

2cc0af7fe

...

gpmf-parser / GPMF_parser.c



dnewman-gpsw version bump 1.6.2

History

1 contributor

2066 lines (1758 sloc) 55.6 KB

...

```
1  /*! @file GPMF_parser.c
2  *
3  * @brief GPMF Parser library
4  *
5  * @version 1.6.2
6  *
7  * (C) Copyright 2017-2020 GoPro Inc (http://gopro.com/).
8  *
9  * Licensed under either:
10 * - Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0
11 * - MIT license, http://opensource.org/licenses/MIT
12 * at your option.
13 *
14 * Unless required by applicable law or agreed to in writing, software
15 * distributed under the License is distributed on an "AS IS" BASIS,
16 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
17 * See the License for the specific language governing permissions and
18 * limitations under the License.
19 *
20 */
21
22 #include <stdlib.h>
23 #include <stdio.h>
24 #include <string.h>
25 #include <stdint.h>
26
27 #include "GPMF_parser.h"
28 #include "GPMF_bitstream.h"
29
30
31 #ifdef DBG
32 #if _WIN32
33 #define DBG_MSG printf
34 #else
35 #define DBG_MSG(...)
36 #endif
37 #else
38 #define DBG_MSG(...)
39 #endif
40
41
42 GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size) // size is in longs not bytes.
43 {
44     if (ms)
45     {
46         uint32_t nestsize = (uint32_t)ms->nest_size[ms->nest_level];
47         if (nestsize == 0 && ms->nest_level == 0)
48             nestsize = ms->buffer_size_long;
49
50         if (size + 2 <= nestsize) return GPMF_OK;
51     }
52     return GPMF_ERROR_BAD_STRUCTURE;
53 }
54
55
56 GPMF_ERR GPMF_Validate(GPMF_stream *ms, GPMF_LEVELS recurse)
57 {
58     if (ms)
59     {
60         uint32_t curpos = ms->pos;
61         uint32_t nestsize = ms->nest_size[ms->nest_level];
62         if (nestsize == 0 && ms->nest_level == 0)
63             nestsize = ms->buffer_size_long;
64
65         while (ms->pos+1 < ms->buffer_size_long && nestsize > 0)
66         {
67             uint32_t key = ms->buffer[ms->pos];
68
69             if (ms->nest_level == 0 && key != GPMF_KEY_DEVICE && ms->device_count == 0 && ms->pos == 0)
70             {
71                 DBG_MSG("ERROR: uninitialized -- GPMF_ERROR_BAD_STRUCTURE\n");
72                 return GPMF_ERROR_BAD_STRUCTURE;
73             }
74
75             if (GPMF_VALID_FOURCC(key))
76             {
77                 uint32_t type_size_repeat = ms->buffer[ms->pos + 1];
78                 uint32_t size = GPMF_DATA_SIZE(type_size_repeat) >> 2;
```

```

79     uint8_t type = GPMF_SAMPLE_TYPE(type_size_repeat);
80     if (size + 2 > nestsize)
81     {
82         DBG_MSG("ERROR: nest size too small within %c%c%c%c-- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
83         return GPMF_ERROR_BAD_STRUCTURE;
84     }
85
86     if (!GPMF_VALID_FOURCC(key))
87     {
88         DBG_MSG("ERROR: invalid 4CC -- GPMF_ERROR_BAD_STRUCTURE\n");
89         return GPMF_ERROR_BAD_STRUCTURE;
90     }
91
92     if (type == GPMF_TYPE_NEST && recurse == GPMF_RECURSE_LEVELS)
93     {
94         uint32_t validnest;
95         ms->pos += 2;
96         ms->nest_level++;
97         if (ms->nest_level > GPMF_NEST_LIMIT)
98         {
99             DBG_MSG("ERROR: nest level within %c%c%c%c too deep -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
100            return GPMF_ERROR_BAD_STRUCTURE;
101        }
102        ms->nest_size[ms->nest_level] = size;
103        validnest = GPMF_Validate(ms, recurse);
104        ms->nest_level--;
105        if (GPMF_OK != validnest)
106        {
107            DBG_MSG("ERROR: invalid nest within %c%c%c%c -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
108            return GPMF_ERROR_BAD_STRUCTURE;
109        }
110        else
111        {
112            if (ms->nest_level == 0)
113                ms->device_count++;
114        }
115
116        ms->pos += size;
117        nestsize -= 2 + size;
118
119        while (ms->pos < ms->buffer_size_longs && nestsize > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
120        {
121            ms->pos++;
122            nestsize--;
123        }
124    }
125    else
126    {
127        ms->pos += 2 + size;
128        nestsize -= 2 + size;
129    }
130
131    if (ms->pos == ms->buffer_size_longs)
132    {
133        ms->pos = currpos;
134        return GPMF_OK;
135    }
136    }
137    else
138    {
139        if (key == GPMF_KEY_END)
140        {
141            do
142            {
143                ms->pos++;
144                nestsize--;
145            } while (ms->pos < ms->buffer_size_longs && nestsize > 0 && ms->buffer[ms->pos] == 0);
146        }
147        else if (ms->nest_level == 0 && ms->device_count > 0)
148        {
149            ms->pos = currpos;
150            return GPMF_OK;
151        }
152        else
153        {
154            DBG_MSG("ERROR: bad struct within %c%c%c%c -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
155            return GPMF_ERROR_BAD_STRUCTURE;
156        }
157    }
158    }
159
160    ms->pos = currpos;
161    return GPMF_OK;
162    }
163    else
164    {
165        DBG_MSG("ERROR: Invalid handle -- GPMF_ERROR_MEMORY\n");
166        return GPMF_ERROR_MEMORY;
167    }
168    }
169
170
171 GPMF_ERR GPMF_ResetState(GPMF_stream *ms)
172 {
173     if (ms)
174     {
175         ms->pos = 0;
176         ms->nest_level = 0;

```

```

177     ms->device_count = 0;
178     ms->nest_size[ms->nest_level] = 0;
179     ms->last_level_pos[ms->nest_level] = 0;
180     ms->last_seek[ms->nest_level] = 0;
181     ms->device_id = 0;
182     ms->device_name[0] = 0;
183
184     return GPMF_OK;
185 }
186
187 return GPMF_ERROR_MEMORY;
188 }
189
190
191 GPMF_ERR GPMF_Init(GPMF_stream *ms, uint32_t *buffer, uint32_t datasize)
192 {
193     if(ms && buffer && datasize > 0)
194     {
195         uint32_t pos = 0;
196         //Validate DEVC GPMF
197         while((pos+1) * 4 < datasize && buffer[pos] == GPMF_KEY_DEVICE)
198         {
199             uint32_t size = GPMF_DATA_SIZE(buffer[pos+1]);
200             pos += 2 + (size >> 2);
201         }
202         if ((pos*4) < datasize && buffer[pos] == GPMF_KEY_END) // NULL terminated GPMF
203         {
204             datasize = pos * 4;
205         }
206         if (pos * 4 == datasize)
207         {
208             ms->buffer = buffer;
209             ms->buffer_size longs = (datasize + 3) >> 2;
210             ms->cbhandle = 0;
211
212             GPMF_ResetState(ms);
213
214             return GPMF_OK;
215         }
216         else
217         {
218             return GPMF_ERROR_BAD_STRUCTURE;
219         }
220     }
221
222     return GPMF_ERROR_MEMORY;
223 }
224
225
226 GPMF_ERR GPMF_CopyState(GPMF_stream *msrc, GPMF_stream *mdst)
227 {
228     if (msrc && mdst)
229     {
230         memcpy(mdst, msrc, sizeof(GPMF_stream));
231         return GPMF_OK;
232     }
233     return GPMF_ERROR_MEMORY;
234 }
235
236
237 GPMF_ERR GPMF_Next(GPMF_stream *ms, GPMF_LEVELS recurse)
238 {
239     if (ms)
240     {
241         if (ms->pos+1 < ms->buffer_size longs)
242         {
243
244             uint32_t key, type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
245             uint32_t size = (GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]) >> 2);
246
247             if (GPMF_OK != IsValidSize(ms, size)) return GPMF_ERROR_BAD_STRUCTURE;
248
249             if (GPMF_TYPE_NEST == type && GPMF_KEY_DEVICE == ms->buffer[ms->pos] && ms->nest_level == 0)
250             {
251                 ms->last_level_pos[ms->nest_level] = ms->pos;
252                 ms->nest_size[ms->nest_level] = size;
253                 if (recurse)
254                     ms->pos += 2;
255                 else
256                     ms->pos += 2 + size;
257             }
258             else
259             {
260                 if (size + 2 > ms->nest_size[ms->nest_level])
261                     return GPMF_ERROR_BAD_STRUCTURE;
262
263                 if (recurse && type == GPMF_TYPE_NEST)
264                 {
265                     ms->last_level_pos[ms->nest_level] = ms->pos;
266                     ms->pos += 2;
267                     ms->nest_size[ms->nest_level] -= size + 2;
268
269                     ms->nest_level++;
270                     if (ms->nest_level > GPMF_NEST_LIMIT)
271                         return GPMF_ERROR_BAD_STRUCTURE;
272
273                     ms->nest_size[ms->nest_level] = size;
274                 }

