# BioHarness™ Bluetooth Logging System Interface





Ph: (443) 569-3603
Fax: (443) 926-9402
Web: www.zephyr-technology.com

# Document History

| Version | Description |
|---|---|
| 20100531 | • Initial Version. |
| 20100602 | • Updated Log Record Format Table in Appendix B. |

# References

| Ref # | ID | Description |
|---|---|---|
| **[1]** | 9700.0110 | BioHarness Bluetooth General Comms Link Specification |
| **[2]** | | IBM/MS RIFF/MCI Specification V3.0<br>(April 15, 1994) |

# Document Notes

All numbers in this document are written in decimal, except hexadecimal numbers which are prefixed by '0x'. For example 5436 is decimal, while 0x5436 is hexadecimal.

# Contents







Ph: (443) 569-3603
Fax: (443) 926-9402
Web: www.zephyr-technology.com

# 1 Data Format

The commands required to Read Data from logging memory are detailed in the BioHarness™ Bluetooth Comms Link Specification [1] document.

The data format uses the standard [2] RIFF structure. This groups the data contents into separate chunks, each containing its own header and data bytes. The chunk header specifies the type and size of the chunk data bytes. This organization method allows programs that do not use or recognize particular types of chunks to easily skip over them and continue processing following known chunks.

## *1.1 Chunk structure*

Chunks are, in general, arranged as shown in Table 1-1 General Chunk Structure.

| Part | Description |
|---|---|
| ckID | A four-character code that identifies the representation of the chunk data. A program reading a RIFF file can skip over any chunk whose chunk ID it doesn't recognize; it simply skips the number of bytes specified by ckSize plus the pad byte, if present. |
| ckSize | A 32-bit unsigned value identifying the size of ckData. This size value does not include the size of the ckID or ckSize fields or the pad byte at the end of ckData. |
| ckData | Binary data of fixed or variable size. The start of ckData is word-aligned with respect to the start of the RIFF file. If the chunk size is an odd number of bytes, a pad byte with value zero is written after ckData. Word aligning improves access speed (for chunks resident in memory) and maintains compatibility with EA IFF. The ckSize value does not include the pad byte. |

**Table 1-1 General Chunk Structure**

There are 2 special types of chunk which may contain sub-chunks, these are the RIFF chunk, which encapsulates the whole file, and a LIST chunk. (Note, JUNK, RIFF and LIST are the only ckIDs which are permitted to contain upper case letters.)
These chunks still comply with the above structure; however ckData has a specific format. The format of the ckData segment of the RIFF and LIST chunks is illustrated in Table 1-2 LIST or RIFF chunk ckData format.

| Part | Description |
|---|---|
| Form Type | A four-character code value identifying the data representation within this RIFF or LIST chunk.<br>The number of sub-chunks, mandatory and /or optional types of sub-chunk or order of sub-chunks may be defined for a particular form type. |
| Sub-Chunk | A sub-chunk, which may be a standard chunk or a LIST chunk |
| Sub-Chunk | A sub-chunk, which may be a standard chunk or a LIST chunk |
| … | … |

**Table 1-2 LIST or RIFF chunk ckData format**

An example of the type of structure that is possible using this technique is illustrated in Figure 1-1 Example Structure.

