I. APPENDIX - WAVE FILES

The WAV files recorded with one of the OROS recorder modes can be played with a Sound Card if its sampling frequency is equal to 51.2 kHz.
Nevertheless, these files cannot be processed because some information contained in chunks can only be processed with the OROS Analyzer.

I.1 Description of the Wave File Format - Version 2.4

The OROS-OR763 / OR773 supports Microsoft 'WAVE' format files containing signal samples in 16-bit PCM. The WAVE format chunks are as follows:

```
<WAVE-form>  ->     RIFF( 'WAVE'        // form type
                    <fmt-ck>            // Wave format
                    <oros-ck>           // OROS setup data chunk
                    <data-ck> )         // waveform data
```

- <fmt-ck> -> fmt(<wave-format>) //general format info

```
<wave-format>                       ->  struct {
                    WORD  wFormatTag;           // = WAVE_FORMAT_PCM
                    WORD  nChannels;            // number of channels
                    DWORD nSamplesPerSec;       // sampling rate
                    DWORD nAvgBytesPerSec       // for buffering
                    WORD  nBlockAlign;          // block alignment of channels
                    WORD  nBitsPerSample;       // sample size = 16
                    WORD  cbSize;               // extra byte size = 0
                    }
```

- <oros-ck>

```
-> oros(
   <trig-ck>        // trigger information
   <drpm-ck>        // delta rpm info
   <freq-ck>        // frequency info
   <misc-ck>        // further misc. info
   <trac-ck>        // Order tracking info
   <info-ck>        // channel unit info
   <vers-ck>        // Version information
   <more-ck>        // More info (user info)
   <ch00-ck>        // channel 1 info
   <ch01-ck>        // channel 2 info
   ...
   <chxx-ck>        // channel xx info
   <tr00-ck>        // transducer channel 1 info
   <tr01-ck>        // transducer channel 2 info
   ...
   <trxx-ck> )      // transducer channel xx info
```

- <trig-ck> -> trig(<trig-data>) *

```
<trig-data>  -> struct {
                short TriggerDeltaRPM;
                short TriggerLowRPM;
```

```
                short TriggerHighRPM;
                }
```

- <drpm-ck> -> trig(<drpm-data>) *

<drpm-data> ->
```
struct {
    long TriggerMode;    // 0: Level, 1: Free run, 2: Internal...
    long TriggerSource;  // -1 external trigger, 0 ch. 1, 1 ch. 2 ...
    long TriggerLevel;   // -99% to +99%
    long TriggerSlope;   // 0: positive slope, 1: negative slope
    long TriggerDelay;   // -4096 < delay < 1000000
    long TriggerDelayOff;
    }
```

- <freq-ck> -> freq(<freq-data>) *

<freq-data> ->
```
struct {
    short FreqIndex;      // sampling rate index (1)
        short OscIndex;       // Oscillator index (1)
    short FreqRange;          // frequency range (0: 0-20kHz, 1: 0-10kHz.)
    }
```

- <misc-ck> -> misc(<misc-data>) *

<misc_data> ->
```
struct {
    long RecordMask;     // 1 bit per recorded input channel , bit 0 = channel 1
    long RecordNumber;   // number of records in the wave file
    long RecordSize;     // size of each record in number of block (of 1024
        samples/channel)
    long RecordDelay;    // time between records (in number of samples)
    short Mode;          // Analyzer running mode
    }
```

- <trac-ck> -> trac(<trac-data>) *

<trac_data> ->
```
struct {
    long  min_rpm;       // min tracked rpm
    long  max_rpm;       // max tracked rpm
    float  conv_rpm;     // RPM conversion: Pulses/Revolution
    short  half_coef;    // Half coef: for 0.5 Pulse/Revolution (if != 0)
    short delta_order;   // tracking delta order (0: 1, 1: 1/2, 2: 1/4,  3: 1/8, 4:
                         1/16 and 5: 1/32)
    }
```

- <tach-ck> -> tach(<tach-data>) *

<tach_data> ->
```
struct {
    long  TachoChannel;
    long  TachoMode;
    long TachoHysteresis;
    long  TachoSlope;
    long  TachoLowPassFilter;
    long  TachoHighPassFilter;
    long TachoHoldOff;
    float TachoConvRPM;
    long TachoHalfCoeff;
```

```
                    }
```

- <topr-ck> -> topr(<topr-data>) *

```
<topr_data>  -> struct {
                    long TopChannel;
                    long TopMode;
                    long TopHysteresis;
                    long TopSlope;
                    long TopLowPassFilter;
                    long TopHighPassFilter;
                    long TopHoldOff;
                    }
```

- <extr-ck> -> extr(<extr-data>) *

```
<extr_data>  -> struct {
                    long ExternalTrigMode;
                    long ExternalTrigSwitch;
                    long ExternalTrigPreDiv;
                    long ExternalTrigThreshold;
                    long ExternalTrigHysteresis;
                    long ExternalTrigSlope;
                    long ExternalTrigLPFilter;
                    long ExternalTrigHoldOff;
                    }
```

- <avtr-ck> -> avtr(<avtr-data>) *

```
<avtr_data>  -> struct {
                    long TriggerAvrgMode;
                    long TriggerAvrgPrdNum;
                    long TriggerAvrgMeasNum;
                    long Trigger2AvrgMode;
                    long Trigger2AvrgPrdNum;
                    long Trigger2AvrgMeasNum;
                    float triggerConvRPM;
                    }
```

