman/Shannon-Fano codes.

    5.5.2 The compressed data is stored in blocks with a header describing

    the block and the Huffman codes used in the data block.  The header

    format is as follows:

       Bit 0: Last Block bit     This bit is set to 1 if this is the last

                                 compressed block in the data.

       Bits 1-2: Block type

          00 (0) - Block is stored - All stored data is byte aligned.

                   Skip bits until next byte, then next word = block

                   length, followed by the ones compliment of the block

                   length word. Remaining data in block is the stored

                   data.

          01 (1) - Use fixed Huffman codes for literal and distance codes.

                   Lit Code    Bits             Dist Code   Bits

--------- ---- --------- ----

0 - 143 8 0 - 31 5

144 - 255 9

256 - 279 7

280 - 287 8

                   Literal codes 286-287 and distance codes 30-31 are

                   never used but participate in the huffman construction.

          10 (2) - Dynamic Huffman codes.  (See expanding Huffman codes)

          11 (3) - Reserved - Flag a "Error in compressed data" if seen.

    5.5.3 Expanding Huffman Codes

    If the data block is stored with dynamic Huffman codes, the Huffman

    codes are sent in the following compressed format:

       5 Bits: # of Literal codes sent - 256 (256 - 286)

               All other codes are never sent.

       5 Bits: # of Dist codes - 1           (1 - 32)

       4 Bits: # of Bit Length codes - 3     (3 - 19)

    The Huffman codes are sent as bit lengths and the codes are built as

    described in the implode algorithm.  The bit lengths themselves are

    compressed with Huffman codes.  There are 19 bit length codes:

       0 - 15: Represent bit lengths of 0 - 15

           16: Copy the previous bit length 3 - 6 times.

               The next 2 bits indicate repeat length (0 = 3, ... ,3 = 6)

                  Example:  Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will

                            expand to 12 bit lengths of 8 (1 + 6 + 5)

           17: Repeat a bit length of 0 for 3 - 10 times. (3 bits of length)

           18: Repeat a bit length of 0 for 11 - 138 times (7 bits of length)

    The lengths of the bit length codes are sent packed 3 bits per value

    (0 - 7) in the following order:

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

    The Huffman codes SHOULD be built as described in the Implode algorithm

    except codes are assigned starting at the shortest bit length, i.e. the

    shortest code SHOULD be all 0's rather than all 1's.  Also, codes with

    a bit length of zero do not participate in the tree construction.  The

    codes are then used to decode the bit lengths for the literal and

    distance tables.

    The bit lengths for the literal tables are sent first with the number

    of entries sent described by the 5 bits sent earlier.  There are up

    to 286 literal characters; the first 256 represent the respective 8

    bit character, code 256 represents the End-Of-Block code, the remaining

    29 codes represent copy lengths of 3 thru 258.  There are up to 30

    distance codes representing distances from 1 thru 32k as described

    below.

                                 Length Codes

------------

          Extra             Extra              Extra              Extra

     Code Bits Length  Code Bits Lengths  Code Bits Lengths  Code Bits Length(s)

---- ---- ------ ---- ---- ------- ---- ---- ------- ---- ---- ---------

257 0 3 265 1 11,12 273 3 35-42 281 5 131-162

258 0 4 266 1 13,14 274 3 43-50 282 5 163-194

259 0 5 267 1 15,16 275 3 51-58 283 5 195-226

260 0 6 268 1 17,18 276 3 59-66 284 5 227-257

261 0 7 269 2 19-22 277 4 67-82 285 0 258

262 0 8 270 2 23-26 278 4 83-98

263 0 9 271 2 27-30 279 4 99-114

264 0 10 272 2 31-34 280 4 115-130

                                Distance Codes

--------------

          Extra           Extra             Extra               Extra

     Code Bits Dist  Code Bits  Dist   Code Bits Distance  Code Bits Distance

---- ---- ---- ---- ---- ------ ---- ---- -------- ---- ---- --------

0 0 1 8 3 17-24 16 7 257-384 24 11 4097-6144

1 0 2 9 3 25-32 17 7 385-512 25 11 6145-8192

2 0 3 10 4 33-48 18 8 513-768 26 12 8193-12288

3 0 4 11 4 49-64 19 8 769-1024 27 12 12289-16384

4 1 5,6 12 5 65-96 20 9 1025-1536 28 13 16385-24576

5 1 7,8 13 5 97-128 21 9 1537-2048 29 13 24577-32768

6 2 9-12 14 6 129-192 22 10 2049-3072

7 2 13-16 15 6 193-256 23 10 3073-4096

    5.5.4 The compressed data stream begins immediately after the

    compressed header data.  The compressed data stream can be

    interpreted as follows:

    do

       read header from input stream.

       if stored block

          skip bits until byte aligned

          read count and 1's compliment of count

          copy count bytes data block

       otherwise

          loop until end of block code sent

             decode literal character from input stream

             if literal < 256

                copy character to the output stream

             otherwise

                if literal = end of block

                   break from loop

                otherwise

                   decode distance from input stream

                   move backwards distance bytes in the output stream, and

                   copy length characters from this position to the output

                   stream.

          end loop

    while not last block

    if data descriptor exists

       skip bits until byte aligned

       read crc and sizes

    endif

5.6 Enhanced Deflating - Method 9

---------------------------------

    5.6.1 The Enhanced Deflating algorithm is similar to Deflate but uses

    a sliding dictionary of up to 64K. Deflate64(tm) is supported

    by the Deflate extractor.

5.7 BZIP2 - Method 12

---------------------

    5.7.1 BZIP2 is an open-source data compression algorithm developed by

    Julian Seward.  Information and source code for this algorithm

    can be found on the internet.

5.8 LZMA - Method 14

---------------------

    5.8.1 LZMA is a block-oriented, general purpose data compression

    algorithm developed and maintained by Igor Pavlov.  It is a derivative

    of LZ77 that utilizes Markov chains and a range coder.  Information and

    source code for this algorithm can be found on the internet.  Consult

    with the author of this algorithm for information on terms or

    restrictions on use.

    Support for LZMA within the ZIP format is defined as follows:

    5.8.2 The Compression method field within the ZIP Local and Central

    Header records will be set to the value 14 to indicate data was

    compressed using LZMA.

    5.8.3 The Version needed to extract field within the ZIP Local and

    Central Header records will be set to 6.3 to indicate the minimum

    ZIP format version supporting this feature.

    5.8.4 File data compressed using the LZMA algorithm MUST be placed

    immediately following the Local Header for the file.  If a standard

    ZIP encryption header is required, it will follow the Local Header

    and will precede the LZMA compressed file data segment.  The location

    of LZMA compressed data segment within the ZIP format will be as shown:

        [local header file 1]

        [encryption header file 1]

        [LZMA compressed data segment for file 1]

        [data descriptor 1]

        [local header file 2]

    5.8.5 The encryption header and data descriptor records MAY

    be conditionally present.  The LZMA Compressed Data Segment

    will consist of an LZMA Properties Header followed by the

    LZMA Compressed Data as shown:

        [LZMA properties header for file 1]

        [LZMA compressed data for file 1]

    5.8.6 The LZMA Compressed Data will be stored as provided by the

    LZMA compression library.  Compressed size, uncompressed size and

    other file characteristics about the file being compressed MUST be

    stored in standard ZIP storage format.

    5.8.7 The LZMA Properties Header will store specific data required

    to decompress the LZMA compressed Data.  This data is set by the

    LZMA compression engine using the function WriteCoderProperties()

    as documented within the LZMA SDK.

    5.8.8 Storage fields for the property information within the LZMA

    Properties Header are as follows:

         LZMA Version Information 2 bytes

         LZMA Properties Size 2 bytes

         LZMA Properties Data variable, defined by "LZMA Properties Size"

       5.8.8.1 LZMA Version Information - this field identifies which version

       of the LZMA SDK was used to compress a file.  The first byte will

       store the major version number of the LZMA SDK and the second

       byte will store the minor number.