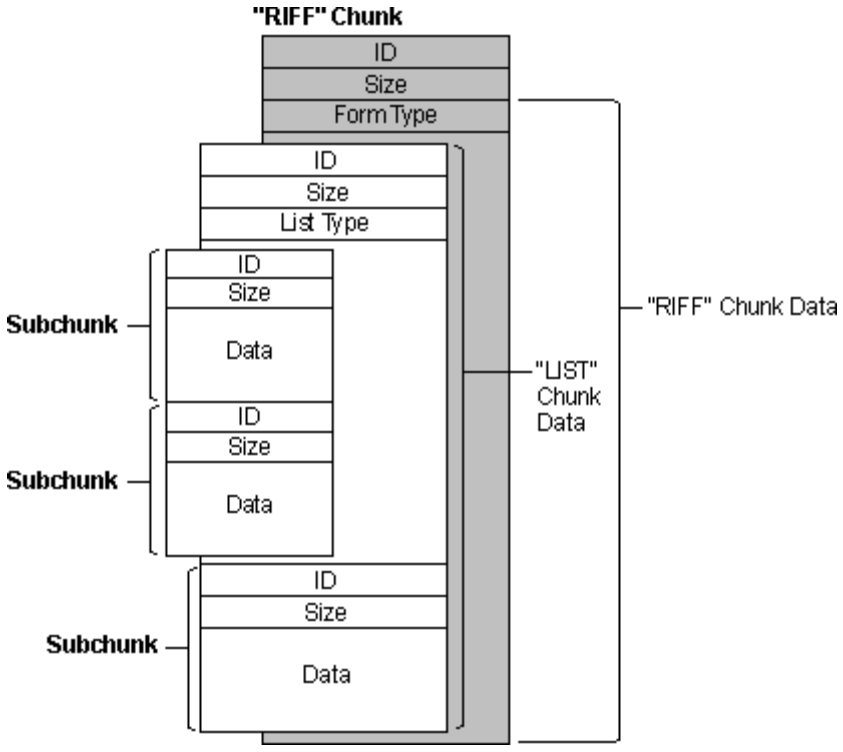



**Figure 1-1 Example Structure**

# 2 Chunk Types

## *2.1 RIFF File Identifier*

The RIFF Chunk Descriptor is the first information resident in the RIFF file, firstly specifying the file is of "RIFF" type and secondly the format of the RIFF file. This enables the PC parsing to process the file based on known sub-chunks.
In the current implementation, there is only one "RIFF" chunk.

| Field Value | Size (bytes) | Description |
|---|---|---|
| "RIFF" | 4 | Chunk ID |
| <file size> | 4 | Size of chunk (bytes) |
| "BIO " | 4 | Format |

**Table 2-1 RIFF Chunk Descriptor**

For the BioHarness the RIFF file format is always "BIO ".

## *2.2 Zephyr File Identifier*

The Zephyr File Identifier sub-chunk is the first sub-chunk chunk in the "RIFF" chunk and contains information relating to the formatting of the file data.
In the current implementation there is only one Zephyr File Identifier chunk.

| Field Value | Size (bytes) | Description |
|---|---|---|
| "zphr" | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| "bhns" | 4 | Project identifier |
| "0003" | 4 | Project version |

**Table 2-2 Zephyr File Identifier Sub-Chunk**

The project identifier and version enables the PC application to process the RIFF data within the log.
The PC application may process log data from different projects and versions and therefore this information is used to parse the data correctly.

## *2.3 File Management Sectors*

The next sub-chunk is the file management sectors chunk. This chunk is used by the device to manage the file system. The format of this chunk varies depending on the product and firmware implementation.

It is not intended that the PC application should use this chunk.





Ph: (443) 569-3603
Fax: (443) 926-9402
Web: www.zephyr-technology.com

## *2.4 Log Header*

The Log Header sub-chunk is written for each new logging session within the logging file and contains information relating to the time of the session, the logging period (how often each set of logging channels is written to the memory) and the number of channels written to the log.

| Field Value | Size (bytes) | Description |
|---|---|---|
| “logh” | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| <timestamp> | 8 | Log session start time |
| <logging period> | 4 | Period in milliseconds |
| <number of channels> | 2 | Channels logged |
| <”logr” pad size> | 2 | Padding bytes in ”logr” sub-chunk |
| <spare> | 488 | Unused data |

**Table 2-3 Log Header Sub-Chunk**

The PC application uses this information to parse the logging data chunks which follows. Note that the logging data sub-chunk uses “channels” which are essentially signed 16-bit data items.

### 2.4.1 Time Stamp Record

| Time Stamp Record | | |
|---|---|---|
| **Description** | **Length** | |
| Year (low byte) | 1 | |
| Year (high byte) | 1 | |
| Month | 1 | |
| Day of Month | 1 | |
| Milliseconds since midnight | 4 | LS byte |
| | | |
| | | |
| | | MS byte |
| Total | 8 bytes | |

**Table 2-4 Time Stamp Record packing format**

The time stamp record identifies the start time of the log to the nearest ms.

## *2.5 Split Log Header 1 & Split Log Header 2*

The Split Log Header 1 and Split Log Header 2 sub-chunks are in exactly the same format as the Log Header sub-chunk, but denotes a partitioned session.

| Field Value | Size (bytes) | Description |
|---|---|---|
| “log1” | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| <timestamp> | 8 | Log session start time |
| <logging period> | 4 | Period in milliseconds |
| <number of channels> | 2 | Channels logged |
| <logr pad size> | 2 | Padding bytes in ”logr” sub-chunk |
| <spare> | 488 | Unused data |

**Table 2-5 Split Log Header 1 Sub-Chunk**

| Field Value | Size (bytes) | Description |
|---|---|---|
| “log2” | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| <timestamp> | 8 | Log session start time |
| <logging period> | 4 | Period in milliseconds |
| <number of channels> | 2 | Channels logged |
| <logr pad size> | 2 | Padding bytes in ”logr” sub-chunk |
| <spare> | 488 | Unused data |

**Table 2-6 Split Log Header 2 Sub-Chunk**

If a session recording reaches the end of the RIFF file, the session is finalized, but written as a Split Log Header 1 sub-chunk instead of the usual Log Header sub-chunk to indicate that this is the first of two session partitions.
The second partition is written at the start of the RIFF file as a Split Log Header 2 sub-chunk, but the PC logging parser should process the Split Log Header 1 and Split Log Header 2 chunks as one session of data.

