1 Einleitung

Das Dateiformat ist ein Archiv mit einer Ordnerstruktur und verschiedenen Dateien. Das Dateiarchiv kann mit einem Passwort geschützt werden. Die Verifizierung der Verschlüsselungsfaktoren benötigt eine gewisse Zeit. Somit kann ein Passwort mit vier Buchstaben schon als sicher gelten, weil es viel Zeit in Anspruch nimmt alle Möglichkeiten auszuprobieren. Das geschützte Dateiarchiv kann mit dem Programm "FileProtected" erstellt werden. Die Dateierweiterung wird ".pdata" (Protected Data) genannt. Bei mehrfachen Dateien werden die Erweiterungen mit ".001" bis ".254" angegeben.

2 Wertetypen

Typ	Beschreibung	Bereich
INT8	8Bit mit Vorzeichen	-128 bis 127
INT16	16Bit mit Vorzeichen	-32.768 bis 32.767
INT32	32Bit mit Vorzeichen	-2.147.483.648 bis 2.147.483.647
INT64	64Bit mit Vorzeichen	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
BYTE	8Bit ohne Vorzeichen	0 bis 255
UINT16	16Bit ohne Vorzeichen	0 bis 65.535
UINT32	32Bit ohne Vorzeichen	0 bis 4.294.967.295
UINT64	64Bit ohne Vorzeichen	0 bis 18.446.744.073.709.551.615
CHAR	8Bit Zeichen	0 bis 255
WCHAR	16Bit Zeichen	0 bis 65.535
FLOAT	32Bit Gleitkommazahl	± 1.5e-45 zu ± 3.4e38
DOUBLE	64Bit Gleitkommazahl	± 5.0e-324 zu ± 1.7e308
MEMORY	Speicher in Bytes	
...[]	Wertemenge	siehe Abschnitt 2.1
?	Verschiedene Werte	siehe Abschnitt 2.2
*	Kodierter Wert	siehe Abschnitt 2.3
...	Nächste Tabelle	siehe Abschnitt 2.4
-> {	Beginn der Schleife	siehe Abschnitt 2.5
} <-	Ende der Schleife	siehe Abschnitt 2.5

Tabelle 2: Wertetypen

2.1 Wertemenge

Die Menge besteht aus einem bestimmten Wertetyp. Die Anzahl der Menge wird in den Informationen detailliert angegeben und ist meistens der vorherige Formatwert.

Beispiel: `INT16[] = { INT16, INT16, INT16, INT16, ... }.`

2.2 Verschiedene Werte

Es können verschiedene Wertetypen in dem Format vorkommen. Die Abhängigkeit des Typs wird in der Information beschrieben.

2.3 Kodierter Wert

Der Wert muss mit einer bestimmten XOR Verknüpfung entschlüsselt werden. In der Information wird beschrieben, wie die Verknüpfung durchzuführen ist.

2.4 Nächste Tabelle

In dem angegebenen Abschnitt und der Tabelle wird das Dateiformat weiter angezeigt.

2.5 Die Schleife

Bei einer Schleife wird das Format wiederholt durchlaufen. Die Anzahl der Durchläufe wird in den Informationen detailliert angegeben und ist meistens der vorherige Wert.

3 Beschreibungen

3.1 Dateiformat

Typ	Name	Beschreibung	Info
UINT32	IDNumber	Die Datei kann die ID Nummer (0x54414450) haben.	4.1
UINT32	LoopMax	Die Anzahl der Entschlüsselungsdurchläufe.	4.2
BYTE	PassVersion	Die Versionsnummer und eine Bestätigung des Passwortes.	4.3
BYTE	FileCount	Die Anzahl der Archivdateien.	4.4
BYTE*	FactorCount	Die Anzahl der Faktoren für die Entschlüsselung.	4.5
BYTE[]*	FactorMemory	Die Faktoren für den Entschlüsselungsalgorithmus.	4.6
BYTE[]*	ReferenceMemory	Eine Wertemenge zur Verifizierung der Faktoren.	4.7
UINT32*	CodeValueCount	Die Anzahl von Entschlüsselungswerten.	4.8
UINT32*	CodePageCount	Die Anzahl zusätzlicher Entschlüsselungswerte und Version	4.9
...		3.2 Archivformat, Tabelle 3.2	