```

```

275         else
276         {
277             if (recurse)
278             {
279                 ms->pos += size + 2;
280                 ms->nest_size[ms->nest_level] -= size + 2;
281             }
282             else
283             {
284                 if (ms->nest_size[ms->nest_level] - (size + 2) > 0)
285                 {
286                     ms->pos += size + 2;
287                     ms->nest_size[ms->nest_level] -= size + 2;
288                 }
289                 else
290                 {
291                     return GPMF_ERROR_LAST;
292                 }
293             }
294         }
295     }
296
297     while (ms->pos < ms->buffer_size_long && ms->nest_size[ms->nest_level] > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
298     {
299         ms->pos++;
300         ms->nest_size[ms->nest_level]--;
301     }
302
303     while (ms->nest_level > 0 && ms->nest_size[ms->nest_level] == 0)
304     {
305         ms->nest_level--;
306         //if (ms->nest_level == 0)
307         //{
308             //    ms->device_count++;
309         //}
310     }
311
312     if (ms->pos < ms->buffer_size_long)
313     {
314         while (ms->pos < ms->buffer_size_long && ms->nest_size[ms->nest_level] > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
315         {
316             ms->pos++;
317             ms->nest_size[ms->nest_level]--;
318         }
319
320         key = ms->buffer[ms->pos];
321         if (!GPMF_VALID_FOURCC(key))
322             return GPMF_ERROR_BAD_STRUCTURE;
323
324         if (key == GPMF_KEY_DEVICE_ID)
325             ms->device_id = BYTESWAP32(ms->buffer[ms->pos + 2]);
326         if (key == GPMF_KEY_DEVICE_NAME)
327         {
328             size = GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]); // in bytes
329             if (size > sizeof(ms->device_name) - 1)
330                 size = sizeof(ms->device_name) - 1;
331             memcpy(ms->device_name, &ms->buffer[ms->pos + 2], size);
332             ms->device_name[size] = 0;
333         }
334     }
335     else
336     {
337         // end of buffer
338         return GPMF_ERROR_BUFFER_END;
339     }
340
341     return GPMF_OK;
342 }
343
344 else
345 {
346     // end of buffer
347     return GPMF_ERROR_BUFFER_END;
348 }
349
350 return GPMF_ERROR_MEMORY;
351 }
352
353
354 GPMF_ERR GPMF_FindNext(GPMF_stream *ms, uint32_t fourcc, GPMF_LEVELS recurse)
355 {
356     GPMF_stream prevstate;
357
358     if (ms)
359     {
360         memcpy(&prevstate, ms, sizeof(GPMF_stream));
361
362         if (ms->pos < ms->buffer_size_long)
363         {
364             while (0 == GPMF_Next(ms, recurse))
365             {
366                 if (ms->buffer[ms->pos] == fourcc)
367                 {
368                     return GPMF_OK; //found match
369                 }
370             }
371
372             // restore read position

```

```

373         memcpy(ms, &prevstate, sizeof(GPMF_stream));
374         return GPMF_ERROR_FIND;
375     }
376 }
377 return GPMF_ERROR_FIND;
378 }
379
380 GPMF_ERR GPMF_Reserved(uint32_t key)
381 {
382     if(key == GPMF_KEY_DEVICE)
383         return GPMF_ERROR_RESERVED;
384
385     if(key == GPMF_KEY_DEVICE_ID)
386         return GPMF_ERROR_RESERVED;
387
388     if(key == GPMF_KEY_DEVICE_NAME)
389         return GPMF_ERROR_RESERVED;
390
391     if(key == GPMF_KEY_STREAM)
392         return GPMF_ERROR_RESERVED;
393
394     if(key == GPMF_KEY_STREAM_NAME)
395         return GPMF_ERROR_RESERVED;
396
397     if(key == GPMF_KEY_SI_UNITS)
398         return GPMF_ERROR_RESERVED;
399
400     if(key == GPMF_KEY_UNITS)
401         return GPMF_ERROR_RESERVED;
402
403     if(key == GPMF_KEY_SCALE)
404         return GPMF_ERROR_RESERVED;
405
406     if(key == GPMF_KEY_TYPE)
407         return GPMF_ERROR_RESERVED;
408
409     if(key == GPMF_KEY_TOTAL_SAMPLES)
410         return GPMF_ERROR_RESERVED;
411
412     if(key == GPMF_KEY_TICK)
413         return GPMF_ERROR_RESERVED;
414
415     if(key == GPMF_KEY_TOCK)
416         return GPMF_ERROR_RESERVED;
417
418     if(key == GPMF_KEY_EMPTY_PAYLOADS)
419         return GPMF_ERROR_RESERVED;
420
421     if(key == GPMF_KEY_REMARK)
422         return GPMF_ERROR_RESERVED;
423
424     return GPMF_OK;
425 }
426
427 uint32_t GPMF_PayloadSampleCount(GPMF_stream *ms)
428 {
429     uint32_t count = 0;
430     if (ms)
431     {
432         uint32_t fourcc = GPMF_Key(ms);
433
434         GPMF_stream find_stream;
435         GPMF_CopyState(ms, &find_stream);
436
437         if (GPMF_OK == GPMF_FindNext(&find_stream, fourcc, GPMF_CURRENT_LEVEL)) // Count the instances, not the repeats
438         {
439             count=2;
440             while (GPMF_OK == GPMF_FindNext(&find_stream, fourcc, GPMF_CURRENT_LEVEL))
441             {
442                 count++;
443             }
444         }
445         else
446         {
447             count = GPMF_Repeat(ms);
448             if (count == 0) // this can happen with an empty FACE, yet this is still a FACE fouce
449                 count = 1;
450         }
451     }
452     return count;
453 }
454
455
456 GPMF_ERR GPMF_SeekToSamples(GPMF_stream *ms)
457 {
458     GPMF_stream prevstate;
459
460     if (ms)
461     {
462         if (ms->pos+1 < ms->buffer_size_long)
463         {
464             uint32_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
465
466             memcpy(&prevstate, ms, sizeof(GPMF_stream));
467
468             if (type == GPMF_TYPE_NEST)
469                 GPMF_Next(ms, GPMF_RECURSE_LEVELS); // open STRM and recurse in
470

```

```

471         while (0 == GPMF_Next(ms, GPMF_CURRENT_LEVEL))
472         {
473             uint32_t size = (GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]) >> 2);
474             if (GPMF_OK != IsValidSize(ms, size))
475             {
476                 memcpy(ms, &prevstate, sizeof(GPMF_stream));
477                 return GPMF_ERROR_BAD_STRUCTURE;
478             }
479
480             type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
481
482             if (type == GPMF_TYPE_NEST) // Nest with-in nest
483             {
484                 return GPMF_OK; //found match
485             }
486
487             if (size + 2 == ms->nest_size[ms->nest_level])
488             {
489                 uint32_t key = GPMF_Key(ms);
490
491                 if (GPMF_ERROR_RESERVED == GPMF_Reserved(key))
492                     return GPMF_ERROR_FIND;
493
494                 return GPMF_OK; //found match
495             }
496
497             if (ms->buffer[ms->pos] == ms->buffer[ms->pos + size + 2]) // Matching tags
498             {
499                 return GPMF_OK; //found match
500             }
501         }
502
503         // restore read position
504         memcpy(ms, &prevstate, sizeof(GPMF_stream));
505         return GPMF_ERROR_FIND;
506     }
507 }
508 return GPMF_ERROR_FIND;
509 }
510 }
511 }
512
513 GPMF_ERR GPMF_FindPrev(GPMF_stream *ms, uint32_t fourcc, GPMF_LEVELS recurse)
514 {
515     GPMF_stream prevstate;
516
517     if (ms)
518     {
519         uint32_t curr_level = ms->nest_level;
520
521         memcpy(&prevstate, ms, sizeof(GPMF_stream));
522
523         if (ms->pos < ms->buffer_size_long && curr_level > 0)
524         {
525             do
526             {
527                 ms->last_seek[curr_level] = ms->pos;
528                 ms->pos = ms->last_level_pos[curr_level - 1] + 2;
529                 ms->nest_size[curr_level] += ms->last_seek[curr_level] - ms->pos;
530                 do
531                 {
532                     if (ms->last_seek[curr_level] > ms->pos && ms->buffer[ms->pos] == fourcc)
533                     {
534                         return GPMF_OK; //found match
535                     }
536                     } while (ms->last_seek[curr_level] > ms->pos && 0 == GPMF_Next(ms, GPMF_CURRENT_LEVEL));
537
538                 curr_level--;
539             } while (recurse == GPMF_RECURSE_LEVELS && curr_level > 0);
540
541             // restore read position
542             memcpy(ms, &prevstate, sizeof(GPMF_stream));
543             return GPMF_ERROR_FIND;
544         }
545     }
546
547     return GPMF_ERROR_FIND;
548 }
549
550 uint32_t GPMF_Key(GPMF_stream *ms)
551 {
552     if (ms)
553     {
554         uint32_t key = ms->buffer[ms->pos];
555         return key;
556     }
557     return 0;
558 }