       5.8.8.2 LZMA Properties Size - this field defines the size of the

       remaining property data.  Typically this size SHOULD be determined by

       the version of the SDK.  This size field is included as a convenience

       and to help avoid any ambiguity arising in the future due

       to changes in this compression algorithm.

       5.8.8.3 LZMA Property Data - this variable sized field records the

       required values for the decompressor as defined by the LZMA SDK.

       The data stored in this field SHOULD be obtained using the

       WriteCoderProperties() in the version of the SDK defined by

       the "LZMA Version Information" field.

       5.8.8.4 The layout of the "LZMA Properties Data" field is a function of

       the LZMA compression algorithm.  It is possible that this layout MAY be

       changed by the author over time.  The data layout in version 4.3 of the

       LZMA SDK defines a 5 byte array that uses 4 bytes to store the dictionary

       size in little-endian order. This is preceded by a single packed byte as

       the first element of the array that contains the following fields:

         PosStateBits

         LiteralPosStateBits

         LiteralContextBits

       Refer to the LZMA documentation for a more detailed explanation of

       these fields.

    5.8.9 Data compressed with method 14, LZMA, MAY include an end-of-stream

    (EOS) marker ending the compressed data stream.  This marker is not

    required, but its use is highly recommended to facilitate processing

    and implementers SHOULD include the EOS marker whenever possible.

    When the EOS marker is used, general purpose bit 1 MUSY be set.  If

    general purpose bit 1 is not set, the EOS marker is not present.

5.9 WavPack - Method 97

-----------------------

    5.9.1 Information describing the use of compression method 97 is

    provided by WinZIP International, LLC.  This method relies on the

    open source WavPack audio compression utility developed by David Bryant.

    Information on WavPack is available at www.wavpack.com.  Please consult

    with the author of this algorithm for information on terms and

    restrictions on use.

    5.9.2 WavPack data for a file begins immediately after the end of the

    local header data.  This data is the output from WavPack compression

    routines.  Within the ZIP file, the use of WavPack compression is

    indicated by setting the compression method field to a value of 97

    in both the local header and the central directory header.  The Version

    needed to extract and version made by fields use the same values as are

    used for data compressed using the Deflate algorithm.

    5.9.3 An implementation note for storing digital sample data when using

    WavPack compression within ZIP files is that all of the bytes of

    the sample data SHOULD be compressed.  This includes any unused

    bits up to the byte boundary.  An example is a 2 byte sample that

    uses only 12 bits for the sample data with 4 unused bits.  If only

    12 bits are passed as the sample size to the WavPack routines, the 4

    unused bits will be set to 0 on extraction regardless of their original

    state.  To avoid this, the full 16 bits of the sample data size

    SHOULD be provided.

5.10 PPMd - Method 98

---------------------

    5.10.1 PPMd is a data compression algorithm developed by Dmitry Shkarin

    which includes a carryless rangecoder developed by Dmitry Subbotin.

    This algorithm is based on predictive phrase matching on multiple

    order contexts.  Information and source code for this algorithm

    can be found on the internet. Consult with the author of this

    algorithm for information on terms or restrictions on use.

    5.10.2 Support for PPMd within the ZIP format currently is provided only

    for version I, revision 1 of the algorithm.  Storage requirements

    for using this algorithm are as follows:

    5.10.3 Parameters needed to control the algorithm are stored in the two

    bytes immediately preceding the compressed data.  These bytes are

    used to store the following fields:

    Model order - sets the maximum model order, default is 8, possible

                  values are from 2 to 16 inclusive

    Sub-allocator size - sets the size of sub-allocator in MB, default is 50,

                    possible values are from 1MB to 256MB inclusive

    Model restoration method - sets the method used to restart context

                    model at memory insufficiency, values are:

                    0 - restarts model from scratch - default

                    1 - cut off model - decreases performance by as much as 2x

                    2 - freeze context tree - not recommended

    5.10.4 An example for packing these fields into the 2 byte storage field is

    illustrated below.  These values are stored in Intel low-byte/high-byte

    order.

    wPPMd = (Model order - 1) +

            ((Sub-allocator size - 1) << 4) +

            (Model restoration method << 12)

5.11 AE-x Encryption marker - Method 99

-------------------------------------------

5.12 JPEG variant - Method 96

-------------------------------------------

5.13 PKWARE Data Compression Library Imploding -  Method 10

-----------------------------------------------------------

5.14 Reserved -  Method 11

-------------------------------------------

5.15 Reserved -  Method 13

-------------------------------------------

5.16 Reserved -  Method 15

-------------------------------------------

5.17 IBM z/OS CMPSC Compression - Method 16

-------------------------------------------

Method 16 utilizes the IBM hardware compression facility available

on most IBM mainframes.  Hardware compression can significantly

increase the speed of data compression.  This method uses a variant

of the LZ78 algorithm.  CMPSC hardware compression is performed

using the COMPRESSION CALL instruction.

ZIP archives can be created using this method only on mainframes

supporting the CP instruction.  Extraction MAY occur on any

platform supporting this compression algorithm.  Use of this

algorithm requires creation of a compression dictionary and

an expansion dictionary.  The expansion dictionary MUST be

placed into the ZIP archive for use on the system where

extraction will occur.

Additional information on this compression algorithm and dictionaries

can be found in the IBM provided document titled IBM ESA/390 Data

Compression (SA22-7208-01). Storage requirements for using CMPSC

compression are as follows.

The format for the compressed data stream placed into the ZIP

archive following the Local Header is:

    [dictionary header]

    [expansion dictionary]

    [CMPSC compressed data]

If encryption is used to encrypt a file compressed with CMPSC, these

sections MUST be encrypted as a single entity.

The format of the dictionary header is:

          Value            Size          Description

----- ---- -----------

          Version          1 byte        1

          Flags/Symsize    1 byte        Processing flags and

                                         symbol size

          DictionaryLen    4 bytes       Length of the

                                         expansion dictionary

Explanation of processing flags and symbol size:

The high 4 bits are used to store the processing flags.  The low

4 bits represent the size of a symbol, in bits (values range

from 9-13).  Flag values are defined below.

    0x80 - expansion dictionary

    0x40 - expansion dictionary is compressed using Deflate

    0x20 - Reserved

    0x10 - Reserved

5.18 Reserved -  Method 17

-------------------------------------------

5.19 IBM TERSE -  Method 18

-------------------------------------------

5.20 IBM LZ77 z Architecture -  Method 19

-----------------------------------------

6.0  Traditional PKWARE Encryption

----------------------------------

    6.0.1 The following information discusses the decryption steps

    required to support traditional PKWARE encryption.  This

    form of encryption is considered weak by today's standards

    and its use is recommended only for situations with

    low security needs or for compatibility with older .ZIP

    applications.

6.1 Traditional PKWARE Decryption

---------------------------------

    6.1.1 PKWARE is grateful to Mr. Roger Schlafly for his expert

    contribution towards the development of PKWARE's traditional

    encryption.

    6.1.2 PKZIP encrypts the compressed data stream.  Encrypted files

    MUST be decrypted before they can be extracted to their original

    form.

    6.1.3 Each encrypted file has an extra 12 bytes stored at the start

    of the data area defining the encryption header for that file.  The

    encryption header is originally set to random values, and then

    itself encrypted, using three, 32-bit keys.  The key values are

    initialized using the supplied encryption password.  After each byte

    is encrypted, the keys are then updated using pseudo-random number

    generation techniques in combination with the same CRC-32 algorithm

    used in PKZIP and described elsewhere in this document.