Tabelle 3.1: Dateiformat

3.2 Archivformat

Typ	Name	Beschreibung	Info
-> {	EndOfFile		5.1
UINT16*	HeaderSize	Die Formatgröße ohne Dateigröße in Bytes.	5.2
BYTE*	HeaderFlag	Gibt den Formattype und die Wertegrößen an.	5.3
...		3.3 Archivordnerformat oder 3.4 Archivdateiformat	5.4
} <-	EndOfFile		5.1

Tabelle 3.2: Archivformat

3.3 Archivordnerformat

Typ	Name	Beschreibung	Info
?*	FolderIndex	Der Index für den übergeordneten Ordner.	6.1
?*	FolderNameSize	Die Größe des Namenspeichers in Bytes.	6.2
?[]*	FolderName	Der Name des Ordners.	6.3

Tabelle 3.3: Archivordnerformat

3.4 Archivdateiformat

Typ	Name	Beschreibung	Info
?*	FolderIndex	Der Index für den Dateiordner.	7.1
?*	FileSize	Die Größe des Datei in Bytes.	7.2
?*	FileNameSize	Die Größe des Namenspeichers in Bytes.	7.3
?[]*	FileName	Der Name und die Erweiterung der Datei.	7.4
MEMORY*	FileMemory	Der Dateispeicher in der angegebenen Größe.	7.5

Tabelle 3.4: Archivdateiformat

4 Informationen zum Dateiformat

4.1 Identifikationsnummer

Die Identifikationsnummer kennzeichnet das Dateiformat. Die Nummer kann auch mit 4 Buchstaben (PDAT: Protected Data) dargestellt werden. Das Archiv kann auch ohne Dateikennzeichnung erstellt werden. Dann muss die Identifikationsnummer aus einem Zufallsgenerator erzeugt werden und kann jeden Wert des Typs (UINT32) annehmen. Das hat Auswirkung auf die Werte in den Abschnitten 4.2 und 4.3.

4.2 Maximale Anzahl der Entschlüsselungsdurchläufe

Der Wert gibt die maximale Anzahl der Durchläufe für die Schleife im Entschlüsselungsalgorithmus an. Der Wert kann nicht kleiner als 100.000 und größer als 2.147.483.647 sein. Die Anzahl ist zwischen 2 bis 4 mal größer als der tatsächliche Schleifenwert. Ist ein falsches Passwort eingegeben worden, wird nur bis zu dieser Anzahl geprüft. Hat das Dateiarchiv keine Dateikennzeichnung (4.1) ist der Wert unbestimmt und muss mit einem Zufallsgenerator erzeugt werden. Die maximale Anzahl der Durchläufe ist somit 2.147.483.647.

4.3 Dateiversion und Passworteingabe

Die Versionsnummer ist für diese Formatbeschreibung immer 1. Wenn das höchste Bit (0x80) gesetzt ist, wird ein Passwort erwartet. Hat das Dateiarchiv keine Dateikennzeichnung (4.1) ist der Wert unbestimmt und muss mit einem Zufallsgenerator erzeugt werden. Somit bestimmt der Benutzer, ob ein Passwort eingegeben werden muss oder nicht.

