```
1  ////////////////////////////////////////////////////////////////////////////
2  // X-Tension API - template for new X-Tensions
3  // Copyright X-Ways Software Technology AG
4  // Please consult
5  // http://x-ways.com/forensics/x-tensions/api.html
6  // for current documentation
7
8  ////////////////////////////// BPListParser v. 0.1 ////////////////////////////
9  //   Dieses Projekt entstand im Rahmen einer Bachelorarbeit im Studiengang      //
10 //   Allgemeine und Digitale Forensik an der Hochschule Mittweida im Jahr       //
11 //   2018. Betreut wurde diese Arbeit durch Prof. Christian Hummert und         //
12 //   Prof. Dirk Pawlaszczyk. Der Quellcode darf unter Beachtung des Copyright   //
13 //   der X-Ways Software Technology AG kostenlos verwendet, verändert oder      //
14 //   weitergegeben werden. Es gibt keine Gewährleistung auf Funktionalität      //
15 //   oder Aktualisierung.   gez. Kittan Michael     michaelkittan@gmx.de        //
16 ////////////////////////////////////////////////////////////////////////////////
17
18 #include "x-tension.h"
19 #include <math.h>
20 #include <stdint.h>
21 #include <time.h>
22 #include <wchar.h>
23
24 #define EXPORT extern "C" __declspec(dllexport)
25
26 ////////////////////Funktionen definieren/////////////////
27 // Parsen INT Counters ( rechtes Nibble, Offset, Buffer der Datei, Rückgabewert Int
   Zahl )
28 uint64_t parseint(int objinfo, uint64_t inhalt, BYTE *file_buf);
29
30 // Containerdateitypen verarbeiten ( rechtes Nibble, Objekt in Offset Tabelle,
   Datei, Offsets der Offsettabelle, Ausgabebuffer, Rückgabewert: Ausgabebuffer)
31 wchar_t* parseArray(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
   markeroffset[4500], wchar_t *buffer);
32 wchar_t* parseSet(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
   markeroffset[4500], wchar_t *buffer);
33 wchar_t* parseDict(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
   markeroffset[4500], wchar_t *buffer);
34
35 // Inhalte zum Ausgabebuffer hinzufügen ( Zielbufferpointer, Zeichenanzahl zum
   zählen, Inhalt, wieviele Zeichen zusätzlich, Rückgabewert: Ausgabebuffer )
36 wchar_t *Ausgabe(wchar_t *buffer, const wchar_t *Zeile, wchar_t *Ausgabebuf, int
   zeichencount);
37
38 // Aufruf der Objektauswahl ( Datei, Offsets der Offset Tabelle, Objekt der Offset
   Tabelle, Ausgabebuffer
39 wchar_t *parseobjekt(BYTE *file_buf, uint64_t markeroffset[4500], int markernummer,
   wchar_t *buffer);
40
41 ////////////////////// Globale Variablendefinition //////////////////////
42 size_t SpeicherZugeordnet = 512;     // Reservierter Speicher
43 size_t Speichernutzung = 0;          // Aktuelle Speichernutzung
44 HANDLE hitem1 = 0;                   // Variable für Hitem Global
45 size_t file_size = 0;                // Dateigröße
46 int static offsize = 0, refsize = 0;// Offsize und efsize Global
47 INT64 numobjekt = 0;                 // Anzahl gespeicherter Objekte in der Datei
48 INT64 markerOfsettable = 0;          // Marker für Verschiebung der Offsettable
49 wchar_t Stringbuffer[3000];          // Buffer für String Ausgabe
50 int64_t negativint = 0;              // Variable für negativ Angabe Integer
51
52
53                                      // Variablen für die Zählung und Anzeige der
                                        Array,Set oder Dict
54 int ebene = 0;                       // Zähler für die Ebene
55 int ebenenlaenge[40];                // Speicherung der Länge für die Ebene
56 int stringlength = 0;                // Länge der einzelnen Strings
57 wchar_t tiefenanzeige[3000];         // Speicherung der kompletten Ebenenanzeige
58 size_t nutzungebene = 10;            // Anfangslänge der Ebenausgabe
59 bool table = 0;                      // Tabellenausgabe ja oder nein
60
61
62 EXPORT LONG __stdcall XT_Init(DWORD nVersion, DWORD nFlags, HANDLE hMainWnd,
63     void* lpReserved)
```

```
64  {
65      // Initialisieren der X-Ways API Funktionen
66      XT_RetrieveFunctionPointers();
67      XWF_OutputMessage(L"Viewer X-Tension BPList Parser von Kittan Michael v0.1
        geladen", 0);
68      return 1;
69  }
70  EXPORT PVOID __stdcall XT_View(HANDLE hItem, LONG nItemID, HANDLE hVolume, HANDLE
    hEvidence, PVOID lpReserved, PINT64 nResSize)
71  {
72      using namespace std;
73
74      // hitem1 und file_size global setzen
75      hitem1 = hItem;
76      file_size = (size_t)XWF_GetSize(hItem, (LPVOID)1);
77
78      wsprintf(tiefenanzeige, L"<b>Ebene:");
79
80      ////////////////////// Prüfen X-Tension Zuständig ////////////////////
81      *nResSize = -1;     // Rückgabewert das die Datei nicht mit dieser Xtention
        betrachtet werden soll
82
83      char sig_buffer[9] = {};
84      XWF_Read(hItem, 0, (BYTE*)sig_buffer, 8);
85
86      if (strcmp("bplist00", sig_buffer) != 0)
87      {
88          return NULL;
89      }
90
91      wchar_t *buffer = (wchar_t*)calloc(SpeicherZugeordnet,  // Anzahl zu
        reservierenden Elemente
92          sizeof(wchar_t));                                   // Größe des jeweeilgen
            Elements ( Hier 2 Byte für wchar_t)
93
94                                                              // Buffer mit Nullen
                                                                füllen
95      wmemset(buffer, 0, SpeicherZugeordnet);
96
97      // Prüfen auf mögliche Fehler beim Einlesen der Datei
98      if (buffer == NULL)
99      {
100         XWF_OutputMessage(L"BPListParser ERROR: Fehler beim Zuordnen des Speichers",
            0);
101         return NULL;
102     }
103
104     if (file_size <= 40)
105     {
106         *nResSize = 150;
107         wsprintf(buffer, L"ERROR: Dieses BPLIST File scheint Fehlerhaft zu
            sein(Kleiner als 40 Byte)");
108         return buffer;
109     }
110
111     if (file_size >= 100000)
112     {
113         *nResSize = 124;
114         wsprintf(buffer, L"ERROR: Dateien größer als 100 KB werden noch nicht
            unterstützt");
115         return buffer;
116     }
117
118     *nResSize = -2;     // -2 gibt Fehler beim Abarbeiten der Xtention an
119
120                         ////////////////////// Trailer auslesen ////////////////////
121
122     HANDLE heap = GetProcessHeap();
123     void *ptr;                      // Pointer
124     INT64  offsetot = 0;            // das Offset der Offsettable
125
126     BYTE *trailer = (BYTE*)HeapAlloc(heap, 0, file_size);
127
```

```
128         XWF_Read(hItem, file_size - 32, trailer, file_size);
129
130         // Null Bytes von index 0 -5
131         offsize = trailer[6];
132         refsize = trailer[7];
133
134         // numobjekt            Anzahl der Kodierten Objekte
135         ptr = &trailer[8];      // Zeiger zum Auslesen des 64 Bit Wertes
136         numobjekt = _byteswap_uint64(*((uint64_t *)ptr));
137
138         // topobjekt
139         ptr = &trailer[16];
140         markerOfsettable = _byteswap_uint64(*((uint64_t *)ptr));
141
142         // Offset Tabellen Offset
143         ptr = &trailer[24];
144         offsetot = _byteswap_uint64(*((uint64_t *)ptr));
145
146         if (offsetot > file_size || offsetot <9)
147         {
148             *nResSize = 90;
149             wsprintf(buffer, L"ERROR: Falsche Angabe des Offsets im Trailer");
150             HeapFree(heap, 0, trailer);
151             return buffer;
152         }
153
154         HeapFree(heap, 0, trailer);         // Speicher vom Trailer wieder Freigeben
155
156                                             // ---------------------- Offsets Tabelle
                                                auslesen -----------------------
157
158         BYTE *file_buf = (BYTE*)HeapAlloc(heap, 0, file_size);