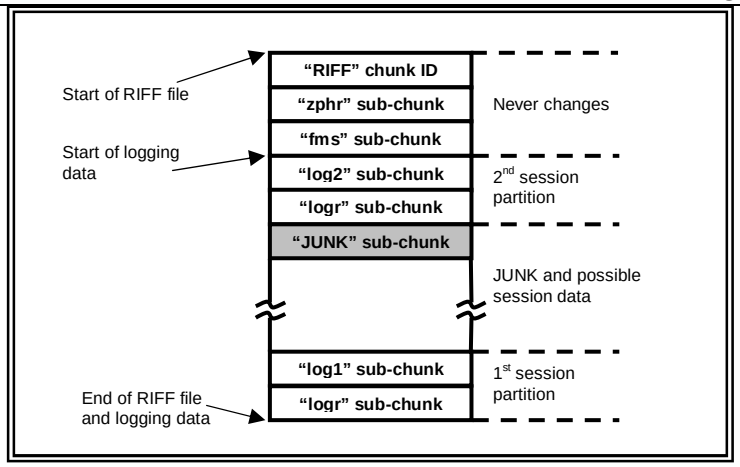



**Figure 2-1 Session partitions**

## 2.6 Log Raw Data

The Log Raw Data sub-chunk is written to memory for each new logging session within the logging file and contains the log data for a single session.

| Field Value | Size (bytes) | Description |
|---|---|---|
| "logr" | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| <spare padding> | (see log header for padding size) | Padding (spare) bytes |
| <logging data> | <chunksize – padding size> | Raw logging session data |

**Table 2-7 Log Raw Data Sub-Chunk**

The Log Raw Data sub-chunk contains data for one logging session.
The number of records, and hence the length of the recording can be calculated based on the number of channels indicated in the the associated log header chunk and the size of the Log Raw Data sub-chunk.

The arrangement of data in the channels for each log record is detailed in section Appendix B – BioHarness™ Log Record format.

## 2.7 "JUNK" Sub-chunk

The "JUNK" sub-chunk is used to specify areas of memory that the PC can ignore e.g. padding bytes. These are used for 2 main reasons:

- Padding bytes due to the minimum "page" write size associated the storage media.
  For example, if a session is completed halfway through a page, because a page is the minimum area that can be written, JUNK must be written for the remainder of the page so the PC does not process the remainder of the data.
- Because the size of the RIFF file on the memory device is typically of the order of several hundreds of Mega-Bytes, when a session has been completed, a JUNK sub-chunk can be written to the end of the file (or until the oldest recorded session data) so the PC doesn't need to process any more data (it ignores JUNK data sub-chunks).

| Field Value | Size (bytes) | Description |
|---|---|---|
| "JUNK" | 4 | Sub-chunk ID |
| <chunk size> | 4 | Size of chunk (bytes) |
| <unused data> | <chunk size> | Junk data (ignored) |

**Table 2-8 "JUNK" Sub-Chunk**

Essentially, the PC parsing of the RIFF data will find a JUNK sub-chunk and jump to the end of the chunk to process the next chunk.

## *2.8 Example RIFF File*

This section provides examples to indicate the storage of data within the RIFF file and is intended to aid understanding and design of a RIFF file parser for the logging data. In the examples the file size is set to 0x140000 (10 128KB blocks), although typically the file size would be approx 256MB in size.

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x000000 | “RIFF” | RIFF chunk descriptor |
| 0x000004 | 0x0013FFF8 | File size (minus first 8 bytes). |
| 0x000008 | “BIO ” | RIFF file format |
| 0x00000C | “zphr” | Sub-chunk ID |
| 0x000010 | 0x00000008 | Size of sub-chunk |
| 0x000014 | “bhns” | Project identifier |
| 0x000018 | “0003” | Project version |
| 0x00001C | “JUNK” | JUNK sub-chunk |
| 0x000020 | 0x000001D4 | JUNK chunk size (up to 0x000001F8) |
| 0x0001F8 | “fms ” | Sub-chunk ID |
| 0x0001FC | 0x0003FE00 | “fms” chunk size (up to 0x40000) |
| 0x040000 | “logh” | Sub-chunk ID |
| 0x040004 | 0x000001F8 | Size of sub-chunk |
| 0x040008 | 0x07D8, 0x01, 0x0F, 0x036D4608 | Timestamp (2008, Jan, 15th, 15:58:13) |
| 0x040010 | 0x000003F0 | Logging Period (1008 milliseconds). |
| 0x040014 | 0x0040 | Number of logging channels (64). |
| 0x040016 | 0x01F8 | “logr” padding size (504 bytes). |
| 0x040018 | <spare data> | 488 spare bytes up to 0x40200 |
| 0x040200 | “JUNK” | JUNK sub-chunk |
| 0x040204 | 0x0001FBF8 | JUNK chunk size (up to 0x0005FE00) |
| 0x05FE00 | “logr” | Sub-chunk ID |
| 0x05FE04 | 0x000003F8 | Size of sub-chunk |
| 0x060000 | <raw logging data> | Note – starts after “logr” padding. |
| 0x060200 | “JUNK” | JUNK sub-chunk after session stopped. |
| 0x060204 | 0x0001FDF8 | JUNK chunk size (up to 0x00080000) |
| 0x080000 | “JUNK” | JUNK sub-chunk after session stopped. |
| 0x080004 | 0x00BFFF8 | JUNK chunk size (up to 0x00140000) |

