```cpp
////////////////////////////////////////////////////////////////////////////////
// X-Tension API - template for new X-Tensions
// Copyright X-Ways Software Technology AG
// Please consult
// http://x-ways.com/forensics/x-tensions/api.html
// for current documentation
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// ESEDBViewer
// Dieses Projekt entstand im Rahmen einer Bachelorarbeit im Studiengang
// Allgemeine und digitale Forensik an der Hochschule Mittweida im Jahr 2023.
// Die Arbeit wurde von Prof. Ronny Bodach und M.Sc. Stefan Schildbach betreut.
// Der Quellcode darf unter Beachtung des Copyright der X-Ways Software
// Technology AG kostenlos verwendet, verändert oder weitergegeben werden.
// Es gibt keine Gewährleistung auf Funktionalität oder Aktualisiserung.
// gez. Linda Becker
////////////////////////////////////////////////////////////////////////////////


#include "X-Tension.h"
#include <windows.h>
#include <wchar.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>


////////////////////////////////////////////////////////////////////////////////
// Globale Variablendefinition
size_t SpeicherZugeordnet = 1024;   // Reservierter Speicher
size_t Speichernutzung = 0;         // Aktuelle Speichernutzung
INT16 jump_size = 0;                // Jump Size in Key
size_t dd_header_offset = 0;        // Offset data definition header (start last fixed id)
bool flag_deleted = 0;              // Flag Datensatz gelöscht
int last_fixed_id = 0;              // last fixed size data ID
int last_variable_id = 0;           // last variable size data ID
INT16 relative_variable_offset = 0; // first variable size data offset
void* ptr;                          // Pointer
size_t variable_offset = 0;         // eigentlicher Offset der
```

```
    variablen Daten
42  size_t record_offset = 0;           // Record Offset nach Data
    Definition Header
43  int data_8bit = 0;                  // Daten 8 Bit
44  INT16 data_16bit = 0;               // Daten 16 Bit
45  INT32 data_32bit = 0;               // Daten 32 Bit
46  INT64 data_64bit = 0;               // Daten 64 Bit
47  float data_float = 0;               // Daten Float
48  double data_double = 0;             // Daten Double
49  char zeichen = 0;                   // einzelnes Zeichen
50  size_t page_offset = 0;             // aktueller Seiten Offset
51
52
53  ////////////////////////////////////////////////////////////////////
    /////////
54  // Funktionsdefintionen
55
56  // Inhalte zum Ausgabebuffer hinzufügen: (Zielpufferpointer,
    Zeichenanzahl zum zählen, Inhalt, wieviele Zeichen zusätzlich,
    Rückgabewert: Ausgabebuffer )
57  wchar_t* Ausgabe(wchar_t* buffer, const wchar_t* Zeile, wchar_t*
    Ausgabebuf, int zeichencount);
58
59  // Page Key Flags überprüfen und Offset Data Definition Header Offset
    bestimmen: (Seiten-Schlüssel-Flags, Schlüssel-Offset, aktueller
    Seiten-Puffer, Rückgabe: Data Defintion Header Offset)
60  size_t ddHeaderOffset(int page_key_flags, size_t key_offset, BYTE*
    page_buf);
61
62  // Data Definition Header auswerten: Last fixed size id, last variable
     size id, variable size offset; (akuteller Seiten-Puffer, Data
    Definition Header Offset, kein Rückgabewert)
63  void ddHeader(BYTE* page_buf, size_t dd_header_offset);
64
65  // Page Offset berechnen
66  size_t pageOffset(size_t page_number, INT32 page_size);
67
68
69  ////////////////////////////////////////////////////////////////////
    /////////
70  // XT_Init
71
72  LONG __stdcall XT_Init(DWORD nVersion, DWORD nFlags, HANDLE hMainWnd,
    void* lpReserved)
73  {
74      XT_RetrieveFunctionPointers();
75      XWF_OutputMessage (L"XT_ESEDBViewer initialized", 0);
76      return 1;
77  }
78
79
80  ////////////////////////////////////////////////////////////////////
    /////////
```

```
 81 // XT_View
 82
 83 PVOID __stdcall XT_View(HANDLE hItem, LONG nItemID, HANDLE hVolume,
      HANDLE hEvidence, PVOID lpReserved, PINT64 lpResSize)
 84 {
 85     // Dateigröße
 86     size_t file_size = 0;
 87     file_size = (size_t)XWF_GetProp(hItem, (INT64)1, 0);        //
        logische Dateigroesse
 88
 89     // Zuständigkeit prüfen
 90     // lpResSize: -2 = Fehler, -1 = nicht zuständig, 0 = keine Daten
        darstellen, >0 = Adresse Puffer der Daten
 91     *lpResSize = -1;                                            //
        nicht zuständig
 92
 93     // Signatur prüfen
 94     char file_signatur[5] = {};                                 //
        Signatur der Datei
 95     char signatur[5] = {0xef, 0xcd, 0xab, 0x89, 0x00};
 96
 97     XWF_Read(hItem, 4, (BYTE*)file_signatur, 4);                // ab
        Offset 4 lesen, 4 Bytes
 98
 99     if (strcmp(signatur, file_signatur) != 0)
100     {
101         return NULL;    // Sobald eine return Anweisung ausgeführt
            wird und der Wert vorher auf -1 gesetzt wurde
102                         // weiß XWF, dass die X - Tension nicht für
                    diese Datei zuständig ist.
103     }
104
105
106     // Buffer: wird als Rückgabewert für XWF verwendet
107     wchar_t* buffer = (wchar_t*)calloc(SpeicherZugeordnet, sizeof
        (wchar_t));
108
109     wmemset(buffer, 0, SpeicherZugeordnet);
                    // Puffer mit Nullen füllen
110
111     *lpResSize = -2;
                    // gibt Fehler beim Abarbeiten in XTension an
112
113     // Header auslesen
114     INT32 file_type = 0;                                        //
        file type
115     INT32 page_size = 0;                                        //
        page size
116
117     HANDLE heap = GetProcessHeap();                             //
        Ruft einen Handle zum Standard-Heap des Aufrufvorgangs ab.