159
160         XWF_Read(hItem,             // Handle auf die Datei
161             offsetot,               // von wo an gelesen werden soll
162             file_buf,               // wo rein geschrieben werden soll
163             file_size);             // bis wohin gelesen werden soll
164
165                                     // in dem OffsetArray stehen danach alle Offsets in
                                        der Datei wo ein Objekt beginnt, das i gibt dabei
                                        die Nummer des Objekts an ( Marker )
166         uint64_t markeroffset[4500];
167         if (numobjekt > 4500)
168         {
169             *nResSize = 90;
170             wsprintf(buffer, L"ERROR: Die BPLIST hat mehr als 4500 Objekte.");
171             HeapFree(heap, 0, file_buf);
172             return buffer;
173         }
174         for (int i = 0; i < numobjekt; i++)
175         {
176             if (offsize == 1)
177             {
178                 markeroffset[i] = file_buf[i];
                    // 8 Bit lesen
179             }
180             else if (offsize == 2)
                // 16 Bit lesen
181             {
182                 markeroffset[i] = (uint64_t)(((uint16_t *)file_buf)[i]);
183                 markeroffset[i] = _byteswap_ushort((uint16_t)markeroffset[i]);
184             }
185             else if (offsize == 4)
186             {
187                 markeroffset[i] = (uint64_t)(((uint32_t *)file_buf)[i]);
                    // 32 Bit lesen
188                 markeroffset[i] = _byteswap_ulong((uint32_t)markeroffset[i]);
189             }
190             else if (offsize == 8)
191             {
192                 markeroffset[i] = (uint64_t)(((uint64_t *)file_buf)[i]);
                    // 64 Bit lesen
193                 markeroffset[i] = _byteswap_uint64((uint64_t)markeroffset[i]);
```

```
194             }
195             else
196             {
197                 *nResSize = 110;
198                 wsprintf(buffer, L"ERROR: Fehler in der Datei, nicht unterstützte
                    Offsize");
199                 HeapFree(heap, 0, file_buf);
200                 return buffer;
201             }
202
203             // Prüfen Werte der Offset Tabelle
204             if (markeroffset[i] > file_size)
205             {
206                 *nResSize = 94;
207                 wsprintf(buffer, L"ERROR: Fehler beim Auslesen der Offset Tabelle");
208                 HeapFree(heap, 0, file_buf);
209                 return buffer;
210             }
211         }
212         HeapFree(heap, 0, file_buf);            // den File Buffer wieder freigeben
213
214                                                 // ----------------------- Auswerten
                                                    derObjectTable
                                                    ----------------------------
215
216         file_buf = (BYTE*)HeapAlloc(heap, 0, file_size);
217         XWF_Read(hItem, 0, file_buf, file_size);            // Komplette Datei wird in
            file_buf geschrieben
218
219                                                             /// HTML Einbindung
220         wchar_t Ausgabebuf[500];    // Temporärer Buffer für Ausgabe
221         buffer[0] = 0xFeFF;         // BOW für HTML Code
222         Speichernutzung = 1;        // Zähler wieviel Byte im Buffer aktuell gespeichert
            sind
223         int zeichencount = 1;       // Zähler für Länge des Int Wertes
224         uint64_t intlaengeu;        // Variable zum Berechnen der Zeichenlänge eines
            Integers
225
226         wsprintf(Ausgabebuf, L"<HTML><BODY><center><b> BPLIST File: %s</b>
            </center><br>", XWF_GetItemName(nItemID));
227         buffer = Ausgabe(buffer, L"<HTML><BODY><center><b> BPLIST File: </b>
            </center><br>", Ausgabebuf, wcslen(XWF_GetItemName(nItemID)));
228
229         intlaengeu = numobjekt;
230         while ((intlaengeu = intlaengeu / 10) != 0) zeichencount++;
231
232         wsprintf(Ausgabebuf, L"Anzahl gespeicherter Objekte in der Objekt Tabelle:
            %d<br><br>", numobjekt);
233         buffer = Ausgabe(buffer, L"Anzahl gespeicherter Objekte in der Objekt Tabelle:
            <br><br>", Ausgabebuf, zeichencount);
234
235         // ---------------------------
236
237         // Aufruf des 0 Objekts
238         buffer = parseobjekt(file_buf,      // Gespeicherter Dateiinhalt
239             markeroffset,                   // Gespeicherte Offsets der Offset Tabelle
240             0,                              // Objektnummer
241             buffer);                        // Buffer in den geschrieben wird
242
243                                             // Html schließen
244         wsprintf(Ausgabebuf, L"</BODY></HTML>");
245         buffer = Ausgabe(buffer, L"</BODY></HTML>", Ausgabebuf, 0);
246
247         *nResSize = Speichernutzung * 2;    // mal 2 weil wchar
248
249         return buffer;
250     }
251
252     wchar_t *parseobjekt(BYTE *file_buf, uint64_t markeroffset[4500], int markernummer,
        wchar_t *buffer) {
253
254         // --------------------- Variablen Definieren ---------------------
255         uint64_t inhalt = markeroffset[markernummer];   // Offset des Objekts ( Alle
```

```
        Werte der Offsetable )
256     uint64_t ergebnis = 0;                  // Ergebnisvariable
257     int objtyp = 0;                         // Objekt Typen Variable festlegen z.B: int,
        fil, bool
258     int objinfo = 0;                        // Info zu dem Typ festlegen z.B. Länge
259     struct tm ts;                           // Hilfsvariable Datumsausgabe
260     double datum;                           // Hilfsvariable für Datumsausgabe
261     int length = 0;                         // Byte länge wenn Int Counter
262     wchar_t *StringZeiger = Stringbuffer;   // Zeiger für ASCII Ausgabe
263     float ergfloat = 0;                     // Real 4 Byte Variable
264     double ergdouble = 0;                   // Real 8 Byte Variable
265     void *ptr = 0;                          // Pointer für UID und Byteswap
266     uint64_t position = 0;                  // Position für Auslesen der Werte
267     wchar_t Ausgabebuf[2000];               // Temporärer Buffer für Ausgabe
268     int64_t intlaenge = 0;                  // Hilfsvariable bei Zeichenberechnung von
        Int 8 Byte Werten
269     uint64_t intlaengeu = 0;                // Hilfsvariable bei Zeichenberechnung von
        Int 8 Byte Werten unsignd
270     HANDLE heap1;                           // Zusätzlichen Speicher initialisieren
271     char *file_tmp;                         // Variable für Temp Buffer
272     char* hex;                              // Pointer für Data Ausgabe
273     int j = 0;                              // Hilfszählvariable
274     int zeichencount = 0;                   // Zählvariable für Länge der Int Zahlen
275     wchar_t *file_tmp16;                    // Filetemp 16 Bit für UNICODE
276     HANDLE heapunicode;                     // Speicher für Unicode initalisieren
277
278                                             // -------------------- Auswahl des
                                                Objekttyps ---------------------
279
280     objtyp = (file_buf[inhalt] & 0xF0);         // Bitweise UND Verknüpfung
281     objtyp = objtyp >> 4;                       // 4 Bits nach rechts verschieben
282     objinfo = (file_buf[inhalt] & 0x0F);        // zweites Nibble Speichern
283
284     switch (objtyp)
285     {
286     case 0x0:
287         switch (objinfo)
288         {
289         case 0x0:
290             //  XWF_OutputMessage(L" Datentyp: NULL ", 0);
291             wsprintf(Ausgabebuf, L"NA");
292             return Ausgabe(buffer, L"NA", Ausgabebuf, 0);
293
294         case 0x8:
295             //  XWF_OutputMessage(L" Datentyp: BOOL = FALSE", 0);
296             wsprintf(Ausgabebuf, L"Bool</td><td>False");
297             return Ausgabe(buffer, L"Bool</td><td>False", Ausgabebuf, 0);
298
299         case 0x9:
300             //XWF_OutputMessage(L" Datentyp: BOOL = TRUE", 0);
301             wsprintf(Ausgabebuf, L"Bool</td><td>True");
302             return Ausgabe(buffer, L"Bool</td><td>True", Ausgabebuf, 0);
303
304         case 0xF:
305             //XWF_OutputMessage(L" Datentyp: FILL BYTE", 0);
306             wsprintf(Ausgabebuf, L"Fill Byte");
307             return Ausgabe(buffer, L"Fill Byte", Ausgabebuf, 0);
308
309         default:
310             XWF_OutputMessage(L"SIMPLE PARSE - Nichts\n", 0);
311             wsprintf(Ausgabebuf, L"NA");
312             return Ausgabe(buffer, L"NA", Ausgabebuf, 0);
313         }
314
315     case 0x1:                                                   // Datentyp INT
316         ergebnis = parseint(objinfo,    // Länge der Folgenden Bytes
317             inhalt,                     // Position innerhalb der Datei
318             file_buf);                  // Buffer mit der Datei
319
320         if (negativint < 0)
321         {
322             intlaenge = negativint;
323
```

```
324                 zeichencount = 2;
325                 while ((intlaenge = intlaenge / 10) != 0)   zeichencount++;
326
327                 wsprintf(Ausgabebuf, L"Integer:</td><td>%d", negativint);
328                 negativint = 0;
329                 return Ausgabe(buffer, L"Integer:</td><td>", Ausgabebuf, zeichencount);
330             }
331             else
332             {
333                 intlaengeu = ergebnis;
334                 zeichencount = 1;
335                 while ((intlaengeu = intlaengeu / 10) != 0) zeichencount++;
336
337                 wsprintf(Ausgabebuf, L"Integer:</td><td>%I64u", ergebnis);
338                 return Ausgabe(buffer, L"Integer:</td><td>", Ausgabebuf, zeichencount);
339             }
340
341         case 0x2:                                                   // Datentyp
            Real
342             length = 0x1 << objinfo;
343             inhalt++;
344             if (length == 4)
345             {
346                 ergebnis = file_buf[inhalt] << 24;
347                 ergebnis = ergebnis + (file_buf[inhalt + 1] << 16);
348                 ergebnis = ergebnis + (file_buf[inhalt + 2] << 8);
349                 ergebnis = ergebnis + (file_buf[inhalt + 3]);
350
351                 memcpy(&ergfloat, &ergebnis, sizeof(float));
352
353                 intlaengeu = (long int)ergfloat;
354                 swprintf(Ausgabebuf, 200, L"REAL:</td><td>%f", ergfloat);
355
356             }
357             else if (length == 8) {
358
359                 ergebnis = file_buf[inhalt] << 24;
360                 ergebnis = ergebnis + (file_buf[inhalt + 1] << 16);
361                 ergebnis = ergebnis + (file_buf[inhalt + 2] << 8);
362                 ergebnis = ergebnis + ((file_buf[inhalt + 3]));
363                 ergebnis = ergebnis << 32;
364                 ergebnis = ergebnis + (file_buf[inhalt + 4] << 24);
365                 ergebnis = ergebnis + (file_buf[inhalt + 5] << 16);
366                 ergebnis = ergebnis + (file_buf[inhalt + 6] << 8);
367                 ergebnis = ergebnis + (file_buf[inhalt + 7]);
368
369                 memcpy(&ergdouble, &ergebnis, sizeof(double));
370
371                 intlaengeu = (long int)ergdouble;
372
373                 swprintf(Ausgabebuf, 200, L"REAL:</td><td>%f", ergdouble);
374             }
375             else
376             {
377                 wsprintf(Ausgabebuf, L"REAL</td><td> Falsche Byte Länge:%3d Byte lang",
                    length);
378                 return Ausgabe(buffer, L"REAL</td><td> Falsche Byte Länge:%3d Byte
                    lang", Ausgabebuf, 0);
379             }
380
381
382             zeichencount = 1;
383             if (intlaengeu == 0) zeichencount++;
384
385             // Zeichen zählen mit 6 Nachkommastellen
386             else if (ergdouble < 0 || ergfloat < 0)     // Vorzeichenstelle
387             {
388                 zeichencount++;
389                 intlaengeu = intlaengeu * (-1);
390             }
391
392             while ((intlaengeu = intlaengeu / 10) != 0)
393                 zeichencount++;
```

```
394
395             zeichencount = zeichencount + 7;        // 6 Kommastellen + 1 Punkt
396
397             return Ausgabe(buffer, L"REAL:</td><td>", Ausgabebuf, zeichencount);
398
399         case 0x3:                                           // Date Format
400             if (objinfo != 3)
401             {
402                 wsprintf(Ausgabebuf, L"Datum</td><td> Ungültiges Datum");
403                 return Ausgabe(buffer, L"Datum</td><td> Ungültiges Datum", Ausgabebuf, 0);
404             }
405             ptr = &file_buf[inhalt + 1];                // Übergibt Zeiger vom Beginn
                des Datums
406
407             ergebnis = _byteswap_uint64(*((uint64_t *)ptr));
408             memcpy(&datum, &ergebnis, 8);               // Kopiert von Ergebnis in Datum
                ohne Konvertierung
409                                                         // 978307200 = Datum Zeit Format
                                                            geparst vom 01.01.2001 GMT
410             datum += 978307200;                         // + EPOCH Time berechnen
411             ergebnis = (uint64_t)datum;
412
413             localtime_s(&ts, (const time_t *)&ergebnis);
414
415             wcsftime(Ausgabebuf, 150, L"Datum:</td><td>%a %Y-%m-%d %H:%M:%S", &ts);
416             return Ausgabe(buffer, L"Datum:</td><td>                        ",
                Ausgabebuf, 0);
417
418         case 0x4:                                                   // Datenty DATA
419             heap1 = GetProcessHeap();   // Neuen TeBuffer definieren
420             file_tmp = (char*)HeapAlloc(heap1, 0, file_size);
421             hex = file_tmp;
422             if (objinfo != 0xf)
423             {
424                 ergebnis = objinfo;
425                 position = inhalt + 1;
426             }
427             else if (objinfo == 0xf)
428             {
429                 objinfo = (file_buf[inhalt + 1] & 0x0F);
430                 length = (int)pow(2, objinfo);
431                 ergebnis = parseint(objinfo, inhalt + 1, file_buf);
432                 position = inhalt + 2 + length;
433             }
434
435             if (position + ergebnis > file_size)
436             {
437                 wsprintf(Ausgabebuf, L"ERROR: Falsche Byte Länge beim Typ Data");
438                 buffer = Ausgabe(buffer, L"ERROR: Falsche Byte Länge beim Typ Data",
                    Ausgabebuf, 0);
439                 HeapFree(heap1, 0, file_tmp);
440                 return buffer;
441             }
442
443             // File_temp befüllen
444             XWF_Read(hitem1, position, (BYTE *)file_tmp, (DWORD)(position + ergebnis));
445
446             ergebnis = ergebnis * 2;
447
448             if (ergebnis> 400)
449             {
450                 wsprintf(Ausgabebuf, L"Data mit %8d Byte Anzeige der ersten
                    200</td><td>", ergebnis / 2);
451                 buffer = Ausgabe(buffer, L"Data mit          Byte Anzeige der ersten
                    200</td><td>", Ausgabebuf, 0);
452                 ergebnis = 400;
453             }
454             else
455             {