    6.1.4 The following are the basic steps required to decrypt a file:

    1) Initialize the three 32-bit keys with the password.

    2) Read and decrypt the 12-byte encryption header, further

       initializing the encryption keys.

    3) Read and decrypt the compressed data stream using the

       encryption keys.

    6.1.5 Initializing the encryption keys

    Key(0) <- 305419896

    Key(1) <- 591751049

    Key(2) <- 878082192

    loop for i <- 0 to length(password)-1

        update\_keys(password(i))

    end loop

    Where update\_keys() is defined as:

    update\_keys(char):

      Key(0) <- crc32(key(0),char)

      Key(1) <- Key(1) + (Key(0) & 000000ffH)

      Key(1) <- Key(1) \* 134775813 + 1

      Key(2) <- crc32(key(2),key(1) >> 24)

    end update\_keys

    Where crc32(old\_crc,char) is a routine that given a CRC value and a

    character, returns an updated CRC value after applying the CRC-32

    algorithm described elsewhere in this document.

    6.1.6 Decrypting the encryption header

    The purpose of this step is to further initialize the encryption

    keys, based on random data, to render a plaintext attack on the

    data ineffective.

    Read the 12-byte encryption header into Buffer, in locations

    Buffer(0) thru Buffer(11).

    loop for i <- 0 to 11

        C <- buffer(i) ^ decrypt\_byte()

        update\_keys(C)

        buffer(i) <- C

    end loop

    Where decrypt\_byte() is defined as:

    unsigned char decrypt\_byte()

        local unsigned short temp

        temp <- Key(2) | 2

        decrypt\_byte <- (temp \* (temp ^ 1)) >> 8

    end decrypt\_byte

    After the header is decrypted,  the last 1 or 2 bytes in Buffer

    SHOULD be the high-order word/byte of the CRC for the file being

    decrypted, stored in Intel low-byte/high-byte order.  Versions of

    PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is

    used on versions after 2.0.  This can be used to test if the password

    supplied is correct or not.

    6.1.7 Decrypting the compressed data stream

    The compressed data stream can be decrypted as follows:

    loop until done

        read a character into C

        Temp <- C ^ decrypt\_byte()

        update\_keys(temp)

        output Temp

    end loop

7.0 Strong Encryption Specification

-----------------------------------

   7.0.1 Portions of the Strong Encryption technology defined in this

   specification are covered under patents and pending patent applications.

   Refer to the section in this document entitled "Incorporating

   PKWARE Proprietary Technology into Your Product" for more information.

7.1 Strong Encryption Overview

------------------------------

   7.1.1 Version 5.x of this specification introduced support for strong

   encryption algorithms.  These algorithms can be used with either

   a password or an X.509v3 digital certificate to encrypt each file.

   This format specification supports either password or certificate

   based encryption to meet the security needs of today, to enable

   interoperability between users within both PKI and non-PKI

   environments, and to ensure interoperability between different

   computing platforms that are running a ZIP program.

   7.1.2 Password based encryption is the most common form of encryption

   people are familiar with.  However, inherent weaknesses with

   passwords (e.g. susceptibility to dictionary/brute force attack)

   as well as password management and support issues make certificate

   based encryption a more secure and scalable option.  Industry

   efforts and support are defining and moving towards more advanced

   security solutions built around X.509v3 digital certificates and

   Public Key Infrastructures(PKI) because of the greater scalability,

   administrative options, and more robust security over traditional

   password based encryption.

   7.1.3 Most standard encryption algorithms are supported with this

   specification. Reference implementations for many of these

   algorithms are available from either commercial or open source

   distributors.  Readily available cryptographic toolkits make

   implementation of the encryption features straight-forward.

   This document is not intended to provide a treatise on data

   encryption principles or theory.  Its purpose is to document the

   data structures required for implementing interoperable data

   encryption within the .ZIP format.  It is strongly recommended that

   you have a good understanding of data encryption before reading

   further.

   7.1.4 The algorithms introduced in Version 5.0 of this specification

   include:

      RC2 40 bit, 64 bit, and 128 bit

      RC4 40 bit, 64 bit, and 128 bit

      DES

      3DES 112 bit and 168 bit

   Version 5.1 adds support for the following:

      AES 128 bit, 192 bit, and 256 bit

   7.1.5 Version 6.1 introduces encryption data changes to support

   interoperability with Smartcard and USB Token certificate storage

   methods which do not support the OAEP strengthening standard.

   7.1.6 Version 6.2 introduces support for encrypting metadata by compressing

   and encrypting the central directory data structure to reduce information

   leakage.   Information leakage can occur in legacy ZIP applications

   through exposure of information about a file even though that file is

   stored encrypted.  The information exposed consists of file

   characteristics stored within the records and fields defined by this

   specification.  This includes data such as a file's name, its original

   size, timestamp and CRC32 value.

   7.1.7 Version 6.3 introduces support for encrypting data using the Blowfish

   and Twofish algorithms.  These are symmetric block ciphers developed

   by Bruce Schneier.  Blowfish supports using a variable length key from

   32 to 448 bits.  Block size is 64 bits.  Implementations SHOULD use 16

   rounds and the only mode supported within ZIP files is CBC. Twofish

   supports key sizes 128, 192 and 256 bits.  Block size is 128 bits.

   Implementations SHOULD use 16 rounds and the only mode supported within

   ZIP files is CBC.  Information and source code for both Blowfish and

   Twofish algorithms can be found on the internet.  Consult with the author

   of these algorithms for information on terms or restrictions on use.

   7.1.8 Central Directory Encryption provides greater protection against

   information leakage by encrypting the Central Directory structure and

   by masking key values that are replicated in the unencrypted Local

   Header.   ZIP compatible programs that cannot interpret an encrypted

   Central Directory structure cannot rely on the data in the corresponding

   Local Header for decompression information.

   7.1.9 Extra Field records that MAY contain information about a file that SHOULD

   not be exposed SHOULD NOT be stored in the Local Header and SHOULD only

   be written to the Central Directory where they can be encrypted.  This

   design currently does not support streaming.  Information in the End of

   Central Directory record, the Zip64 End of Central Directory Locator,

   and the Zip64 End of Central Directory records are not encrypted.  Access

   to view data on files within a ZIP file with an encrypted Central Directory

   requires the appropriate password or private key for decryption prior to

   viewing any files, or any information about the files, in the archive.

   7.1.10 Older ZIP compatible programs not familiar with the Central Directory

   Encryption feature will no longer be able to recognize the Central

   Directory and MAY assume the ZIP file is corrupt.  Programs that

   attempt streaming access using Local Headers will see invalid

   information for each file.  Central Directory Encryption need not be

   used for every ZIP file.  Its use is recommended for greater security.

   ZIP files not using Central Directory Encryption SHOULD operate as

   in the past.

   7.1.11 This strong encryption feature specification is intended to provide for

   scalable, cross-platform encryption needs ranging from simple password

   encryption to authenticated public/private key encryption.

   7.1.12 Encryption provides data confidentiality and privacy.  It is

   recommended that you combine X.509 digital signing with encryption

   to add authentication and non-repudiation.

7.2 Single Password Symmetric Encryption Method

-----------------------------------------------

   7.2.1 The Single Password Symmetric Encryption Method using strong

   encryption algorithms operates similarly to the traditional

   PKWARE encryption defined in this format.  Additional data

   structures are added to support the processing needs of the

   strong algorithms.

   The Strong Encryption data structures are:

   7.2.2 General Purpose Bits - Bits 0 and 6 of the General Purpose bit

   flag in both local and central header records.  Both bits set

   indicates strong encryption.  Bit 13, when set indicates the Central

   Directory is encrypted and that selected fields in the Local Header

   are masked to hide their actual value.

    7.2.3 Extra Field 0x0017 in central header only.

    Fields to consider in this record are:

       7.2.3.1 Format - the data format identifier for this record.  The only

       value allowed at this time is the integer value 2.