118
119     BYTE* header = (BYTE*)HeapAlloc(heap, 0, file_size);
```

```
120
121     XWF_Read(hItem, 0, header, 668);                                    //      ↵
          Header einlesen
122
123     ptr = &header[12];
124     file_type = *((uint32_t*)ptr);                                      //      ↵
          file type
125
126     // File Type überprüfen
127     if (file_type != 0)                                                 // 0:   ↵
          Datenbank Datei, 1: Streaming File
128     {
129         *lpResSize = 46;
130         wsprintf(buffer, L"ERROR: Streaming Datei");
131         HeapFree(heap, 0, header);
132         return buffer;
133     }
134
135     ptr = &header[236];
136     page_size = *((uint32_t*)ptr);                                      //      ↵
          Seitengroesse
137
138     // Offset Datenbereich bestimmen
139     size_t data_offset = 0;
140
141     if (page_size >= 0x4000)
142     {
143         data_offset = 0x50;
144     }
145     else
146     {
147         data_offset = 0x28;
148     }
149
150     HeapFree(heap, 0, header);                                          //      ↵
          Speicher vom Header wieder freigeben
151
152     // Anzahl Seiten überprüfen
153     size_t page_count = 0;                                              //      ↵
          Anzahl Seiten
154     page_count = (file_size / page_size) - 2;                           //      ↵
          Anzahl Seiten; -2 weil ersten zwei seiten sind Header + Kopie
155     if (page_count < 4)
156     {
157         *lpResSize = 80;
158         wsprintf(buffer, L"ERROR: Datenbank enthält keine Tabellen");
159         return buffer;
160     }
161
162     // MSysObjects auslesen -> Data Definitions
163     INT32 page_number = 0;  // aktuelle Seiten-Nummer
164     page_number = 4;    // Root
165
```

```
166     BYTE* msysobjects_buffer = (BYTE*)HeapAlloc(heap, 0, file_size);
167     XWF_Read(hItem, 0, msysobjects_buffer, file_size);
168
169     INT32 msysobjects_array[500] = {};              // Array für
          Seitenzahlen der MSysObjects
170     msysobjects_array[0] = 4;                       // Seite 4
          hinzufügen = Root
171     uint16_t tag_array[1000] = {};                  // Array für alle
          auf einer Seite gespeicherten Tags
172     int current_page = 1;                           // aktuelle Seite
173     int next_page = 0;                              // nächste Seite
          die ausgewertet werden muss
174     int data_definition_count = 0;                  // Anzahl Einträge
           Data Definition Array
175     int variable_count = 0;                         // Anzahl variable
           size Spalten
176     char data_definitions_name[1000][256] = {};     // Array für Namen
           der Data Defintions
177     INT32 data_definitions[1000][7] = {};           // alle data
          definitions
178     int page_flags = 0;                             // Seiten Flags
179     INT16 page_tags = 0;                            // Anzahl Page
          Tags
180     size_t tag_offset = 0;                          // Start Offset
          Tag-Bereich
181     int page_key_flags = 0;                         // Page Key Flags
182     boolean key_flags = 0;                          // Flags in Key
          oder Tag, 0: in Key, 1: in Tag
183     INT16 offset = 0;                               // Offset in Tag
184     size_t key_offset = 0;                          // Key Offset
185     INT32 fdp_id = 0;                               // FDP ID im
          Record (b+baum)
186     INT16 catalog_type = 0;                         // Catalog Type
187     INT32 id = 0;                                   // ID in Data
          Definition
188     INT32 column_type_or_fdp = 0;                   // Spalten-Typ
          oder FDP Nummer in Record
189     INT32 space = 0;                                // Platzbedarf
          Anzahl Seiten (Tabelle, Index, LV) / Bytes (Spalten)
190     INT32 flags = 0;                                // Flags in Record
191     size_t variable_data_offset = 0;                // Offset Variable
           Sized Data, nach Array
192     INT16 variable_size = 0;                        // Größe variable
          sized data
193
194
195     // alle Seiten die MSysObjects Tabelle enthalten auslesen und
          Datendefinitionen abspeichern
196     while (msysobjects_array[next_page] != 0)
197     {
198         page_offset = pageOffset(page_number, page_size);   // Offset
              der Seite berechnen
199
```

```
200         // Anzahl Tags
201         ptr = &msysobjects_buffer[page_offset + 34];
202         page_tags = *((uint16_t*)ptr);
203
204         if (page_tags > 500) // mehr als 500 Tags -> passen sonst
              nicht mehr in Tag-Array
205         {
206             XWF_OutputMessage(L"ERROR: Seite hat mehr als 500
                  Einträge", 0);
207             return NULL;
208         }
209
210         // Tag-Array befüllen
211         tag_offset = page_offset + page_size - 2;
212         for (int i = 0; i < (page_tags * 2); i++)          // *2, da
              Offset und Länge je Tag
213         {
214             ptr = &msysobjects_buffer[tag_offset];         //
                  current_page auf aktuellen Tag-Offset
215             tag_array[i] = *((uint16_t*)ptr);              // 16 Bit
                  Wert lesen
216             tag_offset = tag_offset - 2;                   // Tag-
                  Offset - 2
217         }                                                  //
              Position 0: Offset, Position 1: Länge
218
219         // prüfen ob Flags in Key oder Tag
220         offset = tag_array[2];
221         if ((offset & 0xE000) == 0)
222         {
223             key_flags = 0; // Flags in Key
224         }
225         else
226         {
227             key_flags = 1; // Flags in Tag
228         }
229
230         // Seitenflags prüfen
231         page_flags = msysobjects_buffer[page_offset + 36];
232
233         if ((page_flags & 2) != 2)                         // ist
              keine Blatt-Seite
234         {
235             // Page Tags auswerten -> zeigen auf Seiten eine Ebene
                  weiter unten
236             for (int j = 1; j < page_tags; j++)            // für
                  jeden Page Tag, außer Tag 0 = Header
237             {
238
239                 // key offset und page key flags bestimmen
240                 if (key_flags == 0)
241                 {
242                     key_offset = page_offset + data_offset + tag_array
```

```
                    [j * 2];        // Key Offset = Start eines Datensatzes berechnen
243                   page_key_flags = msysobjects_buffer[key_offset + 1];         // Flags im Schlüssel auslesen
244                 }
245                 else
246                 {
247                     key_offset = page_offset + data_offset + (tag_array[j * 2] & 0x1FFF);
248                   page_key_flags = (int)tag_array[j * 2] >> 8;        // Flags im Tag auslesen
249                 }
250
251                 // Jump Size und DD Header Offset bestimmen
252                 ddHeaderOffset(page_key_flags, key_offset, msysobjects_buffer);        // return: dd_header_offset = start 4 byte page number
253                 ptr = &msysobjects_buffer[dd_header_offset];
254                 page_number = *((uint32_t*)ptr);                        // Seiten Nummer auslesen
255                 msysobjects_array[current_page] = page_number;
256                 current_page++;
257
258             }
259         }
260         else // Blattseite: Datendefinitionen auslesen
261         {
262             for (int j = 1; j < page_tags; j++)
263             {
264                 // Tabellendefintion der MSysObjects Tabelle
265                 if (key_flags == 0)
266                 {
267                     key_offset = page_offset + data_offset + tag_array[j * 2];        // Offset Schlüssel berechnen
268                   page_key_flags = msysobjects_buffer[key_offset + 1];         // Flags im Schlüssel auslesen
269                 }
270                 else
271                 {
272                     key_offset = page_offset + data_offset + (tag_array[j * 2] & 0x1FFF);    // Offset Schlüssel berechnen, ohne 3 MSB in Offset
273                   page_key_flags = (int)tag_array[j * 2] >> 8;                                // Flags im Tag auslesen
274                 }
275
276                 // Jump Size und DD Header Offset bestimmen
277                 ddHeaderOffset(page_key_flags, key_offset, msysobjects_buffer);
278
279                 // DD Header auswerten
280                 ddHeader(msysobjects_buffer, dd_header_offset);
281
```

```
282                 // Datendefinitionen auslesen
283                 ptr = &msysobjects_buffer[record_offset];
284                 fdp_id = *((uint32_t*)ptr);             // fdp id des
                      Records
285
286                 ptr = &msysobjects_buffer[record_offset + 4];
287                 catalog_type = *((uint16_t*)ptr);       // Catalog
                      Type: Tabelle, Spalte, Index, LV
288
289                 ptr = &msysobjects_buffer[record_offset + 6];
290                 id = *((uint32_t*)ptr);                 // id
291
292                 ptr = &msysobjects_buffer[record_offset + 10];
293                 column_type_or_fdp = *((uint32_t*)ptr); // Spaltentyp
                      oder fdp
294
295                 ptr = &msysobjects_buffer[record_offset + 14];
296                 space = *((uint32_t*)ptr);              // Platzbedarf
297
298                 ptr = &msysobjects_buffer[record_offset + 18];
299                 flags = *((uint32_t*)ptr);              // Flags
300
301                 // Data Defintions Array auffüllen
302                 // Spalte 0: fdp_id, Spalte 1: catalog_type, Spalte 2:
                       ID, Spalte 3: Column Type or fdp, Spalte 4: Space,