```

```

569 GPMF_SampleType GPMF_Type(GPMF_stream *ms)
570 {
571     if (ms && ms->pos+1 < ms->buffer_size_longs)
572     {
573         GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos+1]);
574         if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
575         {
576             type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 2]);
577         }
578         return type;
579     }
580     return GPMF_TYPE_ERROR;
581 }
582
583
584 uint32_t GPMF_StructSize(GPMF_stream *ms)
585 {
586     if (ms && ms->pos+1 < ms->buffer_size_longs)
587     {
588         uint32_t ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
589         uint32_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
590         if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
591         {
592             ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2]);
593         }
594         return ssize;
595     }
596     return 0;
597 }
598
599
600 uint32_t GPMF_ElementsInStruct(GPMF_stream *ms)
601 {
602     if (ms && ms->pos+1 < ms->buffer_size_longs)
603     {
604         uint32_t ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
605         GPMF_SampleType type = (GPMF_SampleType) GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
606
607         if (type != GPMF_TYPE_NEST && type != GPMF_TYPE_COMPLEX && type != GPMF_TYPE_COMPRESSED)
608         {
609             uint32_t tsize = GPMF_SizeOfType(type);
610             if (tsize > 0)
611                 return ssize / tsize;
612             else
613                 return 0;
614         }
615         if (type == GPMF_TYPE_COMPLEX)
616         {
617             GPMF_stream find_stream;
618             GPMF_CopyState(ms, &find_stream);
619
620             if (GPMF_OK == GPMF_FindPrev(&find_stream, GPMF_KEY_TYPE, GPMF_CURRENT_LEVEL))
621             {
622                 char tmp[64] = "";
623                 uint32_t tmpsize = sizeof(tmp);
624                 char *data = (char *)GPMF_RawData(&find_stream);
625                 uint32_t size = GPMF_RawDataSize(&find_stream);
626
627                 if (GPMF_OK == GPMF_ExpandComplexTYPE(data, size, tmp, &tmpsize))
628                     return tmpsize;
629             }
630         }
631         if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
632         {
633             type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 2]);
634             ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2]);
635             uint32_t tsize = GPMF_SizeOfType(type);
636             if (tsize > 0)
637                 return ssize / tsize;
638             else
639                 return 0;
640         }
641     }
642     return 0;
643 }
644
645
646 uint32_t GPMF_Repeat(GPMF_stream *ms)
647 {
648     if (ms && ms->pos+1 < ms->buffer_size_longs)
649     {
650         GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
651         uint32_t repeat = GPMF_SAMPLES(ms->buffer[ms->pos + 1]);
652         if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
653         {
654             repeat = GPMF_SAMPLES(ms->buffer[ms->pos + 2]);
655         }
656         return repeat;
657     }
658     return 0;
659 }
660
661
662 uint32_t GPMF_RawDataSize(GPMF_stream *ms)
663 {
664     if (ms && ms->pos+1 < ms->buffer_size_longs)
665     {
666         uint32_t size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
667         if (GPMF_OK != IsValidSize(ms, size >> 2)) return 0;

```

```

667         return size;
668     }
669     return 0;
670 }
671
672 uint32_t GPMF_FormattedDataSize(GPMF_stream *ms)
673 {
674     if (ms && ms->pos + 1 < ms->buffer_size_longs)
675     {
676         GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
677         uint32_t size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1])*GPMF_SAMPLES(ms->buffer[ms->pos + 1]);
678
679         if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
680         {
681             size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2])*GPMF_SAMPLES(ms->buffer[ms->pos + 2]);
682         }
683         return size;
684     }
685     return 0;
686 }
687
688 uint32_t GPMF_ScaledDataSize(GPMF_stream *ms, GPMF_SampleType type)
689 {
690     if (ms && ms->pos + 1 < ms->buffer_size_longs)
691     {
692         uint32_t elements = GPMF_ElementsInStruct(ms);
693         uint32_t samples = GPMF_Repeat(ms);
694         return GPMF_SizeOfType(type) * elements * samples;
695     }
696     return 0;
697 }
698
699 uint32_t GPMF_NestLevel(GPMF_stream *ms)
700 {
701     if (ms)
702     {
703         return ms->nest_level;
704     }
705     return 0;
706 }
707
708 uint32_t GPMF_DeviceID(GPMF_stream *ms)
709 {
710     if (ms)
711     {
712         return ms->device_id;
713     }
714     return 0;
715 }
716
717 GPMF_ERR GPMF_DeviceName(GPMF_stream *ms, char *devicenamebuf, uint32_t devicename_buf_size)
718 {
719     if (ms && devicenamebuf)
720     {
721         uint32_t len = (uint32_t)strlen(ms->device_name);
722         if (len >= devicename_buf_size)
723             return GPMF_ERROR_MEMORY;
724
725         memcpy(devicenamebuf, ms->device_name, len);
726         devicenamebuf[len] = 0;
727         return GPMF_OK;
728     }
729     return GPMF_ERROR_MEMORY;
730 }
731
732 void *GPMF_RawData(GPMF_stream *ms)
733 {
734     if (ms)
735     {
736         return (void *)&ms->buffer[ms->pos + 2];
737     }
738     return NULL;
739 }
740
741 int32_t GPMFTypeEndianSize(int type)
742 {
743     int32_t ssize = -1;
744
745     switch ((int)type)
746     {
747         case GPMF_TYPE_STRING_ASCII:      ssize = 1; break;
748         case GPMF_TYPE_SIGNED_BYTE:       ssize = 1; break;
749         case GPMF_TYPE_UNSIGNED_BYTE:     ssize = 1; break;
750         case GPMF_TYPE_STRING_UTF8:       ssize = 1; break;
751
752         // These datatype can always be stored in Big-Endian
753         case GPMF_TYPE_SIGNED_SHORT:      ssize = 2; break;
754         case GPMF_TYPE_UNSIGNED_SHORT:    ssize = 2; break;
755         case GPMF_TYPE_FLOAT:             ssize = 4; break;
756         case GPMF_TYPE_FOURCC:            ssize = 1; break;
757         case GPMF_TYPE_SIGNED_LONG:       ssize = 4; break;
758         case GPMF_TYPE_UNSIGNED_LONG:     ssize = 4; break;
759         case GPMF_TYPE_Q15_16_FIXED_POINT: ssize = 4; break;
760     }
761 }

```



```

765     case GPMF_TYPE_Q31_32_FIXED_POINT:  ssize = 8; break;
766     case GPMF_TYPE_DOUBLE:              ssize = 8; break;
767     case GPMF_TYPE_SIGNED_64BIT_INT:    ssize = 8; break;
768     case GPMF_TYPE_UNSIGNED_64BIT_INT:  ssize = 8; break;
769
770     case GPMF_TYPE_GUID:                 ssize = 1; break; // Do not byte swap
771     case GPMF_TYPE_UTC_DATE_TIME:       ssize = 1; break; // Do not byte swap
772
773     //All unknown, complex or larger than 8-bytes store as is:
774     default:                             ssize = -1; // unsupported for structsize type
775 }
776
777 return ssize;
778 }
779
780
781 void ByteSwap2Buffer(uint32_t* input, uint32_t* output, GPMF_SampleType data_type, uint32_t structSize, uint32_t repeat)
782 {
783     int32_t i, len = 0, endianSize = GPMFTypeEndianSize(data_type);
784     if (endianSize == 8) // 64-bit swap required
785     {
786         for (i = 0; i < (int32_t)((repeat * structSize + 3) / sizeof(int32_t)); i += 2)
787         {
788             output[len++] = BYTESWAP32(input[i + 1]);
789             output[len++] = BYTESWAP32(input[i]);
790         }
791     }
792     else if (endianSize >= 1)
793     {
794         for (i = 0; i < (int32_t)((repeat * structSize + 3) / sizeof(int32_t)); i++)
795         {
796             switch (endianSize)
797             {
798                 case 2:      output[len++] = BYTESWAP2x16(input[i]); break;
799                 case 4:      output[len++] = BYTESWAP32(input[i]); break;
800                 default:     output[len++] = input[i]; break;
801             }
802         }
803     }
804 }
805
806
807
808 //find and inplace overwrite a GPMF KLV with new KLV, if the lengths match.
809 GPMF_ERR GPMF_Modify(GPMF_stream* ms, uint32_t origfourCC, uint32_t newfourCC,
810                     GPMF_SampleType newType, uint32_t newStructSize, uint32_t newRepeat, void* newData)
811 {
812     uint32_t dataSizeLongs = (newStructSize * newRepeat + 3) >> 2;
813
814     if (ms && ms->pos + 1 + dataSizeLongs < ms->buffer_size_long)
815     {
816         GPMF_stream fs;
817         GPMF_CopyState(ms, &fs);
818
819         uint32_t key = fs.buffer[fs.pos];
820         uint32_t tsr = fs.buffer[fs.pos + 1];
821         uint32_t ssize = GPMF_SAMPLE_SIZE(tsr);
822         uint32_t repeat = GPMF_SAMPLES(tsr);
823
824         if (key == origfourCC && (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2))) // no find required and data will fit
825         {
826             fs.buffer[fs.pos] = newfourCC;
827             fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);
828
829             ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
830             return GPMF_OK;
831         }
832         else
833         {
834             // search forward from the current position at this level
835             if (GPMF_OK == GPMF_FindNext(&fs, origfourCC, GPMF_CURRENT_LEVEL))
836             {
837                 tsr = fs.buffer[fs.pos + 1];
838                 ssize = GPMF_SAMPLE_SIZE(tsr);
839                 repeat = GPMF_SAMPLES(tsr);
840
841                 if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2)) //will the new data fit
842                 {
843                     fs.buffer[fs.pos] = newfourCC;
844                     fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);
845
846                     ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
847                     return GPMF_OK;
848                 }
849                 return GPMF_ERROR_BAD_STRUCTURE; // sizes don't match
850             }
851             // search backward from the current position at this level
852             else if (GPMF_OK == GPMF_FindPrev(&fs, origfourCC, GPMF_CURRENT_LEVEL))
853             {
854                 tsr = fs.buffer[fs.pos + 1];
855                 ssize = GPMF_SAMPLE_SIZE(tsr);
856                 repeat = GPMF_SAMPLES(tsr);
857
858                 if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2)) //will the new data fit
859                 {
860                     fs.buffer[fs.pos] = newfourCC;
861                     fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);
862