       7.2.3.2 AlgId - integer identifier of the encryption algorithm from the

       following range

                 0x6601 - DES

                 0x6602 - RC2 (version needed to extract < 5.2)

                 0x6603 - 3DES 168

                 0x6609 - 3DES 112

                 0x660E - AES 128

                 0x660F - AES 192

                 0x6610 - AES 256

                 0x6702 - RC2 (version needed to extract >= 5.2)

                 0x6720 - Blowfish

                 0x6721 - Twofish

                 0x6801 - RC4

                 0xFFFF - Unknown algorithm

       7.2.3.3 Bitlen - Explicit bit length of key

                 32 - 448 bits

       7.2.3.4 Flags - Processing flags needed for decryption

                 0x0001 - Password is required to decrypt

                 0x0002 - Certificates only

                 0x0003 - Password or certificate required to decrypt

                 Values > 0x0003 reserved for certificate processing

   7.2.4 Decryption header record preceding compressed file data.

                 -Decryption Header:

                  Value     Size     Description

----- ---- -----------

                  IVSize    2 bytes  Size of initialization vector (IV)

                  IVData    IVSize   Initialization vector for this file

                  Size      4 bytes  Size of remaining decryption header data

                  Format    2 bytes  Format definition for this record

                  AlgID     2 bytes  Encryption algorithm identifier

                  Bitlen    2 bytes  Bit length of encryption key

                  Flags     2 bytes  Processing flags

                  ErdSize   2 bytes  Size of Encrypted Random Data

                  ErdData   ErdSize  Encrypted Random Data

                  Reserved1 4 bytes  Reserved certificate processing data

                  Reserved2 (var)    Reserved for certificate processing data

                  VSize     2 bytes  Size of password validation data

                  VData     VSize-4  Password validation data

                  VCRC32    4 bytes  Standard ZIP CRC32 of password validation data

       7.2.4.1 IVData - The size of the IV SHOULD match the algorithm block size.

       The IVData can be completely random data.  If the size of

       the randomly generated data does not match the block size

       it SHOULD be complemented with zero's or truncated as

       necessary.  If IVSize is 0,then IV = CRC32 + Uncompressed

       File Size (as a 64 bit little-endian, unsigned integer value).

       7.2.4.2 Format - the data format identifier for this record.  The only

       value allowed at this time is the integer value 3.

       7.2.4.3 AlgId - integer identifier of the encryption algorithm from the

       following range

                     0x6601 - DES

                     0x6602 - RC2 (version needed to extract < 5.2)

                     0x6603 - 3DES 168

                     0x6609 - 3DES 112

                     0x660E - AES 128

                     0x660F - AES 192

                     0x6610 - AES 256

                     0x6702 - RC2 (version needed to extract >= 5.2)

                     0x6720 - Blowfish

                     0x6721 - Twofish

                     0x6801 - RC4

                     0xFFFF - Unknown algorithm

        7.2.4.4 Bitlen - Explicit bit length of key

                     32 - 448 bits

        7.2.4.5 Flags - Processing flags needed for decryption

                     0x0001 - Password is required to decrypt

                     0x0002 - Certificates only

                     0x0003 - Password or certificate required to decrypt

                     Values > 0x0003 reserved for certificate processing

        7.2.4.6 ErdData - Encrypted random data is used to store random data that

        is used to generate a file session key for encrypting

        each file.  SHA1 is used to calculate hash data used to

        derive keys.  File session keys are derived from a master

        session key generated from the user-supplied password.

        If the Flags field in the decryption header contains

        the value 0x4000, then the ErdData field MUST be

        decrypted using 3DES. If the value 0x4000 is not set,

        then the ErdData field MUST be decrypted using AlgId.

        7.2.4.7 Reserved1 - Reserved for certificate processing, if value is

        zero, then Reserved2 data is absent.  See the explanation

        under the Certificate Processing Method for details on

        this data structure.

        7.2.4.8 Reserved2 - If present, the size of the Reserved2 data structure

        is located by skipping the first 4 bytes of this field

        and using the next 2 bytes as the remaining size.  See

        the explanation under the Certificate Processing Method

        for details on this data structure.

        7.2.4.9 VSize - This size value will always include the 4 bytes of the

        VCRC32 data and will be greater than 4 bytes.

        7.2.4.10 VData - Random data for password validation.  This data is VSize

        in length and VSize MUST be a multiple of the encryption

        block size.  VCRC32 is a checksum value of VData.

        VData and VCRC32 are stored encrypted and start the

        stream of encrypted data for a file.

    7.2.5 Useful Tips

        7.2.5.1 Strong Encryption is always applied to a file after compression. The

        block oriented algorithms all operate in Cypher Block Chaining (CBC)

        mode.  The block size used for AES encryption is 16.  All other block

        algorithms use a block size of 8.  Two IDs are defined for RC2 to

        account for a discrepancy found in the implementation of the RC2

        algorithm in the cryptographic library on Windows XP SP1 and all

        earlier versions of Windows.  It is recommended that zero length files

        not be encrypted, however programs SHOULD be prepared to extract them

        if they are found within a ZIP file.

        7.2.5.2 A pseudo-code representation of the encryption process is as follows:

            Password = GetUserPassword()

            MasterSessionKey = DeriveKey(SHA1(Password))

            RD = CryptographicStrengthRandomData()

            For Each File

               IV = CryptographicStrengthRandomData()

               VData = CryptographicStrengthRandomData()

               VCRC32 = CRC32(VData)

               FileSessionKey = DeriveKey(SHA1(IV + RD)

               ErdData = Encrypt(RD,MasterSessionKey,IV)

               Encrypt(VData + VCRC32 + FileData, FileSessionKey,IV)

            Done

        7.2.5.3 The function names and parameter requirements will depend on

        the choice of the cryptographic toolkit selected.  Almost any

        toolkit supporting the reference implementations for each

        algorithm can be used.  The RSA BSAFE(r), OpenSSL, and Microsoft

        CryptoAPI libraries are all known to work well.

 7.3 Single Password - Central Directory Encryption

--------------------------------------------------

    7.3.1 Central Directory Encryption is achieved within the .ZIP format by

    encrypting the Central Directory structure.  This encapsulates the metadata

    most often used for processing .ZIP files.  Additional metadata is stored for

    redundancy in the Local Header for each file.  The process of concealing

    metadata by encrypting the Central Directory does not protect the data within

    the Local Header.  To avoid information leakage from the exposed metadata

    in the Local Header, the fields containing information about a file are masked.

    7.3.2 Local Header

    Masking replaces the true content of the fields for a file in the Local

    Header with false information.  When masked, the Local Header is not

    suitable for streaming access and the options for data recovery of damaged

    archives is reduced.  Extra Data fields that MAY contain confidential

    data SHOULD NOT be stored within the Local Header.  The value set into

    the Version needed to extract field SHOULD be the correct value needed to

    extract the file without regard to Central Directory Encryption. The fields

    within the Local Header targeted for masking when the Central Directory is

    encrypted are:

            Field Name                     Mask Value

------------------ ---------------------------

            compression method              0

            last mod file time              0

            last mod file date              0

            crc-32                          0

            compressed size                 0

            uncompressed size               0

            file name (variable size)       Base 16 value from the

                                            range 1 - 0xFFFFFFFFFFFFFFFF

                                            represented as a string whose

                                            size will be set into the

                                            file name length field

    The Base 16 value assigned as a masked file name is simply a sequentially

    incremented value for each file starting with 1 for the first file.

    Modifications to a ZIP file MAY cause different values to be stored for

    each file.  For compatibility, the file name field in the Local Header

    SHOULD NOT be left blank.  As of Version 6.2 of this specification,

    the Compression Method and Compressed Size fields are not yet masked.