456                 wsprintf(Ausgabebuf, L"Data mit %8d Byte</td><td>", ergebnis / 2);
457                 buffer = Ausgabe(buffer, L"Data mit          Byte</td><td>", Ausgabebuf,
                    0);
458             }
```

```
459
460             j = 0;
461             for (int i = 0; i < ergebnis; i++)  // Zeichen einzeln in den Buffer
                schreiben ( *2 wegen whitechar)
462             {
463                 wsprintf(Ausgabebuf + i * 2 + j, L"%02X", (*(hex + i) & 0xFF));
464                 // Leerzeichen alle 16 Bytes
465                 if (((i + 1) % 16 == 0) && i != 0)
466                 {
467                     j = j + 2;
468                     wsprintf(Ausgabebuf + i * 2 + j, L" ");
469                 }
470             }
471
472             // Erweiterung des Speichers wenn Speicherbedarf zu Groß
473             while (Speichernutzung + ergebnis + j / 2 >= SpeicherZugeordnet)
474             {
475                 // Exponentielle Vergößerung
476                 SpeicherZugeordnet *= 2;
477
478                 // Speicherzuordnung erfolgreich
479                 buffer = (wchar_t *)realloc(buffer, SpeicherZugeordnet * sizeof(wchar_t));
480                 if ((buffer == NULL))
481                 {
482                     XWF_OutputMessage(L"BPListParser ERROR: Fehler beim Zuordnen des
                    größeren Speichers", 0);
483                     HeapFree(heap1, 0, file_tmp);
484                     return NULL;
485                 }
486             }
487
488             // Neuen Inhalt an Speicher anhängen
489             wmemcpy(buffer + Speichernutzung        // Zielbuffer
490                 , Ausgabebuf                        // Quellbuffer
491                 , (size_t)ergebnis + j / 2 - 1);        // Anzahl der zu kopierenden
                    Zeichen
492
493
494
495                                                     // Aktuelle Speichernutzung
                                                        festhalten
496             Speichernutzung += (size_t)ergebnis + j / 2 - 1;
497             HeapFree(heap1, 0, file_tmp);
498             return buffer;
499
500         case
            0x5:
                    // ASCII
501
502             heap1 = GetProcessHeap();               // Neuen Temporären Buffer definieren
503             file_tmp = (char*)HeapAlloc(heap1, 0, file_size);
504             if (objinfo != 0xf)
505             {
506                 ergebnis = objinfo;
507                 position = inhalt + 1;
508             }
509             else if (objinfo == 0xf)
510             {
511                 objinfo = (file_buf[inhalt + 1] & 0x0F);                // objektInfo
                    neu einlesen da ein Byte weiter
512
513                 length = (int)pow(2, objinfo);
514                 ergebnis = parseint(objinfo, inhalt + 1, file_buf);
515                 position = inhalt + 2 + length;
516             }
517
518             if (position + ergebnis > file_size)
519             {
520                 wsprintf(Ausgabebuf, L"ERROR: Falsche Byte Länge beim Typ String");
521                 buffer = Ausgabe(buffer, L"ERROR: Falsche Byte Länge beim Typ String",
                    Ausgabebuf, 0);
522                 HeapFree(heap1, 0, file_tmp);
523                 return buffer;
```

```
524         }
525         // File_temp befüllen von Anfang bis Ende des Strings
526         XWF_Read(hitem1, position, (BYTE *)file_tmp, (DWORD)(position + ergebnis));
527
528         wsprintf(Ausgabebuf, L"String:</td><td>");
529         buffer = Ausgabe(buffer, L"String:</td><td>", Ausgabebuf, 0); ;
530
531         StringZeiger = Stringbuffer;
532
533         // Prüfen ob String länger als Stringbuffer sonst abschneiden
534         if (ergebnis > 1495)
535             ergebnis = 1495;
536
537         for (int i = 0; i < ergebnis; i++)
538             *(StringZeiger++) = file_tmp[i];
539
540         // Variable für EbenenAngabe
541         stringlength = (size_t)ergebnis;
542
543         // Erweiterung des Speichers wenn Speicherbedarf zu Groß
544         while (Speichernutzung + ergebnis >= SpeicherZugeordnet)
545         {
546             // Exponentielle Vergößerung
547             SpeicherZugeordnet *= 2;
548
549             // Speicherzurdnung erfolgreich
550             buffer = (wchar_t *)realloc(buffer, SpeicherZugeordnet * sizeof(wchar_t));
551             if ((buffer == NULL))
552             {
553                 XWF_OutputMessage(L"Fehler beim Zuordnen des größeren Speichers", 0);
554                 HeapFree(heap1, 0, file_tmp);
555                 return NULL;
556             }
557         }
558         // Neuen Inhalt an Speicher anhängen
559         wmemcpy(buffer + Speichernutzung        // Zielbuffer
560             , Stringbuffer                      // Quellbuffer
561             , (size_t)ergebnis);                // Anzahl der zu kopierenden Zeichen
562
563                                                 // Aktuelle Speichernutzung festhalten
564         Speichernutzung += (size_t)ergebnis;
565         //      wsprintf(Ausgabebuf, L"</td>");
566         //      buffer = Ausgabe(buffer, L"</td>", Ausgabebuf,0);
567         HeapFree(heap1, 0, file_tmp);
568         return buffer;
569
570     case 0x6:                                   // UNICODE
571
572                                                 // Buffer defninieren
573         heapunicode = GetProcessHeap();
574         file_tmp16 = (wchar_t*)HeapAlloc(heapunicode, 0, (uint16_t)file_size);
575         if (objinfo != 0xf)
576         {
577             ergebnis = (objinfo);
578             position = inhalt + 1;
579         }
580         else if (objinfo == 0xf)
581         {
582             objinfo = (file_buf[inhalt + 1] & 0x0F);                // objektInfo
                neu einlesen da ein Byte weiter
583             length = (int)pow(2, objinfo);
584             ergebnis = (parseint(objinfo, inhalt + 1, file_buf));
585             position = inhalt + 2 + length;
586         }
587
588         if (position + ergebnis * 2 > file_size)
589         {
590             wsprintf(Ausgabebuf, L"ERROR: Falsche Byte Länge beim Typ Unicode");
591             buffer = Ausgabe(buffer, L"ERROR: Falsche Byte Länge beim Typ Unicode",
                Ausgabebuf, 0);
592             HeapFree(heapunicode, 0, file_tmp16);
593             return buffer;
594         }
```

```
595
596             // Einlesen in file_tmp vom Ende des Intcount bis zum Ende des Strings
597             XWF_Read(hitem1, position, (BYTE *)file_tmp16, (DWORD)(position + ergebnis *
                2));
598
599             wsprintf(Ausgabebuf, L"Unicode:</td><td>");
600             buffer = Ausgabe(buffer, L"Unicode:</td><td>", Ausgabebuf, 0);
601
602             StringZeiger = Stringbuffer;
603
604             if (ergebnis > 1495)
605                 ergebnis = 1495;
606
607             for (int i = 0; i < ergebnis; i++)
608                 *(StringZeiger++) = _byteswap_ushort((uint16_t)file_tmp16[i]);
609
610             // Variable für EbenenAngabe
611             stringlength = (size_t)ergebnis;
612
613             while (Speichernutzung + ergebnis >= SpeicherZugeordnet)
614             {
615                 // Exponentielle Vergößerung
616                 SpeicherZugeordnet *= 2;
617
618                 // Speicherzuordnung erfolgreich
619                 buffer = (wchar_t *)realloc(buffer, SpeicherZugeordnet * sizeof(wchar_t));
620                 if ((buffer == NULL))
621                 {
622                     XWF_OutputMessage(L"Fehler beim Zuordnen des größeren Speichers", 0);
623                     HeapFree(heapunicode, 0, file_tmp16);