```

```

863         ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
864         return GPMF_OK;
865     }
866     return GPMF_ERROR_BAD_STRUCTURE; // sizes don't match
867 }
868 else
869 {
870     // search from the beginning through all levels
871     GPMF_ResetState(&fs);
872     if (GPMF_OK == GPMF_FindNext(&fs, origfourCC, GPMF_RECURSE_LEVELS))
873     {
874         tsr = fs.buffer[fs.pos + 1];
875         ssize = GPMF_SAMPLE_SIZE(tsr);
876         repeat = GPMF_SAMPLES(tsr);
877
878         if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2)) //will the new data fit
879         {
880             fs.buffer[fs.pos] = newfourCC;
881             fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);
882
883             ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
884             return GPMF_OK;
885         }
886         return GPMF_ERROR_BAD_STRUCTURE; // sizes don't match
887     }
888     else
889         return GPMF_ERROR_FIND; // if can't find the data to replace.
890 }
891 }
892 }
893 return GPMF_ERROR_BAD_STRUCTURE; // sizes don't match
894 }
895
896
897
898 uint32_t GPMF_SizeOfType(GPMF_SampleType type)
899 {
900     uint32_t ssize = 0;
901
902     switch (type)
903     {
904     case GPMF_TYPE_STRING_ASCII:        ssize = 1; break;
905     case GPMF_TYPE_SIGNED_BYTE:         ssize = 1; break;
906     case GPMF_TYPE_UNSIGNED_BYTE:       ssize = 1; break;
907
908     // These datatypes are always be stored in Big-Endian
909     case GPMF_TYPE_SIGNED_SHORT:        ssize = 2; break;
910     case GPMF_TYPE_UNSIGNED_SHORT:      ssize = 2; break;
911     case GPMF_TYPE_FLOAT:               ssize = 4; break;
912     case GPMF_TYPE_FOURCC:              ssize = 4; break;
913     case GPMF_TYPE_SIGNED_LONG:         ssize = 4; break;
914     case GPMF_TYPE_UNSIGNED_LONG:       ssize = 4; break;
915     case GPMF_TYPE_Q15_16_FIXED_POINT:  ssize = 4; break;
916     case GPMF_TYPE_Q31_32_FIXED_POINT:  ssize = 8; break;
917     case GPMF_TYPE_DOUBLE:              ssize = 8; break;
918     case GPMF_TYPE_SIGNED_64BIT_INT:    ssize = 8; break;
919     case GPMF_TYPE_UNSIGNED_64BIT_INT:  ssize = 8; break;
920
921     //All unknown or larger than 8-bytes stored as is:
922     case GPMF_TYPE_GUID:                ssize = 16; break;
923     case GPMF_TYPE_UTC_DATE_TIME:       ssize = 16; break;
924     default: ssize = 0; break;
925     }
926
927     return ssize;
928 }
929
930 uint32_t GPMF_ExpandComplexType(char *src, uint32_t srcsize, char *dst, uint32_t *dstsize)
931 {
932     uint32_t i = 0, k = 0, count = 0;
933
934     while (i < srcsize && k < *dstsize)
935     {
936         if (src[i] == '[' && i > 0)
937         {
938             uint32_t j = 1;
939             count = 0;
940             while (src[i + j] >= '0' && src[i + j] <= '9')
941             {
942                 count *= 10;
943                 count += (uint32_t) (src[i + j] - '0');
944                 j++;
945             }
946
947             if (count > 1)
948             {
949                 uint32_t l;
950                 for (l = 1; l < count; l++)
951                 {
952                     dst[k] = src[i - 1];
953                     k++;
954                 }
955             }
956             i += j;
957             if (src[i] == ']') i++;
958         }
959         else
960         {

```

```

961         dst[k] = src[i];
962         if (dst[k] == 0) break;
963         i++;
964         k++;
965     }
966 }
967
968 if (k >= *dstsize)
969     return GPMF_ERROR_MEMORY; // bad structure formed
970
971 dst[k] = 0;
972 *dstsize = k;
973
974 return GPMF_OK;
975 }
976
977
978
979 uint32_t GPMF_SizeOfComplexTYPE(char *type, uint32_t typestringlength)
980 {
981     char *typearray = type;
982     uint32_t size = 0, expand = 0;
983     uint32_t i, len = typestringlength;
984
985     for (i = 0; i < len; i++)
986         if (typearray[i] == '[')
987             expand = 1;
988
989     if (expand)
990     {
991         char exptypearray[64];
992         uint32_t dstsize = sizeof(exptypearray);
993
994         if (GPMF_OK == GPMF_ExpandComplexTYPE(typearray, len, exptypearray, &dstsize))
995         {
996             typearray = exptypearray;
997             len = dstsize;
998         }
999         else
1000             return 0;
1001     }
1002
1003     for (i = 0; i < len; i++)
1004     {
1005         uint32_t typesize = GPMF_SizeOfType((GPMF_SampleType)typearray[i]);
1006
1007         if (typesize < 1) return 0;
1008         size += typesize;
1009     }
1010
1011     return size;
1012 }
1013
1014 }
1015
1016
1017 GPMF_ERR GPMF_FormattedData(GPMF_stream *ms, void *buffer, uint32_t buffersize, uint32_t sample_offset, uint32_t read_samples)
1018 {
1019     if (ms && buffer)
1020     {
1021         uint8_t *data = (uint8_t *)&ms->buffer[ms->pos + 2];
1022         uint8_t *output = (uint8_t *)buffer;
1023         uint32_t sample_size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
1024         uint32_t remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
1025         uint8_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
1026         uint32_t typesize = 1;
1027         uint32_t elements = 0;
1028         uint32_t typestringlength = 1;
1029         char complextype[64] = "L";
1030
1031         if (type == GPMF_TYPE_NEST)
1032             return GPMF_ERROR_BAD_STRUCTURE;
1033
1034         if (GPMF_OK != IsValidSize(ms, remaining_sample_size>>2))
1035             return GPMF_ERROR_BAD_STRUCTURE;
1036
1037         if (type == GPMF_TYPE_COMPRESSED)
1038         {
1039             if (GPMF_OK == GPMF-Decompress(ms, (uint32_t *)output, buffersize))
1040             {
1041                 uint32_t compressed_typesize = ms->buffer[ms->pos + 2];
1042                 sample_size = GPMF_SAMPLE_SIZE(compressed_typesize);
1043                 remaining_sample_size = GPMF_DATA_PACKEDSIZE(compressed_typesize);
1044                 type = GPMF_SAMPLE_TYPE(compressed_typesize);
1045                 data = output;
1046             }
1047             else
1048                 return GPMF_ERROR_MEMORY;
1049         }
1050
1051         if (sample_size * read_samples > buffersize)
1052             return GPMF_ERROR_MEMORY;
1053
1054         remaining_sample_size -= sample_offset * sample_size; // skip samples
1055         data += sample_offset * sample_size;
1056
1057         if (remaining_sample_size < sample_size * read_samples)
1058             return GPMF_ERROR_MEMORY;

```

```

1059
1060     if (type == GPMF_TYPE_COMPLEX)
1061     {
1062         GPMF_stream find_stream;
1063         GPMF_CopyState(ms, &find_stream);
1064
1065         if (GPMF_OK == GPMF_FindPrev(&find_stream, GPMF_KEY_TYPE, GPMF_RECURSE_LEVELS))
1066         {
1067             char *data1 = (char *)GPMF_RawData(&find_stream);
1068             uint32_t size = GPMF_RawDataSize(&find_stream);
1069
1070             typestringlength = sizeof(complextype);
1071             if (GPMF_OK == GPMF_ExpandComplexTYPE(data1, size, complextype, &typestringlength))
1072             {
1073                 elements = (uint32_t)strlen(complextype);
1074
1075                 if (sample_size != GPMF_SizeOfComplexTYPE(complextype, typestringlength))
1076                     return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1077             }
1078             else
1079                 return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1080         }
1081         else
1082             return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1083     }
1084     else
1085     {
1086         typesize = GPMF_SizeOfType((GPMF_SampleType)type);
1087
1088         if (type == GPMF_TYPE_FOURCC)
1089             typesize = 1; // Do not ByteSWAP
1090
1091         if (typesize == 0)
1092             return GPMF_ERROR_MEMORY;
1093
1094         elements = sample_size / typesize;
1095     }
1096
1097     while (read_samples--)
1098     {
1099         uint32_t i,j;
1100
1101         for (i = 0; i < elements; i++)
1102         {
1103             if (type == GPMF_TYPE_COMPLEX)
1104             {
1105                 if (complextype[i] == GPMF_TYPE_FOURCC)
1106                 {
1107                     *output++ = *data++;
1108                     *output++ = *data++;
1109                     *output++ = *data++;
1110                     *output++ = *data++;
1111                     typesize = 0;
1112                 }
1113                 else
1114                     typesize = GPMF_SizeOfType((GPMF_SampleType) complextype[i]);
1115             }
1116
1117             switch (typesize)
1118             {
1119             case 2:
1120             {
1121                 uint16_t *data16 = (uint16_t *)data;
1122                 uint16_t *output16 = (uint16_t *)output;
1123                 *output16 = BYTESWAP16(*data16);
1124                 output16++;
1125                 data16++;
1126
1127                 data = (uint8_t *)data16;
1128                 output = (uint8_t *)output16;
1129             }
1130             break;
1131             case 4:
1132             {
1133                 uint32_t *data32 = (uint32_t *)data;
1134                 uint32_t *output32 = (uint32_t *)output;
1135                 *output32 = BYTESWAP32(*data32);
1136                 output32++;
1137                 data32++;
1138
1139                 data = (uint8_t *)data32;
1140                 output = (uint8_t *)output32;
1141             }
1142             break;
1143             case 8:
1144             {
1145                 uint32_t *data32 = (uint32_t *)data;
1146                 uint32_t *output32 = (uint32_t *)output;
1147                 *(output32+1) = BYTESWAP32(*data32);
1148                 *(output32) = BYTESWAP32(*(data32+1));
1149                 data32 += 2;
1150                 output32 += 2;
1151
1152                 data = (uint8_t *)data32;
1153                 output = (uint8_t *)output32;
1154             }
1155             break;
1156             default: //1, 16 or more not byteswapped