    Fields having a value of 0xFFFF or 0xFFFFFFFF for the ZIP64 format

    SHOULD NOT be masked.

    7.3.3 Encrypting the Central Directory

    Encryption of the Central Directory does not include encryption of the

    Central Directory Signature data, the Zip64 End of Central Directory

    record, the Zip64 End of Central Directory Locator, or the End

    of Central Directory record.  The ZIP file comment data is never

    encrypted.

    Before encrypting the Central Directory, it MAY optionally be compressed.

    Compression is not required, but for storage efficiency it is assumed

    this structure will be compressed before encrypting.  Similarly, this

    specification supports compressing the Central Directory without

    requiring that it also be encrypted.  Early implementations of this

    feature will assume the encryption method applied to files matches the

    encryption applied to the Central Directory.

    Encryption of the Central Directory is done in a manner similar to

    that of file encryption.  The encrypted data is preceded by a

    decryption header.  The decryption header is known as the Archive

    Decryption Header.  The fields of this record are identical to

    the decryption header preceding each encrypted file.  The location

    of the Archive Decryption Header is determined by the value in the

    Start of the Central Directory field in the Zip64 End of Central

    Directory record.  When the Central Directory is encrypted, the

    Zip64 End of Central Directory record will always be present.

    The layout of the Zip64 End of Central Directory record for all

    versions starting with 6.2 of this specification will follow the

    Version 2 format.  The Version 2 format is as follows:

    The leading fixed size fields within the Version 1 format for this

    record remain unchanged.  The record signature for both Version 1

    and Version 2 will be 0x06064b50.  Immediately following the last

    byte of the field known as the Offset of Start of Central

    Directory With Respect to the Starting Disk Number will begin the

    new fields defining Version 2 of this record.

    7.3.4 New fields for Version 2

    Note: all fields stored in Intel low-byte/high-byte order.

              Value                 Size       Description

----- ---- -----------

              Compression Method    2 bytes    Method used to compress the

                                               Central Directory

              Compressed Size       8 bytes    Size of the compressed data

              Original   Size       8 bytes    Original uncompressed size

              AlgId                 2 bytes    Encryption algorithm ID

              BitLen                2 bytes    Encryption key length

              Flags                 2 bytes    Encryption flags

              HashID                2 bytes    Hash algorithm identifier

              Hash Length           2 bytes    Length of hash data

              Hash Data             (variable) Hash data

     The Compression Method accepts the same range of values as the

     corresponding field in the Central Header.

     The Compressed Size and Original Size values will not include the

     data of the Central Directory Signature which is compressed or

     encrypted.

     The AlgId, BitLen, and Flags fields accept the same range of values

     the corresponding fields within the 0x0017 record.

     Hash ID identifies the algorithm used to hash the Central Directory

     data.  This data does not have to be hashed, in which case the

     values for both the HashID and Hash Length will be 0.  Possible

     values for HashID are:

              Value         Algorithm

------ ---------

             0x0000          none

             0x0001          CRC32

             0x8003          MD5

             0x8004          SHA1

             0x8007          RIPEMD160

             0x800C          SHA256

             0x800D          SHA384

             0x800E          SHA512

     7.3.5 When the Central Directory data is signed, the same hash algorithm

     used to hash the Central Directory for signing SHOULD be used.

     This is recommended for processing efficiency, however, it is

     permissible for any of the above algorithms to be used independent

     of the signing process.

     The Hash Data will contain the hash data for the Central Directory.

     The length of this data will vary depending on the algorithm used.

     The Version Needed to Extract SHOULD be set to 62.

     The value for the Total Number of Entries on the Current Disk will

     be 0.  These records will no longer support random access when

     encrypting the Central Directory.

     7.3.6 When the Central Directory is compressed and/or encrypted, the

     End of Central Directory record will store the value 0xFFFFFFFF

     as the value for the Total Number of Entries in the Central

     Directory.  The value stored in the Total Number of Entries in

     the Central Directory on this Disk field will be 0.  The actual

     values will be stored in the equivalent fields of the Zip64

     End of Central Directory record.

     7.3.7 Decrypting and decompressing the Central Directory is accomplished

     in the same manner as decrypting and decompressing a file.

 7.4 Certificate Processing Method

---------------------------------

    The Certificate Processing Method for ZIP file encryption

    defines the following additional data fields:

    7.4.1 Certificate Flag Values

    Additional processing flags that can be present in the Flags field of both

    the 0x0017 field of the central directory Extra Field and the Decryption

    header record preceding compressed file data are:

         0x0007 - reserved for future use

         0x000F - reserved for future use

         0x0100 - Indicates non-OAEP key wrapping was used.  If this

                  this field is set, the version needed to extract MUST

                  be at least 61.  This means OAEP key wrapping is not

                  used when generating a Master Session Key using

                  ErdData.

         0x4000 - ErdData MUST be decrypted using 3DES-168, otherwise use the

                  same algorithm used for encrypting the file contents.

         0x8000 - reserved for future use

    7.4.2 CertData - Extra Field 0x0017 record certificate data structure

    The data structure used to store certificate data within the section

    of the Extra Field defined by the CertData field of the 0x0017

    record are as shown:

          Value     Size     Description

----- ---- -----------

          RCount    4 bytes  Number of recipients.

          HashAlg   2 bytes  Hash algorithm identifier

          HSize     2 bytes  Hash size

          SRList    (var)    Simple list of recipients hashed public keys

         RCount    This defines the number intended recipients whose

                   public keys were used for encryption.  This identifies

                   the number of elements in the SRList.

         HashAlg   This defines the hash algorithm used to calculate

                   the public key hash of each public key used

                   for encryption. This field currently supports

                   only the following value for SHA-1

                   0x8004 - SHA1

         HSize     This defines the size of a hashed public key.

         SRList    This is a variable length list of the hashed

                   public keys for each intended recipient.  Each

                   element in this list is HSize.  The total size of

                   SRList is determined using RCount \* HSize.

    7.4.3 Reserved1 - Certificate Decryption Header Reserved1 Data

          Value     Size     Description

----- ---- -----------

          RCount    4 bytes  Number of recipients.

          RCount   This defines the number intended recipients whose

                   public keys were used for encryption.  This defines

                   the number of elements in the REList field defined below.

    7.4.4 Reserved2 - Certificate Decryption Header Reserved2 Data Structures

          Value     Size     Description

----- ---- -----------

          HashAlg   2 bytes  Hash algorithm identifier

          HSize     2 bytes  Hash size

          REList    (var)    List of recipient data elements

         HashAlg   This defines the hash algorithm used to calculate

                   the public key hash of each public key used

                   for encryption. This field currently supports

                   only the following value for SHA-1

                       0x8004 - SHA1

         HSize     This defines the size of a hashed public key

                   defined in REHData.

         REList    This is a variable length of list of recipient data.

                   Each element in this list consists of a Recipient

                   Element data structure as follows:

        Recipient Element (REList) Data Structure:

              Value     Size     Description

----- ---- -----------

              RESize    2 bytes  Size of REHData + REKData

              REHData   HSize    Hash of recipients public key

              REKData   (var)    Simple key blob

             RESize    This defines the size of an individual REList

                       element.  This value is the combined size of the

                       REHData field + REKData field.  REHData is defined by

                       HSize.  REKData is variable and can be calculated

                       for each REList element using RESize and HSize.

             REHData   Hashed public key for this recipient.

             REKData   Simple Key Blob.  The format of this data structure

                       is identical to that defined in the Microsoft

                       CryptoAPI and generated using the CryptExportKey()

                       function.  The version of the Simple Key Blob

                       supported at this time is 0x02 as defined by

                       Microsoft.