4.3.1 Passworteingabe

Das Passwort (PassLetterArray) muss in eine Wertemenge von Bytes umgewandelt werden. Die maximale Größe der Menge (PassArray) ist 515 Bytes. Wurde kein Passwort eingegeben ist die Größe 0. Bei jedem Durchlauf der Schleife wird das aktuelle Zeichen mit einem Bytewert (PassSpace) addiert. Wurden alle Zeichen vom Passwort gelesen, wird dieser Wert übernommen und der Index (PassIndex) wieder auf 0 zurück gesetzt. Ist das aktuelle Zeichen aus dem Passwort kleiner als 256 wird es einfach in die Wertemenge (PassArray) übernommen. Wenn das Zeichen größer als 255 ist, wird zuerst das unter Byte und dann das höhere Byte in die Menge eingefügt.

```
UInt16[] PassLetterArray; //user input, can all ASCII letter
Byte[] PassArray = new Byte[0];

if(PassLetterArray != null && PassLetterArray.Length > 0) {
  PassArray = new Byte[515]; //size: 2 * 256 + 1 + 2

  Byte PassSpace = 0;
  Int32 PassIndex = 0;

  for(Int32 i = 0; i <= 513; i++) { //use size: 2 * 256 + 1
    PassSpace += (Byte) PassLetterArray[PassIndex];

    if(PassLetterArray[PassIndex] > 255) {
      PassArray[i++] = (Byte) PassLetterArray[PassIndex];
      PassArray[i] = (Byte) (PassLetterArray[PassIndex++] / 0x0100);
    } else {
      PassArray[i] = (Byte) PassLetterArray[PassIndex++];
    }

    if(PassIndex == PassLetterArray.Length) {
      PassArray[++i] = PassSpace;
      PassIndex = 0;
    }
  }
}
```

Programm 4.3.1: Passworteingabe

4.3.2 Passwortbestimmung

Die Identifikationsnummer und der Bytewert (PassVersion) werden aus der Datei gelesen. Ist die Identifikation vorhanden, wird als erstes geprüft, ob die Version 1 ist. Danach wird abgefragt, ob kein Passwort verwendet wurde.

```
UInt32 IDNumber;  //read from file
Byte PassVersion; //read from file
Byte[] PassArray; //see Program 4.3.1

Boolean UseFileIdentifier = IDNumber == 0x54414450 ? true : false;
Boolean UsePassword = PassArray.Length > 0 ? true : false;

if(UseFileIdentifier) {
  if((PassVersion & 0x7F) != 1) { //wrong version }

  if((PassVersion & 0x80) != 0x80) { //no password
    UsePassword = false;
  }
}
```

Programm 4.3.2: Passwortbestimmung

4.4 Anzahl der Archivdateien

Ein Archiv kann in mehrere Dateien unterteilt werden. Der Wert gibt die Anzahl der Dateien an. Enthält das Dateiarchiv keine Kennzeichnung (4.1) muss der Wert mit einem Zufallsgenerator erstellt werden.

Beispiel: File.pdata, File.001, File.002, ... , File.254

4.5 Anzahl der Faktoren

Die Faktoren sind eine Menge von Bytewerten, die für den Entschlüsselungsalgorithmus verwendet werden. Die Anzahl muss mit einem Zufallsgenerator erzeugt werden. Das höchste Bit (0x80) wird nicht verwendet, weil die Anzahl der Faktoren nicht kleiner als 128 sein darf. Wird ein Passwort verwendet, muss der gelesene Wert mit dem ersten Byte der Passwortmenge exklusive Oder verknüpft werden. Das Ergebnis ist dann die Anzahl.

```
Byte FactorCount;    //read from file
Byte[] PassArray;    //see Program 4.3.1
Boolean UsePassword; //see Program 4.3.2

if(UsePassword) FactorCount ^= PassArray[0]; //XOR

FactorCount = (Byte) (FactorCount | 0x80);
```

Programm 4.5: Faktoranzahl

4.6 Faktorspeicher

Die Faktoren sind eine Menge von Bytewerten, die für den Entschlüsselungsalgorithmus verwendet werden. Die Werte wurden mit einem Zufallsgenerator erzeugt. Die Anzahl der Faktoren (4.5) wird in Bytes angegeben. Wird ein Passwort verwendet, müssen die gelesenen Werte mit den Bytes der Passwortmenge exklusive Oder verknüpft werden.

```
Byte[] PassArray;    //see Program 4.3.1
Boolean UsePassword; //see Program 4.3.2
Byte FactorCount;    //see Program 4.5
Byte[] FactorMemory; //read from file

if(UsePassword) {
  for(Int32 i = 0; i < FactorCount; i++)
    FactorMemory[i] ^= PassArray[i + 1]; //XOR
}
```