```

```

1157         for (j = 0; j < typesize; j++)
1158             *output++ = *data++;
1159         break;
1160     }
1161 }
1162 }
1163
1164     return GPMF_OK;
1165 }
1166
1167     return GPMF_ERROR_MEMORY;
1168 }
1169
1170
1171 #define MACRO_CAST_SCALE_UNSIGNED_TYPE(casttype) \
1172 {
1173     casttype *tmp = (casttype *)output;
1174     switch (scal_type)
1175     {
1176     case GPMF_TYPE_SIGNED_BYTE:         *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((int8_t *)scal_data8); break; \
1177     case GPMF_TYPE_UNSIGNED_BYTE:       *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((uint8_t *)scal_data8); break; \
1178     case GPMF_TYPE_SIGNED_SHORT:        *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((int16_t *)scal_data8); break; \
1179     case GPMF_TYPE_UNSIGNED_SHORT:      *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((uint16_t *)scal_data8); break; \
1180     case GPMF_TYPE_SIGNED_LONG:         *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((int32_t *)scal_data8); break; \
1181     case GPMF_TYPE_UNSIGNED_LONG:       *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((uint32_t *)scal_data8); break; \
1182     case GPMF_TYPE_FLOAT:               *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)((float *)scal_data8); break; \
1183     default: break;
1184     }
1185     output = (uint8_t *)tmp;
1186 }
1187
1188 #define MACRO_CAST_SCALE_SIGNED_TYPE(casttype) \
1189 {
1190     casttype *tmp = (casttype *)output;
1191     switch (scal_type)
1192     {
1193     case GPMF_TYPE_SIGNED_BYTE:         *tmp++ = (casttype)*val / (casttype)((int8_t *)scal_data8); break; \
1194     case GPMF_TYPE_UNSIGNED_BYTE:       *tmp++ = (casttype)*val / (casttype)((uint8_t *)scal_data8); break; \
1195     case GPMF_TYPE_SIGNED_SHORT:        *tmp++ = (casttype)*val / (casttype)((int16_t *)scal_data8); break; \
1196     case GPMF_TYPE_UNSIGNED_SHORT:      *tmp++ = (casttype)*val / (casttype)((uint16_t *)scal_data8); break; \
1197     case GPMF_TYPE_SIGNED_LONG:         *tmp++ = (casttype)*val / (casttype)((int32_t *)scal_data8); break; \
1198     case GPMF_TYPE_UNSIGNED_LONG:       *tmp++ = (casttype)*val / (casttype)((uint32_t *)scal_data8); break; \
1199     case GPMF_TYPE_FLOAT:               *tmp++ = (casttype)*val / (casttype)((float *)scal_data8); break; \
1200     default: break;
1201     }
1202     output = (uint8_t *)tmp;
1203 }
1204
1205 #define MACRO_CAST_SCALE \
1206     switch (outputType) { \
1207     case GPMF_TYPE_SIGNED_BYTE:     MACRO_CAST_SCALE_SIGNED_TYPE(int8_t) break; \
1208     case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_CAST_SCALE_UNSIGNED_TYPE(uint8_t) break; \
1209     case GPMF_TYPE_SIGNED_SHORT:    MACRO_CAST_SCALE_SIGNED_TYPE(int16_t) break; \
1210     case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_CAST_SCALE_UNSIGNED_TYPE(uint16_t) break; \
1211     case GPMF_TYPE_FLOAT:           MACRO_CAST_SCALE_SIGNED_TYPE(float) break; \
1212     case GPMF_TYPE_SIGNED_LONG:     MACRO_CAST_SCALE_SIGNED_TYPE(int32_t) break; \
1213     case GPMF_TYPE_UNSIGNED_LONG:   MACRO_CAST_SCALE_UNSIGNED_TYPE(uint32_t) break; \
1214     case GPMF_TYPE_DOUBLE:         MACRO_CAST_SCALE_SIGNED_TYPE(double) break; \
1215     default: break; \
1216     } \
1217
1218 #define MACRO_CAST_UNSIGNED_SCALE \
1219     switch (outputType) { \
1220     case GPMF_TYPE_SIGNED_BYTE:     MACRO_CAST_SCALE_SIGNED_TYPE(int8_t) break; \
1221     case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_CAST_SCALE_SIGNED_TYPE(uint8_t) break; \
1222     case GPMF_TYPE_SIGNED_SHORT:    MACRO_CAST_SCALE_SIGNED_TYPE(int16_t) break; \
1223     case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_CAST_SCALE_SIGNED_TYPE(uint16_t) break; \
1224     case GPMF_TYPE_FLOAT:           MACRO_CAST_SCALE_SIGNED_TYPE(float) break; \
1225     case GPMF_TYPE_SIGNED_LONG:     MACRO_CAST_SCALE_SIGNED_TYPE(int32_t) break; \
1226     case GPMF_TYPE_UNSIGNED_LONG:   MACRO_CAST_SCALE_SIGNED_TYPE(uint32_t) break; \
1227     case GPMF_TYPE_DOUBLE:         MACRO_CAST_SCALE_SIGNED_TYPE(double) break; \
1228     default: break; \
1229     } \
1230
1231 #define MACRO_BSWAP_CAST_SCALE(swap, inputcast, tempcast) \
1232 { \
1233     inputcast *val; \
1234     tempcast temp, *datatemp = (tempcast *)data; \
1235     temp = swap(*datatemp); \
1236     val = (inputcast *)&temp; \
1237     MACRO_CAST_SCALE \
1238     datatemp++; \
1239     data = (uint8_t *)datatemp; \
1240 } \
1241
1242
1243 #define MACRO_BSWAP_CAST_UNSIGNED_SCALE(swap, inputcast, tempcast) \
1244 { \
1245     inputcast *val; \
1246     tempcast temp, *datatemp = (tempcast *)data; \
1247     temp = swap(*datatemp); \
1248     val = (inputcast *)&temp; \
1249     MACRO_CAST_UNSIGNED_SCALE \
1250     datatemp++; \
1251     data = (uint8_t *)datatemp; \
1252 } \
1253
1254 #define MACRO_NOSWAP_CAST_SCALE(inputcast) \