624                     return NULL;
625                 }
626             }
627
628             // Neuen Inhalt an Speicher anhängen
629             wmemcpy(buffer + Speichernutzung        // Zielbuffer
630                 , Stringbuffer                      // Quellbuffer
631                 , (size_t)ergebnis);                // Anzahl der zu kopierenden Zeichen
632
633                                                     // Aktuelle Speichernutzung festhalten
634             Speichernutzung += (size_t)ergebnis;
635             HeapFree(heapunicode, 0, file_tmp16);
636
637             return buffer;
638
639         case 0x8:                                                   // Datentyp
            UID
640             ergebnis = objinfo + 1;
641
642             ptr = &file_buf[inhalt + 1];            // liest den Inhalt von dieser
                Position aus
643             switch (ergebnis) {
644             case 1:
645                 ergebnis = *((uint8_t *)ptr);               // hier kein Byteswap da nur
                    8 Bit
646                 break;
647             case 2:
648                 ergebnis = (uint64_t)_byteswap_ushort(*((uint16_t *)ptr));      //
                    Byteswap wechselt Big Endian zu Little Endian mit 16 Bit
649                 break;
650             case 4:
651                 ergebnis = (uint64_t)_byteswap_ulong(*((uint32_t *)ptr));       //
                    Byteswap wechselt Big Endian zu Little Endian mit 32 Bit
652                 break;
653             case 8:
654                 ergebnis = (uint64_t)_byteswap_uint64(*((uint64_t *)ptr));      //
                    Byteswap wechselt Big Endian zu Little Endian mit 64 Bit
655                 break;
656
657             default:
658
659                 wsprintf(Ausgabebuf, L"Uid:</td><td>Fehler beim Auslesen des UID Wertes");
660                 return Ausgabe(buffer, L"Uid</td><td>Fehler beim Auslesen des UID
```

```
                Wertes", Ausgabebuf, 0);
661         }
662
663         wsprintf(Ausgabebuf, L"Uid:</td><td>%4d", ergebnis);
664         return Ausgabe(buffer, L"Uid:</td><td>    ", Ausgabebuf, 0);
665
666     case 0xA:                                                           //
        Datentyp Array
667         buffer = parseArray(objinfo, markernummer, file_buf, markeroffset, buffer);
668         return buffer;
669
670     case 0xB:                                                           //
        Datentyp SET
671         buffer = parseSet(objinfo, markernummer, file_buf, markeroffset, buffer);
672         return buffer;
673
674     case 0xD:                                                           //
        Datentyp Dict
675         buffer = parseDict(objinfo, markernummer, file_buf, markeroffset, buffer);
676         return buffer;
677
678     default:
679         wsprintf(Ausgabebuf, L"Unbekannter Datentyp");
680         return Ausgabe(buffer, L"Unbekannter Datentyp", Ausgabebuf, 0);
681
682     }
683     wsprintf(Ausgabebuf, L"Unbekannter Datentyp");
684     return Ausgabe(buffer, L"Unbekannter Datentyp", Ausgabebuf, 0);
685
686 }
687
688 // Speicher freigeben Variablen zurücksetzen
689 EXPORT BOOL __stdcall XT_ReleaseMem(PVOID lpBuffer)
690 {
691     if (NULL != lpBuffer)
692     {
693         free(lpBuffer);
694         SpeicherZugeordnet = 512;
695         Speichernutzung = 0;
696         nutzungebene = 10;
697         ebene = 0;
698         wsprintf(tiefenanzeige, L"<b>Ebene:");
699         return true;
700     }
701     return false;
702 }
703
704 // Int Counter auswerten Rückgabewert in Int
705 // Übergabe: Länge ( vor ^2), Stelle im Buffer, Buffer
706 uint64_t parseint(int objinfo, uint64_t inhalt, BYTE *file_buf) {
707
708     uint64_t ergebnis = 0;
709
710     void *ptr = &file_buf[inhalt + 1];          // liest den Inhalt von dieser
        Position aus
711     switch (0x01 << objinfo) {                  // verschiebt den Wert bitweise nach
        Links für die Anzahl von objinfo
712                                                 // ergibt dann automatisch die
                                                    Potenzen von 2
713     case 1:
714         ergebnis = *((uint8_t *)ptr);               // hier kein Byteswap da nur 8 Bit
715         break;
716
717     case 2:
718         ergebnis = (uint64_t)_byteswap_ushort(*((uint16_t *)ptr));      // Byteswap
            wechselt Big Endian zu Little Endian mit 16 Bit
719         break;
720
721     case 4:
722         ergebnis = (uint64_t)_byteswap_ulong(*((uint32_t *)ptr));       // Byteswap
            wechselt Big Endian zu Little Endian mit 32 Bit
723         break;
724
```

```
725         case 8:     // 8 Byte Integer werden nicht unsigned definiert
726             ergebnis = (int64_t)_byteswap_uint64(*((int64_t *)ptr));        // Byteswap
                wechselt Big Endian zu Little Endian mit 64 Bit hier kein uint
727             negativint = (int64_t)_byteswap_uint64(*((int64_t *)ptr));      // Variable
                für Minus Werte
728             break;
729
730         default:
731             XWF_OutputMessage(L"BPListParser ERROR: Fehler bei der Integer Berechnung",
                0);
732             break;
733         }
734         return ergebnis;
735     }
736
737     wchar_t* parseDict(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
        markeroffset[4500], wchar_t *buffer) {
738
739         uint64_t inhalt = markeroffset[markernummer], objektpaare = 0, position = 0;
740         wchar_t Ausgabebuf[500];            // Temporärer Buffer für Ausgabe
741         int objektnummer, objektnummer2, j = 0;
742
743         int length = 0;
744         if (objinfo != 0xf)
745         {
746             objektpaare = objinfo;      // Anzahl der Objektpaare
747         }
748         else if (objinfo == 0xf)
749         {
750             inhalt++;
751             objinfo = (file_buf[inhalt] & 0x0F);                // objektInfo neu
                einlesen da ein Byte weiter
752             length = (int)pow(2, objinfo);                      // length rechnet für
                die eigene Funktion
753             objektpaare = parseint(objinfo, inhalt, file_buf);  // in INT wird länge des
                Int selber berechnet
754         }
755
756         if (objektpaare != 0)
757         {
758             ebene++;
759             if (markernummer == 0)
760             {
761                 wsprintf(Ausgabebuf, L"Dictionary mit %4d Objekten<br>", objektpaare);
762                 buffer = Ausgabe(buffer, L"Dictionary mit      Objekten<br>",
                    Ausgabebuf, 0);
763             }
764             else
765             {
766                 wsprintf(Ausgabebuf, L"Dictionary</td> <td>mit %4d
                    Objekten</td></tr></table><br>", objektpaare);
767                 buffer = Ausgabe(buffer, L"Dictionary</td> <td> mit
                    Objekten</td></tr></table><br>", Ausgabebuf, 0);
768
769                 // nutzungsebene = Aktueller Pointer auf Array tiefenanzeige
770                 // Stringbuffer = letzer gespeicherter String
771                 // stringlength = Länge des letzten Objekts
772
773                 // Ausgabe Ebene und Prüfung auf Max Länge der Buffer
774                 if (((nutzungebene + stringlength + 4) < 1500) && (ebene <40))
775                 {
776                     wsprintf(tiefenanzeige + nutzungebene, L"--> %s", Stringbuffer);
777                     nutzungebene = nutzungebene + stringlength + 4;
778                     buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
779                     ebenenlaenge[ebene] = stringlength + 4;
780
781                     wsprintf(Ausgabebuf, L"</b><br><br>");