Programm 4.6: Faktorspeicher

4.7 Verifizierungsspeicher

Der Speicher ist eine Menge von Bytewerten, die als Referenzwerte zur Verifizierung des Dateiformats benutzt werden. Die Anzahl der Werte (4.5) ist identisch mit den Faktoren. Wird ein Passwort verwendet, müssen die gelesenen Werte mit den Bytes der Passwortmenge exklusive Oder verknüpft werden.

```
Byte[] PassArray;        //see Program 4.3.1
Boolean UsePassword;     //see Program 4.3.2
Byte FactorCount;        //see Program 4.5
Byte[] ReferenceMemory; //read from file

if(UsePassword) {
  for(Int32 i = 0; i < FactorCount; i++)
    ReferenceMemory[i] ^= PassArray[i + FactorCount + 1]; //XOR
}
```

Programm 4.7.1: Verifizierungsspeicher

Die Verifizierung des Dateiformats wird mit dem Programm 4.7.2 bestimmt. Der Entschlüsselungs-algorithmus arbeitet mit den Faktoren und den Durchläufen der Schleifen. Die Anzahl der Berech-nungen benötigt eine bestimmte Zeitdauer, sie liegt zwischen eine bis zwei Sekunden. Der Wert (CodeValue) und der Index (CodeIndex) werden später weiter verwendet und sind im Programm global definiert.

```
UInt32 LoopMax;            //see Program 4.2
Byte FactorCount;          //see Program 4.5
Byte[] FactorMemory;       //see Program 4.6
Byte[] ReferenceMemory;    //see Program 4.7.1

this.CodeValue = 0; //global value: UInt64
this.CodeIndex = 1; //global value: UInt64

Int32 ReferenceIndex = 0;

for(UInt64 i = 1; i <= LoopMax + 1; i++) {
  for(UInt64 j = 1; j <= FactorCount; j++)
    this.CodeValue += (FactorMemory[j - 1] + (this.CodeValue + 1) * i) * j;

  this.CodeIndex++;
  this.CodeValue /= FactorCount;

  //equal reference values
  if((Byte) this.CodeValue == ReferenceMemory[ReferenceIndex]) {
    ReferenceIndex++;

    if(ReferenceIndex == FactorCount) {
      return; //file format: OK
    }
  } else {
    ReferenceIndex = 0;
  }
}

//LoopMax: bad file format
```

Programm 4.7.2: Verifizierung

Der Entschlüsselungsspeicher wird mit dem Programm 4.7.3 erstellt. Die Größe (Count) der Werte-menge (CodeMemory) ist unterschiedlich und wird in den Abschnitten 4.8 und 4.9 angegeben. Aus dem Ergebnis (CodeValue) der Berechnungen werden nur die ersten 8Bit in den Entschlüsselungs-speicher geschrieben. Die daraus entstehenden Werte sind nicht periodisch und folgen keinem Muster. Der Wert (CodeValue) und der Index (CodeIndex) werden weiter verwendet und sind in den Abschnitten global definiert. Beide Werte wurden im Programm 4.7.2 das erste Mal verwendet.

```
Byte FactorCount;    //see Program 4.5
Byte[] FactorMemory; //see Program 4.6

this.CodeValue; //global value: UInt64
this.CodeIndex; //global value: UInt64

UInt32 Count;   //various size

Byte[] CodeMemory = new Byte[(Int32) Count];

UInt32 Index = 0;

for(UInt64 i = this.CodeIndex; i < this.CodeIndex + Count; i++) {
  for(UInt64 j = 1; j <= FactorCount; j++)
     this.CodeValue += (FactorMemory[j - 1] + (this.CodeValue + 1) * i) * j;

  this.CodeValue /= FactorCount;

  CodeMemory[Index++] = (Byte) this.CodeValue;
}

this.CodeIndex += Index;
```