```

```

1255 {
1256     inputcast *val;
1257     inputcast temp, *datatemp = (inputcast *)data;
1258     temp = *(inputcast *)data;
1259     val = (inputcast *)&temp;
1260     MACRO_CAST_SCALE
1261     datatemp++;
1262     data = (uint8_t *)datatemp;
1263 }
1264
1265
1266 #define MACRO_NOSWAP_CAST_UNSIGNED_SCALE(inputcast) \
1267 {
1268     inputcast *val;
1269     inputcast temp, *datatemp = (inputcast *)data;
1270     temp = *(inputcast *)data;
1271     val = (inputcast *)&temp;
1272     MACRO_CAST_UNSIGNED_SCALE
1273     datatemp++;
1274     data = (uint8_t *)datatemp;
1275 }
1276
1277 // a sensor matrix with only [1,0,0, 0,-1,0, 0,0,1], is just a form of non-calibrated sensor orientation
1278 #define MACRO_IS_MATRIX_CALIBRATION(inputcast) \
1279 {
1280     uint32_t m;
1281     inputcast *md = (inputcast *)mtrx_data;
1282     inputcast one = (inputcast)1;
1283     inputcast negone = (inputcast)-1;
1284     mtrx_calibration = 0;
1285     for (m = 0; m < elements*elements; m++, md++)
1286     {
1287         if (*md != one && *md != negone && *md != 0)
1288             mtrx_calibration = 1;
1289     }
1290 }
1291
1292
1293 #define MACRO_APPLY_CALIBRATION(matrixcast, outputcast)
1294 {
1295     uint32_t x,y;
1296     outputcast tmpbuf[8];
1297     outputcast *tmp = (outputcast *)output;
1298     tmp -= elements;
1299     matrixcast *mtrx = (matrixcast *)mtrx_data;
1300     for (y = 0; y < elements; y++) tmpbuf[y] = 0;
1301     for (y = 0; y < elements; y++) for (x = 0; x < elements; x++) tmpbuf[y] += tmp[x] * (outputcast)mtrx[y*elements + x];
1302     for (y = 0; y < elements; y++) tmp[y] = tmpbuf[y];
1303 }
1304
1305
1306 #define MACRO_APPLY_MATRIX_CALIBRATION(matrixcast) \
1307 {
1308     switch (outputType) {
1309         case GPMF_TYPE_SIGNED_BYTE: MACRO_APPLY_CALIBRATION(matrixcast, int8_t) break; \
1310         case GPMF_TYPE_UNSIGNED_BYTE: MACRO_APPLY_CALIBRATION(matrixcast, uint8_t) break; \
1311         case GPMF_TYPE_SIGNED_SHORT: MACRO_APPLY_CALIBRATION(matrixcast, int16_t) break; \
1312         case GPMF_TYPE_UNSIGNED_SHORT: MACRO_APPLY_CALIBRATION(matrixcast, uint16_t) break; \
1313         case GPMF_TYPE_SIGNED_LONG: MACRO_APPLY_CALIBRATION(matrixcast, int32_t) break; \
1314         case GPMF_TYPE_UNSIGNED_LONG: MACRO_APPLY_CALIBRATION(matrixcast, uint32_t) break; \
1315         case GPMF_TYPE_FLOAT: MACRO_APPLY_CALIBRATION(matrixcast, float) break; \
1316         case GPMF_TYPE_DOUBLE: MACRO_APPLY_CALIBRATION(matrixcast, double) break; \
1317         default: break;
1318     }
1319 }
1320
1321 #define MACRO_SET_MATRIX(matrixcast, orin, orio, pos) \
1322 {
1323     matrixcast *mtrx = (matrixcast *)mtrx_data;
1324     if (orin == orio)
1325         mtrx[pos] = (matrixcast)1;
1326     else if ((orin - 'a') == (orio - 'A'))
1327         mtrx[pos] = (matrixcast)-1;
1328     else if ((orin - 'A') == (orio - 'a'))
1329         mtrx[pos] = (matrixcast)-1;
1330     else
1331         mtrx[pos] = 0;
1332 }
1333
1334
1335
1336 GPMF_ERR GPMF_ScaledData(GPMF_stream *ms, void *buffer, uint32_t buffersize, uint32_t sample_offset, uint32_t read_samples, GPMF_SampleType outputType)
1337 {
1338     if (ms && buffer)
1339     {
1340         GPMF_ERR ret = GPMF_OK;
1341         uint8_t *data = (uint8_t *)&ms->buffer[ms->pos + 2];
1342         uint8_t *output = (uint8_t *)&buffer;
1343         uint32_t sample_size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
1344         uint32_t output_sample_size = GPMF_SizeOfType(outputType);
1345         uint32_t remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
1346         GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
1347         char complextype[64] = "L";
1348         uint32_t inputtypesize = 0;
1349         uint32_t inputtypeelements = 0;
1350
1351         uint8_t scal_type = 0;
1352         uint8_t scal_count = 0;

```

```

1353     uint32_t scal_typesize = 0;
1354     uint32_t *scal_data = NULL;
1355     uint32_t scal_buffer[64];
1356     uint32_t scal_buffersize = sizeof(scal_buffer);
1357
1358     uint8_t mtrx_type = 0;
1359     uint8_t mtrx_count = 0;
1360     uint32_t mtrx_typesize = 0;
1361     uint32_t mtrx_sample_size = 0;
1362     uint32_t *mtrx_data = NULL;
1363     uint32_t mtrx_buffer[64];
1364     uint32_t mtrx_buffersize = sizeof(mtrx_buffer);
1365     uint32_t mtrx_calibration = 0;
1366
1367     char *orin_data = NULL;
1368     uint32_t orin_len = 0;
1369     char *orio_data = NULL;
1370     uint32_t orio_len = 0;
1371
1372     uint32_t *uncompressedSamples = NULL;
1373     uint32_t elements = 1;
1374     uint32_t noswap = 0;
1375
1376     type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
1377
1378     if (type == GPMF_TYPE_NEST)
1379         return GPMF_ERROR_MEMORY;
1380
1381     if (type == GPMF_TYPE_COMPRESSED)
1382     {
1383         uint32_t neededunc = GPMF_FormattedDataSize(ms);
1384         uint32_t samples = GPMF_Repeat(ms);
1385
1386         remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 2]);
1387
1388         uncompressedSamples = (uint32_t *)malloc(neededunc + 12);
1389         if (uncompressedSamples)
1390         {
1391             if (GPMF_OK == GPMF_FormattedData(ms, uncompressedSamples, neededunc, 0, samples))
1392             {
1393                 read_samples = samples;
1394                 elements = GPMF_ElementsInStruct(ms);
1395                 type = GPMF_Type(ms);
1396                 complextype[0] = (char)type;
1397                 inputtypesize = GPMF_SizeOfType((GPMF_SampleType)type);
1398                 if (inputtypesize == 0)
1399                 {
1400                     ret = GPMF_ERROR_MEMORY;
1401                     goto cleanup;
1402                 }
1403                 inputtypeelements = 1;
1404                 noswap = 1; // data is formatted to LittleEndian
1405
1406                 data = (uint8_t *)uncompressedSamples;
1407
1408                 remaining_sample_size -= sample_offset * sample_size; // skip samples
1409                 data += sample_offset * sample_size;
1410
1411                 if (remaining_sample_size < sample_size * read_samples)
1412                     return GPMF_ERROR_MEMORY;
1413             }
1414         }
1415     }
1416 }
1417 else if (type == GPMF_TYPE_COMPLEX)
1418 {
1419     GPMF_stream find_stream;
1420     GPMF_CopyState(ms, &find_stream);
1421
1422     remaining_sample_size -= sample_offset * sample_size; // skip samples
1423     data += sample_offset * sample_size;
1424
1425     if (remaining_sample_size < sample_size * read_samples)
1426         return GPMF_ERROR_MEMORY;
1427
1428     if (GPMF_OK == GPMF_FindPrev(&find_stream, GPMF_KEY_TYPE, GPMF_RECURSE_LEVELS))
1429     {
1430         char *data1 = (char *)GPMF_RawData(&find_stream);
1431         uint32_t size = GPMF_RawDataSize(&find_stream);
1432         uint32_t typestringlength = sizeof(complextype);
1433         if (GPMF_OK == GPMF_ExpandComplexType(data1, size, complextype, &typestringlength))
1434         {
1435             inputtypeelements = elements = typestringlength;
1436
1437             if (sample_size != GPMF_SizeOfComplexType(complextype, typestringlength))
1438                 return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1439         }
1440         else
1441             return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1442     }
1443     else
1444         return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1445 }
1446 else
1447 {
1448     remaining_sample_size -= sample_offset * sample_size; // skip samples
1449 }

```

```

1451         data += sample_offset * sample_size;
1452
1453         if (remaining_sample_size < sample_size * read_samples)
1454             return GPMF_ERROR_MEMORY;
1455
1456         complextype[0] = type;
1457         inputtypesize = GPMF_SizeOfType((GPMF_SampleType) type);
1458         if (inputtypesize == 0)
1459             return GPMF_ERROR_MEMORY;
1460         inputtypeelements = 1;
1461         elements = sample_size / inputtypesize;
1462     }
1463
1464     if (output_sample_size * elements * read_samples > buffersize)
1465         return GPMF_ERROR_MEMORY;
1466
1467     switch (outputType) {
1468     case GPMF_TYPE_SIGNED_BYTE:
1469     case GPMF_TYPE_UNSIGNED_BYTE:
1470     case GPMF_TYPE_SIGNED_SHORT:
1471     case GPMF_TYPE_UNSIGNED_SHORT:
1472     case GPMF_TYPE_FLOAT:
1473     case GPMF_TYPE_SIGNED_LONG:
1474     case GPMF_TYPE_UNSIGNED_LONG:
1475     case GPMF_TYPE_DOUBLE:
1476         // All supported formats.
1477     {
1478         GPMF_stream fs;
1479         GPMF_CopyState(ms, &fs);
1480
1481         if (GPMF_OK == GPMF_FindPrev(&fs, GPMF_KEY_SCALE, GPMF_CURRENT_LEVEL))
1482         {
1483             scal_data = (uint32_t *)GPMF_RawData(&fs);
1484             scal_type = GPMF_SAMPLE_TYPE(fs.buffer[fs.pos + 1]);
1485
1486             switch (scal_type)
1487             {
1488             case GPMF_TYPE_SIGNED_BYTE:
1489             case GPMF_TYPE_UNSIGNED_BYTE:
1490             case GPMF_TYPE_SIGNED_SHORT:
1491             case GPMF_TYPE_UNSIGNED_SHORT:
1492             case GPMF_TYPE_SIGNED_LONG:
1493             case GPMF_TYPE_UNSIGNED_LONG:
1494             case GPMF_TYPE_FLOAT:
1495                 scal_count = GPMF_SAMPLES(fs.buffer[fs.pos + 1]);
1496                 scal_typesize = GPMF_SizeOfType((GPMF_SampleType)scal_type);
1497
1498                 if (scal_count > 1)
1499                 {
1500                     if (scal_count != elements)
1501                     {
1502                         ret = GPMF_ERROR_SCALE_COUNT;
1503                         goto cleanup;
1504                     }
1505                 }
1506
1507                 GPMF_FormattedData(&fs, scal_buffer, scal_buffersize, 0, scal_count);
1508
1509                 scal_data = (uint32_t *)scal_buffer;
1510                 break;
1511             default:
1512                 return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1513                 break;
1514             }
1515         }
1516     }
1517     else
1518     {
1519         scal_type = 'L';
1520         scal_count = 1;
1521         scal_buffer[0] = 1; // set the scale to 1 is no scale was provided
1522         scal_data = (uint32_t *)scal_buffer;
1523     }
1524
1525     GPMF_CopyState(ms, &fs);
1526     if (GPMF_OK == GPMF_FindPrev(&fs, GPMF_KEY_MATRIX, GPMF_CURRENT_LEVEL))
1527     {
1528         uint32_t mtrx_found_size = 0;
1529         uint32_t matrix_size = elements * elements;
1530         mtrx_data = (uint32_t *)GPMF_RawData(&fs);
1531         mtrx_type = GPMF_SAMPLE_TYPE(fs.buffer[fs.pos + 1]);
1532
1533         switch (mtrx_type)
1534         {
1535         case GPMF_TYPE_SIGNED_BYTE:
1536         case GPMF_TYPE_UNSIGNED_BYTE:
1537         case GPMF_TYPE_SIGNED_SHORT:
1538         case GPMF_TYPE_UNSIGNED_SHORT:
1539         case GPMF_TYPE_SIGNED_LONG:
1540         case GPMF_TYPE_UNSIGNED_LONG:
1541         case GPMF_TYPE_FLOAT:
1542         case GPMF_TYPE_DOUBLE:
1543             mtrx_count = GPMF_SAMPLES(fs.buffer[fs.pos + 1]);
1544             mtrx_sample_size = GPMF_SAMPLE_SIZE(fs.buffer[fs.pos + 1]);
1545             mtrx_typesize = GPMF_SizeOfType((GPMF_SampleType)mtrx_type);
1546             mtrx_found_size = mtrx_count * mtrx_sample_size / mtrx_typesize;
1547             if (mtrx_found_size != matrix_size) // e.g XYZ is a 3x3 matrix, RGBA is a 4x4 matrix
1548             {