**Table 2-9 Example RIFF file (one session recording).**

The table above shows a single session recording started at 15:58:13 on the 15th of January 2008. The session is short, but is for illustration purposes only. Note that the final JUNK chunk covers the remainder of the file. As the logging data has yet to wrap around to the beginning again, the JUNK chunk is written to cover until the end of the file. To be precise, the final JUNK chunk is written after the newest session and covers the file up until the beginning of the oldest session. Each sub-chunk is outlined to clearly identify the data within.





Ph: (443) 569-3603 Fax: (443) 926-9402 Web: www.zephyr-technology.com

# 3 Appendix A – PC Log Import Example Implementation

This section provides an example implementation of how to import logging data with a PC.

The log data can be processed by implementing the “Read Logging Data” and “Delete Log File” commands (see BioHarness Bluetooth Comms Link Specification **[1]**).
As far as the PC is concerned, it is accessing a standard file using these commands. Figure 1-1 should be referenced to see an example layout in the RIFF file.

## *3.1 Verifying the RIFF and Zephyr File Identifier Chunks*

The RIFF Chunk descriptor and Zephyr File Identifier chunk should be read from the device and verified. This should be performed by using the “Read Logging Data” command. Reading the “RIFF Chunk Descriptor” will allow the PC application to determine the size of the file on the device as well as the “file format”; a BioHarness will have a “file format” field of “BIO ”. The Zephyr File Identifier sub-chunk determines the project and version used; the PC may have to perform different processing on the data depending on the Project ID and Project Version found in the file.

The following serial data session shows the data transfer to read the RIFF chunk data from the start of the file:

```
Request: 12/30/2007 6:30:04 PM.61564 (+0.0156 seconds)
 02 01 06 00 00 00 00 00 04 61 03                  .........a.

Answer: 12/30/2007 6:30:04 PM.63064 (+0.0156 seconds)
 02 01 04 52 49 46 46 CA 06                        ...RIFFÊ.

Request: 12/30/2007 6:30:04 PM.63064 (+0.0000 seconds)
 02 01 06 00 04 00 00 00 04 7E 03                  .........~.

Answer: 12/30/2007 6:30:04 PM.64664 (+0.0156 seconds)
 02 01 04 F8 DF 57 0E 2D 06                        ...øßW.-.

Request: 12/30/2007 6:30:04 PM.67764 (+0.0313 seconds)
 02 01 06 00 08 00 00 00 04 5F 03                  .........._.

Answer: 12/30/2007 6:30:04 PM.69364 (+0.0156 seconds)
 02 01 04 42 49 4F 20 F8 06                        ...BIO ø.
```

The transfer above shows the PC application requesting 12 bytes from location zero (start of the logging file) in 3 separate transfers; the device firstly responds with “RIFF” which is the expected data at the start. The next four bytes read indicates the size of the file data area being 0x0E57DFF8; note that the file size takes into account the 8 bytes for the RIFF Chunk ID and Chunk Size (**“RIFF”<0x0E57DFF8>**). The next four bytes read indicates the RIFF File Format being “BIO ”. After reading this information the PC application now knows the file is of RIFF format, its size and also that it is on a BioHarness due to the “BIO ” string.

```
Request: 12/30/2007 6:30:04 PM.69364 (+0.0000 seconds)
 02 01 06 00 0C 00 00 00 04 40 03                 .........@.

Answer: 12/30/2007 6:30:04 PM.70964 (+0.0156 seconds)
 02 01 04 7A 70 68 72 34 06                       ...zphr4.

Request: 12/30/2007 6:30:04 PM.72464 (+0.0156 seconds)
 02 01 06 00 10 00 00 00 04 1D 03      ...............

Answer: 12/30/2007 6:30:04 PM.27164 (+0.0156 seconds)
 02 01 04 08 00 00 00 1C 06                       .........

Request: 12/30/2007 6:30:04 PM.27164 (+0.0000 seconds)
 02 01 06 00 14 00 00 00 08 A1 03                 .........¡.

Answer: 12/30/2007 6:30:04 PM.28764 (+0.0156 seconds)
 02 01 08 62 68 6E 73 30 30 30 33 AA 06           ...bhns0003ª.
```