Programm 4.7.3: Entschlüsselungsspeicher

4.8 Anzahl der Entschlüsselungswerte

Die Anzahl der Entschlüsselungswerte kann nicht kleiner als 500.000 und größer als 500.000.000 sein. Der entsprechende Speicher aus Abschnitt 4.10.1 darf erst nach der Bestimmung der Anzahl der zusätzlichen Entschlüsselungswerte (4.9) erstellt werden.

```
Byte[] CodeMemory; //create with Program 4.7.3, Count = 4
Byte[] ValueArray; //read from file

UInt32[] FactorArray = new UInt32[] { 1, 0x00000100, 0x00010000, 0x01000000 };

UInt32 CodeValueCount = 0; //result for memory count

for(Int32 i = 0; i < 4; i++) {
  ValueArray[i] ^= CodeMemory[i];
  CodeValueCount += FactorArray[i] * ValueArray[i];
}
```

Programm 4.8: Bestimmung der Anzahl der Entschlüsselungswerte

4.9 Anzahl der zusätzlichen Entschlüsselungswerte

Die Anzahl der zusätzlichen Entschlüsselungswerte kann nicht kleiner als 500.000 und größer als 1.000.000 sein. Ist die Anzahl gleich 0 wird der Zusatzspeicher (4.10.2) nicht verwendet. Der Speicher darf erst nach dem einfachen Entschlüsselungsspeicher (4.10.1) erstellt werden. Das höchste Byte gibt die Dateiversion (4.3) an. Die Version wird immer zusätzlich angegeben, weil das Archiv eventuell keine Dateikennzeichnung (4.1) hat.

```
Byte[] CodeMemory; //create with Program 4.7.3, Count = 4
Byte[] ValueArray; //read from file

UInt32[] FactorArray = new UInt32[] { 1, 0x00000100, 0x00010000 };

UInt32 CodePageCount = 0; //result for memory count

Byte FileVersion = (Byte) (ValueArray[3] ^ CodeMemory[3]); //result for version

for(Int32 i = 0; i < 3; i++) {
  ValueArray[i] ^= CodeMemory[i];
  CodePageCount += FactorArray[i] * ValueArray[i];
}
```

Programm 4.9: Bestimmung der Anzahl der zusätzlichen Entschlüsselungswerte

4.10 Entschlüsselungswerte

4.10.1 Einfacher Entschlüsselungsspeicher

Der einfache Entschlüsselungsspeicher wird für alle weiteren Endkodierungen verwenden. Er wird mit der Anzahl aus dem Abschnitt 4.8 und dem Programm 4.7.3 erstellt.

```
this.CodeValueMemory; //create with Program 4.7.3, Count = CodeValueCount (4.8)
```

Ist kein zusätzlicher Entschlüsselungsspeicher vorhanden, wird das Programm 4.10.1 für die Endkodierungen aller weiteren Daten verwendet. Die Dateiposition (FilePosition) muss vor dem Lesen der Daten bestimmt werden. Die Originaldaten werden in die Menge (ByteArray) kopiert und dann entschlüsselt. Die Größe (Count) der Datenmenge ist unterschiedlich und wird im Archivformat (Abschnitt 3.2) angegeben.

```
UInt64 FilePosition;       //file position before reading

UInt32 Count;              //size to read from file
Byte[] ByteArray;          //read from file

UInt32 CodeValueCount;     //Program 4.8

UInt32 ValuePosition = (UInt32) (FilePosition - FilePosition / CodeValueCount *
                                 CodeValueCount);

UInt32 ValueCount = CodeValueCount - ValuePosition;

UInt32 Index = 0;

while(true) {
  if(ValueCount > Count - Index) ValueCount = Count - Index;

  for(UInt32 i = 0; i < ValueCount; i++)
     ByteArray[Index++] ^= this.CodeValueMemory[ValuePosition++];

  if(Index == Count) break;

  ValuePosition = 0;
  ValueCount = CodeValueCount;
}
```

Programm 4.10.1: Verwendung des Entschlüsselungsspeichers

4.10.2 Zusätzlicher Entschlüsselungsspeicher

Der einfache und zusätzliche Entschlüsselungsspeicher wird für alle weiteren Endkodierungen verwenden. Der Speicher wird mit der Anzahl aus dem Abschnitt 4.9 und dem Programm 4.7.3 erstellt.

```
this.CodePageMemory; //create with Program 4.7.3, Count = CodePageCount (4.9)
```