- <topf-ck> -> topf(<topf-data>) *

```
<topf_data>  -> struct {
                    long TopFilterMask;
                    long TopLowPassFreq;
                    long TopHighPassFreq;
                    long TopLowBandPassFreq;
                    long TopHighBandPassFreq;
                    long TopLowStopBandFreq;
                    long TopHighStopBandFreq;
                    }
```

- <info-ck> -> info(<info-data>)

```
<info_data>     -> struct {
                     Type_AE2_Header AE2_Header[channel_number]; // array of recorded channels
                                                                 number Type_AE2_Header
                                                                 structures
                   }

typedef tagAE2_HEADER {
                     float Sens;              // Transducer sensitivity
                     char Dummy1;             // dummy char
                     float Const;             // constant conversion from logical to physical unit
                     float Ref0dB;            // 0 dB reference in physical unit (1.0 by default)
                     char Label[18];          // "OR25-V2.0"
                     char UnitLabel[32];      // Physical Unit label ("V" by default)
                     char Dummy2;             // dummy char
                     short OR9000;            // != 0 if OR9000 is used
                     short ExtAtt;            // != 0 if an external attenuator (by 1/10) is used
                     short MaxOrder;          // max order defined in Order Tracking recorder mode
                                              (0: 400, 1: 200, 2: 100, 3: 50, 4: 25, 5:12.5 and 6: 6.25)
                     char Name[20];           // channel name, example: "Ch1"
                     short Ref;               // reference channel number for the current channel
                     } Type_AE2_Header;
```

- <vers-ck> -> vers(<vers-data>)

```
<vers-data>     -> struct {
                     long VersionNumber; // 0x200 (for the version 2.0)
                     }
```

(chunk available for specific user information)

- <more-ck> -> more(<more-data>) *

```
<more-data>     -> struct {
                     User defined structure
                     }
```

- <chxx-ck> -> chxx(<chxx-data>)

```
<chxx-data>     -> struct {
                     CH_DATA ch_data;    // channel calibration data,
                     }

typedef tagCH_DATA    {
                     short Number;             // channel number
                     short Gain;               // channel gain in dB
                     float Calib_Module[101]:  // calibration data for module, coded in mB from 0 to 20 kHz
                     float Calib_Phase[101]:   // calibration data for phase, coded in 1/100° from 0 to 20kHz
                     } CH_DATA;
```

- <trxx-ck> -> trxx(<trxx-data>) *

```
<trxx-data>     -> struct {
                     Type_Transducer_Info trans_data;    // transducer data,
                     }

/* Define the transducer connected to a physical input, used in the wave form file */
typedef struct TAG_TRANSDUCER_INFO {
                     char  str_ChannelName[32];      // same as Name in Type_AE2_Header structures
                     char  str_SensorFile name[32];  // transducer database file name
                     char  str_SensorFilePath[128];  // transducer database file path
```

```
char  str_CalibFile name[32];      // transducer calibration file name
char  str_CalibFilePath[128];      // transducer calibration file path
char  str_SensorType[128];         // transducer type description
char  str_SensorID[128];           // transducer identifier
char  str_SensorReference[128];    // transducer reference name
char  str_UnitLabel[32];           // same as UnitLabel in Type_AE2_Header structures
char  str_UnitLabelInt1[32];       // single integration unit label string
char  str_UnitLabelInt2[32];       // double integration unit label string
char  str_UnitLabelDiff1[32];      // single differentiation unit label string
char  str_UnitLabelDiff2[32];      // double differentiation unit label string
short int_SensorIndex;             // transducer index in the data base
short int_MeasureIndex;            // measure index in the data base
short int_ApplyProcessTo;          // apply process on signal or on spectrum
short int_DataType;                // transducer data type (see UFF data type)
short int_DataTypeInt1;            // single integration data type
short int_DataTypeInt2;            // double integration data type
short int_DataTypeDiff1;           // single differentiation data type
short int_DataTypeDiff2;           // double differentiation data type
short int_ExtAttenuator;           // external attenuator
short int_Switches;                // input coupling
short int_PhysicalNumber;          // physical input number
short int_ReferenceNumber;         // reference preprocess number
float flt_Sensitivity;             // transducer sensitivity value
float flt_ConstConv;               // constant conversion value
float flt_Ref0dB;                  // 0 dB reference value
float flt_DisplayOffset;           // display offset value
float flt_RPM_rate;                // RPM ratio / channel 1
} Type_Transducer_Info;
```

* These chunks can be ignored when reading an OROS WAVE signal file.

Notes:

- A general WAVE format file supporting application, such as Sound Editor, or WAVECONV can skip the 'oros' chunk and use the general information to read the recorded file. While, the OROS-OR763 / OR773 can read any ordinary WAVE format files in 16-bit PCM coding scheme.

- To convert logical samples values into physical values, use the following expression:

Yphys = Ylog * Ystep[ch]

where
ch = channel number
Ystep[ch] = conversion step for the input channel number ch.

Ystep = YSTEP_0DB / (pow(10.0 , ch_data[ch].Gain / 20.0) x AE2_Header[ch].Const x AE2_Header[ch].Sens)

YSTEP_0DB = 3.16 * 1.414 / 32768 (31.6 * 1.414 is the maximum input voltage)