                      Spalte 5: flags
303                 data_definitions[data_definition_count][0] = fdp_id;
304                 data_definitions[data_definition_count][1] =
                      catalog_type;
305                 data_definitions[data_definition_count][2] = id;
306                 data_definitions[data_definition_count][3] =
                      column_type_or_fdp;
307                 data_definitions[data_definition_count][4] = space;
308                 data_definitions[data_definition_count][5] = flags;
309
310                 // Namen in Namensarray speichern
311                 ptr = &msysobjects_buffer[variable_offset];
312                 variable_size = *((uint16_t*)ptr);              //
                      Länge variable size data = Länge Name
313
314                 data_definitions[data_definition_count][6] =
                      variable_size; // Länge des Namen abspeichern
315
316                 if (last_variable_id >= 0x80)
317                 {
318                     variable_count = last_variable_id - 0x7f;   //
                      Anzahl variable size Einträg berechnen
319                 }
320
321                 // Offset der variable sized data nach array
322                 variable_data_offset = variable_offset + 2 *
                      variable_count;
323
```

```
324                 // Name in Array speichern
325                 for (int k = 0; k < variable_size; k++)
326                 {
327                     data_definitions_name[data_definition_count][k] =
                      msysobjects_buffer[variable_data_offset + k];
328                 }
329                 data_definition_count++;
330             }
331         }
332         next_page++;
333         page_number = msysobjects_array[next_page];
334     }
335
336     HeapFree(heap, 0, msysobjects_buffer);
337
338     // Ausgabe
339     wchar_t Ausgabebuf[500];                // Temporärer Buffer für
          Ausgabe
340     buffer[0] = 0xFEFF;                     // BOM für HTML Code
341     Speichernutzung = 1;                    // Zähler wieviel Byte im
          Buffer aktuell gespeichert sind
342     int zeichencount = 1;                   // Zähler für Länge des
          Int Wertes
343
344     // HTML Einbindung
345     wsprintf(Ausgabebuf, L"<HTML><HEAD><style>table, th, td {border:
          1px solid black; border-collapse: collapse;}</style></
          HEAD><BODY><center><b>ESEDB File: %s</b></center><br>",
          XWF_GetItemName(nItemID));
346     buffer = Ausgabe(buffer, L"<HTML><HEAD><style>table, th, td
          {border: 1px solid black; border-collapse: collapse;}</style></
          HEAD><BODY><center><b>ESEDB File: </b></center><br>", Ausgabebuf,
           wcslen(XWF_GetItemName(nItemID)));
347
348
349     // Data Definitions Array durchgehen
350     BYTE* file_buf = (BYTE*)HeapAlloc(heap, 0, file_size);
351     XWF_Read(hItem, 0, file_buf, file_size);                    //
          Handle auf die Datei, von wo an gelesen werden soll,
352                                                                 //wo
                    rein geschrieben werden soll, bis wohin gelesen
                    werden soll)
353
354     // Tabellen ausgeben
355     for (int i = 0; i < data_definition_count; i++)             // für
          jeden Eintrag im data definition array
356     {
357         if (data_definitions[i][1] == 1)                        //
              wenn Tabelle
358         {
359             // Spaltenanzahl zurücksetzen
360             int column_count = 0;               // Anzahl Spalten in
                  MSysObject
```

```
361
362             // fdp id der Tabelle / B+Baum
363             fdp_id = data_definitions[i][0];
364
365             // Tabelle initiieren
366             wsprintf(Ausgabebuf, L"<table style=\"width:100%%
                  \"><caption style=\"text-align:left\">");
367             buffer = Ausgabe(buffer, L"<table style=\"width:100%%
                  \"><caption style=\"text-align:left\">", Ausgabebuf, 0);
368
369             // Name der Tabelle
370             for (int j = 0; j < data_definitions[i][6]; j++)
371             {
372                 zeichen = data_definitions_name[i][j];
373                 wsprintf(Ausgabebuf, L"%c", zeichen);
374                 buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
375             }
376
377             // Caption schließen
378             wsprintf(Ausgabebuf, L"</caption>");
379             buffer = Ausgabe(buffer, L"</caption>", Ausgabebuf, 0);
380
381             // neue Tabellenzeile öffnen
382             wsprintf(Ausgabebuf, L"<tr>");
383             buffer = Ausgabe(buffer, L"<tr>", Ausgabebuf, 0);
384
385             // Spaltennamen und Definitionen der Tabelle
386             INT32 data_definitions_table[100][6] = {};
387
388             // für jede Spalte in dd die zur gleichen Tabelle gehört
389             for (int x = 0; x < data_definition_count; x++)
390             {
391                 if ((data_definitions[x][0] == fdp_id) &&
                      data_definitions[x][1] == 2)
392                 {
393                     // Data Definition der Tabelle speichern
394                     data_definitions_table[column_count][0] =
                    data_definitions[x][0];
395                     data_definitions_table[column_count][1] =
                    data_definitions[x][1];
396                     data_definitions_table[column_count][2] =
                    data_definitions[x][2];
397                     data_definitions_table[column_count][3] =
                    data_definitions[x][3];
398                     data_definitions_table[column_count][4] =
                    data_definitions[x][4];
399                     data_definitions_table[column_count][5] =
                    data_definitions[x][5];
400
401                     // neue Zelle öffnen
402                     wsprintf(Ausgabebuf, L"<th>");
403                     buffer = Ausgabe(buffer, L"<th>", Ausgabebuf, 0);
404
```

```
405                     // Name der Spalte
406                     for (int j = 0; j < data_definitions[x][6]; j++)
407                     {
408                         zeichen = data_definitions_name[x][j];
409                         wsprintf(Ausgabebuf, L"%c", zeichen);
410                         buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
411                     }
412 
413                     // Zelle schließen
414                     wsprintf(Ausgabebuf, L"</th>");
415                     buffer = Ausgabe(buffer, L"</th>", Ausgabebuf, 0);
416 
417                     column_count++;
418                 }
419             }
420 
421             // Tabellenzeile schließen
422             wsprintf(Ausgabebuf, L"</tr>");
423             buffer = Ausgabe(buffer, L"</tr>", Ausgabebuf, 0);
424 
425             page_number = data_definitions[i][3];   // fdp der Tabelle
426 
427             INT32 page_array[500] = {};             // Array für
                  Seitenzahlen
428             page_array[0] = page_number;            // aktuelle Seite
                  hinzufügen
429 
430             current_page = 1;                       // current_page
                  für aktuelle Seite
431             next_page = 0;                          // nächste Seite
                  die ausgewertet werden muss
432             INT16 tagged_data_size = 0;             // Größe Tagged
                  Data
433             int printed_columns = 0;                // Anzahl
                  ausgegebene Spalten
434 
435             while (page_array[next_page] != 0)
436             {
437                 page_offset = pageOffset(page_number, page_size);   //
                       Offset der Seite berechnen
438 
439                 // Anzahl Tags
440                 ptr = &file_buf[page_offset + 34];
441                 page_tags = *((uint16_t*)ptr);
442 
443                 if (page_tags > 500) // mehr als 500 Tags -> passen
                      sonst nicht mehr in Tag-Array
444                 {
445                     XWF_OutputMessage(L"ERROR: Seite hat mehr als 500
                      Einträge", 0);
446                     return NULL;
447                 }
448 
```

```
449                 // Tag-Array befüllen
450                 tag_offset = page_offset + page_size - 2;
451                 for (int j = 0; j < (page_tags * 2); j++)   // *2, da
                      Offset und Länge je Tag
452                 {
453                     ptr = &file_buf[tag_offset];            //
                    current_page auf aktuellen Tag-Offset
454                     tag_array[j] = *((uint16_t*)ptr);       // 16 Bit
                    Wert lesen
455                     tag_offset = tag_offset - 2;            // Tag-
                    Offset - 2
456                 }
457
458                 // prüfen ob Flags in Key oder Tag
459                 offset = tag_array[2];
460                 if ((offset & 0xE000) == 0)
461                 {
462                     key_flags = 0; // Flags in Key
463                 }
464                 else
465                 {
466                     key_flags = 1; // Flags in Schlüssel
467                 }
468
469                 // Seitenflags prüfen
470                 page_flags = file_buf[page_offset + 36];
471
472                 if ((page_flags & 2) != 2) // ist keine Blatt-Seite
473                 {
474                     // Page Tags auswerten -> zeigen auf Seiten eine
                    Ebene weiter unten
475                     for (int j = 1; j < page_tags; j++)         // für
                     jeden Page Tag, außer Tag 0 = Header
476                     {
477                         // key offset und page key flags bestimmen
478                         if (key_flags == 0)
479                         {