```


[illegible]

```

1647         goto cleanup;
1648         break;
1649     }
1650 }
1651 else
1652 {
1653     switch (complexttype[i % inputtypeelements])
1654     {
1655         case GPMF_TYPE_FLOAT:    MACRO_BSWAP_CAST_SCALE(BYTESWAP32, float, uint32_t) break;
1656         case GPMF_TYPE_SIGNED_BYTE:  MACRO_BSWAP_CAST_SCALE(NOSWAP8, int8_t, uint8_t) break;
1657         case GPMF_TYPE_UNSIGNED_BYTE: MACRO_BSWAP_CAST_UNSIGNED_SCALE(NOSWAP8, uint8_t, uint8_t) break;
1658         case GPMF_TYPE_SIGNED_SHORT:  MACRO_BSWAP_CAST_SCALE(BYTESWAP16, int16_t, uint16_t) break;
1659         case GPMF_TYPE_UNSIGNED_SHORT: MACRO_BSWAP_CAST_UNSIGNED_SCALE(BYTESWAP16, uint16_t, uint16_t) break;
1660         case GPMF_TYPE_SIGNED_LONG:   MACRO_BSWAP_CAST_SCALE(BYTESWAP32, int32_t, uint32_t) break;
1661         case GPMF_TYPE_UNSIGNED_LONG:  MACRO_BSWAP_CAST_UNSIGNED_SCALE(BYTESWAP32, uint32_t, uint32_t) break;
1662         case GPMF_TYPE_SIGNED_64BIT_INT:  MACRO_BSWAP_CAST_SCALE(BYTESWAP64, int64_t, uint64_t) break;
1663         case GPMF_TYPE_UNSIGNED_64BIT_INT: MACRO_BSWAP_CAST_UNSIGNED_SCALE(BYTESWAP64, uint64_t, uint64_t) break;
1664         default:
1665             ret = GPMF_ERROR_TYPE_NOT_SUPPORTED;
1666             goto cleanup;
1667             break;
1668     }
1669 }
1670 if (scal_count > 1)
1671     scal_data8 += scal_typesize;
1672 }
1673
1674 if (inputtypeelements == 1)
1675 {
1676     if (mtrx_calibration)
1677     {
1678         switch (mtrx_type)
1679         {
1680             case GPMF_TYPE_SIGNED_BYTE:  MACRO_APPLY_MATRIX_CALIBRATION(int8_t) break;
1681             case GPMF_TYPE_UNSIGNED_BYTE: MACRO_APPLY_MATRIX_CALIBRATION(uint8_t) break;
1682             case GPMF_TYPE_SIGNED_SHORT:  MACRO_APPLY_MATRIX_CALIBRATION(int16_t) break;
1683             case GPMF_TYPE_UNSIGNED_SHORT: MACRO_APPLY_MATRIX_CALIBRATION(uint16_t) break;
1684             case GPMF_TYPE_SIGNED_LONG:   MACRO_APPLY_MATRIX_CALIBRATION(int32_t) break;
1685             case GPMF_TYPE_UNSIGNED_LONG:  MACRO_APPLY_MATRIX_CALIBRATION(uint32_t) break;
1686             case GPMF_TYPE_UNSIGNED_LONG:  MACRO_APPLY_MATRIX_CALIBRATION(uint32_t) break;
1687             case GPMF_TYPE_FLOAT:    MACRO_APPLY_MATRIX_CALIBRATION(float); break;
1688             case GPMF_TYPE_DOUBLE:  MACRO_APPLY_MATRIX_CALIBRATION(double); break;
1689             default: break;
1690         }
1691     }
1692 }
1693 }
1694 break;
1695
1696 default:
1697     ret = GPMF_ERROR_TYPE_NOT_SUPPORTED;
1698     goto cleanup;
1699     break;
1700 }
1701
1702 cleanup:
1703     if (uncompressedSamples)
1704         free(uncompressedSamples);
1705
1706     return ret;
1707 }
1708
1709 return GPMF_ERROR_MEMORY;
1710 }
1711
1712
1713
1714 GPMF_ERR GPMF-DecompressedSize(GPMF_stream *ms, uint32_t *neededsize)
1715 {
1716     if (ms && neededsize)
1717     {
1718         *neededsize = GPMF_DATA_SIZE(ms->buffer[ms->pos + 2]); // The first 32-bit of data, is the uncompressed type-size-repeat
1719         return GPMF_OK;
1720     }
1721
1722     return GPMF_ERROR_MEMORY;
1723 }
1724
1725
1726 GPMF_ERR GPMF-Decompress(GPMF_stream *ms, uint32_t *localbuf, uint32_t localbuf_size)
1727 {
1728     if (ms && localbuf && localbuf_size)
1729     {
1730         if (ms->cbhandle == 0)
1731             if (GPMF_OK != GPMF_AllocCodebook(&ms->cbhandle))
1732                 return GPMF_ERROR_MEMORY;
1733
1734         memset(localbuf, 0, localbuf_size);
1735
1736         // unpack here
1737         GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 2]); // The first 32-bit of data, is the uncompressed type-size-repeat
1738         uint8_t *start = (uint8_t *) &ms->buffer[ms->pos + 3];
1739         uint16_t quant;
1740         size_t sOffset = 0;
1741         uint16_t *compressed_data;
1742         uint32_t sample_size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2]);
1743         uint32_t sizeoftype = GPMF_SizeofType(type);
1744         uint32_t chn = 0, channels = sample_size / sizeoftype;

```

```

1745 //uint32_t compressed_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
1746 uint32_t uncompressed_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 2]);
1747 uint32_t maxsamples = uncompressed_size / sample_size;
1748 int signed_type = 1;
1749
1750 memset(localbuf, 0, localbuf_size);
1751
1752 GPMF_codebook *cb = (GPMF_codebook *)ms->cbhandle;
1753
1754 if (sizeoftype == 4) // LONGs are handled at two channels of SHORTs
1755 {
1756     sizeoftype = 2;
1757     channels *= 2;
1758
1759     if (type == '1')
1760         type = GPMF_TYPE_SIGNED_SHORT;
1761     else
1762         type = GPMF_TYPE_UNSIGNED_SHORT;
1763 }
1764
1765
1766 if (type == GPMF_TYPE_SIGNED_SHORT || type == GPMF_TYPE_SIGNED_BYTE)
1767     signed_type = -1; //signed
1768
1769
1770 uint16_t *buf_u16 = (uint16_t *)localbuf;
1771 int16_t *buf_s16 = (int16_t *)localbuf;
1772 uint8_t *buf_u8 = (uint8_t *)localbuf;
1773 int8_t *buf_s8 = (int8_t *)localbuf;
1774 int32_t last;
1775 uint32_t pos, end = 0;
1776
1777 memcpy(&buf_u8[0], start, sample_size);
1778
1779 sOffset += sample_size;
1780
1781 for (chn = 0; chn < channels; chn++)
1782 {
1783     pos = 1;
1784
1785     switch ((int)sizeoftype * signed_type)
1786     {
1787     default:
1788     case -2: last = BYTESWAP16(buf_s16[chn]); quant = *((uint16_t *)&start[sOffset]); quant = BYTESWAP16(quant); sOffset += 2; break;
1789     case -1: last = (int8_t)buf_s8[chn]; quant = *((uint8_t *)&start[sOffset]); sOffset++; break;
1790     case 1: last = buf_u8[chn]; quant = *((uint8_t *)&start[sOffset]); sOffset++; break;
1791     case 2: last = BYTESWAP16(buf_u16[chn]); quant = *((uint16_t *)&start[sOffset]); quant = BYTESWAP16(quant); sOffset += 2; break;
1792     }
1793
1794     sOffset = ((sOffset + 1) & (uint32_t)-1); //16-bit aligned compressed data
1795     compressed_data = (uint16_t *)&start[sOffset];
1796
1797     uint16_t currWord = BYTESWAP16(*compressed_data); compressed_data++;
1798     uint16_t nextWord = BYTESWAP16(*compressed_data); compressed_data++;
1799     int currBits = 16;
1800     int nextBits = 16;
1801
1802     do
1803     {
1804         switch (cb[currWord].command)
1805         {
1806         case 0: // store zeros and/or a value
1807             {
1808                 int usedbits = cb[currWord].bits_used;
1809                 uint32_t zeros = cb[currWord].offset;
1810                 int delta = (int)cb[currWord].value * quant;
1811
1812                 last += delta * cb[currWord].bytes_stored;
1813
1814                 if (pos + zeros >= maxsamples)
1815                 {
1816                     end = 1;
1817                     return GPMF_ERROR_MEMORY;
1818                 }
1819                 switch ((int)sizeoftype * signed_type)
1820                 {
1821                 default:
1822                 case -2:
1823                     while (zeros) { buf_s16[channels*pos++ + chn] = (int16_t) BYTESWAP16(last); zeros--; }
1824                     buf_s16[channels*pos + chn] = (int16_t) BYTESWAP16(last);
1825                     break;
1826                 case -1:
1827                     while (zeros) { buf_s8[channels*pos++ + chn] = (int8_t)last; zeros--; }
1828                     buf_s8[channels*pos + chn] = (int8_t)last;
1829                     break;
1830                 case 1:
1831                     while (zeros) { buf_u8[channels*pos++ + chn] = (uint8_t)last; zeros--; }
1832                     buf_u8[channels*pos + chn] = (uint8_t)last;
1833                     break;
1834                 case 2:
1835                     while (zeros) { buf_u16[channels*pos++ + chn] = BYTESWAP16(last); zeros--; }
1836                     buf_u16[channels*pos + chn] = BYTESWAP16(last);
1837                     break;
1838                 }
1839
1840                 pos += (uint32_t) cb[currWord].bytes_stored;
1841                 currWord <= usedbits;
1842                 currBits -= usedbits;

