dnewman-gpsw version bump 1.6.2 History

1 contributor

2066 lines (1758 sloc) | 55.6 KB

```c
/*! @file GPMF_parser.c
 *
 *  @brief GPMF Parser library
 *
 *  @version 1.6.2
 *
 *  (C) Copyright 2017-2020 GoPro Inc (http://gopro.com/).
 *
 *  Licensed under either:
 *  - Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0
 *  - MIT license, http://opensource.org/licenses/MIT
 *  at your option.
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>

#include "GPMF_parser.h"
#include "GPMF_bitstream.h"


#ifdef DBG
#if _WINDOWS
#define DBG_MSG printf
#else
#define DBG_MSG(...)
#endif
#else
#define DBG_MSG(...)
#endif


GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size) // size is in longs not bytes.
{
	if (ms)
	{
		uint32_t nestsize = (uint32_t)ms->nest_size[ms->nest_level];
		if (nestsize == 0 && ms->nest_level == 0)
			nestsize = ms->buffer_size_longs;

		if (size + 2 <= nestsize) return GPMF_OK;
	}
	return GPMF_ERROR_BAD_STRUCTURE;
}


GPMF_ERR GPMF_Validate(GPMF_stream *ms, GPMF_LEVELS recurse)
{
	if (ms)
	{
		uint32_t currpos = ms->pos;
		uint32_t nestsize = ms->nest_size[ms->nest_level];
		if (nestsize == 0 && ms->nest_level == 0)
			nestsize = ms->buffer_size_longs;

		while (ms->pos+1 < ms->buffer_size_longs && nestsize > 0)
		{
			uint32_t key = ms->buffer[ms->pos];

			if (ms->nest_level == 0 && key != GPMF_KEY_DEVICE && ms->device_count == 0 && ms->pos == 0)
			{
				DBG_MSG("ERROR: uninitized -- GPMF_ERROR_BAD_STRUCTURE\n");
				return GPMF_ERROR_BAD_STRUCTURE;
			}

			if (GPMF_VALID_FOURCC(key))
			{
				uint32_t type_size_repeat = ms->buffer[ms->pos + 1];
				uint32_t size = GPMF_DATA_SIZE(type_size_repeat) >> 2;
```

```
                                uint8_t type = GPMF_SAMPLE_TYPE(type_size_repeat);
                                if (size + 2 > nestsize)
                                {
                                        DBG_MSG("ERROR: nest size too small within %c%c%c%c-- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
                                        return GPMF_ERROR_BAD_STRUCTURE;
                                }

                                if (!GPMF_VALID_FOURCC(key))
                                {
                                        DBG_MSG("ERROR: invalid 4CC -- GPMF_ERROR_BAD_STRUCTURE\n");
                                        return GPMF_ERROR_BAD_STRUCTURE;
                                }

                                if (type == GPMF_TYPE_NEST && recurse == GPMF_RECURSE_LEVELS)
                                {
                                        uint32_t validnest;
                                        ms->pos += 2;
                                        ms->nest_level++;
                                        if (ms->nest_level > GPMF_NEST_LIMIT)
                                        {
                                                DBG_MSG("ERROR: nest level within %c%c%c%c too deep -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
                                                return GPMF_ERROR_BAD_STRUCTURE;
                                        }
                                        ms->nest_size[ms->nest_level] = size;
                                        validnest = GPMF_Validate(ms, recurse);
                                        ms->nest_level--;
                                        if (GPMF_OK != validnest)
                                        {
                                                DBG_MSG("ERROR: invalid nest within %c%c%c%c -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
                                                return GPMF_ERROR_BAD_STRUCTURE;
                                        }
                                        else
                                        {
                                                if (ms->nest_level == 0)
                                                        ms->device_count++;
                                        }

                                        ms->pos += size;
                                        nestsize -= 2 + size;

                                        while (ms->pos < ms->buffer_size_longs && nestsize > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
                                        {
                                                ms->pos++;
                                                nestsize--;
                                        }
                                }
                                else
                                {
                                        ms->pos += 2 + size;
                                        nestsize -= 2 + size;
                                }

                                if (ms->pos == ms->buffer_size_longs)
                                {
                                        ms->pos = currpos;
                                        return GPMF_OK;
                                }
                        }
                        else
                        {
                                if (key == GPMF_KEY_END)
                                {
                                        do
                                        {
                                                ms->pos++;
                                                nestsize--;
                                        } while (ms->pos < ms->buffer_size_longs && nestsize > 0 && ms->buffer[ms->pos] == 0);
                                }
                                else if (ms->nest_level == 0 && ms->device_count > 0)
                                {
                                        ms->pos = currpos;
                                        return GPMF_OK;
                                }
                                else
                                {
                                        DBG_MSG("ERROR: bad struct within %c%c%c%c -- GPMF_ERROR_BAD_STRUCTURE\n", PRINTF_4CC(key));
                                        return GPMF_ERROR_BAD_STRUCTURE;
                                }
                        }
                }

                ms->pos = currpos;
                return GPMF_OK;
        }
        else
        {
                DBG_MSG("ERROR: Invalid handle -- GPMF_ERROR_MEMORY\n");
                return GPMF_ERROR_MEMORY;
        }
}


GPMF_ERR GPMF_ResetState(GPMF_stream *ms)
{
        if (ms)
        {
                ms->pos = 0;
                ms->nest_level = 0;
```

```
                ms->device_count = 0;
                ms->nest_size[ms->nest_level] = 0;
                ms->last_level_pos[ms->nest_level] = 0;
                ms->last_seek[ms->nest_level] = 0;
                ms->device_id = 0;
                ms->device_name[0] = 0;

                return GPMF_OK;
        }

        return GPMF_ERROR_MEMORY;
}


GPMF_ERR GPMF_Init(GPMF_stream *ms, uint32_t *buffer, uint32_t datasize)
{
        if(ms && buffer && datasize > 0)
        {
                uint32_t pos = 0;
                //Validate DEVC GPMF
                while((pos+1) * 4 < datasize && buffer[pos] == GPMF_KEY_DEVICE)
                {
                        uint32_t size = GPMF_DATA_SIZE(buffer[pos+1]);
                        pos += 2 + (size >> 2);
                }
                if ((pos*4) < datasize && buffer[pos] == GPMF_KEY_END) // NULL terminated GPMF
                {
                        datasize = pos * 4;
                }
                if (pos * 4 == datasize)
                {
                        ms->buffer = buffer;
                        ms->buffer_size_longs = (datasize + 3) >> 2;
                        ms->cbhandle = 0;

                        GPMF_ResetState(ms);

                        return GPMF_OK;
                }
                else
                {
                        return GPMF_ERROR_BAD_STRUCTURE;
                }
        }

        return GPMF_ERROR_MEMORY;
}


GPMF_ERR GPMF_CopyState(GPMF_stream *msrc, GPMF_stream *mdst)
{
        if (msrc && mdst)
        {
                memcpy(mdst, msrc, sizeof(GPMF_stream));
                return GPMF_OK;
        }
        return GPMF_ERROR_MEMORY;
}


GPMF_ERR GPMF_Next(GPMF_stream *ms, GPMF_LEVELS recurse)
{
        if (ms)
        {
                if (ms->pos+1 < ms->buffer_size_longs)
                {

                        uint32_t key, type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
                        uint32_t size = (GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]) >> 2);

                        if (GPMF_OK != IsValidSize(ms, size)) return GPMF_ERROR_BAD_STRUCTURE;

                        if (GPMF_TYPE_NEST == type && GPMF_KEY_DEVICE == ms->buffer[ms->pos] && ms->nest_level == 0)
                        {
                                ms->last_level_pos[ms->nest_level] = ms->pos;
                                ms->nest_size[ms->nest_level] = size;
                                if (recurse)
                                        ms->pos += 2;
                                else
                                        ms->pos += 2 + size;
                        }
                        else
                        {
                                if (size + 2 > ms->nest_size[ms->nest_level])
                                        return GPMF_ERROR_BAD_STRUCTURE;

                                if (recurse && type == GPMF_TYPE_NEST)
                                {
                                        ms->last_level_pos[ms->nest_level] = ms->pos;
                                        ms->pos += 2;
                                        ms->nest_size[ms->nest_level] -= size + 2;

                                        ms->nest_level++;
                                        if (ms->nest_level > GPMF_NEST_LIMIT)
                                                return GPMF_ERROR_BAD_STRUCTURE;