480                             key_offset = page_offset + data_offset +
                    tag_array[j * 2];       // Key Offset = Start eines
                    Datensatzes berechnen
481                             page_key_flags = file_buf[key_offset + 1];
                                            // Flags im Schlüssel
                    auslesen
482                         }
483                         else
484                         {
485                             key_offset = page_offset + data_offset +
                    (tag_array[j * 2] & 0x1FFF);
486                             page_key_flags = (int)tag_array[j * 2] >>
                    8;                          // Flags im Tag auslesen
487                         }
488
489                         // Jump Size und DD Header Offset bestimmen
```

```
490                         ddHeaderOffset(page_key_flags, key_offset,
                    file_buf); // return: dd_header_offset = start 4 byte
                     page number
491                         ptr = &file_buf[dd_header_offset];
492                         page_number = *((uint32_t*)ptr);        //
                    Seiten Nummer auslesen
493                         page_array[current_page] = page_number;
494                         current_page++;
495                       }
496                   }
497                   else // Blattseite
498                   {
499                       // Seite auswerten und ausgeben
500                       for (int j = 1; j < page_tags; j++)         // für
                     jeden Page Tag, außer Tag 0 = Header
501                       {
502                         printed_columns = 0;                      //
                    ausgegebene Spaltenanzahl zurücksetzen
503
504                         // key offset und page key flags bestimmen
505                         if (key_flags == 0)
506                         {
507                             key_offset = page_offset + data_offset +
                    tag_array[j * 2];       // Key Offset = Start eines
                    Datensatzes berechnen
508                             page_key_flags = file_buf[key_offset + 1];
                                        // Flags im Schlüssel
                    auslesen
509                         }
510                         else
511                         {
512                             key_offset = page_offset + data_offset +
                    (tag_array[j * 2] & 0x1FFF);
513                             page_key_flags = (int)tag_array[j * 2] >>
                    8;                          // Flags im Tag auslesen
514                         }
515
516                         // Jump Size und DD Header Offset bestimmen
517                         ddHeaderOffset(page_key_flags, key_offset,
                    file_buf);  // return: dd_header_offset
518                         ddHeader(file_buf, dd_header_offset);
                              // dd_header auswerten + record offset
                    bestimmen
519
520                         // neue Tabellenzeile öffnen
521                         wsprintf(Ausgabebuf, L"<tr>");
522                         buffer = Ausgabe(buffer, L"<tr>", Ausgabebuf,
                    0);
523
524                         // fixed sized Spalten
525                         // Anzahl bestimmen
526                         int fixed_count = 0;
527                         for (int f = 0; f < column_count; f++)
```

```
528                         {
529                             if (data_definitions_table[f][2] <= 0x7F)
530                             {
531                                 fixed_count++;
532                             }
533                         }
534                         if (last_fixed_id < fixed_count)
535                         {
536                             fixed_count = last_fixed_id;
537                         }
538 
539                         size_t offset_data = 0;
540                         size_t sum_fixed_size = 0;
541                         for (int f = 0; f < fixed_count; f++) // für jede fixed id
542                         {
543                             // neue Zelle öffnen
544                             wsprintf(Ausgabebuf, L"<td>");
545                             buffer = Ausgabe(buffer, L"<td>", Ausgabebuf, 0);
546 
547                             column_type_or_fdp = data_definitions_table[f][3];       // Spaltentyp bestimmen
548                             space = data_definitions_table[f][4];       // Platzbedarf bestimmen
549                             offset_data = record_offset + sum_fixed_size;         // Offset aktualisieren
550 
551                             // Spaltentyp auswerten und ausgegeben
552                             int x = 1;
553                             switch (column_type_or_fdp)
554                             {
555                             case 0:
556                                 break;
557                             case 1:
558                                 data_8bit = file_buf[offset_data];
559                                 wsprintf(Ausgabebuf, L"%3u", data_8bit);
560                                 buffer = Ausgabe(buffer, L"", Ausgabebuf, 3);
561                                 break;
562                             case 2:
563                                 data_8bit = file_buf[offset_data];
564                                 wsprintf(Ausgabebuf, L"%3u", data_8bit);
565                                 buffer = Ausgabe(buffer, L"", Ausgabebuf, 3);
566                                 break;
567                             case 3:
568                                 ptr = &file_buf[offset_data];
569                                 data_16bit = *((uint16_t*)ptr);
570                                 wsprintf(Ausgabebuf, L"%6d",
```

```
                    data_16bit);
571                             buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 6);
572                             break;
573                         case 4:
574                             ptr = &file_buf[offset_data];
575                             data_32bit = *((uint32_t*)ptr);
576                             wsprintf(Ausgabebuf, L"%11d",
                    data_32bit);
577                             buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 11);
578                             break;
579                         case 5:
580                             ptr = &file_buf[offset_data];
581                             data_64bit = *((uint64_t*)ptr);
582                             wsprintf(Ausgabebuf, L"%20d",
                    data_64bit);
583                             buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 20);
584                             break;
585                         case 6:
586                             ptr = &file_buf[offset_data];
587                             data_32bit = *((uint32_t*)ptr);
588                             memcpy(&data_float, &data_32bit,
                    sizeof(float));
589                             wsprintf(Ausgabebuf, L"%08X",
                    data_float);
590                             buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 8);
591                             break;
592                         case 7:
593                             ptr = &file_buf[offset_data];
594                             data_64bit = *((uint64_t*)ptr);
595                             memcpy(&data_float, &data_64bit,
                    sizeof(double));
596                             wsprintf(Ausgabebuf, L"%016X",
                    data_double);
597                             buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 16);
598                             break;
599                         case 8:
600                             for (int z = 0; z < space; z++)
601                             {
602                                 data_8bit = file_buf[offset_data +
                     z];
603                                 wsprintf(Ausgabebuf, L"%02X",
                    data_8bit);
604                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 2);
605                             }
606                             break;
607                         case 9:
608                             for (int z = 0; z < space; z++)
```

```
609                                 {
610                                     data_8bit = file_buf[offset_data + z];
611                                     wsprintf(Ausgabebuf, L"%02X", data_8bit);
612                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 2);
613                                 }
614                                 break;
615                             case 10:
616                                 for (int z = 0; z < space; z++)
617                                 {
618                                     zeichen = file_buf[offset_data + z];
619                                     wsprintf(Ausgabebuf, L"%c", zeichen);
620                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
621                                 }
622                                 break;
623                             case 11:
624                                 for (int z = 0; z < space; z++)
625                                 {
626                                     data_8bit = file_buf[offset_data + z];
627                                     wsprintf(Ausgabebuf, L"%02X", data_8bit);
628                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 2);
629                                 }
630                                 break;
631                             case 12:
632                                 for (int z = 0; z < space; z++)
633                                 {
634                                     zeichen = file_buf[offset_data + z];