7.5 Certificate Processing - Central Directory Encryption

---------------------------------------------------------

    7.5.1 Central Directory Encryption using Digital Certificates will

    operate in a manner similar to that of Single Password Central

    Directory Encryption.  This record will only be present when there

    is data to place into it.  Currently, data is placed into this

    record when digital certificates are used for either encrypting

    or signing the files within a ZIP file.  When only password

    encryption is used with no certificate encryption or digital

    signing, this record is not currently needed. When present, this

    record will appear before the start of the actual Central Directory

    data structure and will be located immediately after the Archive

    Decryption Header if the Central Directory is encrypted.

    7.5.2 The Archive Extra Data record will be used to store the following

    information.  Additional data MAY be added in future versions.

    Extra Data Fields:

    0x0014 - PKCS#7 Store for X.509 Certificates

    0x0016 - X.509 Certificate ID and Signature for central directory

    0x0019 - PKCS#7 Encryption Recipient Certificate List

    The 0x0014 and 0x0016 Extra Data records that otherwise would be

    located in the first record of the Central Directory for digital

    certificate processing. When encrypting or compressing the Central

    Directory, the 0x0014 and 0x0016 records MUST be located in the

    Archive Extra Data record and they SHOULD NOT remain in the first

    Central Directory record.  The Archive Extra Data record will also

    be used to store the 0x0019 data.

    7.5.3 When present, the size of the Archive Extra Data record will be

    included in the size of the Central Directory.  The data of the

    Archive Extra Data record will also be compressed and encrypted

    along with the Central Directory data structure.

7.6 Certificate Processing Differences

--------------------------------------

    7.6.1 The Certificate Processing Method of encryption differs from the

    Single Password Symmetric Encryption Method as follows.  Instead

    of using a user-defined password to generate a master session key,

    cryptographically random data is used.  The key material is then

    wrapped using standard key-wrapping techniques.  This key material

    is wrapped using the public key of each recipient that will need

    to decrypt the file using their corresponding private key.

    7.6.2 This specification currently assumes digital certificates will follow

    the X.509 V3 format for 1024 bit and higher RSA format digital

    certificates.  Implementation of this Certificate Processing Method

    requires supporting logic for key access and management.  This logic

    is outside the scope of this specification.

7.7 OAEP Processing with Certificate-based Encryption

-----------------------------------------------------

    7.7.1 OAEP stands for Optimal Asymmetric Encryption Padding.  It is a

    strengthening technique used for small encoded items such as decryption

    keys.  This is commonly applied in cryptographic key-wrapping techniques

    and is supported by PKCS #1.  Versions 5.0 and 6.0 of this specification

    were designed to support OAEP key-wrapping for certificate-based

    decryption keys for additional security.

    7.7.2 Support for private keys stored on Smartcards or Tokens introduced

    a conflict with this OAEP logic.  Most card and token products do

    not support the additional strengthening applied to OAEP key-wrapped

    data.  In order to resolve this conflict, versions 6.1 and above of this

    specification will no longer support OAEP when encrypting using

    digital certificates.

    7.7.3 Versions of PKZIP available during initial development of the

    certificate processing method set a value of 61 into the

    version needed to extract field for a file.  This indicates that

    non-OAEP key wrapping is used.  This affects certificate encryption

    only, and password encryption functions SHOULD NOT be affected by

    this value.  This means values of 61 MAY be found on files encrypted

    with certificates only, or on files encrypted with both password

    encryption and certificate encryption.  Files encrypted with both

    methods can safely be decrypted using the password methods documented.

7.8 Additional Encryption/Decryption Data Records

-----------------------------------------------------

    7.8.1 Additional information MAY be stored within a ZIP file in support

    of the strong password and certificate encryption methods defined above.

    These include, but are not limited to the following record types.

      0x0021        Policy Decryption Key Record

      0x0022        Smartcrypt Key Provider Record

      0x0023        Smartcrypt Policy Key Data Record

8.0  Splitting and Spanning ZIP files

-------------------------------------

    8.1 Spanned ZIP files

      8.1.1 Spanning is the process of segmenting a ZIP file across

      multiple removable media. This support has typically only

      been provided for DOS formatted floppy diskettes.

    8.2 Split ZIP files

      8.2.1 File splitting is a newer derivation of spanning.

      Splitting follows the same segmentation process as

      spanning, however, it does not require writing each

      segment to a unique removable medium and instead supports

      placing all pieces onto local or non-removable locations

      such as file systems, local drives, folders, etc.

    8.3  File Naming Differences

      8.3.1 A key difference between spanned and split ZIP files is

      that all pieces of a spanned ZIP file have the same name.

      Since each piece is written to a separate volume, no name

      collisions occur and each segment can reuse the original

      .ZIP file name given to the archive.

      8.3.2 Sequence ordering for DOS spanned archives uses the DOS

      volume label to determine segment numbers.  Volume labels

      for each segment are written using the form PKBACK#xxx,

      where xxx is the segment number written as a decimal

      value from 001 - nnn.

      8.3.3 Split ZIP files are typically written to the same location

      and are subject to name collisions if the spanned name

      format is used since each segment will reside on the same

      drive. To avoid name collisions, split archives are named

      as follows.

      Segment 1   = filename.z01

      Segment n-1 = filename.z(n-1)

      Segment n   = filename.zip

      8.3.4 The .ZIP extension is used on the last segment to support

      quickly reading the central directory.  The segment number

      n SHOULD be a decimal value.

    8.4  Spanned Self-extracting ZIP Files

      8.4.1 Spanned ZIP files MAY be PKSFX Self-extracting ZIP files.

      PKSFX files MAY also be split, however, in this case

      the first segment MUST be named filename.exe.  The first

      segment of a split PKSFX archive MUST be large enough to

      include the entire executable program.

    8.5  Capacities and Markers

      8.5.1 Capacities for split archives are as follows:

      Maximum number of segments = 4,294,967,295 - 1

      Maximum .ZIP segment size = 4,294,967,295 bytes

      Minimum segment size = 64K

      Maximum PKSFX segment size = 2,147,483,647 bytes

      8.5.2 Segment sizes MAY be different however by convention, all

      segment sizes SHOULD be the same with the exception of the

      last, which MAY be smaller.  Local and central directory

      header records MUST NOT be split across a segment boundary.

      When writing a header record, if the number of bytes remaining

      within a segment is less than the size of the header record,

      end the current segment and write the header at the start

      of the next segment.  The central directory MAY span segment

      boundaries, but no single record in the central directory

      SHOULD be split across segments.

      8.5.3 Spanned/Split archives created using PKZIP for Windows

      (V2.50 or greater), PKZIP Command Line (V2.50 or greater),

      or PKZIP Explorer will include a special spanning

      signature as the first 4 bytes of the first segment of

      the archive.  This signature (0x08074b50) will be

      followed immediately by the local header signature for

      the first file in the archive.

      8.5.4 A special spanning marker MAY also appear in spanned/split

      archives if the spanning or splitting process starts but

      only requires one segment.  In this case the 0x08074b50

      signature will be replaced with the temporary spanning

      marker signature of 0x30304b50.  Split archives can

      only be uncompressed by other versions of PKZIP that

      know how to create a split archive.

      8.5.5 The signature value 0x08074b50 is also used by some

      ZIP implementations as a marker for the Data Descriptor

      record.  Conflict in this alternate assignment can be

      avoided by ensuring the position of the signature

      within the ZIP file to determine the use for which it

      is intended.

9.0 Change Process

------------------

   9.1 In order for the .ZIP file format to remain a viable technology, this

   specification SHOULD be considered as open for periodic review and

   revision.  Although this format was originally designed with a

   certain level of extensibility, not all changes in technology

   (present or future) were or will be necessarily considered in its

   design.