                                        ms->nest_size[ms->nest_level] = size;
                                }
```

```
                                else
                                {
                                        if (recurse)
                                        {
                                                ms->pos += size + 2;
                                                ms->nest_size[ms->nest_level] -= size + 2;
                                        }
                                        else
                                        {
                                                if (ms->nest_size[ms->nest_level] - (size + 2) > 0)
                                                {
                                                        ms->pos += size + 2;
                                                        ms->nest_size[ms->nest_level] -= size + 2;
                                                }
                                                else
                                                {
                                                        return GPMF_ERROR_LAST;
                                                }
                                        }
                                }
                        }

                        while (ms->pos < ms->buffer_size_longs && ms->nest_size[ms->nest_level] > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
                        {
                                ms->pos++;
                                ms->nest_size[ms->nest_level]--;
                        }

                        while (ms->nest_level > 0 && ms->nest_size[ms->nest_level] == 0)
                        {
                                ms->nest_level--;
                                //if (ms->nest_level == 0)
                                //{
                                //      ms->device_count++;
                                //}
                        }

                        if (ms->pos < ms->buffer_size_longs)
                        {
                                while (ms->pos < ms->buffer_size_longs && ms->nest_size[ms->nest_level] > 0 && ms->buffer[ms->pos] == GPMF_KEY_END)
                                {
                                        ms->pos++;
                                        ms->nest_size[ms->nest_level]--;
                                }

                                key = ms->buffer[ms->pos];
                                if (!GPMF_VALID_FOURCC(key))
                                        return GPMF_ERROR_BAD_STRUCTURE;

                                if (key == GPMF_KEY_DEVICE_ID)
                                        ms->device_id = BYTESWAP32(ms->buffer[ms->pos + 2]);
                                if (key == GPMF_KEY_DEVICE_NAME)
                                {
                                        size = GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]); // in bytes
                                        if (size > sizeof(ms->device_name) - 1)
                                                size = sizeof(ms->device_name) - 1;
                                        memcpy(ms->device_name, &ms->buffer[ms->pos + 2], size);
                                        ms->device_name[size] = 0;
                                }
                        }
                        else
                        {
                                // end of buffer
                                return GPMF_ERROR_BUFFER_END;
                        }

                        return GPMF_OK;
                }
                else
                {
                        // end of buffer
                        return GPMF_ERROR_BUFFER_END;
                }
        }
        return GPMF_ERROR_MEMORY;
}



GPMF_ERR GPMF_FindNext(GPMF_stream *ms, uint32_t fourcc, GPMF_LEVELS recurse)
{
        GPMF_stream prevstate;

        if (ms)
        {
                memcpy(&prevstate, ms, sizeof(GPMF_stream));

                if (ms->pos < ms->buffer_size_longs)
                {
                        while (0 == GPMF_Next(ms, recurse))
                        {
                                if (ms->buffer[ms->pos] == fourcc)
                                {
                                        return GPMF_OK; //found match
                                }
                        }

                        // restore read position
```

```
                        memcpy(ms, &prevstate, sizeof(GPMF_stream));
                        return GPMF_ERROR_FIND;
                }
        }
        return GPMF_ERROR_FIND;
}

GPMF_ERR GPMF_Reserved(uint32_t key)
{
        if(key == GPMF_KEY_DEVICE)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_DEVICE_ID)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_DEVICE_NAME)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_STREAM)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_STREAM_NAME)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_SI_UNITS)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_UNITS)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_SCALE)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_TYPE)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_TOTAL_SAMPLES)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_TICK)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_TOCK)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_EMPTY_PAYLOADS)
                return GPMF_ERROR_RESERVED;

        if(key == GPMF_KEY_REMARK)
                return GPMF_ERROR_RESERVED;

        return GPMF_OK;
}

uint32_t GPMF_PayloadSampleCount(GPMF_stream *ms)
{
        uint32_t count = 0;
        if (ms)
        {
                uint32_t fourcc = GPMF_Key(ms);

                GPMF_stream find_stream;
                GPMF_CopyState(ms, &find_stream);

                if (GPMF_OK == GPMF_FindNext(&find_stream, fourcc, GPMF_CURRENT_LEVEL)) // Count the instances, not the repeats
                {
                        count=2;
                        while (GPMF_OK == GPMF_FindNext(&find_stream, fourcc, GPMF_CURRENT_LEVEL))
                        {
                                count++;
                        }
                }
                else
                {
                        count = GPMF_Repeat(ms);
                        if (count == 0) // this can happen with an empty FACE, yet this is still a FACE fouce
                                count = 1;
                }
        }
        return count;
}


GPMF_ERR GPMF_SeekToSamples(GPMF_stream *ms)
{
        GPMF_stream prevstate;

        if (ms)
        {

                if (ms->pos+1 < ms->buffer_size_longs)
                {
                        uint32_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);

                        memcpy(&prevstate, ms, sizeof(GPMF_stream));

                        if (type == GPMF_TYPE_NEST)
                                GPMF_Next(ms, GPMF_RECURSE_LEVELS); // open STRM and recurse in
```

```
471
472                         while (0 == GPMF_Next(ms, GPMF_CURRENT_LEVEL))
473                         {
474                                 uint32_t size = (GPMF_DATA_SIZE(ms->buffer[ms->pos + 1]) >> 2);
475                                 if (GPMF_OK != IsValidSize(ms, size))
476                                 {
477                                         memcpy(ms, &prevstate, sizeof(GPMF_stream));
478                                         return GPMF_ERROR_BAD_STRUCTURE;
479                                 }
480
481                                 type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
482
483
484                                 if (type == GPMF_TYPE_NEST)  // Nest with-in nest
485                                 {
486                                         return GPMF_OK; //found match
487                                 }
488
489                                 if (size + 2 == ms->nest_size[ms->nest_level])
490                                 {
491                                         uint32_t key = GPMF_Key(ms);
492
493                                         if (GPMF_ERROR_RESERVED == GPMF_Reserved(key))
494                                                 return GPMF_ERROR_FIND;
495
496                                         return GPMF_OK; //found match
497                                 }
498
499                                 if (ms->buffer[ms->pos] == ms->buffer[ms->pos + size + 2]) // Matching tags
500                                 {
501                                         return GPMF_OK; //found match
502                                 }
503                         }
504
505                         // restore read position
506                         memcpy(ms, &prevstate, sizeof(GPMF_stream));
507                         return GPMF_ERROR_FIND;
508                 }
509         }
510         return GPMF_ERROR_FIND;
511 }
512
513
514 GPMF_ERR GPMF_FindPrev(GPMF_stream *ms, uint32_t fourcc, GPMF_LEVELS recurse)
515 {
516         GPMF_stream prevstate;
517
518         if (ms)
519         {
520                 uint32_t curr_level = ms->nest_level;
521
522                 memcpy(&prevstate, ms, sizeof(GPMF_stream));
523
524                 if (ms->pos < ms->buffer_size_longs && curr_level > 0)
525                 {
526
527                         do
528                         {
529                                 ms->last_seek[curr_level] = ms->pos;
530                                 ms->pos = ms->last_level_pos[curr_level - 1] + 2;
531                                 ms->nest_size[curr_level] += ms->last_seek[curr_level] - ms->pos;
532                                 do
533                                 {
534                                         if (ms->last_seek[curr_level] > ms->pos && ms->buffer[ms->pos] == fourcc)
535                                         {