635                                     wsprintf(Ausgabebuf, L"%c", zeichen);
636                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
637                                 }
638                                 break;
639                             case 13:
640                                 break;
641                             case 14:
642                                 ptr = &file_buf[offset_data];
643                                 data_32bit = *((uint32_t*)ptr);
644                                 wsprintf(Ausgabebuf, L"%10u", data_32bit);
645                                 buffer = Ausgabe(buffer, L"", Ausgabebuf, 10);
646                                 break;
647                             case 15:
```

```
648                                 ptr = &file_buf[offset_data];
649                                 data_64bit = *((uint64_t*)ptr);
650                                 wsprintf(Ausgabebuf, L"%20d",
                    data_64bit);
651                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 20);
652                                 break;
653                             case 16:
654                                 for (int z = 1; z <= 4; z++)
655                                 {
656                                     data_8bit = file_buf[offset_data +
                     16 - z]; // vom Ende auslesen -> Little Endian
657                                     wsprintf(Ausgabebuf, L"%02X",
                    data_8bit);
658                                     buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 2);
659                                 }
660                                 wsprintf(Ausgabebuf, L"-");
661                                 buffer = Ausgabe(buffer, L"-",
                    Ausgabebuf, 0);
662                                 for (int z = 1; z <= 3; z++)
663                                 {
664                                     for (int k = 1; k <= 2; k++)
665                                     {
666                                         data_8bit = file_buf
                    [offset_data + 12 - x];
667                                         wsprintf(Ausgabebuf, L"%02X",
                    data_8bit);
668                                         buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 2);
669                                         x++;
670                                     }
671                                     wsprintf(Ausgabebuf, L"-");
672                                     buffer = Ausgabe(buffer, L"-",
                    Ausgabebuf, 0);
673                                 }
674                                 for (int z = 1; z <= 6; z++)
675                                 {
676                                     data_8bit = file_buf[offset_data +
                     6 - z];
677                                     wsprintf(Ausgabebuf, L"%02X",
                    data_8bit);
678                                     buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 2);
679                                 }
680                                 break;
681                             case 17:
682                                 ptr = &file_buf[offset_data];
683                                 data_16bit = *((uint16_t*)ptr);
684                                 wsprintf(Ausgabebuf, L"%5u",
                    data_16bit);
685                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 5);
```

```
686                                 break;
687                             default: // als Binärdaten ausgeben
688                                 for (int z = 0; z < space; z++)
689                                 {
690                                     data_8bit = file_buf[offset_data + z];
691                                     wsprintf(Ausgabebuf, L"%02X", data_8bit);
692                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 2);
693                                 }
694                             }
695                             sum_fixed_size += space;
696 
697                             // Zelle schließen
698                             wsprintf(Ausgabebuf, L"</td>");
699                             buffer = Ausgabe(buffer, L"</td>", Ausgabebuf, 0);
700 
701                             printed_columns++;
702                         }
703 
704 
705                         // variable sized Spalten
706                         INT32 sum_variable_size = 0;
707                         variable_count = 0;
708 
709                         if (last_variable_id >= 0x80)
710                         {
711                             variable_count = last_variable_id - 0x7f;  // Anzahl variable size Spalten berechnen
712                         }
713 
714                         variable_data_offset = variable_offset + 2 * variable_count;     // Offset der variable sized data nach array
715 
716                         // prüfen ob Spalten aufgefüllt werden müssen
717                         if (variable_count > 0)
718                         {
719                             while (data_definitions_table[printed_columns][2] != 0x80)
720                             {
721                                 // leere Spalte
722                                 wsprintf(Ausgabebuf, L"<td></td>");
723                                 buffer = Ausgabe(buffer, L"<td></td>", Ausgabebuf, 0);
724                                 printed_columns++;
725                             }
726                         }
727 
728                         // für jeden variable Datentyp
```

```
729                     for (int v = 0; v < variable_count; v++)
730                     {
731                         // neue Zelle öffnen
732                         wsprintf(Ausgabebuf, L"<td>");
733                         buffer = Ausgabe(buffer, L"<td>",
                    Ausgabebuf, 0);
734
735                         // Größe bestimmen
736                         ptr = &file_buf[variable_offset + 2 * v];
                                        // Größe aus Array
                    bestimmen
737                         variable_size = *((uint16_t*)ptr);
738                         variable_size = (variable_size & 0x7FFF) -
                     sum_variable_size;     // - MSB - Summe
739
740                         // Daten lesen und schreiben
741                         if (variable_size > 0)
742                         {
743                             // column type aus array holen
744                             column_type_or_fdp =
                    data_definitions_table[printed_columns][3];
745
746                             // Spaltentyp auswerten und ausgegeben
747                             int x = 1;
748                             switch (column_type_or_fdp)
749                             {
750                             case 0:
751                                 break;
752                             case 1:
753                                 data_8bit = file_buf
                    [variable_data_offset];
754                                 wsprintf(Ausgabebuf, L"%3u",
                    data_8bit);
755                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 3);
756                                 break;
757                             case 2:
758                                 data_8bit = file_buf
                    [variable_data_offset];
759                                 wsprintf(Ausgabebuf, L"%3u",
                    data_8bit);
760                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 3);
761                                 break;
762                             case 3:
763                                 ptr = &file_buf
                    [variable_data_offset];
764                                 data_16bit = *((uint16_t*)ptr);
765                                 wsprintf(Ausgabebuf, L"%6d",
                    data_16bit);
766                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 6);
767                                 break;
```

```
768                                 case 4:
769                                     ptr = &file_buf                    ↪
                    [variable_data_offset];
770                                     data_32bit = *((uint32_t*)ptr);
771                                     wsprintf(Ausgabebuf, L"%11d",      ↪
                    data_32bit);
772                                     buffer = Ausgabe(buffer, L"",      ↪
                    Ausgabebuf, 11);
773                                     break;
774                                 case 5:
775                                     ptr = &file_buf                    ↪
                    [variable_data_offset];
776                                     data_64bit = *((uint64_t*)ptr);
777                                     wsprintf(Ausgabebuf, L"%20d",      ↪
                    data_64bit);
778                                     buffer = Ausgabe(buffer, L"",      ↪
                    Ausgabebuf, 20);
779                                     break;
780                                 case 6:
781                                     ptr = &file_buf                    ↪
                    [variable_data_offset];
782                                     data_32bit = *((uint32_t*)ptr);
783                                     memcpy(&data_float, &data_32bit,   ↪
                    sizeof(float));
784                                     wsprintf(Ausgabebuf, L"%08X",      ↪
                    data_float);
785                                     buffer = Ausgabe(buffer, L"",      ↪
                    Ausgabebuf, 8);
786                                     break;
787                                 case 7:
788                                     ptr = &file_buf                    ↪
                    [variable_data_offset];
789                                     data_64bit = *((uint64_t*)ptr);