The next serial data shown above indicates the transfer to retrieve the Zephyr File Identifier sub-chunk information. We see that all fields so far are as follows:

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x000000 | “RIFF” | RIFF chunk descriptor |
| 0x000004 | 0x0E57DFF8 | File size (minus first 8 bytes). |
| 0x000008 | “BIO ” | RIFF file format |
| 0x00000C | “zphr” | Sub-chunk ID |
| 0x000010 | 0x00000008 | Size of sub-chunk |
| 0x000014 | “bhns” | Project identifier |
| 0x000018 | “0003” | Project version |

**Table 3-1 Example data retrieved from the RIFF chunk and Zephyr File Identifier**

## *3.2 Processing the File Management Sectors area*

Although the data within the File Management Sectors area does not require processing by the PC application, it is a valid chunk and therefore the chunk length must be retrieved so the application can move to the next chunk:

```
Request: 12/30/2007 6:30:04 PM.30264 (+0.0156 seconds)
 02 01 06 00 1C 00 00 00 04 3C 03                  .........<.

Answer: 12/30/2007 6:30:04 PM.30264 (+0.0000 seconds)
 02 01 04 4A 55 4E 4B 94 06                        ...JUNK?.

Request: 12/30/2007 6:30:04 PM.31864 (+0.0000 seconds)
 02 01 06 00 20 00 00 00 04 99 03                  .... ....?.

Answer: 12/30/2007 6:30:04 PM.33464 (+0.0156 seconds)
 02 01 04 D4 01 00 00 A4 06                        ...Ô...¤.

Request: 12/30/2007 6:30:04 PM.33464 (+0.0000 seconds)
 02 01 06 00 F8 01 00 00 04 76 03                  ....ø....v.

Answer: 12/30/2007 6:30:05 PM.34964 (+0.0156 seconds)
 02 01 04 66 6D 73 20 EC 06                        ...fms ì.

Request: 12/30/2007 6:30:05 PM.34964 (+0.0000 seconds)
 02 01 06 00 FC 01 00 00 04 69 03                  ....ü....i.

Answer: 12/30/2007 6:30:05 PM.36564 (+0.0156 seconds)
 02 01 04 00 14 00 00 D4 06                        .......Ô.
```

The data transfer between the PC and the device shows that the next chunk processed is a JUNK chunk. We simply skip over this chunk by its size and get the next chunk which happens to be the File Management Sectors chunk. Again, because the PC application does not need to process the File Management Sectors data content, it can skip over it much like a JUNK chunk. We see that the data is as expected:

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x00001C | “JUNK” | JUNK sub-chunk |
| 0x000020 | 0x000001D4 | JUNK chunk size |
| 0x0001F8 | “fms ” | Sub-chunk ID |
| 0x0001FC | 0x00001400 | “fms” chunk size |

**Table 3-2 Example data retrieved from a “JUNK” chunk and a File Management Sectors chunk**

## 3.3 Mapping Logging Data

When the main file header chunks have been read, the application should proceed to map out the session recordings in the file. To do this, the PC should record the Log Header chunks and where they are and also the first part of the Log Raw Data chunks i.e. the size of the logging session data and where the Log Raw Data resides in the file. Recording these parts of the Log Header and Log Raw Data chunks will allow the PC application to make a map of each recording session and allow the user to select the session(s) they wish to import.

```
Request: 12/30/2007 6:30:05 PM.36564 (+0.0000 seconds)
 02 01 06 00 00 16 00 00 04 50 03                  .........P.

Answer: 12/30/2007 6:30:05 PM.38064 (+0.0156 seconds)
 02 01 04 6C 6F 67 68 EB 06                        ...loghë.

Request: 12/30/2007 6:30:05 PM.38064 (+0.0000 seconds)
 02 01 06 00 04 16 00 00 04 4F 03                  .........O.

Answer: 12/30/2007 6:30:05 PM.39664 (+0.0156 seconds)
 02 01 04 F8 01 00 00 C6 06                        ...ø...Æ.
```

This transfer shows the next chunk the PC sees is the Log Header chunk, with a size of 0x1F8:

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x001600 | “logh” | Sub-chunk ID |
| 0x001604 | 0x000001F8 | Size of sub-chunk |