536
537                                                 return GPMF_OK; //found match
538                                         }
539                                 } while (ms->last_seek[curr_level] > ms->pos && 0 == GPMF_Next(ms, GPMF_CURRENT_LEVEL));
540
541                                 curr_level--;
542                         } while (recurse == GPMF_RECURSE_LEVELS && curr_level > 0);
543
544                         // restore read position
545                         memcpy(ms, &prevstate, sizeof(GPMF_stream));
546
547                         return GPMF_ERROR_FIND;
548                 }
549         }
550
551         return GPMF_ERROR_FIND;
552 }
553
554
555
556
557
558 uint32_t GPMF_Key(GPMF_stream *ms)
559 {
560         if (ms)
561         {
562                 uint32_t key = ms->buffer[ms->pos];
563                 return key;
564         }
565         return 0;
566 }
567
568
```

```c
GPMF_SampleType GPMF_Type(GPMF_stream *ms)
{
	if (ms && ms->pos+1 < ms->buffer_size_longs)
	{
		GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos+1]);
		if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
		{
			type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 2]);
		}
		return type;
	}
	return GPMF_TYPE_ERROR;
}


uint32_t GPMF_StructSize(GPMF_stream *ms)
{
	if (ms && ms->pos+1 < ms->buffer_size_longs)
	{
		uint32_t ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
		uint32_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
		if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
		{
			ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2]);
		}
		return ssize;
	}
	return 0;
}


uint32_t GPMF_ElementsInStruct(GPMF_stream *ms)
{
	if (ms && ms->pos+1 < ms->buffer_size_longs)
	{
		uint32_t ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
		GPMF_SampleType type = (GPMF_SampleType) GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);

		if (type != GPMF_TYPE_NEST && type != GPMF_TYPE_COMPLEX && type != GPMF_TYPE_COMPRESSED)
		{
			uint32_t tsize = GPMF_SizeofType(type);
			if (tsize > 0)
				return ssize / tsize;
			else
				return 0;
		}
		if (type == GPMF_TYPE_COMPLEX)
		{
			GPMF_stream find_stream;
			GPMF_CopyState(ms, &find_stream);

			if (GPMF_OK == GPMF_FindPrev(&find_stream, GPMF_KEY_TYPE, GPMF_CURRENT_LEVEL))
			{
				char tmp[64] = "";
				uint32_t tmpsize = sizeof(tmp);
				char *data = (char *)GPMF_RawData(&find_stream);
				uint32_t size = GPMF_RawDataSize(&find_stream);

				if (GPMF_OK == GPMF_ExpandComplexTYPE(data, size, tmp, &tmpsize))
					return tmpsize;
			}
		}
		if (type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
		{
			type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 2]);
			ssize = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 2]);
			uint32_t tsize = GPMF_SizeofType(type);
			if (tsize > 0)
				return ssize / tsize;
			else
				return 0;
		}
	}
	return 0;
}


uint32_t GPMF_Repeat(GPMF_stream *ms)
{
	if (ms && ms->pos+1 < ms->buffer_size_longs)
	{
		GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
		uint32_t repeat = GPMF_SAMPLES(ms->buffer[ms->pos + 1]);
		if(type == GPMF_TYPE_COMPRESSED && ms->pos+2 < ms->buffer_size_longs)
		{
			repeat = GPMF_SAMPLES(ms->buffer[ms->pos + 2]);
		}
		return repeat;
	}
	return 0;
}

uint32_t GPMF_RawDataSize(GPMF_stream *ms)
{
	if (ms && ms->pos+1 < ms->buffer_size_longs)
	{
		uint32_t size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
		if (GPMF_OK != IsValidSize(ms, size >> 2)) return 0;
```

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

```
        case GPMF_TYPE_Q31_32_FIXED_POINT:  ssize = 8; break;
        case GPMF_TYPE_DOUBLE:                          ssize = 8; break;
        case GPMF_TYPE_SIGNED_64BIT_INT:        ssize = 8; break;
        case GPMF_TYPE_UNSIGNED_64BIT_INT:  ssize = 8; break;

        case GPMF_TYPE_GUID:                            ssize = 1; break; // Do not byte swap
        case GPMF_TYPE_UTC_DATE_TIME:           ssize = 1; break; // Do not byte swap

        //All unknown,complex or larger than 8-bytes store as is:
        default:                                                        ssize = -1;  // unsupported for structsize type
        }

        return ssize;
}


void ByteSwap2Buffer(uint32_t* input, uint32_t* output, GPMF_SampleType data_type, uint32_t structSize, uint32_t repeat)
{
        int32_t i, len = 0, endianSize = GPMFTypeEndianSize(data_type);
        if (endianSize == 8) // 64-bit swap required
        {
                for (i = 0; i < (int32_t)((repeat * structSize + 3) / sizeof(int32_t)); i += 2)
                {
                        output[len++] = BYTESWAP32(input[i + 1]);
                        output[len++] = BYTESWAP32(input[i]);
                }
        }
        else if (endianSize >= 1)
        {
                for (i = 0; i < (int32_t)((repeat * structSize + 3) / sizeof(int32_t)); i++)
                {
                        switch (endianSize)
                        {
                        case 2:         output[len++] = BYTESWAP2x16(input[i]); break;
                        case 4:         output[len++] = BYTESWAP32(input[i]); break;
                        default:        output[len++] = input[i]; break;
                        }
                }
        }
}



//find and inplace overwrite a GPMF KLV with new KLV, if the lengths match.
GPMF_ERR GPMF_Modify(GPMF_stream* ms, uint32_t origfourCC, uint32_t newfourCC,
        GPMF_SampleType newType, uint32_t newStructSize, uint32_t newRepeat, void* newData)
{
        uint32_t dataSizeLongs = (newStructSize * newRepeat + 3) >> 2;

        if (ms && ms->pos + 1 + dataSizeLongs < ms->buffer_size_longs)
        {
                GPMF_stream fs;
                GPMF_CopyState(ms, &fs);

                uint32_t key = fs.buffer[fs.pos];
                uint32_t tsr = fs.buffer[fs.pos + 1];
                uint32_t ssize = GPMF_SAMPLE_SIZE(tsr);
                uint32_t repeat = GPMF_SAMPLES(tsr);

                if (key == origfourCC && (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2)))  // no find required and data will fit
                {
                        fs.buffer[fs.pos] = newfourCC;
                        fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);

                        ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
                        return GPMF_OK;
                }
                else
                {
                        // search forward from the current position at this level
                        if (GPMF_OK == GPMF_FindNext(&fs, origfourCC, GPMF_CURRENT_LEVEL))
                        {
                                tsr = fs.buffer[fs.pos + 1];
                                ssize = GPMF_SAMPLE_SIZE(tsr);
                                repeat = GPMF_SAMPLES(tsr);

                                if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2))  //will the new data fit
                                {
                                        fs.buffer[fs.pos] = newfourCC;
                                        fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);

                                        ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
                                        return GPMF_OK;
                                }
                                return GPMF_ERROR_BAD_STRUCTURE;  // sizes don't match
                        }
                        // search backward from the current position at this level
                        else if (GPMF_OK == GPMF_FindPrev(&fs, origfourCC, GPMF_CURRENT_LEVEL))
                        {
                                tsr = fs.buffer[fs.pos + 1];
                                ssize = GPMF_SAMPLE_SIZE(tsr);
                                repeat = GPMF_SAMPLES(tsr);

                                if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2))  //will the new data fit
                                {
                                        fs.buffer[fs.pos] = newfourCC;
                                        fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);

```

```
                                        ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
                                        return GPMF_OK;
                                }
                                return GPMF_ERROR_BAD_STRUCTURE;  // sizes don't match
                        }
                        else
                        {
                                // search from the beginning through all levels
                                GPMF_ResetState(&fs);
                                if (GPMF_OK == GPMF_FindNext(&fs, origfourCC, GPMF_RECURSE_LEVELS))
                                {
                                        tsr = fs.buffer[fs.pos + 1];
                                        ssize = GPMF_SAMPLE_SIZE(tsr);
                                        repeat = GPMF_SAMPLES(tsr);

                                        if (((ssize * repeat + 3) >> 2) == ((newStructSize * newRepeat + 3) >> 2))  //will the new data fit
                                        {
                                                fs.buffer[fs.pos] = newfourCC;
                                                fs.buffer[fs.pos + 1] = GPMF_MAKE_TYPE_SIZE_COUNT(newType, newStructSize, newRepeat);

                                                ByteSwap2Buffer((uint32_t*)newData, (uint32_t*)&fs.buffer[fs.pos + 2], newType, newStructSize, newRepeat);
                                                return GPMF_OK;
                                        }
                                        return GPMF_ERROR_BAD_STRUCTURE;  // sizes don't match
                                }
                                else
                                        return GPMF_ERROR_FIND; // if can't find the data to replace.