790                                     memcpy(&data_float, &data_64bit,   ↪
                    sizeof(double));
791                                     wsprintf(Ausgabebuf, L"%016X",     ↪
                    data_double);
792                                     buffer = Ausgabe(buffer, L"",      ↪
                    Ausgabebuf, 16);
793                                     break;
794                                 case 8:
795                                     for (int z = 0; z < variable_size; ↪
                     z++)
796                                     {
797                                         data_8bit = file_buf           ↪
                    [offset_data + z];
798                                         wsprintf(Ausgabebuf, L"%02X",  ↪
                    data_8bit);
799                                         buffer = Ausgabe(buffer, L"",  ↪
                    Ausgabebuf, 2);
800                                     }
801                                     break;
802                                 case 9:
```

```
803                                 for (int z = 0; z < variable_size; z++)
804                                 {
805                                     data_8bit = file_buf[variable_data_offset + z];
806                                     wsprintf(Ausgabebuf, L"%02X", data_8bit);
807                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 2);
808                                 }
809                                 break;
810                             case 10:
811                                 for (int z = 0; z < variable_size; z++)
812                                 {
813                                     zeichen = file_buf[variable_data_offset + z];
814                                     wsprintf(Ausgabebuf, L"%c", zeichen);
815                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
816                                 }
817                                 break;
818                             case 11:
819                                 for (int z = 0; z < variable_size; z++)
820                                 {
821                                     data_8bit = file_buf[variable_data_offset + z];
822                                     wsprintf(Ausgabebuf, L"%02X", data_8bit);
823                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 2);
824                                 }
825                                 break;
826                             case 12:
827                                 for (int z = 0; z < variable_size; z++)
828                                 {
829                                     zeichen = file_buf[variable_data_offset + z];
830                                     wsprintf(Ausgabebuf, L"%c", zeichen);
831                                     buffer = Ausgabe(buffer, L"", Ausgabebuf, 1);
832                                 }
833                                 break;
834                             case 13:
835                                 break;
836                             case 14:
837                                 ptr = &file_buf[variable_data_offset];
838                                 data_32bit = *((uint32_t*)ptr);
```

```
839                                 wsprintf(Ausgabebuf, L"%10u",       ↪
                    data_32bit);
840                                 buffer = Ausgabe(buffer, L"",       ↪
                    Ausgabebuf, 10);
841                                 break;
842                             case 15:
843                                 ptr = &file_buf                     ↪
                    [variable_data_offset];
844                                 data_64bit = *((uint64_t*)ptr);
845                                 wsprintf(Ausgabebuf, L"%20d",       ↪
                    data_64bit);
846                                 buffer = Ausgabe(buffer, L"",       ↪
                    Ausgabebuf, 20);
847                                 break;
848                             case 16:
849                                 for (int z = 1; z <= 4; z++)
850                                 {
851                                     data_8bit = file_buf            ↪
                    [variable_data_offset + 16 - z]; // vom Ende auslesen↪
                     -> Little Endian
852                                     wsprintf(Ausgabebuf, L"%02X",   ↪
                    data_8bit);
853                                     buffer = Ausgabe(buffer, L"",   ↪
                    Ausgabebuf, 2);
854                                 }
855                                 wsprintf(Ausgabebuf, L"-");
856                                 buffer = Ausgabe(buffer, L"-",      ↪
                    Ausgabebuf, 0);
857                                 for (int z = 1; z <= 3; z++)
858                                 {
859                                     for (int k = 1; k <= 2; k++)
860                                     {
861                                         data_8bit = file_buf        ↪
                    [variable_data_offset + 12 - x];
862                                         wsprintf(Ausgabebuf, L"%    ↪
                    02X", data_8bit);
863                                         buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 2);
864                                         x++;
865                                     }
866                                     wsprintf(Ausgabebuf, L"-");
867                                     buffer = Ausgabe(buffer, L"-",  ↪
                     Ausgabebuf, 0);
868                                 }
869                                 for (int z = 1; z <= 6; z++)
870                                 {
871                                     data_8bit = file_buf            ↪
                    [variable_data_offset + 6 - z];
872                                     wsprintf(Ausgabebuf, L"%02X",   ↪
                    data_8bit);
873                                     buffer = Ausgabe(buffer, L"",   ↪
                    Ausgabebuf, 2);
874                                 }
```

```
875                                 break;
876                             case 17:
877                                 ptr = &file_buf
                    [variable_data_offset];
878                                 data_16bit = *((uint16_t*)ptr);
879                                 wsprintf(Ausgabebuf, L"%5u",
                    data_16bit);
880                                 buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 5);
881                                 break;
882                             default: // als Binärdaten ausgeben
883                                 for (int z = 0; z < variable_size;
                     z++)
884                                 {
885                                     data_8bit = file_buf
                    [variable_data_offset + z];
886                                     wsprintf(Ausgabebuf, L"%02X",
                    data_8bit);
887                                     buffer = Ausgabe(buffer, L"",
                    Ausgabebuf, 2);
888                                 }
889                             }
890
891                         }
892
893                         // Offset aktualisieren
894                         variable_data_offset += variable_size;
895
896                         // Gesamtgröße aufsummieren
897                         sum_variable_size += variable_size;
898
899                         // Zelle schließen
900                         wsprintf(Ausgabebuf, L"</td>");
901                         buffer = Ausgabe(buffer, L"</td>",
                    Ausgabebuf, 0);
902
903                         printed_columns++;
904                     }
905
906
907                     // tagged Spalten
908                     bool tagged_data = 0;               // Tagged
                    Data vorhanden ja / nein
909                     INT16 tagged_data_offset = 0;       // Offset
                    Tagged Data
910                     INT16 tagged_id = 0;                // ID
                    Tagged Data Column
911                     size_t tagged_offset = 0;           // Offset
                    Tagged Data Array
912
913                     if (variable_count > 0) // variable sized
                    spalten vorhanden
914                     {
```

```
915                         // prüfen ob tagged Spalten vorhanden
916                         if (page_offset + data_offset + tag_array
                    [2 * j] + tag_array[2 * j + 1] ==
                    variable_data_offset) // Ende Record = Ende variable
                    Data?
917                         {
918                             // leere Spalten auffüllen
919                             for (printed_columns; printed_columns
                    < column_count; printed_columns++)
920                             {
921                                 // leere Zelle anfügen
922                                 wsprintf(Ausgabebuf, L"<td></
                    td>");
923                                 buffer = Ausgabe(buffer, L"<td></
                    td>", Ausgabebuf, 0);
924                             }
925                             tagged_data = 0;
926
927                         }
928                         else
929                         {
930                             tagged_offset = variable_data_offset;
931                             tagged_data = 1;
932                         }
933                     }
934                     else // variable_count = 0: keine variable
                    size data
935                     {
936                         // prüfen ob tagged Spalten vorhanden
937                         if (page_offset + data_offset + tag_array
                    [2 * j] + tag_array[2 * j + 1] == variable_offset) //
                     Ende Record = Start variable Data?