   9.2 If your application requires new definitions to the

   extensible sections in this format, or if you would like to

   submit new data structures or new capabilities, please forward

   your request to zipformat@pkware.com.  All submissions will be

   reviewed by the ZIP File Specification Committee for possible

   inclusion into future versions of this specification.

   9.3 Periodic revisions to this specification will be published as

   DRAFT or as FINAL status to ensure interoperability. We encourage

   comments and feedback that MAY help improve clarity or content.

10.0 Incorporating PKWARE Proprietary Technology into Your Product

------------------------------------------------------------------

   10.1 The Use or Implementation in a product of APPNOTE technological

   components pertaining to either strong encryption or patching requires

   a separate, executed license agreement from PKWARE. Please contact

   PKWARE at zipformat@pkware.com or +1-414-289-9788 with regard to

   acquiring such a license.

   10.2 Additional information regarding PKWARE proprietary technology is

   available at http://www.pkware.com/appnote.

11.0 Acknowledgements

---------------------

   In addition to the above mentioned contributors to PKZIP and PKUNZIP,

   PKWARE would like to extend special thanks to Robert Mahoney for

   suggesting the extension .ZIP for this software.

12.0 References

---------------

   Fiala, Edward R., and Greene, Daniel H., "Data compression with

      finite windows",  Communications of the ACM, Volume 32, Number 4,

      April 1989, pages 490-505.

   Held, Gilbert, "Data Compression, Techniques and Applications,

      Hardware and Software Considerations", John Wiley & Sons, 1987.

   Huffman, D.A., "A method for the construction of minimum-redundancy

      codes", Proceedings of the IRE, Volume 40, Number 9, September 1952,

      pages 1098-1101.

   Nelson, Mark, "LZW Data Compression", Dr. Dobbs Journal, Volume 14,

      Number 10, October 1989, pages 29-37.

   Nelson, Mark, "The Data Compression Book",  M&T Books, 1991.

   Storer, James A., "Data Compression, Methods and Theory",

      Computer Science Press, 1988

   Welch, Terry, "A Technique for High-Performance Data Compression",

      IEEE Computer, Volume 17, Number 6, June 1984, pages 8-19.

   Ziv, J. and Lempel, A., "A universal algorithm for sequential data

      compression", Communications of the ACM, Volume 30, Number 6,

       June 1987, pages 520-540.

   Ziv, J. and Lempel, A., "Compression of individual sequences via

      variable-rate coding", IEEE Transactions on Information Theory,

      Volume 24, Number 5, September 1978, pages 530-536.

APPENDIX A - AS/400 Extra Field (0x0065) Attribute Definitions

--------------------------------------------------------------

A.1 Field Definition Structure:

   a. field length including length             2 bytes Big Endian

   b. field code                                2 bytes

   c. data                                      x bytes

A.2 Field Code  Description

   4001     Source type i.e. CLP etc

   4002     The text description of the library

   4003     The text description of the file

   4004     The text description of the member

   4005     x'F0' or 0 is PF-DTA,  x'F1' or 1 is PF\_SRC

   4007     Database Type Code                  1 byte

   4008     Database file and fields definition

   4009     GZIP file type                      2 bytes

   400B     IFS code page                       2 bytes

   400C     IFS Time of last file status change 4 bytes

   400D     IFS Access Time                     4 bytes

   400E     IFS Modification time               4 bytes

   005C     Length of the records in the file   2 bytes

   0068     GZIP two words                      8 bytes

APPENDIX B - z/OS Extra Field (0x0065) Attribute Definitions

------------------------------------------------------------

B.1 Field Definition Structure:

   a. field length including length             2 bytes Big Endian

   b. field code                                2 bytes

   c. data                                      x bytes

B.2 Field Code  Description

   0001     File Type                           2 bytes

   0002     NonVSAM Record Format               1 byte

   0003     Reserved

   0004     NonVSAM Block Size                  2 bytes Big Endian

   0005     Primary Space Allocation            3 bytes Big Endian

   0006     Secondary Space Allocation          3 bytes Big Endian

   0007     Space Allocation Type1 byte flag

   0008     Modification Date                   Retired with PKZIP 5.0 +

   0009     Expiration Date                     Retired with PKZIP 5.0 +

   000A     PDS Directory Block Allocation      3 bytes Big Endian binary value

   000B     NonVSAM Volume List                 variable

   000C     UNIT Reference                      Retired with PKZIP 5.0 +

   000D     DF/SMS Management Class             8 bytes EBCDIC Text Value

   000E     DF/SMS Storage Class                8 bytes EBCDIC Text Value

   000F     DF/SMS Data Class                   8 bytes EBCDIC Text Value

   0010     PDS/PDSE Member Info.               30 bytes

   0011     VSAM sub-filetype                   2 bytes

   0012     VSAM LRECL                          13 bytes EBCDIC "(num\_avg num\_max)"

   0013     VSAM Cluster Name                   Retired with PKZIP 5.0 +

   0014     VSAM KSDS Key Information           13 bytes EBCDIC "(num\_length num\_position)"

   0015     VSAM Average LRECL                  5 bytes EBCDIC num\_value padded with blanks

   0016     VSAM Maximum LRECL                  5 bytes EBCDIC num\_value padded with blanks

   0017     VSAM KSDS Key Length                5 bytes EBCDIC num\_value padded with blanks

   0018     VSAM KSDS Key Position              5 bytes EBCDIC num\_value padded with blanks

   0019     VSAM Data Name                      1-44 bytes EBCDIC text string

   001A     VSAM KSDS Index Name                1-44 bytes EBCDIC text string

   001B     VSAM Catalog Name                   1-44 bytes EBCDIC text string

   001C     VSAM Data Space Type                9 bytes EBCDIC text string

   001D     VSAM Data Space Primary             9 bytes EBCDIC num\_value left-justified

   001E     VSAM Data Space Secondary           9 bytes EBCDIC num\_value left-justified

   001F     VSAM Data Volume List               variable EBCDIC text list of 6-character Volume IDs

   0020     VSAM Data Buffer Space              8 bytes EBCDIC num\_value left-justified

   0021     VSAM Data CISIZE                    5 bytes EBCDIC num\_value left-justified

   0022     VSAM Erase Flag                     1 byte flag

   0023     VSAM Free CI %                      3 bytes EBCDIC num\_value left-justified

   0024     VSAM Free CA %                      3 bytes EBCDIC num\_value left-justified

   0025     VSAM Index Volume List              variable EBCDIC text list of 6-character Volume IDs

   0026     VSAM Ordered Flag                   1 byte flag

   0027     VSAM REUSE Flag                     1 byte flag

   0028     VSAM SPANNED Flag                   1 byte flag

   0029     VSAM Recovery Flag                  1 byte flag

   002A     VSAM  WRITECHK  Flag                1 byte flag

   002B     VSAM Cluster/Data SHROPTS           3 bytes EBCDIC "n,y"

   002C     VSAM Index SHROPTS                  3 bytes EBCDIC "n,y"

   002D     VSAM Index Space Type               9 bytes EBCDIC text string

   002E     VSAM Index Space Primary            9 bytes EBCDIC num\_value left-justified

   002F     VSAM Index Space Secondary          9 bytes EBCDIC num\_value left-justified