                        }
                }
        }
        return GPMF_ERROR_BAD_STRUCTURE;  // sizes don't match
}



uint32_t GPMF_SizeofType(GPMF_SampleType type)
{
        uint32_t ssize = 0;

        switch (type)
        {
        case GPMF_TYPE_STRING_ASCII:            ssize = 1; break;
        case GPMF_TYPE_SIGNED_BYTE:                     ssize = 1; break;
        case GPMF_TYPE_UNSIGNED_BYTE:           ssize = 1; break;

        // These datatypes are always be stored in Big-Endian
        case GPMF_TYPE_SIGNED_SHORT:            ssize = 2; break;
        case GPMF_TYPE_UNSIGNED_SHORT:          ssize = 2; break;
        case GPMF_TYPE_FLOAT:                           ssize = 4; break;
        case GPMF_TYPE_FOURCC:                          ssize = 4; break;
        case GPMF_TYPE_SIGNED_LONG:                     ssize = 4; break;
        case GPMF_TYPE_UNSIGNED_LONG:           ssize = 4; break;
        case GPMF_TYPE_Q15_16_FIXED_POINT:  ssize = 4; break;
        case GPMF_TYPE_Q31_32_FIXED_POINT:  ssize = 8; break;
        case GPMF_TYPE_DOUBLE:                          ssize = 8; break;
        case GPMF_TYPE_SIGNED_64BIT_INT:        ssize = 8; break;
        case GPMF_TYPE_UNSIGNED_64BIT_INT:  ssize = 8; break;

        //All unknown or larger than 8-bytes stored as is:
        case GPMF_TYPE_GUID:                            ssize = 16; break;
        case GPMF_TYPE_UTC_DATE_TIME:           ssize = 16; break;
        default: ssize = 0; break;
        }

        return ssize;
}

uint32_t GPMF_ExpandComplexTYPE(char *src, uint32_t srcsize, char *dst, uint32_t *dstsize)
{
        uint32_t i = 0, k = 0, count = 0;

        while (i<srcsize && k<*dstsize)
        {
                if (src[i] == '[' && i>0)
                {
                        uint32_t j = 1;
                        count = 0;
                        while (src[i + j] >= '0' && src[i + j] <= '9')
                        {
                                count *= 10;
                                count += (uint32_t) (src[i + j] - '0');
                                j++;
                        }

                        if (count > 1)
                        {
                                uint32_t l;
                                for (l = 1; l<count; l++)
                                {
                                        dst[k] = src[i - 1];
                                        k++;
                                }
                        }
                        i += j;
                        if (src[i] == ']') i++;
                }
                else
                {
```

```
                        dst[k] = src[i];
                        if (dst[k] == 0) break;
            i++;
            k++;
                }
        }

        if (k >= *dstsize)
                return GPMF_ERROR_MEMORY; // bad structure formed

        dst[k] = 0;
        *dstsize = k;

        return GPMF_OK;
}



uint32_t GPMF_SizeOfComplexTYPE(char *type, uint32_t typestringlength)
{
        char *typearray = type;
        uint32_t size = 0, expand = 0;
        uint32_t i, len = typestringlength;


        for (i = 0; i < len; i++)
                if (typearray[i] == '[')
                        expand = 1;

        if (expand)
        {
                char exptypearray[64];
                uint32_t dstsize = sizeof(exptypearray);

                if (GPMF_OK == GPMF_ExpandComplexTYPE(typearray, len, exptypearray, &dstsize))
                {
                        typearray = exptypearray;
                        len = dstsize;
                }
                else
                        return 0;
        }


        for (i = 0; i < len; i++)
        {
                uint32_t typesize = GPMF_SizeofType((GPMF_SampleType)typearray[i]);

                if (typesize < 1) return 0;
                size += typesize;
        }

        return size;
}


GPMF_ERR GPMF_FormattedData(GPMF_stream *ms, void *buffer, uint32_t buffersize, uint32_t sample_offset, uint32_t read_samples)
{
        if (ms && buffer)
        {
                uint8_t *data = (uint8_t *)&ms->buffer[ms->pos + 2];
                uint8_t *output = (uint8_t *)buffer;
                uint32_t sample_size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
                uint32_t remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
                uint8_t type = GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
                uint32_t typesize = 1;
                uint32_t elements = 0;
                uint32_t typestringlength = 1;
                char complextype[64] = "L";

                if (type == GPMF_TYPE_NEST)
                        return GPMF_ERROR_BAD_STRUCTURE;

                if (GPMF_OK != IsValidSize(ms, remaining_sample_size>>2))
                        return GPMF_ERROR_BAD_STRUCTURE;

                if (type == GPMF_TYPE_COMPRESSED)
                {
                        if (GPMF_OK == GPMF_Decompress(ms, (uint32_t *)output, buffersize))
                        {
                                uint32_t compressed_typesize = ms->buffer[ms->pos + 2];
                                sample_size = GPMF_SAMPLE_SIZE(compressed_typesize);
                                remaining_sample_size = GPMF_DATA_PACKEDSIZE(compressed_typesize);
                                type = GPMF_SAMPLE_TYPE(compressed_typesize);
                                data = output;
                        }
                        else
                                return GPMF_ERROR_MEMORY;
                }

                if (sample_size * read_samples > buffersize)
                        return GPMF_ERROR_MEMORY;

                remaining_sample_size -= sample_offset * sample_size; // skip samples
                data += sample_offset * sample_size;

                if (remaining_sample_size < sample_size * read_samples)
                        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
```

```
                                        for (j = 0; j < typesize; j++)
                                                *output++ = *data++;
                                        break;
                                }
                        }
                }

                return GPMF_OK;
        }

        return GPMF_ERROR_MEMORY;
}


#define MACRO_CAST_SCALE_UNSIGNED_TYPE(casttype)                \
{
        casttype *tmp = (casttype *)output;
        switch (scal_type)
        {
        case GPMF_TYPE_SIGNED_BYTE:             *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((int8_t *)scal_data8);   break;  \
        case GPMF_TYPE_UNSIGNED_BYTE:   *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((uint8_t *)scal_data8);  break;  \
        case GPMF_TYPE_SIGNED_SHORT:    *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((int16_t *)scal_data8);  break;  \
        case GPMF_TYPE_UNSIGNED_SHORT:  *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((uint16_t *)scal_data8); break;  \
        case GPMF_TYPE_SIGNED_LONG:             *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((int32_t *)scal_data8);  break;  \
        case GPMF_TYPE_UNSIGNED_LONG:   *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((uint32_t *)scal_data8); break;  \
        case GPMF_TYPE_FLOAT:                   *tmp++ = (casttype)(*val < 0 ? 0 : *val) / (casttype)*((float *)scal_data8);    break;  \
        default: break;
        }
        output = (uint8_t *)tmp;
}

#define MACRO_CAST_SCALE_SIGNED_TYPE(casttype)          \
{
        casttype *tmp = (casttype *)output;
        switch (scal_type)
        {