**Table 3-3 Example summary data read from a Log Header chunk**

The PC application then proceeds to read the rest of the Log Header chunk data;

```
Request: 12/30/2007 6:30:05 PM.39664 (+0.0000 seconds)
 02 01 06 00 08 16 00 00 80 83 03                  ........??.

Answer: 12/30/2007 6:30:05 PM.42764 (+0.0313 seconds)
 02 01 80 D0 07 01 01 A8 B5 4A 00 F0 03 00 00 40   ..?Ð...¨µJ.ð...@
 00 F8 01 00 00 00 00 00 00 00 00 00 00 00 00 00   .ø..............
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 65 06                                    ...e.
```

We can see that 128 bytes are read this time from address 0x1608, although we could have read up to and including the Log Header pad size data only. We see the following data from the transfer:

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x001608 | 0x07D0, 0x01, 0x01, 0x004AB5A8 | Timestamp (2000, Jan, 1st, 01:21:36.168) |
| 0x001610 | 0x000003F0 | Logging Period (1008 milliseconds). |
| 0x001614 | 0x0040 | Number of logging channels (64). |
| 0x001616 | 0x01F8 | “logr” padding size (504 bytes). |

| | | |
|---|---|---|
| 0x001618 | <spare data> | 488 spare bytes up to 0x00001800 |

**Table 3-4 Example full data transfer for a Log Header chunk**

The next operation is to read the chunk after the Log Header sub-chunk:

```
Request: 12/30/2007 6:30:05 PM.53764 (+0.0000 seconds)
 02 01 06 00 00 18 00 00 04 45 03                  .........E.

Answer: 12/30/2007 6:30:05 PM.55264 (+0.0156 seconds)
 02 01 04 6C 6F 67 72 08 06                        ...logr..

Request: 12/30/2007 6:30:05 PM.55264 (+0.0000 seconds)
 02 01 06 00 04 18 00 00 04 5A 03                  .........Z.

Answer: 12/30/2007 6:30:05 PM.56864 (+0.0156 seconds)
 02 01 04 78 31 00 00 C1 06                        ...x1..Á.

Request: 12/30/2007 6:30:05 PM.58464 (+0.0156 seconds)
 02 01 06 00 80 49 00 00 04 D9 03                   ....?I...Ù.....?

Answer: 12/30/2007 6:30:05 PM.13064 (+0.0156 seconds)
 02 01 04 4A 55 4E 4B 94 06                        ...JUNK?.

Request: 12/30/2007 6:30:05 PM.13064 (+0.0000 seconds)
 02 01 06 00 84 49 00 00 04 C6 03                  ....?I...Æ.

Answer: 12/30/2007 6:30:05 PM.14664 (+0.0156 seconds)
 02 01 04 78 00 00 00 B4 06                        ...x...´.
```

In this case, we see that the Log Raw Data chunk is 0x3178 in size and from the previous Log Header chunk we know that the Log Raw Data pad size is 0x1F8 bytes, therefore the PC application (having recorded the Log Header and Log Raw Data details and location, but not the raw data), moves on to the next chunk at address 0x4980 (0x1808 + 0x3178). Note that the pad size is 0x1F8, so the raw data starts at 0x1808 + 0x1F8 = 0x1A00; the PC will read data from here if the user selects this recording to import:

| Offset (bytes) | File Entry | Description |
|---|---|---|
| 0x001800 | “logr” | Sub-chunk ID |
| 0x001804 | 0x00003178 | Size of sub-chunk |
| 0x001A00 | <raw logging data> | Note – starts after “logr” padding. |
| 0x004980 | “JUNK” | Sub-chunk ID |
| 0x004984 | 0x00000078 | Size of sub-chunk |

**Table 3-5 Example Summary Data from a Log Raw Data chunk**

We see that the next chunk is a JUNK chunk and can therefore be skipped over to get to address 0x4A00:

```
Request: 12/30/2007 6:30:05 PM.14664 (+0.0000 seconds)
 02 01 06 00 00 4A 00 00 04 9A 03                  .....J...?.

Answer: 12/30/2007 6:30:05 PM.16264 (+0.0156 seconds)
 02 01 04 6C 6F 67 68 EB 06                        ...loghë.
```

The next transfer (shown above) indicates that the chunk directly after the JUNK chunk is another Log Header chunk at address 0x4A00 which can be processed as before.

This cycle continues until the PC reaches the end of the file and has mapped out the locations and sizes of all sessions (Log Header + Log Raw Data) details.