938                         {
939                             // leere Spalten auffüllen
940                             for (printed_columns; printed_columns
                    < column_count; printed_columns++)
941                             {
942                                 // leere Zelle anfügen
943                                 wsprintf(Ausgabebuf, L"<td></
                    td>");
944                                 buffer = Ausgabe(buffer, L"<td></
                    td>", Ausgabebuf, 0);
945                             }
946                             tagged_data = 0;
947                         }
948                         else
949                         {
950                             tagged_offset = variable_offset;
951                             tagged_data = 1;
952                         }
953                     }
954
955                     if (tagged_data == 1)
```

```
956                         {
957                             // tagged data offset
958                             ptr = &file_buf[tagged_offset + 2];
959                             tagged_data_offset = *((uint16_t*)ptr); // ersten Offset lesen
960
961                             INT16 tagged_data_array[50][2] = {}; // leeres Array initialisieren -> id und offset speichern
962
963                             int tagged_count = 0;
964                             tagged_count = tagged_data_offset / 4; // Anzahl tagged Data Einträge berechnen: jeder Eintrag im Array = 4 Byte
965
966                             for (int t = 0; t < tagged_count; t++) // für jeden tagged data type: tagged data array füllen -> id und offset
967                             {
968                                 // tagged id
969                                 ptr = &file_buf[tagged_offset + 4 * t];
970                                 tagged_id = *((uint16_t*)ptr);
971
972                                 // tagged data offset
973                                 ptr = &file_buf[tagged_offset + 2 + 4 * t];
974                                 tagged_data_offset = *((uint16_t*)ptr);
975
976                                 tagged_data_array[t][0] = tagged_id;
977                                 tagged_data_array[t][1] = tagged_data_offset;
978                             }
979
980                             // id spalte zuordnen
981                             for (int a = 0; a < tagged_count; a++)
982                             {
983                                 tagged_id = tagged_data_array[a][0];
984
985                                 // Größe Tagged data berechnen: bis nächster Offset oder Ende Record
986                                 if (a + 1 < tagged_count) // nicht letzter tagged data type -> nächster Offset - aktueller Offset
987                                 {
988                                     tagged_data_size = tagged_data_array[a + 1][1] - tagged_data_array[a][1] - 1; // - 1, weil erstes Byte ist Flag
989                                 }
990                                 else // letzter tagged data type -> Ende Record - Offset
991                                 {
```

```
992                             tagged_data_size = (page_offset + data_offset + tag_array[2 * j] + tag_array[2 * j + 1]) - (tagged_offset + tagged_data_array[a][1]) - 1;
993                         }
994
995                         for (int b = 0; b < column_count; b++) // id suchen
996                         {
997                             if (data_definitions_table[b][2] == tagged_id) // wenn id übereinstimmt
998                             {
999                                 for (printed_columns; printed_columns < b; printed_columns++) // leere Zellen auffüllen
1000                                {
1001                                    wsprintf(Ausgabebuf, L"<td></td>"); // 1 leere Zelle anfügen
1002                                    buffer = Ausgabe(buffer, L"<td></td>", Ausgabebuf, 0);
1003                                }
1004                                wsprintf(Ausgabebuf, L"<td>"); // Zelle öffnen
1005                                buffer = Ausgabe(buffer, L"<td>", Ausgabebuf, 0);
1006
1007                                // column type aus array holen
1008                                column_type_or_fdp = data_definitions_table[b][3];
1009
1010                                // Offset berechnen
1011                                offset_data = tagged_offset + tagged_data_array[a][1] + 1;
1012
1013                                // Spaltentyp auswerten und ausgegeben
1014                                int x = 1;
1015                                switch (column_type_or_fdp)
1016                                {
1017                                case 0:
1018                                    break;
1019                                case 1:
1020                                    data_8bit = file_buf [offset_data];
1021                                    wsprintf(Ausgabebuf, L"%3u", data_8bit);
1022                                    buffer = Ausgabe(buffer, L"", Ausgabebuf, 3);
1023                                    break;
1024                                case 2:
1025                                    data_8bit = file_buf [offset_data];
1026                                    wsprintf(Ausgabebuf, L"%3u", data_8bit);
```

```
1027                                        buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 3);
1028                                        break;
1029                                    case 3:
1030                                        ptr = &file_buf             ↪
                    [offset_data];
1031                                        data_16bit = *((uint16_t*)  ↪
                    ptr);
1032                                        wsprintf(Ausgabebuf, L"%    ↪
                    6d", data_16bit);
1033                                        buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 6);
1034                                        break;
1035                                    case 4:
1036                                        ptr = &file_buf             ↪
                    [offset_data];
1037                                        data_32bit = *((uint32_t*)  ↪
                    ptr);
1038                                        wsprintf(Ausgabebuf, L"%    ↪
                    11d", data_32bit);
1039                                        buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 11);
1040                                        break;
1041                                    case 5:
1042                                        ptr = &file_buf             ↪
                    [offset_data];
1043                                        data_64bit = *((uint64_t*)  ↪
                    ptr);
1044                                        wsprintf(Ausgabebuf, L"%    ↪
                    20d", data_64bit);
1045                                        buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 20);
1046                                        break;
1047                                    case 6:
1048                                        ptr = &file_buf             ↪
                    [offset_data];
1049                                        data_32bit = *((uint32_t*)  ↪
                    ptr);
1050                                        memcpy(&data_float,         ↪
                    &data_32bit, sizeof(float));
1051                                        wsprintf(Ausgabebuf, L"%    ↪
                    08X", data_float);
1052                                        buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 8);
1053                                        break;
1054                                    case 7:
1055                                        ptr = &file_buf             ↪
                    [offset_data];
1056                                        data_64bit = *((uint64_t*)  ↪
                    ptr);
1057                                        memcpy(&data_float,         ↪
                    &data_64bit, sizeof(double));
1058                                        wsprintf(Ausgabebuf, L"%    ↪
```

```
                    016X", data_double);
1059                                    buffer = Ausgabe(buffer,    ↪
                    L"", Ausgabebuf, 16);
1060                                    break;
1061                                case 8:
1062                                    for (int z = 0; z <         ↪
                    tagged_data_size; z++)
1063                                    {
1064                                        data_8bit = file_buf    ↪
                    [offset_data + z];
1065                                        wsprintf(Ausgabebuf,    ↪
                    L"%02X", data_8bit);
1066                                        buffer = Ausgabe        ↪
                    (buffer, L"", Ausgabebuf, 2);
1067                                    }
1068                                    break;
1069                                case 9:
1070                                    for (int z = 0; z <         ↪
                    tagged_data_size; z++)
1071                                    {
1072                                        data_8bit = file_buf    ↪
                    [offset_data + z];
1073                                        wsprintf(Ausgabebuf,    ↪
                    L"%02X", data_8bit);
1074                                        buffer = Ausgabe        ↪
                    (buffer, L"", Ausgabebuf, 2);
1075                                    }
1076                                    break;
1077                                case 10:
1078                                    for (int z = 0; z <         ↪
                    tagged_data_size; z++)
1079                                    {
1080                                        zeichen = file_buf      ↪
                    [offset_data + z];
1081                                        wsprintf(Ausgabebuf,    ↪
                    L"%c", zeichen);
1082                                        buffer = Ausgabe        ↪
                    (buffer, L"", Ausgabebuf, 1);
1083                                    }
1084                                    break;
1085                                case 11:
1086                                    for (int z = 0; z <         ↪
                    tagged_data_size; z++)
1087                                    {
1088                                        data_8bit = file_buf    ↪
                    [offset_data + z];
1089                                        wsprintf(Ausgabebuf,    ↪