        case GPMF_TYPE_SIGNED_BYTE:             *tmp++ = (casttype)*val / (casttype)*((int8_t *)scal_data8);    break;  \
        case GPMF_TYPE_UNSIGNED_BYTE:   *tmp++ = (casttype)*val / (casttype)*((uint8_t *)scal_data8);   break;  \
        case GPMF_TYPE_SIGNED_SHORT:    *tmp++ = (casttype)*val / (casttype)*((int16_t *)scal_data8);   break;  \
        case GPMF_TYPE_UNSIGNED_SHORT:  *tmp++ = (casttype)*val / (casttype)*((uint16_t *)scal_data8);  break;  \
        case GPMF_TYPE_SIGNED_LONG:             *tmp++ = (casttype)*val / (casttype)*((int32_t *)scal_data8);   break;  \
        case GPMF_TYPE_UNSIGNED_LONG:   *tmp++ = (casttype)*val / (casttype)*((uint32_t *)scal_data8);  break;  \
        case GPMF_TYPE_FLOAT:                   *tmp++ = (casttype)*val / (casttype)*((float *)scal_data8);             break;  \
        default: break;
        }
        output = (uint8_t *)tmp;
}

#define MACRO_CAST_SCALE                                                                                                        \
                switch (outputType)     {                                                                                               \
                case GPMF_TYPE_SIGNED_BYTE:     MACRO_CAST_SCALE_SIGNED_TYPE(int8_t)    break;          \
                case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_CAST_SCALE_UNSIGNED_TYPE(uint8_t) break;          \
                case GPMF_TYPE_SIGNED_SHORT:    MACRO_CAST_SCALE_SIGNED_TYPE(int16_t)   break;          \
                case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_CAST_SCALE_UNSIGNED_TYPE(uint16_t)        break;  \
                case GPMF_TYPE_FLOAT:                   MACRO_CAST_SCALE_SIGNED_TYPE(float)     break;                  \
                case GPMF_TYPE_SIGNED_LONG:             MACRO_CAST_SCALE_SIGNED_TYPE(int32_t)   break;          \
                case GPMF_TYPE_UNSIGNED_LONG:   MACRO_CAST_SCALE_UNSIGNED_TYPE(uint32_t)        break;  \
                case GPMF_TYPE_DOUBLE:                  MACRO_CAST_SCALE_SIGNED_TYPE(double)    break;          \
                default: break;                                                                                                 \
                }

#define MACRO_CAST_UNSIGNED_SCALE                                                                                               \
                switch (outputType)     {                                                                                               \
                case GPMF_TYPE_SIGNED_BYTE:     MACRO_CAST_SCALE_SIGNED_TYPE(int8_t)    break;          \
                case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_CAST_SCALE_SIGNED_TYPE(uint8_t)   break;          \
                case GPMF_TYPE_SIGNED_SHORT:    MACRO_CAST_SCALE_SIGNED_TYPE(int16_t)   break;          \
                case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_CAST_SCALE_SIGNED_TYPE(uint16_t)  break;          \
                case GPMF_TYPE_FLOAT:                   MACRO_CAST_SCALE_SIGNED_TYPE(float)             break;          \
                case GPMF_TYPE_SIGNED_LONG:             MACRO_CAST_SCALE_SIGNED_TYPE(int32_t)   break;          \
                case GPMF_TYPE_UNSIGNED_LONG:   MACRO_CAST_SCALE_SIGNED_TYPE(uint32_t)  break;          \
                case GPMF_TYPE_DOUBLE:                  MACRO_CAST_SCALE_SIGNED_TYPE(double)    break;          \
                default: break;                                                                                                 \
                }

#define MACRO_BSWAP_CAST_SCALE(swap, inputcast, tempcast)       \
{                                                                                                               \
        inputcast *val;                                                                         \
        tempcast temp,  *datatemp = (tempcast *)data;           \
        temp = swap(*datatemp);                                                                 \
        val = (inputcast *)&temp;                                                               \
        MACRO_CAST_SCALE                                                                        \
        datatemp++;                                                                                     \
        data = (uint8_t *)datatemp;                                                             \
}


#define MACRO_BSWAP_CAST_UNSIGNED_SCALE(swap, inputcast, tempcast)      \
{                                                                                                               \
        inputcast *val;                                                                         \
        tempcast temp,  *datatemp = (tempcast *)data;           \
        temp = swap(*datatemp);                                                                 \
        val = (inputcast *)&temp;                                                               \
        MACRO_CAST_UNSIGNED_SCALE                                                               \
        datatemp++;                                                                                     \
        data = (uint8_t *)datatemp;                                                             \
}

#define MACRO_NOSWAP_CAST_SCALE(inputcast)      \
```

```
{                                                                                                       \
        inputcast *val;                                                                         \
        inputcast temp,  *datatemp = (inputcast *)data;         \
        temp = *(inputcast *)data;                                                      \
        val = (inputcast *)&temp;                                                       \
        MACRO_CAST_SCALE                                                                        \
        datatemp++;                                                                                     \
        data = (uint8_t *)datatemp;                                                     \
}


#define MACRO_NOSWAP_CAST_UNSIGNED_SCALE(inputcast)     \
{                                                                                                       \
        inputcast *val;                                                                         \
        inputcast temp,  *datatemp = (inputcast *)data;         \
        temp = *(inputcast *)data;                                                      \
        val = (inputcast *)&temp;                                                       \
        MACRO_CAST_UNSIGNED_SCALE                                               \
        datatemp++;                                                                                     \
        data = (uint8_t *)datatemp;                                                     \
}

// a sensor matrix with only [1,0,0, 0,-1,0, 0,0,1], is just a form of non-calibrated sensor orientation
#define MACRO_IS_MATRIX_CALIBRATION(inputcast)                          \
{                                                                                                                                       \
        uint32_t m;                                                                                                     \
        inputcast *md = (inputcast *)mtrx_data;                                 \
        inputcast one = (inputcast)1;                                                   \
        inputcast negone = (inputcast)-1;                                               \
        mtrx_calibration = 0;                                                                   \