## *3.4 Processing Logging Data*

When the user selects the session(s) they wish to import (based on the map of logging sessions created in the previous section), the PC application needs to retrieve the logging data from the device. In the previous section the location of each session's data chunk within the file was mapped/recorded, therefore when the user selects the session to import, the PC application should be able to read only the required raw logging data from the file:

```
Request: 12/30/2007 6:32:10 PM.88764 (+117.6280 seconds)
 02 01 06 00 00 FE 03 00 80 0C 03                  .....þ..?..

Answer: 12/30/2007 6:32:10 PM.91864 (+0.0313 seconds)
 02 01 80 06 00 32 00 1C 01 CB FF 15 00 AA 0F 5A   ..?..2...Ëÿ..ª.Z
 10 24 0C 00 00 BC 01 00 00 1E 01 C7 FF C8 FF 20   .$...¼.....ÇÿÈÿ
 00 22 00 3B 00 3D 00 1F 00 1F 00 1F 00 1F 00 1F   .".;.=..........
 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F   ................
 00 1F 00 1F 00 1F 00 1F 00 1F 00 EC 00 EC 00 EC   ...........ì.ì.ì
 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC   .ì.ì.ì.ì.ì.ì.ì.ì
 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 00   .ì.ì.ì.ì.ì.ì.ì..
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 C6 06                                    ...Æ.

Request: 12/30/2007 6:32:10 PM.93464 (+0.0000 seconds)
 02 01 06 00 80 FE 03 00 80 C7 03                  ....?þ..?Ç.

Answer: 12/30/2007 6:32:10 PM.94964 (+0.0156 seconds)
 02 01 80 00 00 32 00 1E 01 CE FF 10 00 6E 13 4E   ..?..2...Îÿ..n.N
 10 5E 10 00 00 BC 01 00 00 1E 01 B7 FF C9 FF 00   .^...¼.....·ÿÉÿ.
 00 21 00 30 00 46 00 1F 00 1F 00 1F 00 1F 00 1F   .!.0.F..........
 00 72 00 1A 02 B4 04 91 07 53 0A 6E 0C B4 0D 6D   .r...´.?.S.n.´.m
 0E D7 0E 12 0F 33 0F 2B 0F D5 0E EC 00 EC 00 EC   .×...3.+.Õ.ì.ì.ì
 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC 00 EC   .ì.ì.ì.ì.ì.ì.ì.ì
 00 EC 00 EC 00 EC 00 EC 00 B0 EB B0 EB 00         .ì.ì.ì.ì.ì.°ë°ë.
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 00 00 00 60 06                                    ...`.
```

Note that the example transfers above do not read the first session from the device (which would have been at 0x1A00), but another recording at 0x3FE00. The data above only shows the 1st two data transfers from the device with raw session data.
If the 1st recording were chosen, we would instead see:

```
Request: 12/30/2007 6:32:10 PM.88764 (+117.6280 seconds)
 02 01 06 00 00 1A 00 00 80 xx 03                  .....þ..?..

Answer: 12/30/2007 6:32:10 PM.91864 (+0.0313 seconds)
 02 01 80 06 00 32 00 1C 01 CB FF 15 00 AA 0F 5A   ..?..2...Ëÿ..ª.Z
 10 24 0C 00 00 BC 01 00 00 1E 01 C7 FF C8 FF 20   .$...¼.....ÇÿÈÿ
 00 22 00 3B 00 3D 00 1F 00 1F 00 1F 00 1F 00 1F   .".;.=..........
 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F 00 1F   ................
```

Because we know the size of the first session to be 0x4980 – 0x1A00 = 2F80, the PC continues to retrieve data until all 0x2F80 bytes have been retrieved.
The data can then be interpreted as a series of log records, formatted as described in Appendix B – BioHarness™ Log Record format.