Ist der zusätzliche Entschlüsselungsspeicher vorhanden, wird das Programm 4.10.2 für die Endkodierungen aller weiteren Daten verwendet. Die Dateiposition (FilePosition) muss vor dem Lesen der Daten bestimmt werden. Die Originaldaten werden in die Menge (ByteArray) kopiert und dann entschlüsselt. Die Größe (Count) der Datenmenge ist unterschiedlich und wird im Archivformat (Abschnitt 3.2) angegeben.

```
UInt64 FilePosition;       //file position before reading

UInt32 Count;              //size to read from file
Byte[] ByteArray;          //read from file

UInt32 CodeValueCount;     //Program 4.8
```

```
UInt32 PagePosition = (UInt32) (FilePosition / CodeValueCount);
UInt32 ValuePosition = (UInt32) (FilePosition - PagePosition * CodeValueCount);

UInt32 ValueCount = CodeValueCount - ValuePosition;

UInt32 Index = 0;

while(true) {
  if(ValueCount > Count - Index) ValueCount = Count - Index;

  Byte Codepage = this.CodePageMemory[PagePosition];

  for(UInt32 i = 0; i < ValueCount; i++)
     ByteArray[Index++] ^= (Byte) (FilePosition++ *
                                   this.CodeValueMemory[ValuePosition++] +
                                   Codepage);

  if(Index == Count) break;

  PagePosition++;

  ValuePosition = 0;
  ValueCount = CodeValueCount;
}
```

Programm 4.10.2: Verwendung der beiden Entschlüsselungsspeicher

4.10.3 Beispiel zum Auslesen der verschlüsselten Daten

In den weiteren Formatbeschreibungen müssen die Daten entschlüsselt ausgelesen werden. Weil die Entschlüsselung als eine Menge von Bytes definiert ist, kann hierfür die Klasse "BinaryReader" verwendet werden. Die Menge (ByteArray) wird in den Programmteilen 4.10.1 und 4.10.2 erzeugt. Die Anzahl der Bytes ergibt sich aus der Größe, der zu lesenden Daten.

```
BinaryReader HeaderReader; //System.IO.BinaryReader
MemoryStream HeaderStream; //System.IO.MemoryStream

if(this.CodePageMemory == null) {
  UInt32 Count;      //size to read from file

  Byte[] ByteArray; //create with Program 4.10.1

  HeaderStream = new MemoryStream(ByteArray);
} else {
  UInt32 Count;      //size to read from file

  Byte[] ByteArray; //create with Program 4.10.2

  HeaderStream = new MemoryStream(ByteArray);
}

HeaderReader = new BinaryReader(HeaderStream);
```

Programm 4.10.3: Auslesen der verschlüsselten Daten

5 Informationen zum Archivformat

Das Archivformat muss mit dem Programm 4.10.3 entschlüsselt werden. Es kann Ordner- oder Dateiinformationen enthalten.

5.1 Die Schleife

Das Archivformat wird sooft ausgelesen, bis das Dateiende erreicht ist. Wird das Ende vorher oder nicht vollständig erreicht ist die Datei fehlerhaft.

5.2 Größe des Archivformats

Die Größe des Archivformats in Bytes, ohne diesen 16Bit Wert. Wird ein Dateiformat verwenden, ist die Dateigröße nicht in der Archivgröße enthalten. Der Wert kann nicht kleiner als 4 oder größer als 535 Bytes sein.

5.3 Zustände im Archivformat

Die Zustände sind als Bits definiert. Wie die Bits verwendet werden, wird später beschrieben.