        for (m = 0; m < elements*elements; m++, md++)           \
        {                                                                                                                               \
                if (*md != one && *md != negone && *md != 0)            \
                        mtrx_calibration = 1;                                                   \
        }                                                                                                                               \
}


#define MACRO_APPLY_CALIBRATION(matrixcast, outputcast)
{
        uint32_t x,y;
        outputcast tmpbuf[8];
        outputcast *tmp = (outputcast *)output;
        tmp -= elements;
        matrixcast *mtrx = (matrixcast *)mtrx_data;
        for (y = 0; y < elements; y++) tmpbuf[y] = 0;
        for (y = 0; y < elements; y++) for (x = 0; x < elements; x++)  tmpbuf[y] += tmp[x] * (outputcast)mtrx[y*elements + x];  \
        for (y = 0; y < elements; y++) tmp[y] = tmpbuf[y];
}


#define MACRO_APPLY_MATRIX_CALIBRATION(matrixcast)                                                                                                      \
{
        switch (outputType)     {                                                                                                                                       \
                case GPMF_TYPE_SIGNED_BYTE:     MACRO_APPLY_CALIBRATION(matrixcast, int8_t)             break;  \
                case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_APPLY_CALIBRATION(matrixcast, uint8_t)    break;  \
                case GPMF_TYPE_SIGNED_SHORT:    MACRO_APPLY_CALIBRATION(matrixcast, int16_t)    break;  \
                case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_APPLY_CALIBRATION(matrixcast, uint16_t)   break;  \
                case GPMF_TYPE_SIGNED_LONG:             MACRO_APPLY_CALIBRATION(matrixcast, int32_t)    break;  \
                case GPMF_TYPE_UNSIGNED_LONG:   MACRO_APPLY_CALIBRATION(matrixcast, uint32_t)   break;  \
                case GPMF_TYPE_FLOAT:                   MACRO_APPLY_CALIBRATION(matrixcast, float)              break;  \
                case GPMF_TYPE_DOUBLE:                  MACRO_APPLY_CALIBRATION(matrixcast, double)             break;  \
                default: break;                                                                                                                                 \
        }
}

#define MACRO_SET_MATRIX(matrixcast, orin, orio, pos)   \
{                                                                                                       \
        matrixcast *mtrx = (matrixcast *)mtrx_data;                     \
        if (orin == orio)                                                                       \
                mtrx[pos] = (matrixcast)1;                                                      \
        else if ((orin - 'a') == (orio - 'A'))                          \
                mtrx[pos] = (matrixcast)-1;                                             \
        else if ((orin - 'A') == (orio - 'a'))                          \
                mtrx[pos] = (matrixcast)-1;                                             \
        else                                                                                                    \
                mtrx[pos] = 0;                                                                  \
}



GPMF_ERR GPMF_ScaledData(GPMF_stream *ms, void *buffer, uint32_t buffersize, uint32_t sample_offset, uint32_t read_samples, GPMF_SampleType outputType)
{
        if (ms && buffer)
        {
                GPMF_ERR ret = GPMF_OK;
                uint8_t *data = (uint8_t *)&ms->buffer[ms->pos + 2];
                uint8_t *output = (uint8_t *)buffer;
                uint32_t sample_size = GPMF_SAMPLE_SIZE(ms->buffer[ms->pos + 1]);
                uint32_t output_sample_size = GPMF_SizeofType(outputType);
                uint32_t remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
                GPMF_SampleType type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);
                char complextype[64] = "L";
                uint32_t inputtypesize = 0;
                uint32_t inputtypeelements = 0;

                uint8_t scal_type = 0;
                uint8_t scal_count = 0;
```

```
		uint32_t scal_typesize = 0;
		uint32_t *scal_data = NULL;
		uint32_t scal_buffer[64];
		uint32_t scal_buffersize = sizeof(scal_buffer);

		uint8_t mtrx_type = 0;
		uint8_t mtrx_count = 0;
		uint32_t mtrx_typesize = 0;
		uint32_t mtrx_sample_size = 0;
		uint32_t *mtrx_data = NULL;
		uint32_t mtrx_buffer[64];
		uint32_t mtrx_buffersize = sizeof(mtrx_buffer);
		uint32_t mtrx_calibration = 0;

		char *orin_data = NULL;
		uint32_t orin_len = 0;
		char *orio_data = NULL;
		uint32_t orio_len = 0;

		uint32_t *uncompressedSamples = NULL;
		uint32_t elements = 1;
		uint32_t noswap = 0;

		type = (GPMF_SampleType)GPMF_SAMPLE_TYPE(ms->buffer[ms->pos + 1]);

		if (type == GPMF_TYPE_NEST)
			return GPMF_ERROR_MEMORY;

		if (type == GPMF_TYPE_COMPRESSED)
		{
			uint32_t neededunc = GPMF_FormattedDataSize(ms);
			uint32_t samples = GPMF_Repeat(ms);

			remaining_sample_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 2]);

			uncompressedSamples = (uint32_t *)malloc(neededunc + 12);
			if (uncompressedSamples)
			{
				if (GPMF_OK == GPMF_FormattedData(ms, uncompressedSamples, neededunc, 0, samples))
				{
					read_samples = samples;
					elements = GPMF_ElementsInStruct(ms);
					type = GPMF_Type(ms);
					complextype[0] = (char)type;
					inputtypesize = GPMF_SizeofType((GPMF_SampleType)type);
					if (inputtypesize == 0)
					{
						ret = GPMF_ERROR_MEMORY;
						goto cleanup;
					}
					inputtypeelements = 1;
					noswap = 1; // data is formatted to LittleEndian

					data = (uint8_t *)uncompressedSamples;

					remaining_sample_size -= sample_offset * sample_size; // skip samples
					data += sample_offset * sample_size;

					if (remaining_sample_size < sample_size * read_samples)
						return GPMF_ERROR_MEMORY;

				}
			}
		}
		else if (type == GPMF_TYPE_COMPLEX)
		{

			GPMF_stream find_stream;
			GPMF_CopyState(ms, &find_stream);

			remaining_sample_size -= sample_offset * sample_size; // skip samples
			data += sample_offset * sample_size;

			if (remaining_sample_size < sample_size * read_samples)
				return GPMF_ERROR_MEMORY;

			if (GPMF_OK == GPMF_FindPrev(&find_stream, GPMF_KEY_TYPE, GPMF_RECURSE_LEVELS))
			{
				char *data1 = (char *)GPMF_RawData(&find_stream);
				uint32_t size = GPMF_RawDataSize(&find_stream);
				uint32_t typestringlength = sizeof(complextype);
				if (GPMF_OK == GPMF_ExpandComplexTYPE(data1, size, complextype, &typestringlength))
				{
					inputtypeelements = elements = typestringlength;

					if (sample_size != GPMF_SizeOfComplexTYPE(complextype, typestringlength))
						return GPMF_ERROR_TYPE_NOT_SUPPORTED;
				}
				else
					return GPMF_ERROR_TYPE_NOT_SUPPORTED;
			}
			else
				return GPMF_ERROR_TYPE_NOT_SUPPORTED;
		}
		else
		{

			remaining_sample_size -= sample_offset * sample_size; // skip samples
```

```
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
1468            switch (outputType)     {
1469            case GPMF_TYPE_SIGNED_BYTE:
1470            case GPMF_TYPE_UNSIGNED_BYTE:
1471            case GPMF_TYPE_SIGNED_SHORT:
1472            case GPMF_TYPE_UNSIGNED_SHORT:
1473            case GPMF_TYPE_FLOAT:
1474            case GPMF_TYPE_SIGNED_LONG:
1475            case GPMF_TYPE_UNSIGNED_LONG:
1476            case GPMF_TYPE_DOUBLE:
1477                    // All supported formats.