782                     buffer = Ausgabe(buffer, L"</b><br><br>", Ausgabebuf, 0);
783                 }
784                 else
785                 {
786                     wsprintf(Ausgabebuf, L"<b>Zu viele Ebenen für Anzeige</b><br><br>");
787                     buffer = Ausgabe(buffer, L"<b>Zu viele Ebenen für
```

```
                Anzeige</b><br><br>", Ausgabebuf, 0);
788             }
789         }
790
791         wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=70> Item </th><th
            width=250>Wert</th> <th width=100>Typ</th><th width=550>Inhalt</th></tr>");
792         buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=70> Item
            </th><th width=250>Wert</th> <th width=100>Typ</th><th
            width=550>Inhalt</th></tr>", Ausgabebuf, 0);
793         table = 0;
794
795         // objektnummer = Objektnummer im DICT ( z.B. 01,02,03 ) --> neue markernummer
796         // markernummer = Objekt in der Offsettabelle
797         // markeroffset = Offsets der Offsettabelle ( mit allen offsize
798
799         for (int i = 1; i <= objektpaare; i++)
800         {
801             if (table == 1)
802             {
803                 wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=70> Item
                    </th><th width=250>Wert</th> <th width=100>Typ</th><th
                    width=550>Inhalt</th></tr>");
804                 buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=70>
                    Item </th><th width=250>Wert</th> <th width=100>Typ</th><th
                    width=550>Inhalt</th></tr>", Ausgabebuf, 0);
805                 table = 0;
806             }
807
808             position = inhalt   // Aktuelle Position in der Datei
809                 + i         // Nummer des auszulesenden Objekts
810                 + length    // Länge des Int Counters
811                 + j;        // Byte zähler für refsize 2
812
813             if (position + objektpaare * 2> file_size)
814             {
815                 wsprintf(Ausgabebuf, L"</table><br><b> Fehler: Auslesewert außerhalb
                    der Datei</b><br>");
816                 buffer = Ausgabe(buffer, L"</table><br><b> Fehler: Auslesewert
                    außerhalb der Datei</b><br>", Ausgabebuf, 0);
817                 return buffer;
818             }
819
820             if (refsize == 1)
821             {
822                 objektnummer = file_buf[position];
823                 objektnummer2 = file_buf[position + objektpaare];
824
825             }
826             else if (refsize == 2)
827             {
828                 objektnummer = (file_buf[position]) <<
                    8;                                  // 1 Byte
829                 objektnummer = objektnummer + file_buf[position +
                    1];                         // 2 Byte
830                 objektnummer2 = (file_buf[position + objektpaare * 2]) <<
                    8;                      // 1 Byte
831                 objektnummer2 = objektnummer2 + file_buf[position + 1 + objektpaare
                    * 2];   // 2 Byte
832                 j++;                // Zähler für das doppelbyte
833             }
834             else
835             {
836                 XWF_OutputMessage(L"BPListParser ERROR: Unsupportet Dict refsize", 0);
837                 return buffer;
838             }
839
840             // Prüfen Objektnummer
841             if (objektnummer > numobjekt && objektnummer2 > numobjekt)
842             {
843                 wsprintf(Ausgabebuf, L"</table><br><b> Fehler: nicht vorhandene
                    Objektnummer</b><br>");
844                 buffer = Ausgabe(buffer, L"</table><br><b> Fehler: nicht vorhandene
                    Objektnummer</b><br>", Ausgabebuf, 0);
```

```
845                     return buffer;
846                 }
847
848                 wsprintf(Ausgabebuf, L"<td>");
849                 buffer = Ausgabe(buffer, L"<td>", Ausgabebuf, 0);
850                 buffer = parseobjekt(file_buf, markeroffset, objektnummer, buffer);
851                 wsprintf(Ausgabebuf, L"</td><td>");
852                 buffer = Ausgabe(buffer, L"</td><td>", Ausgabebuf, 0);
853                 buffer = parseobjekt(file_buf, markeroffset, objektnummer2, buffer);
854                 wsprintf(Ausgabebuf, L"</td></tr>");
855                 buffer = Ausgabe(buffer, L"</td></tr>", Ausgabebuf, 0);
856
857             }
858             if (ebene<40)
859                 nutzungebene = nutzungebene - ebenenlaenge[ebene];
860
861             wsprintf(Ausgabebuf, L"</table><br>");
862             buffer = Ausgabe(buffer, L"</table><br>", Ausgabebuf, 0);
863             ebene--;
864             table = 1;
865
866             if (ebene > 1)
867             {
868                 buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
869                 wsprintf(Ausgabebuf, L"<br><br></b>");
870                 buffer = Ausgabe(buffer, L"<br><br></b>", Ausgabebuf, 0);
871             }
872             else if (ebene > 0)
873             {
874                 wsprintf(Ausgabebuf, L"<b>Ebene: Root</b><br><br>");
875                 buffer = Ausgabe(buffer, L"<b>Ebene: Root</b><br><br>", Ausgabebuf, 0);
876             }
877         }
878         else
879         {
880             wsprintf(Ausgabebuf, L"Dictionary</td><td> mit 0 Objekten</td></tr>");
881             buffer = Ausgabe(buffer, L"Dictionary</td><td> mit 0 Objekten</td></tr>",
                Ausgabebuf, 0);
882         }
883         return buffer;
884     }
885
886
887     wchar_t* parseArray(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
    markeroffset[4500], wchar_t *buffer)
888     {
889         uint64_t inhalt = markeroffset[markernummer], objekte = 0, position = 0;;
890         int length = 0, objektnummer = 0;
891         wchar_t Ausgabebuf[500];            // Temporärer Buffer für Ausgabe
892         int j = 0;
893         if (objinfo != 0xf)
894         {
895             objekte = objinfo;      // Anzahl der Objektpaare
896         }
897         else if (objinfo == 0xf)
898         {
899             inhalt++;
900             objinfo = (file_buf[inhalt] & 0x0F);                // objektInfo neu
                einlesen da ein Byte weiter
901             length = (int)pow(2, objinfo);
902             objekte = parseint(objinfo, inhalt, file_buf);
903         }
904
905         if (objekte != 0)
906         {
907             ebene++;
908             if (markernummer == 0)
909             {
910                 wsprintf(Ausgabebuf, L"Array mit %4d Objekten<br>", objekte);
911                 buffer = Ausgabe(buffer, L"Array mit      Objekten<br>", Ausgabebuf, 0);
912             }
913             else
914             {
```

```
915                 wsprintf(Ausgabebuf, L"Array</td> <td>mit %4d
                    Objekten</td></tr></table><br>", objekte);
916                 buffer = Ausgabe(buffer, L"Array</td> <td> mit
                    Objekten</td></tr></table><br>", Ausgabebuf, 0);
917
918                 // Prüfen auf Überschreitung der Buffer in Ebenenlaenge und Tiefenanzeige
919                 if (((nutzungebene + stringlength + 4) < 1500) && (ebene <40))
920                 {
921                     wsprintf(tiefenanzeige + nutzungebene, L"--> %s", Stringbuffer);
922                     nutzungebene = nutzungebene + stringlength + 4;
923                     buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
924                     ebenenlaenge[ebene] = stringlength + 4;
925
926                     wsprintf(Ausgabebuf, L"</b><br><br>");
927                     buffer = Ausgabe(buffer, L"</b><br><br>", Ausgabebuf, 0);