                    L"%02X", data_8bit);
1090                                        buffer = Ausgabe        ↪
                    (buffer, L"", Ausgabebuf, 2);
1091                                    }
1092                                    break;
1093                                case 12:
```

```
1094                                    for (int z = 0; z <
                    tagged_data_size; z++)
1095                                    {
1096                                        zeichen = file_buf
                    [offset_data + z];
1097                                        wsprintf(Ausgabebuf,
                    L"%c", zeichen);
1098                                        buffer = Ausgabe
                    (buffer, L"", Ausgabebuf, 1);
1099                                    }
1100                                    break;
1101                                case 13:
1102                                    break;
1103                                case 14:
1104                                    ptr = &file_buf
                    [offset_data];
1105                                    data_32bit = *((uint32_t*)
                    ptr);
1106                                    wsprintf(Ausgabebuf, L"%
                    10u", data_32bit);
1107                                    buffer = Ausgabe(buffer,
                    L"", Ausgabebuf, 10);
1108                                    break;
1109                                case 15:
1110                                    ptr = &file_buf
                    [offset_data];
1111                                    data_64bit = *((uint64_t*)
                    ptr);
1112                                    wsprintf(Ausgabebuf, L"%
                    20d", data_64bit);
1113                                    buffer = Ausgabe(buffer,
                    L"", Ausgabebuf, 20);
1114                                    break;
1115                                case 16:
1116                                    for (int z = 1; z <= 4; z+
                    +)
1117                                    {
1118                                        data_8bit = file_buf
                    [offset_data + 16 - z]; // vom Ende auslesen ->
                    Little Endian
1119                                        wsprintf(Ausgabebuf,
                    L"%02X", data_8bit);
1120                                        buffer = Ausgabe
                    (buffer, L"", Ausgabebuf, 2);
1121                                    }
1122                                    wsprintf(Ausgabebuf,
                    L"-");
1123                                    buffer = Ausgabe(buffer,
                    L"-", Ausgabebuf, 0);
1124                                    for (int z = 1; z <= 3; z+
                    +)
1125                                    {
1126                                        for (int k = 1; k <=
```

```
                    2; k++)
1127                                    {
1128                                        data_8bit =
                    file_buf[offset_data + 12 - x];
1129                                        wsprintf
                    (Ausgabebuf, L"%02X", data_8bit);
1130                                        buffer = Ausgabe
                    (buffer, L"", Ausgabebuf, 2);
1131                                        x++;
1132                                    }
1133                                    wsprintf(Ausgabebuf,
                    L"-");
1134                                    buffer = Ausgabe
                    (buffer, L"-", Ausgabebuf, 0);
1135                                }
1136                                for (int z = 1; z <= 6; z+
                    +)
1137                                {
1138                                    data_8bit = file_buf
                    [offset_data + 6 - z];
1139                                    wsprintf(Ausgabebuf,
                    L"%02X", data_8bit);
1140                                    buffer = Ausgabe
                    (buffer, L"", Ausgabebuf, 2);
1141                                }
1142                                break;
1143                            case 17:
1144                                ptr = &file_buf
                    [offset_data];
1145                                data_16bit = *((uint16_t*)
                    ptr);
1146                                wsprintf(Ausgabebuf, L"%
                    5u", data_16bit);
1147                                buffer = Ausgabe(buffer,
                    L"", Ausgabebuf, 5);
1148                                break;
1149                            default: // als Binärdaten
                    ausgeben
1150                                for (int z = 0; z <
                    tagged_data_size; z++)
1151                                {
1152                                    data_8bit = file_buf
                    [offset_data + z];
1153                                    wsprintf(Ausgabebuf,
                    L"%02X", data_8bit);
1154                                    buffer = Ausgabe
                    (buffer, L"", Ausgabebuf, 2);
1155                                }
1156                            }
1157
1158                            wsprintf(Ausgabebuf, L"</
                    td>"); // Zelle schließen
1159                            buffer = Ausgabe(buffer, L"</
```

```
                    td>", Ausgabebuf, 0);
1160                                    printed_columns++;
1161                                }
1162                            }
1163
1164                            }
1165                        }
1166
1167                        if (printed_columns < column_count)
1168                        {
1169                            for (printed_columns; printed_columns <
                    column_count; printed_columns++)
1170                            {
1171                                wsprintf(Ausgabebuf, L"<td></td>"); //
                     1 leere Zelle anfügen
1172                                buffer = Ausgabe(buffer, L"<td></td>",
                     Ausgabebuf, 0);
1173                            }
1174                        }
1175
1176                        // Tabellenzeile schließen
1177                        wsprintf(Ausgabebuf, L"</tr>");
1178                        buffer = Ausgabe(buffer, L"</tr>", Ausgabebuf,
                    0);
1179
1180                     }
1181
1182                }
1183                page_number = page_array[next_page];
1184                next_page++;
1185            }
1186
1187            // Tabelle schließen
1188            wsprintf(Ausgabebuf, L"</table><br><br>");
1189            buffer = Ausgabe(buffer, L"</table><br><br>", Ausgabebuf,
                  0);
1190        }
1191    }
1192
1193    HeapFree(heap, 0, file_buf);
1194
1195    // HTML schließen
1196    wsprintf(Ausgabebuf, L"</BODY></HTML>");
1197    buffer = Ausgabe(buffer, L"</BODY></HTML>", Ausgabebuf, 0);
1198
1199    *lpResSize = Speichernutzung * 2;
1200
1201    return buffer;
1202 }
1203
1204
1205
1206 ////////////////////////////////////////////////////////////////////////
```

```
      /////////
// XT_ReleaseMem

BOOL __stdcall XT_ReleaseMem(PVOID lpBuffer)
{
    if (lpBuffer != 0)
    {
        free(lpBuffer);
        SpeicherZugeordnet = 1024;
        Speichernutzung = 0;
        page_offset = 0;
        dd_header_offset = 0;
        variable_offset = 0;
        record_offset = 0;
        return TRUE;
    }
    else {
        return FALSE;
    }
}


////////////////////////////////////////////////////////////////////// ↵
      /////////
// XT_Done
// optional

LONG __stdcall XT_Done(void* lpReserved)
{
    XWF_OutputMessage(L"XT_ESEDBViewer done", 0);
    return 0;
}


wchar_t* Ausgabe(wchar_t* buffer, const wchar_t* Zeile, wchar_t*       ↵
  Ausgabebuf, int zeichencount)
{
    // Anzahl zu schreibenden Zeichen zählen
    size_t Zeilenlaenge = wcslen(Zeile) + zeichencount;             // ↵
       wcslen: Länge der Zeichenfolge

    // Erweiterung des Speichers wenn Speicherbedarf zu groß
    while (Speichernutzung + Zeilenlaenge >= SpeicherZugeordnet)
    {
        // Vergößerung
        SpeicherZugeordnet *= 2;

        // Speicherzuordnung erfolgreich
        buffer = (wchar_t*)realloc(buffer, SpeicherZugeordnet * sizeof ↵
          (wchar_t));     // realloc: Neubelegung von                  ↵
          Arbeitsspeicherblöcken.
        if ((buffer == NULL))
        {
```

```
1254              XWF_OutputMessage(L"ERROR: Fehler beim Zuordnen des
                    größeren Speichers", 0);
1255              return NULL;
1256          }
1257      }
1258      // Neuen Inhalt an Speicher anhängen
1259      wmemcpy(buffer + Speichernutzung, Ausgabebuf, Zeilenlaenge);
                        // wmemcpy: Kopiert Bytes zwischen Puffern.
1260
                // (Zielbuffer + aktueller Inhalt, Quellbuffer, Anzahl der
     zu kopierenden Zeichen)
1261
1262      // Aktuelle Speichernutzung festhalten
1263      Speichernutzung += Zeilenlaenge;
1264
1265      return buffer;
1266  }
1267
1268  size_t ddHeaderOffset(int page_key_flags, size_t key_offset, BYTE*
       file_buf)
1269  {
1270      // Schlüssel Flags auswerten
1271      if ((page_key_flags & 128) != 128) // case 1, flags: 0xx
1272      {
1273          // Jump Size an Key_Offset und dd_header Offset bestimmen
1274          jump_size = file_buf[key_offset];
1275          dd_header_offset = key_offset + 2 + jump_size;
1276      }
1277      else // case 2, flags 1xx
1278      {
1279          // Jump Size an key_offset + 2 und dd_header offset bestimmen
1280          ptr = &file_buf[key_offset + 2];
1281          jump_size = *((uint16_t*)ptr);
1282          dd_header_offset = key_offset + 4 + jump_size;
1283      }
1284      return dd_header_offset;
1285  }
1286
1287  void ddHeader(BYTE* file_buf, size_t dd_header_offset)
1288  {
1289      last_fixed_id = file_buf[dd_header_offset];
1290      last_variable_id = file_buf[dd_header_offset + 1];
1291      ptr = &file_buf[dd_header_offset + 2];
1292      relative_variable_offset = *((uint16_t*)ptr);
                  // relativ zu dd_header
1293      variable_offset = dd_header_offset + relative_variable_offset;
1294      record_offset = dd_header_offset + 4;
                  // Start der Daten nach Data Definition Header
1295  }
1296
1297
1298  size_t pageOffset(size_t page_number, INT32 page_size)
1299  {
```

```
1300      // Offset zu PageNumber bestimmen
1301      page_offset = (page_number + 1) * page_size;
1302      return page_offset;
1303  }
1304
1305
```