1478            {
1479                    GPMF_stream fs;
1480                    GPMF_CopyState(ms, &fs);
1481
1482                    if (GPMF_OK == GPMF_FindPrev(&fs, GPMF_KEY_SCALE, GPMF_CURRENT_LEVEL))
1483                    {
1484                            scal_data = (uint32_t *)GPMF_RawData(&fs);
1485                            scal_type = GPMF_SAMPLE_TYPE(fs.buffer[fs.pos + 1]);
1486
1487                            switch (scal_type)
1488                            {
1489                            case GPMF_TYPE_SIGNED_BYTE:
1490                            case GPMF_TYPE_UNSIGNED_BYTE:
1491                            case GPMF_TYPE_SIGNED_SHORT:
1492                            case GPMF_TYPE_UNSIGNED_SHORT:
1493                            case GPMF_TYPE_SIGNED_LONG:
1494                            case GPMF_TYPE_UNSIGNED_LONG:
1495                            case GPMF_TYPE_FLOAT:
1496                                    scal_count = GPMF_SAMPLES(fs.buffer[fs.pos + 1]);
1497                                    scal_typesize = GPMF_SizeofType((GPMF_SampleType)scal_type);
1498
1499                                    if (scal_count > 1)
1500                                    {
1501                                            if (scal_count != elements)
1502                                            {
1503                                                    ret = GPMF_ERROR_SCALE_COUNT;
1504                                                    goto cleanup;
1505                                            }
1506                                    }
1507
1508                                    GPMF_FormattedData(&fs, scal_buffer, scal_buffersize, 0, scal_count);
1509
1510                                    scal_data = (uint32_t *)scal_buffer;
1511                                    break;
1512                            default:
1513                                    return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1514                                    break;
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                                    {
```

```
1549                                    ret = GPMF_ERROR_SCALE_COUNT;
1550                                    goto cleanup;
1551                            }
1552
1553                            GPMF_FormattedData(&fs, mtrx_buffer, mtrx_buffersize, 0, mtrx_count);
1554                            mtrx_data = (uint32_t *)mtrx_buffer;
1555                            break;
1556                    default:
1557                            return GPMF_ERROR_TYPE_NOT_SUPPORTED;
1558                            break;
1559                    }
1560
1561                    switch (mtrx_type)
1562                    {
1563                    case GPMF_TYPE_SIGNED_BYTE:  MACRO_IS_MATRIX_CALIBRATION(int8_t) break;
1564                    case GPMF_TYPE_UNSIGNED_BYTE:  MACRO_IS_MATRIX_CALIBRATION(uint8_t) break;
1565                    case GPMF_TYPE_SIGNED_SHORT:  MACRO_IS_MATRIX_CALIBRATION(int16_t) break;
1566                    case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_IS_MATRIX_CALIBRATION(uint16_t) break;
1567                    case GPMF_TYPE_SIGNED_LONG:  MACRO_IS_MATRIX_CALIBRATION(int32_t) break;
1568                    case GPMF_TYPE_UNSIGNED_LONG:  MACRO_IS_MATRIX_CALIBRATION(uint32_t) break;
1569                    case GPMF_TYPE_FLOAT: MACRO_IS_MATRIX_CALIBRATION(float); break;
1570                    case GPMF_TYPE_DOUBLE: MACRO_IS_MATRIX_CALIBRATION(double); break;
1571                    }
1572            }
1573
1574            if (!mtrx_calibration)
1575            {
1576                    GPMF_CopyState(ms, &fs);
1577                    if (GPMF_OK == GPMF_FindPrev(&fs, GPMF_KEY_ORIENTATION_IN, GPMF_CURRENT_LEVEL))
1578                    {
1579                            orin_data = (char *)GPMF_RawData(&fs);
1580                            orin_len = GPMF_DATA_PACKEDSIZE(fs.buffer[fs.pos + 1]);
1581                    }
1582                    GPMF_CopyState(ms, &fs);
1583                    if (GPMF_OK == GPMF_FindPrev(&fs, GPMF_KEY_ORIENTATION_OUT, GPMF_CURRENT_LEVEL))
1584                    {
1585                            orio_data = (char *)GPMF_RawData(&fs);
1586                            orio_len = GPMF_DATA_PACKEDSIZE(fs.buffer[fs.pos + 1]);
1587                    }
1588                    if (orio_len == orin_len && orin_len > 1 && orio_len == elements)
1589                    {
1590                            uint32_t x, y, pos = 0;
1591                            mtrx_type = outputType;
1592
1593                            for (y = 0; y < elements; y++)
1594                            {
1595                                    for (x = 0; x < elements; x++)
1596                                    {
1597                                            switch (mtrx_type)
1598                                            {
1599                                            case GPMF_TYPE_FLOAT:                   MACRO_SET_MATRIX(float,       orio_data[y], orin_data[x], pos);  break;
1600                                            case GPMF_TYPE_DOUBLE:                  MACRO_SET_MATRIX(double,      orio_data[y], orin_data[x], pos);  break;
1601                                            case GPMF_TYPE_SIGNED_BYTE:             MACRO_SET_MATRIX(int8_t,      orio_data[y], orin_data[x], pos);   break;
1602                                            case GPMF_TYPE_UNSIGNED_BYTE:   MACRO_SET_MATRIX(uint8_t,     orio_data[y], orin_data[x], pos);  break;
1603                                            case GPMF_TYPE_SIGNED_SHORT:    MACRO_SET_MATRIX(int16_t,     orio_data[y], orin_data[x], pos);  break;
1604                                            case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_SET_MATRIX(uint16_t,    orio_data[y], orin_data[x], pos);  break;
1605                                            case GPMF_TYPE_SIGNED_LONG:             MACRO_SET_MATRIX(int32_t,     orio_data[y], orin_data[x], pos);  break;
1606                                            case GPMF_TYPE_UNSIGNED_LONG:   MACRO_SET_MATRIX(uint32_t,    orio_data[y], orin_data[x], pos);  break;
1607                                            case GPMF_TYPE_SIGNED_64BIT_INT:  MACRO_SET_MATRIX(int64_t,   orio_data[y], orin_data[x], pos);  break;
1608                                            case GPMF_TYPE_UNSIGNED_64BIT_INT: MACRO_SET_MATRIX(uint64_t, orio_data[y], orin_data[x], pos);  break;
1609                                            default:
1610                                                    ret = GPMF_ERROR_TYPE_NOT_SUPPORTED;
1611                                                    goto cleanup;
1612                                                    break;
1613                                            }
1614
1615                                            pos++;
1616                                    }
1617                            }
1618
1619                            mtrx_calibration = 1;
1620                    }
1621
1622            }
1623    }
1624
1625    while (read_samples--)
1626    {
1627            uint32_t i;
1628            uint8_t *scal_data8 = (uint8_t *)scal_data;
1629
1630            for (i = 0; i < elements; i++)
1631            {
1632                    if (noswap)
1633                    {
1634                            switch (complextype[i % inputtypeelements])
1635                            {
1636                            case GPMF_TYPE_FLOAT:  MACRO_NOSWAP_CAST_SCALE(float) break;
1637                            case GPMF_TYPE_SIGNED_BYTE:  MACRO_NOSWAP_CAST_SCALE(int8_t) break;
1638                            case GPMF_TYPE_UNSIGNED_BYTE:  MACRO_NOSWAP_CAST_UNSIGNED_SCALE(uint8_t) break;
1639                            case GPMF_TYPE_SIGNED_SHORT:  MACRO_NOSWAP_CAST_SCALE(int16_t) break;
1640                            case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_NOSWAP_CAST_UNSIGNED_SCALE(uint16_t) break;
1641                            case GPMF_TYPE_SIGNED_LONG:  MACRO_NOSWAP_CAST_SCALE(int32_t) break;
1642                            case GPMF_TYPE_UNSIGNED_LONG:  MACRO_NOSWAP_CAST_UNSIGNED_SCALE(uint32_t) break;