928                 }
929                 else
930                 {
931                     wsprintf(Ausgabebuf, L"<b>Zu viele Ebenen für Anzeige</b><br><br>");
932                     buffer = Ausgabe(buffer, L"<b>Zu viele Ebenen für
                        Anzeige</b><br><br>", Ausgabebuf, 0);
933                 }
934             }
935
936             // HTML Einbindung
937             wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=150> Item Nr. </th>
                <th width=100>Typ </th> <th width=720>Inhalt</th></tr>");
938             buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=150> Item Nr.
                </th> <th width=100>Typ </th> <th width=720>Inhalt</th></tr>", Ausgabebuf, 0);
939             table = 0;
940
941             for (int i = 1; i <= objekte; i++)
942             {
943                 if (table == 1)
944                 {
945                     wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=150> Item
                        Nr. </th> <th width=100>Typ </th> <th width=720>Inhalt</th></tr>");
946                     buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=150>
                        Item Nr. </th> <th width=100>Typ </th> <th
                        width=720>Inhalt</th></tr>", Ausgabebuf, 0);
947                     table = 0;
948                 }
949
950                 wsprintf(Ausgabebuf, L"<tr><td>Array Item Nr: %4d </td><td>", i);
951                 buffer = Ausgabe(buffer, L"<tr><td>Array Item Nr:      </td><td>",
                    Ausgabebuf, 0);
952
953                 wsprintf(Stringbuffer, L"Array Item Nr: %4d", i);
954                 stringlength = wcslen(L"Array Item Nr:     ");
955
956                 position = inhalt   // Aktuelle Position in der Datei
957                     + i             // Nummer des auszulesenden Objekts
958                     + length        // Länge des Int Counters
959                     + j;            // Byte zähler für refsize 2
960
961                 if (position + 1> file_size)
962                 {
963                     wsprintf(Ausgabebuf, L"</table><br><b> Fehler: Auslesewert außerhalb
                        der Datei</b><br>");
964                     buffer = Ausgabe(buffer, L"</table><br><b> Fehler: Auslesewert
                        außerhalb der Datei</b><br>", Ausgabebuf, 0);
965                     return buffer;
966                 }
967
968                 if (refsize == 1)
969                     objektnummer = file_buf[position];
970                 else if (refsize == 2)
971                 {
972                     objektnummer = (file_buf[position]) << 8;               // 1 Byte
973                     objektnummer = objektnummer + file_buf[position + 1];   // 2 Byte
974                     j++;
975                 }
976                 else
```

```
977                 {
978                     XWF_OutputMessage(L"BPListParser ERROR: Nicht unterstützte Array
                        refsize", 0);
979                     return buffer;
980                 }
981 
982                 // Prüfen Objektnummer in Datei vorhanden
983                 if (objektnummer > numobjekt)
984                 {
985                     wsprintf(Ausgabebuf, L"</table><br><b> Fehler: nicht vorhandene
                        Objektnummer</b><br>");
986                     buffer = Ausgabe(buffer, L"</table><br><b> Fehler: nicht vorhandene
                        Objektnummer</b><br>", Ausgabebuf, 0);
987                     return buffer;
988                 }
989 
990                 buffer = parseobjekt(file_buf, markeroffset, objektnummer, buffer);
991                 wsprintf(Ausgabebuf, L"</td></tr>");
992                 buffer = Ausgabe(buffer, L"</td></tr>", Ausgabebuf, 0);
993             }
994 
995             if (ebene<40)
996                 nutzungebene = nutzungebene - ebenenlaenge[ebene];
997 
998             wsprintf(Ausgabebuf, L"</table><br>");
999             buffer = Ausgabe(buffer, L"</table><br>", Ausgabebuf, 0);
1000            ebene--;
1001            table = 1;
1002 
1003            if (ebene > 1)
1004            {
1005                buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
1006                wsprintf(Ausgabebuf, L"<br><br></b>");
1007                buffer = Ausgabe(buffer, L"<br><br></b>", Ausgabebuf, 0);
1008            }
1009            else if (ebene > 0)
1010            {
1011                wsprintf(Ausgabebuf, L"<b>Ebene: Root<br><br></b>");
1012                buffer = Ausgabe(buffer, L"<b>Ebene: Root<br><br></b>", Ausgabebuf, 0);
1013            }
1014        }
1015        else {
1016            wsprintf(Ausgabebuf, L"Array</td><td> mit 0 Objekten</td></tr>");
1017            buffer = Ausgabe(buffer, L"Array</td><td> mit 0 Objekten</td></tr>",
                Ausgabebuf, 0);
1018        }
1019        return buffer;
1020    }
1021 
1022    wchar_t* parseSet(int objinfo, uint64_t markernummer, BYTE *file_buf, uint64_t
    markeroffset[4500], wchar_t *buffer)
1023    {
1024        uint64_t inhalt = markeroffset[markernummer], objekte = 0, position = 0;;
1025        int length = 0, objektnummer = 0;
1026        wchar_t Ausgabebuf[500];            // Temporärer Buffer für Ausgabe
1027        int j = 0;
1028        if (objinfo != 0xf)
1029        {
1030            objekte = objinfo;      // Anzahl der Objektpaare
1031        }
1032        else if (objinfo == 0xf)
1033        {
1034            inhalt++;
1035            objinfo = (file_buf[inhalt] & 0x0F);                // objektInfo neu
                einlesen da ein Byte weiter
1036            length = (int)pow(2, objinfo);
1037            objekte = parseint(objinfo, inhalt, file_buf);
1038        }
1039 
1040        if (objekte != 0)
1041        {
1042            ebene++;
1043            if (markernummer == 0)
```

```
1044            {
1045                wsprintf(Ausgabebuf, L"Set mit %4d Objekten<br>", objekte);
1046                buffer = Ausgabe(buffer, L"Set mit      Objekten<br>", Ausgabebuf, 0);
1047            }
1048            else
1049            {
1050                wsprintf(Ausgabebuf, L"Set</td> <td>mit %4d
                    Objekten</td></tr></table><br>", objekte);
1051                buffer = Ausgabe(buffer, L"Set</td> <td> mit
                    Objekten</td></tr></table><br>", Ausgabebuf, 0);
1052
1053                // Prüfen auf Überschreitung der Buffer in Ebenenlaenge und Tiefenanzeige
1054                if (((nutzungebene + stringlength + 4) < 1500) && (ebene <40))
1055                {
1056                    wsprintf(tiefenanzeige + nutzungebene, L"--> %s", Stringbuffer);
1057                    nutzungebene = nutzungebene + stringlength + 4;
1058                    buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
1059                    ebenenlaenge[ebene] = stringlength + 4;
1060
1061                    wsprintf(Ausgabebuf, L"</b><br><br>");
1062                    buffer = Ausgabe(buffer, L"</b><br><br>", Ausgabebuf, 0);
1063                }
1064                else
1065                {
1066                    wsprintf(Ausgabebuf, L"<b>Zu viele Ebenen für Anzeige</b><br><br>");
1067                    buffer = Ausgabe(buffer, L"<b>Zu viele Ebenen für
                        Anzeige</b><br><br>", Ausgabebuf, 0);
1068                }
1069            }
1070
1071            // HTML Einbindung
1072            wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=150> Set Nr. </th>
                <th width=100>Typ </th> <th width=720>Inhalt</th></tr>");
1073            buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=150> Set Nr.