   0030     VSAM Index CISIZE                   5 bytes EBCDIC num\_value left-justified

   0031     VSAM Index IMBED                    1 byte flag

   0032     VSAM Index Ordered Flag             1 byte flag

   0033     VSAM REPLICATE Flag                 1 byte flag

   0034     VSAM Index REUSE Flag               1 byte flag

   0035     VSAM Index WRITECHK Flag            1 byte flag Retired with PKZIP 5.0 +

   0036     VSAM Owner                          8 bytes EBCDIC text string

   0037     VSAM Index Owner                    8 bytes EBCDIC text string

   0038     Reserved

   0039     Reserved

   003A     Reserved

   003B     Reserved

   003C     Reserved

   003D     Reserved

   003E     Reserved

   003F     Reserved

   0040     Reserved

   0041     Reserved

   0042     Reserved

   0043     Reserved

   0044     Reserved

   0045     Reserved

   0046     Reserved

   0047     Reserved

   0048     Reserved

   0049     Reserved

   004A     Reserved

   004B     Reserved

   004C     Reserved

   004D     Reserved

   004E     Reserved

   004F     Reserved

   0050     Reserved

   0051     Reserved

   0052     Reserved

   0053     Reserved

   0054     Reserved

   0055     Reserved

   0056     Reserved

   0057     Reserved

   0058     PDS/PDSE Member TTR Info.           6 bytes  Big Endian

   0059     PDS 1st LMOD Text TTR               3 bytes  Big Endian

   005A     PDS LMOD EP Rec #                   4 bytes  Big Endian

   005B     Reserved

   005C     Max Length of records               2 bytes  Big Endian

   005D     PDSE Flag                           1 byte flag

   005E     Reserved

   005F     Reserved

   0060     Reserved

   0061     Reserved

   0062     Reserved

   0063     Reserved

   0064     Reserved

   0065     Last Date Referenced                4 bytes  Packed Hex "yyyymmdd"

   0066     Date Created                        4 bytes  Packed Hex "yyyymmdd"

   0068     GZIP two words                      8 bytes

   0071     Extended NOTE Location              12 bytes Big Endian

   0072     Archive device UNIT                 6 bytes  EBCDIC

   0073     Archive 1st Volume                  6 bytes  EBCDIC

   0074     Archive 1st VOL File Seq#           2 bytes  Binary

   0075     Native I/O Flags                    2 bytes

   0081     Unix File Type                      1 byte   enumerated

   0082     Unix File Format                    1 byte   enumerated

   0083     Unix File Character Set Tag Info    4 bytes

   0090     ZIP Environmental Processing Info   4 bytes

   0091     EAV EATTR Flags                     1 byte

   0092     DSNTYPE Flags                       1 byte

   0093     Total Space Allocation (Cyls)       4 bytes  Big Endian

   009D     NONVSAM DSORG                       2 bytes

   009E     Program Virtual Object Info         3 bytes

   009F     Encapsulated file Info              9 bytes

   400C     Unix File Creation Time             4 bytes

   400D     Unix File Access Time               4 bytes

   400E     Unix File Modification time         4 bytes

   4101     IBMCMPSC Compression Info           variable

   4102     IBMCMPSC Compression Size           8 bytes  Big Endian

APPENDIX C - Zip64 Extensible Data Sector Mappings

---------------------------------------------------

         -Z390   Extra Field:

          The following is the general layout of the attributes for the

          ZIP 64 "extra" block for extended tape operations.

          Note: some fields stored in Big Endian format.  All text is

          in EBCDIC format unless otherwise specified.

          Value       Size          Description

----- ---- -----------

  (Z390)  0x0065      2 bytes       Tag for this "extra" block type

          Size        4 bytes       Size for the following data block

          Tag         4 bytes       EBCDIC "Z390"

          Length71    2 bytes       Big Endian

          Subcode71   2 bytes       Enote type code

          FMEPos      1 byte

          Length72    2 bytes       Big Endian

          Subcode72   2 bytes       Unit type code

          Unit        1 byte        Unit

          Length73    2 bytes       Big Endian

          Subcode73   2 bytes       Volume1 type code

          FirstVol    1 byte        Volume

          Length74    2 bytes       Big Endian

          Subcode74   2 bytes       FirstVol file sequence

          FileSeq     2 bytes       Sequence

APPENDIX D - Language Encoding (EFS)

------------------------------------

D.1 The ZIP format has historically supported only the original IBM PC character

encoding set, commonly referred to as IBM Code Page 437.  This limits storing

file name characters to only those within the original MS-DOS range of values

and does not properly support file names in other character encodings, or

languages. To address this limitation, this specification will support the

following change.

D.2 If general purpose bit 11 is unset, the file name and comment SHOULD conform

to the original ZIP character encoding.  If general purpose bit 11 is set, the

filename and comment MUST support The Unicode Standard, Version 4.1.0 or

greater using the character encoding form defined by the UTF-8 storage

specification.  The Unicode Standard is published by the The Unicode

Consortium (www.unicode.org).  UTF-8 encoded data stored within ZIP files

is expected to not include a byte order mark (BOM).

D.3 Applications MAY choose to supplement this file name storage through the use

of the 0x0008 Extra Field.  Storage for this optional field is currently

undefined, however it will be used to allow storing extended information

on source or target encoding that MAY further assist applications with file

name, or file content encoding tasks.  Please contact PKWARE with any

requirements on how this field SHOULD be used.

D.4 The 0x0008 Extra Field storage MAY be used with either setting for general

purpose bit 11.  Examples of the intended usage for this field is to store

whether "modified-UTF-8" (JAVA) is used, or UTF-8-MAC.  Similarly, other

commonly used character encoding (code page) designations can be indicated

through this field.  Formalized values for use of the 0x0008 record remain

undefined at this time.  The definition for the layout of the 0x0008 field

will be published when available.  Use of the 0x0008 Extra Field provides

for storing data within a ZIP file in an encoding other than IBM Code

Page 437 or UTF-8.

D.5 General purpose bit 11 will not imply any encoding of file content or

password.  Values defining character encoding for file content or

password MUST be stored within the 0x0008 Extended Language Encoding

Extra Field.

D.6 Ed Gordon of the Info-ZIP group has defined a pair of "extra field" records

that can be used to store UTF-8 file name and file comment fields.  These

records can be used for cases when the general purpose bit 11 method

for storing UTF-8 data in the standard file name and comment fields is

not desirable.  A common case for this alternate method is if backward

compatibility with older programs is required.

D.7 Definitions for the record structure of these fields are included above

in the section on 3rd party mappings for "extra field" records.  These

records are identified by Header ID's 0x6375 (Info-ZIP Unicode Comment

Extra Field) and 0x7075 (Info-ZIP Unicode Path Extra Field).

D.8 The choice of which storage method to use when writing a ZIP file is left

to the implementation.  Developers SHOULD expect that a ZIP file MAY

contain either method and SHOULD provide support for reading data in

either format. Use of general purpose bit 11 reduces storage requirements

for file name data by not requiring additional "extra field" data for

each file, but can result in older ZIP programs not being able to extract

files.  Use of the 0x6375 and 0x7075 records will result in a ZIP file

that SHOULD always be readable by older ZIP programs, but requires more

storage per file to write file name and/or file comment fields.

APPENDIX E - AE-x encryption marker

-----------------------------------

E.1 AE-x defines an alternate password-based encryption method used

in ZIP files that is based on a file encryption utility developed by

Dr. Brian Gladman.  Information on Dr. Gladman's method is available at

   http://www.gladman.me.uk/cryptography\_technology/fileencrypt/

E.2 AE-x uses AES with CTR (counter mode) and HMAC-SHA1.  It defines

encryption using key sizes of 128 bits or 256 bits.  It does not

restrict support for decrypting 192 bits.

E.3 This method uses the standard ZIP encryption bit (bit 0)

of the general purpose bit flag (section 4.4.4) to indicate a

file is encrypted.

E.4 The compression method field (section 4.4.5) is set to 99

to indicate a file has been encrypted using this method.

E.5 The actual compression method is stored in an extra field

structure identified by a Header ID of 0x9901. Information on this

record structure can be found at http://www.winzip.com/aes\_info.htm.

E.6 Two versions are defined for the 0x9901 structure.

   E.6.1 Version 1 stores the file CRC value in the CRC-32 field

   (section 4.4.7).

   E.6.2 Version 2 stores a value of 0 in the CRC-32 field.