1643                            case GPMF_TYPE_SIGNED_64BIT_INT:  MACRO_NOSWAP_CAST_SCALE(int64_t) break;
1644                            case GPMF_TYPE_UNSIGNED_64BIT_INT:  MACRO_NOSWAP_CAST_UNSIGNED_SCALE(uint64_t) break;
1645                            default:
1646                                    ret = GPMF_ERROR_TYPE_NOT_SUPPORTED;
```

```
1647                                            goto cleanup;
1648                                            break;
1649                                    }
1650                            }
1651                            else
1652                            {
1653                                    switch (complextype[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
1675                    if (inputtypeelements == 1)
1676                    {
1677                            if (mtrx_calibration)
1678                            {
1679                                    switch (mtrx_type)
1680                                    {
1681                                    case GPMF_TYPE_SIGNED_BYTE:  MACRO_APPLY_MATRIX_CALIBRATION(int8_t) break;
1682                                    case GPMF_TYPE_UNSIGNED_BYTE:  MACRO_APPLY_MATRIX_CALIBRATION(uint8_t) break;
1683                                    case GPMF_TYPE_SIGNED_SHORT:  MACRO_APPLY_MATRIX_CALIBRATION(int16_t) break;
1684                                    case GPMF_TYPE_UNSIGNED_SHORT:  MACRO_APPLY_MATRIX_CALIBRATION(uint16_t) break;
1685                                    case GPMF_TYPE_SIGNED_LONG:  MACRO_APPLY_MATRIX_CALIBRATION(int32_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 uncomresseded 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 uncomresseded 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;
```

```
		//uint32_t compressed_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 1]);
		uint32_t uncompressed_size = GPMF_DATA_PACKEDSIZE(ms->buffer[ms->pos + 2]);
		uint32_t maxsamples = uncompressed_size / sample_size;
		int signed_type = 1;

		memset(localbuf, 0, localbuf_size);

		GPMF_codebook *cb = (GPMF_codebook *)ms->cbhandle;

		if (sizeoftype == 4) // LONGs are handled at two channels of SHORTs
		{
			sizeoftype = 2;
			channels *= 2;

			if (type == 'l')
				type = GPMF_TYPE_SIGNED_SHORT;
			else
				type = GPMF_TYPE_UNSIGNED_SHORT;
		}


		if (type == GPMF_TYPE_SIGNED_SHORT || type == GPMF_TYPE_SIGNED_BYTE)
			signed_type = -1; //signed


		uint16_t *buf_u16 = (uint16_t *)localbuf;
		int16_t *buf_s16 = (int16_t *)localbuf;
		uint8_t *buf_u8 = (uint8_t *)localbuf;
		int8_t *buf_s8 = (int8_t *)localbuf;
		int32_t last;
		uint32_t pos, end = 0;

		memcpy(&buf_u8[0], start, sample_size);

		sOffset += sample_size;

		for (chn = 0; chn<channels; chn++)
		{
			pos = 1;

			switch ((int)sizeoftype * signed_type)
			{
			default:
			case -2: last = BYTESWAP16(buf_s16[chn]); quant = *((uint16_t *)&start[sOffset]); quant = BYTESWAP16(quant); sOffset += 2; break;
			case -1: last = (int8_t)buf_s8[chn]; quant = *((uint8_t *)&start[sOffset]); sOffset++; break;
			case 1: last = buf_u8[chn]; quant = *((uint8_t *)&start[sOffset]); sOffset++; break;
			case 2: last = BYTESWAP16(buf_u16[chn]); quant = *((uint16_t *)&start[sOffset]); quant = BYTESWAP16(quant); sOffset += 2;  break;
			}

			sOffset = ((sOffset + 1) & (uint32_t)~1); //16-bit aligned compressed data
			compressed_data = (uint16_t *)&start[sOffset];

			uint16_t currWord = BYTESWAP16(*compressed_data); compressed_data++;
			uint16_t nextWord = BYTESWAP16(*compressed_data); compressed_data++;
			int currBits = 16;
			int nextBits = 16;

			do
			{
				switch (cb[currWord].command)
				{
				case 0:  // store zeros and/or a value
					{
						int usedbits = cb[currWord].bits_used;
						uint32_t zeros = cb[currWord].offset;
						int delta = (int)cb[currWord].value * quant;

						last += delta * cb[currWord].bytes_stored;

						if (pos + zeros >= maxsamples)
						{
							end = 1;
							return GPMF_ERROR_MEMORY;
						}
						switch ((int)sizeoftype * signed_type)
						{
						default:
						case -2:
							while (zeros) { buf_s16[channels*pos++ + chn] = (int16_t) BYTESWAP16(last); zeros--; }
							buf_s16[channels*pos + chn] = (int16_t) BYTESWAP16(last);
							break;
						case -1:
							while (zeros) { buf_s8[channels*pos++ + chn] = (int8_t)last; zeros--; }
							buf_s8[channels*pos + chn] = (int8_t)last;
							break;
						case 1:
							while (zeros) { buf_u8[channels*pos++ + chn] = (uint8_t)last; zeros--; }
							buf_u8[channels*pos + chn] = (uint8_t)last;
							break;
						case 2:
							while (zeros) { buf_u16[channels*pos++ + chn] = BYTESWAP16(last); zeros--; }
							buf_u16[channels*pos + chn] = BYTESWAP16(last);
							break;
						}

						pos += (uint32_t) cb[currWord].bytes_stored;
						currWord <<= usedbits;
						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                                    {
```

```
                                        nextWord = BYTESWAP16(*compressed_data);
                                        compressed_data++;
                                        nextBits = 16;
                                }
                        }
                } while (!end);

                if (nextBits == 16) compressed_data--;
                sOffset = (size_t)compressed_data - (size_t)start;
                end = 0;
        }

        return GPMF_OK;
    }

    return GPMF_ERROR_MEMORY;
}


GPMF_ERR GPMF_AllocCodebook(size_t *cbhandle)
{
    *cbhandle = (size_t)malloc(65536 * sizeof(GPMF_codebook));
    if (*cbhandle)
    {
        int i,v,z;
        GPMF_codebook *cb = (GPMF_codebook *)*cbhandle;

        for (i = 0; i <= 0xffff; i++)
        {
            uint16_t code = (uint16_t)i;
            uint16_t mask = 0x8000;
            int zeros = 0, used = 0;

            cb->command = 0;

            // all commands are 16-bits long
            if (code == enccontrolcodestable.entries[HUFF_ESC_CODE_ENTRY].bits)
            {
                cb->command = 2;
                cb->bytes_stored = 1;
                cb->bits_used = 16;
                cb->offset = 0;
                cb++;
                continue;
            }
            if (code == enccontrolcodestable.entries[HUFF_END_CODE_ENTRY].bits)
            {
                cb->command = 1;
                cb->bytes_stored = 0;
                cb->bits_used = 16;
                cb->offset = 0;
                cb++;
                continue;
            }

            for (z = enczerorunstable.length-1; z >= 0; z--)
            {
                if (16 - used >= enczerorunstable.entries[z].size)
                {
                    if ((code >> (16 - enczerorunstable.entries[z].size)) == enczerorunstable.entries[z].bits)
                    {
                        zeros += enczerorunstable.entries[z].count;
                        used  += enczerorunstable.entries[z].size;
                        mask >>= enczerorunstable.entries[z].size;
                        break;
                    }
                }
                else break;
            }

            // count single zeros.
            while (!(code & mask) && mask)
            {
                zeros++;
                used++;
                mask >>= 1;
            }

            //move the code word up to see if is a complete code for a value following the zeros.
            code <<= used;

            cb->bytes_stored = 0;
            for (v=enchuftable.length-1; v>0; v--)
            {
                if (16-used >= enchuftable.entries[v].size+1) // codeword + sign bit
                {
                    if ((code >> (16 - enchuftable.entries[v].size)) == enchuftable.entries[v].bits)
                    {
                        int sign = 1-(((code >> (16 - (enchuftable.entries[v].size + 1))) & 1)<<1); // last bit is the sign.
                        cb->value = enchuftable.entries[v].value * (int16_t)sign;
                        used += enchuftable.entries[v].size+1;
                        cb->bytes_stored = 1;
                        break;
                    }
                }
            }

            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
```