                </th> <th width=100>Typ </th> <th width=720>Inhalt</th></tr>", Ausgabebuf, 0);
1074            table = 0;
1075
1076            for (int i = 1; i <= objekte; i++)
1077            {
1078                if (table == 1)
1079                {
1080                    wsprintf(Ausgabebuf, L"<table border=\"1\"><tr><th width=150> Set
                        Nr. </th> <th width=100>Typ </th> <th width=720>Inhalt</th></tr>");
1081                    buffer = Ausgabe(buffer, L"<table border=\"1\"><tr><th width=150>
                        Set Nr. </th> <th width=100>Typ </th> <th
                        width=720>Inhalt</th></tr>", Ausgabebuf, 0);
1082                    table = 0;
1083                }
1084
1085                wsprintf(Ausgabebuf, L"<tr><td>Set Item Nr: %4d </td><td>", i);
1086                buffer = Ausgabe(buffer, L"<tr><td>Set Item Nr:      </td><td>",
                    Ausgabebuf, 0);
1087
1088                wsprintf(Stringbuffer, L"Set Item Nr: %4d", i);
1089                stringlength = wcslen(L"Set Item Nr:     ");
1090
1091                position = inhalt   // Aktuelle Position in der Datei
1092                    + i             // Nummer des auszulesenden Objekts
1093                    + length        // Länge des Int Counters
1094                    + j;            // Byte zähler für refsize 2
1095
1096                if (position + 1> file_size)
1097                {
1098                    wsprintf(Ausgabebuf, L"</table><br><b> Fehler: Auslesewert außerhalb
                        der Datei</b><br>");
1099                    buffer = Ausgabe(buffer, L"</table><br><b> Fehler: Auslesewert
                        außerhalb der Datei</b><br>", Ausgabebuf, 0);
1100                    return buffer;
1101                }
1102
1103                if (refsize == 1)
1104                    objektnummer = file_buf[position];
1105                else if (refsize == 2)
```

```
1106                {
1107                    objektnummer = (file_buf[position]) << 8;                // 1 Byte
1108                    objektnummer = objektnummer + file_buf[position + 1];    // 2 Byte
1109                    j++;
1110                }
1111                else
1112                {
1113                    XWF_OutputMessage(L"BPListParser ERROR: Nicht unterstützte Set
                    refsize", 0);
1114                    return buffer;
1115                }
1116
1117                // Prüfen Objektnummer in Datei vorhanden
1118                if (objektnummer > numobjekt)
1119                {
1120                    wsprintf(Ausgabebuf, L"</table><br><b> Fehler: nicht vorhandene
                    Objektnummer</b><br>");
1121                    buffer = Ausgabe(buffer, L"</table><br><b> Fehler: nicht vorhandene
                    Objektnummer</b><br>", Ausgabebuf, 0);
1122                    return buffer;
1123                }
1124
1125                buffer = parseobjekt(file_buf, markeroffset, objektnummer, buffer);
1126                wsprintf(Ausgabebuf, L"</td></tr>");
1127                buffer = Ausgabe(buffer, L"</td></tr>", Ausgabebuf, 0);
1128            }
1129
1130            if (ebene<40)
1131                nutzungebene = nutzungebene - ebenenlaenge[ebene];
1132
1133            wsprintf(Ausgabebuf, L"</table><br>");
1134            buffer = Ausgabe(buffer, L"</table><br>", Ausgabebuf, 0);
1135            ebene--;
1136            table = 1;
1137
1138            if (ebene > 1)
1139            {
1140                buffer = Ausgabe(buffer, L"", tiefenanzeige, nutzungebene);
1141                wsprintf(Ausgabebuf, L"<br><br></b>");
1142                buffer = Ausgabe(buffer, L"<br><br></b>", Ausgabebuf, 0);
1143            }
1144            else if (ebene > 0)
1145            {
1146                wsprintf(Ausgabebuf, L"<b>Ebene: Root<br><br></b>");
1147                buffer = Ausgabe(buffer, L"<b>Ebene: Root<br><br></b>", Ausgabebuf, 0);
1148            }
1149        }
1150        else {
1151            wsprintf(Ausgabebuf, L"Set</td><td> mit 0 Objekten</td></tr>");
1152            buffer = Ausgabe(buffer, L"Set</td><td> mit 0 Objekten</td></tr>",
                Ausgabebuf, 0);
1153        }
1154        return buffer;
1155    }
1156
1157    wchar_t *Ausgabe(wchar_t *buffer, const wchar_t* Zeile, wchar_t* Ausgabebuf, int
        zeichencount) {
1158
1159        // Anzahl zu schreibenden Zeichen zählen
1160        size_t Zeilenlänge = wcslen(Zeile) + zeichencount;
1161
1162        // Erweiterung des Speichers wenn Speicherbedarf zu Groß
1163        while (Speichernutzung + Zeilenlänge >= SpeicherZugeordnet)
1164        {
1165            // Exponentielle Vergößerung
1166            SpeicherZugeordnet *= 2;
1167
1168            // Speicherzuordnung erfolgreich
1169            buffer = (wchar_t *)realloc(buffer, SpeicherZugeordnet * sizeof(wchar_t));
1170            if ((buffer == NULL))
1171            {
1172                XWF_OutputMessage(L"BPListParser ERROR: Fehler beim Zuordnen des
                    größeren Speichers", 0);
```

```
1173                return NULL;
1174            }
1175        }
1176        // Neuen Inhalt an Speicher anhängen
1177        wmemcpy(buffer + Speichernutzung        // Zielbuffer + aktueller Inhalt
1178            , Ausgabebuf                        // Quellbuffer
1179            , Zeilenlänge);                     // Anzahl der zu kopierenden Zeichen
1180
1181                                                // Aktuelle Speichernutzung festhalten
1182        Speichernutzung += Zeilenlänge;
1183
1184        return buffer;
1185    }
```