Name	Werte	Beschreibung
Folder	0x80	Ordner
FolderIndexInt32	0x40	Ordnernummer als 32Bit Wert
NameWChar	0x20	Zeichen im Namen als 16Bit Werte
NameSizeUInt16	0x10	Speichergröße vom Namen als 16Bit Wert
FileSizeUInt64	0x03	Angabe der Dateigröße mit einem 64Bit Wert
FileSizeUInt32	0x02	Angabe der Dateigröße mit einem 32Bit Wert
FileSizeUInt16	0x01	Angabe der Dateigröße mit einem 16Bit Wert
FileSizeByte	0x00	Angabe der Dateigröße mit einem 8Bit Wert

Tabelle 5.3: Zustände im Archivformat

5.4 Verzweigung zum bestimmten Archivformat

Ist der Zustand "Folder" (Tabelle 5.3) enthalten, wird das Archivorderformat verwendet. Ansonsten das Archivdateiformat.

```
BinaryReader HeaderReader; //create with program 4.10.3, Count = 2
UInt16 HeaderSize = HeaderReader.ReadUInt16();

HeaderReader; //create again, Count = HeaderSize

Byte HeaderFlags = HeaderReader.ReadByte();

if((HeaderFlags & 0x80) == 0x80) //Table 5.3: Folder
  //jump to folder format
else
  //jump to file format
```

Programm 5.4: Verzweigung zum bestimmten Archivformat

6 Informationen zum Archivordnerformat

6.1 Ordnernummer

Die Ordnernummer kennzeichnet die Zugehörigkeit des Ordners. Ist der Wert -1 wird der Ordner als Stammordner verwenden, ansonsten ist der Index vom übergeordneten Ordner enthalten. Alle Ordner werden durch nummeriert. Der erste Ordner hat die Nummer 0. Ist der Zustand "FolderIndexInt32" (Tabelle 5.3) vorhanden, wird ein 32Bit Wert verwendet andernfalls ein 8Bit Wert.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4
Int32 FolderIndex;

if((HeaderFlags & 0x40) == 0x40) //Table 5.3: FolderIndexInt32
  FolderIndex = HeaderReader.ReadInt32(); //Int32
else
  FolderIndex = HeaderReader.ReadSByte(); //Int8
```

Programm 6.1: Ordnernummer

6.2 Ordnernamengröße

Die Speichergröße vom Ordnernamen wird in Bytes angegeben. Ist der Zustand "NameSizeUInt16" (Tabelle 5.3) enthalten, wird ein 16Bit Wert verwendet andernfalls ein 8Bit Wert. Der Wert kann nicht 0 oder größer als 520 Bytes sein.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt16 NameSize;

if((HeaderFlags & 0x10) == 0x10) //Table 5.3: NameSizeUInt16
  NameSize = HeaderReader.ReadUInt16(); //UInt16
else
  NameSize = HeaderReader.ReadByte(); //UInt8
```

Programm 6.2: Ordnernamengröße

6.3 Ordnername

Ist der Zustand "NameWChar" (Tabelle 5.3) vorhanden, werden 16Bit Zeichen verwendet ansonsten 8Bit Zeichen. Die Speichergröße wird in Abschnitt 6.2 bestimmt.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt16 NameSize;           //see Program 6.2

String Name = String.Empty;

if((HeaderFlags & 0x20) == 0x20) { //Table 5.3: NameWChar
  for(UInt16 i = 0; i < NameSize / 2; i++)
    Name += (Char) HeaderReader.ReadUInt16(); //WChar
} else {
  for(UInt16 i = 0; i < NameSize; i++)
    Name += (Char) HeaderReader.ReadByte();   //Char
}
```

Programm 6.3: Ordnername

7 Informationen zum Archivdateiformat

7.1 Ordnernummer

Die Nummer kennzeichnet den Archivordner, in dem die Datei vorhanden ist. Alle Ordner werden durch nummeriert. Der erste Ordner hat die Nummer 0. Ist der Zustand "FolderIndexInt32" (Tabelle 5.3) vorhanden, wird ein 32Bit Wert (Int32) verwendet andernfalls ein 8Bit Wert (Int8). Der Aufbau ist identisch mit dem Programm 6.1.

7.2 Dateigröße

Die Dateigröße ist in Bytes angegeben. Eine Datei kann die Größe 0 haben. Der Zustand (Tabelle 5.3) wird mit dem Wert 0x03 hex. Und verknüpft. Das Ergebnis ist eine Zahl von 0 bis 3. In der Tabelle 5.3 werden die entsprechenden Nummern mit "FileSizeByte", "FileSizeUInt16", "FileSizeUInt32" und "FileSizeUInt64" gekennzeichnet. Die Dateigröße muss entweder als 8Bit, 16Bit, 32Bit oder 64Bit Wert gelesen werden.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt64 FileSize;
```

```
switch(HeaderFlags & 0x03) { //FileSizeMask = 0x03
  case 0: { //Table 5.3: FileSizeByte
    FileSize = HeaderReader.ReadByte(); //Byte
    break;
  }
  case 1: { //Table 5.3: FileSizeUInt16
    FileSize = HeaderReader.ReadUInt16(); //UInt16
    break;
  }
  case 2: { //Table 5.3: FileSizeUInt32
    FileSize = HeaderReader.ReadUInt32(); //UInt32
    break;
  }
  case 3: { //Table 5.3: FileSizeUInt64
    FileSize = HeaderReader.ReadUInt64(); //UInt64
    break;
  }
}
```