```

```

1843     }
1844     break;
1845
1846 case 1: //channel END code detected, store the remaining zero deltas
1847     {
1848         int zeros = (int)(uncompressed_size/(channels*sizeoftype) - pos);
1849         switch ((int)sizeoftype*signed_type)
1850         {
1851             default:
1852             case -2:
1853                 while (zeros) { buf_s16[channels*pos++ + chn] = (int16_t) BYTESWAP16(last); zeros--; }
1854                 break;
1855             case -1:
1856                 while (zeros) { buf_s8[channels*pos++ + chn] = (int8_t)last; zeros--; }
1857                 break;
1858             case 1:
1859                 while (zeros) { buf_u8[channels*pos++ + chn] = (uint8_t)last; zeros--; }
1860                 break;
1861             case 2:
1862                 while (zeros) { buf_u16[channels*pos++ + chn] = (uint16_t) BYTESWAP16(last); zeros--; }
1863                 break;
1864         }
1865     }
1866     end = 1;
1867     break;
1868
1869 case 2: //ESC code, next byte or short contains the delta.
1870     {
1871         int usedBits = cb[currWord].bits_used;
1872         int delta;
1873         currWord <= usedbits;
1874         currBits -= usedbits;
1875
1876         //Get more bits
1877         while (currBits < 16)
1878         {
1879             int needed = 16 - currBits;
1880             currWord |= nextWord >> currBits;
1881             if (nextBits >= needed) currBits = 16; else currBits += nextBits;
1882             nextWord <= needed;
1883             nextBits -= needed;
1884             if (nextBits <= 0)
1885             {
1886                 nextWord = BYTESWAP16(*compressed_data);
1887                 compressed_data++;
1888                 nextBits = 16;
1889             }
1890         }
1891
1892         switch ((int)sizeoftype*signed_type)
1893         {
1894             default:
1895             case -2:
1896                 delta = (int16_t)(currWord);
1897                 delta *= quant;
1898                 last += delta;
1899                 buf_s16[channels*pos++ + chn] = (int16_t) BYTESWAP16(last);
1900                 break;
1901             case -1:
1902                 delta = (int8_t)(currWord >> 8);
1903                 delta *= quant;
1904                 last += delta;
1905                 buf_s8[channels*pos++ + chn] = (int8_t)last;
1906                 break;
1907             case 1:
1908                 delta = (int8_t)(currWord >> 8);
1909                 delta *= quant;
1910                 last += delta;
1911                 buf_u8[channels*pos++ + chn] = (uint8_t)last;
1912                 break;
1913             case 2:
1914                 delta = (int16_t)(currWord);
1915                 delta *= quant;
1916                 last += delta;
1917                 buf_u16[channels*pos++ + chn] = BYTESWAP16(last);
1918                 break;
1919         }
1920         currWord <= 8 * sizeoftype;
1921         currBits -= 8 * sizeoftype;
1922     }
1923     break;
1924
1925 default: //Invalid codeword read
1926     end = 1;
1927     return GPMF_ERROR_MEMORY;
1928     break;
1929 }
1930
1931 //Get more bits
1932 while (currBits < 16)
1933 {
1934     int needed = 16 - currBits;
1935     currWord |= nextWord >> currBits;
1936     if (nextBits >= needed) currBits = 16; else currBits += nextBits;
1937     nextWord <= needed;
1938     nextBits -= needed;
1939     if (nextBits <= 0)
1940     {

```

```

1941         nextWord = BYTESWAP16(*compressed_data);
1942         compressed_data++;
1943         nextBits = 16;
1944     }
1945 }
1946 } while (!end);
1947
1948 if (nextBits == 16) compressed_data--;
1949 sOffset = (size_t)compressed_data - (size_t)start;
1950 end = 0;
1951 }
1952
1953 return GPMF_OK;
1954 }
1955
1956 return GPMF_ERROR_MEMORY;
1957 }
1958
1959
1960 GPMF_ERR GPMF_AllocCodebook(size_t *cbhandle)
1961 {
1962     *cbhandle = (size_t)malloc(65536 * sizeof(GPMF_codebook));
1963     if (*cbhandle)
1964     {
1965         int i,v,z;
1966         GPMF_codebook *cb = (GPMF_codebook *)*cbhandle;
1967
1968         for (i = 0; i <= 0xffff; i++)
1969         {
1970             uint16_t code = (uint16_t)i;
1971             uint16_t mask = 0x8000;
1972             int zeros = 0, used = 0;
1973
1974             cb->command = 0;
1975
1976             // all commands are 16-bits long
1977             if (code == enccontrolcodestable.entries[HUFF_ESC_CODE_ENTRY].bits)
1978             {
1979                 cb->command = 2;
1980                 cb->bytes_stored = 1;
1981                 cb->bits_used = 16;
1982                 cb->offset = 0;
1983                 cb++;
1984                 continue;
1985             }
1986             if (code == enccontrolcodestable.entries[HUFF_END_CODE_ENTRY].bits)
1987             {
1988                 cb->command = 1;
1989                 cb->bytes_stored = 0;
1990                 cb->bits_used = 16;
1991                 cb->offset = 0;
1992                 cb++;
1993                 continue;
1994             }
1995
1996             for (z = enczerorunstable.length-1; z >= 0; z--)
1997             {
1998                 if (16 - used >= enczerorunstable.entries[z].size)
1999                 {
2000                     if ((code >> (16 - enczerorunstable.entries[z].size)) == enczerorunstable.entries[z].bits)
2001                     {
2002                         zeros += enczerorunstable.entries[z].count;
2003                         used += enczerorunstable.entries[z].size;
2004                         mask >>= enczerorunstable.entries[z].size;
2005                         break;
2006                     }
2007                 }
2008                 else break;
2009             }
2010
2011             // count single zeros.
2012             while (!(code & mask) && mask)
2013             {
2014                 zeros++;
2015                 used++;
2016                 mask >>= 1;
2017             }
2018
2019             //move the code word up to see if is a complete code for a value following the zeros.
2020             code <<= used;
2021
2022             cb->bytes_stored = 0;
2023             for (v=enchtuple.length-1; v>0; v--)
2024             {
2025                 if (16-used >= enchtuple.entries[v].size+1) // codeword + sign bit
2026                 {
2027                     if ((code >> (16 - enchtuple.entries[v].size)) == enchtuple.entries[v].bits)
2028                     {
2029                         int sign = 1-(((code >> (16 - (enchtuple.entries[v].size + 1))) & 1)<1); // last bit is the sign.
2030                         cb->value = enchtuple.entries[v].value * (int16_t)sign;
2031                         used += enchtuple.entries[v].size+1;
2032                         cb->bytes_stored = 1;
2033                         break;
2034                     }
2035                 }
2036             }
2037
2038             if (used == 0)

```

```

2039         {
2040             used = 16;
2041             cb->command = -1; // ERROR invalid code
2042         }
2043         cb->bits_used = (uint8_t)used;
2044         cb->offset = (uint8_t)zeros;
2045         cb++;
2046     }
2047
2048     return GPMF_OK;
2049 }
2050
2051 return GPMF_ERROR_MEMORY;
2052 }
2053
2054 GPMF_ERR GPMF_FreeCodebook(size_t cbhandle)
2055 {
2056     GPMF_codebook *cb = (GPMF_codebook *)cbhandle;
2057
2058     if (cb)
2059     {
2060         free(cb);
2061
2062         return GPMF_OK;
2063     }
2064     return GPMF_ERROR_MEMORY;
2065 }
2066

```