Programm 7.2: Dateigröße

7.3 Dateinamengröße

Die Speichergröße vom Dateinamen wird in Bytes angegeben. Ist der Zustand "NameSizeUInt16" (Tabelle 5.3) enthalten, wird ein 16Bit Wert verwendet andernfalls ein 8Bit Wert. Der Wert kann nicht 0 oder größer als 520 Bytes sein. Der Aufbau ist identisch mit dem Programm 6.2.

7.4 Dateiname

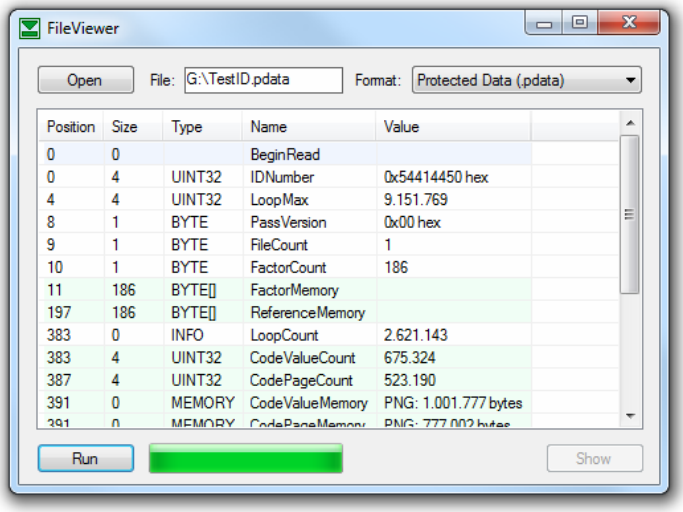
Ist der Zustand "NameWChar" (Tabelle 5.3) vorhanden, werden 16Bit Zeichen verwendet ansonsten 8Bit Zeichen. Die Speichergröße wird in Abschnitt 7.3 bestimmt. Der Aufbau ist identisch mit dem Programm 6.3.

7.5 Dateispeicher

Die Größe des Dateispeichers wird in Abschnitt 7.2 bestimmt. Die Daten können mit dem Programm 4.10.1 oder 4.10.2 ausgelesen werden. Es empfiehlt sich die Dateidaten in mehreren Teilen zu entschlüsselt. Zum Beispiel mit einer Speichergröße von 66000 Bytes.

8 Programm zum Auslesen des Dateiformats

Auf der Internetseite von PanotiSoft ist unter technische Dokumente ein Testprogramm vorhanden, mit dem das Dateiformat strukturiert ausgelesen werden kann. Zusätzlich kann auch der Programmkode herunter geladen werden. Das Programm wurde unter Visual Studio 2008 mit der Programmiersprache C# geschrieben.



Programm: FileViewerX64.zip
FileViewerX32.zip

Projektdatei: FileViewerCode.zip

Beschreibung: FileViewer.pdf

FileViewerCode:

Formatdatei: FileViewerFormat.cs
Formatklasse: FileViewerProtectedData