# 4 Appendix B – BioHarness™ Log Record format

**BioHarness™ Log Records**

| Item | Byte Offset | Length | Description |
|---|---|---|---|
| Heart Rate | 0 | 2 | 0 – 240 Beats Per Minute (in 1 BPM units) |
| Respiration Rate | 2 | 2 | 0 – 170 Breaths Per Minute (in 0.1 BPM units) |
| Skin Temperature | 4 | 2 | 10 – 60℃ (in 0.1℃ units) |
| Posture | 6 | 2 | 0 – 90° (in 1° units) |
| Vector Magnitude | 8 | 2 | (in 0.01g units) |
| Peak Acceleration | 10 | 2 | (in 0.01g units) |
| Battery Voltage | 12 | 2 | 0 – 4.2v (in mV units) |
| ~~Not Currently Used~~ | ~~14~~ | ~~2~~ | |
| GSR Level | 16 | 2 | (in 1nS units) |
| ~~Not Currently Used~~ | ~~18~~ | ~~2~~ | |
| ~~Not Currently Used~~ | ~~20~~ | ~~2~~ | |
| ~~Not Currently Used~~ | ~~22~~ | ~~2~~ | |
| X Acceleration Min | 24 | 2 | (in 0.01g units) |
| X Acceleration Peak | 26 | 2 | (in 0.01g units) |
| Y Acceleration Min | 28 | 2 | (in 0.01g units) |
| Y Acceleration Peak | 30 | 2 | (in 0.01g units) |
| Z Acceleration Min | 32 | 2 | (in 0.01g units) |
| Z Acceleration Peak | 34 | 2 | (in 0.01g units) |
| Breathing Data 1 | 36 | 2 | 0 – 4095 (raw ADC data) – sample 1. |
| Breathing Data 2 | 38 | 2 | 0 – 4095 (raw ADC data) – sample 2. |
| Breathing Data 3 | 40 | 2 | 0 – 4095 (raw ADC data) – sample 3. |
| Breathing Data 4 | 42 | 2 | 0 – 4095 (raw ADC data) – sample 4. |
| Breathing Data 5 | 44 | 2 | 0 – 4095 (raw ADC data) – sample 5. |
| Breathing Data 6 | 46 | 2 | 0 – 4095 (raw ADC data) – sample 6. |
| Breathing Data 7 | 48 | 2 | 0 – 4095 (raw ADC data) – sample 7. |
| Breathing Data 8 | 50 | 2 | 0 – 4095 (raw ADC data) – sample 8. |
| Breathing Data 9 | 52 | 2 | 0 – 4095 (raw ADC data) – sample 9. |
| Breathing Data 10 | 54 | 2 | 0 – 4095 (raw ADC data) – sample 10. |
| Breathing Data 11 | 56 | 2 | 0 – 4095 (raw ADC data) – sample 11. |
| Breathing Data 12 | 58 | 2 | 0 – 4095 (raw ADC data) – sample 12. |
| Breathing Data 13 | 60 | 2 | 0 – 4095 (raw ADC data) – sample 13. |
| Breathing Data 14 | 62 | 2 | 0 – 4095 (raw ADC data) – sample 14. |
| Breathing Data 15 | 64 | 2 | 0 – 4095 (raw ADC data) – sample 15. |
| Breathing Data 16 | 66 | 2 | 0 – 4095 (raw ADC data) – sample 16. |
| Breathing Data 17 | 68 | 2 | 0 – 4095 (raw ADC data) – sample 17. |
| Breathing Data 18 | 70 | 2 | 0 – 4095 (raw ADC data) – sample 18. |
| R to R time 1 | 72 | 2 | Last R to R period (ms) – sample 1. |
| R to R time 2 | 74 | 2 | Last R to R period (ms) – sample 2. |
| R to R time 3 | 76 | 2 | Last R to R period (ms) – sample 3. |
| R to R time 4 | 78 | 2 | Last R to R period (ms) – sample 4. |

| | | | |
|---|---|---|---|
| R to R time 5 | 80 | 2 | Last R to R period (ms) – sample 5. |
| R to R time 6 | 82 | 2 | Last R to R period (ms) – sample 6. |
| R to R time 7 | 84 | 2 | Last R to R period (ms) – sample 7. |
| R to R time 8 | 86 | 2 | Last R to R period (ms) – sample 8. |
| R to R time 9 | 88 | 2 | Last R to R period (ms) – sample 9. |
| R to R time 10 | 90 | 2 | Last R to R period (ms) – sample 10. |
| R to R time 11 | 92 | 2 | Last R to R period (ms) – sample 11. |
| R to R time 12 | 94 | 2 | Last R to R period (ms) – sample 12. |
| R to R time 13 | 96 | 2 | Last R to R period (ms) – sample 13. |
| R to R time 14 | 98 | 2 | Last R to R period (ms) – sample 14. |
| R to R time 15 | 100 | 2 | Last R to R period (ms) – sample 15. |
| R to R time 16 | 102 | 2 | Last R to R period (ms) – sample 16. |
| R to R time 17 | 104 | 2 | Last R to R period (ms) – sample 17. |
| R to R time 18 | 106 | 2 | Last R to R period (ms) – sample 18. |
| Not Currently Used | 108 | 2 | |
| Not Currently Used | 110 | 2 | |
| Not Currently Used | 112 | 2 | |
| Not Currently Used | 114 | 2 | |
| Not Currently Used | 116 | 2 | |
| Not Currently Used | 118 | 2 | |
| Not Currently Used | 120 | 2 | |
| Not Currently Used | 122 | 2 | |
| Not Currently Used | 124 | 2 | |
| Not Currently Used | 126 | 2 | |
| Total Bytes | | 128 | |

All items are in Little Endian format and are 16-bit signed numbers.

Each entry is 2 bytes in size.

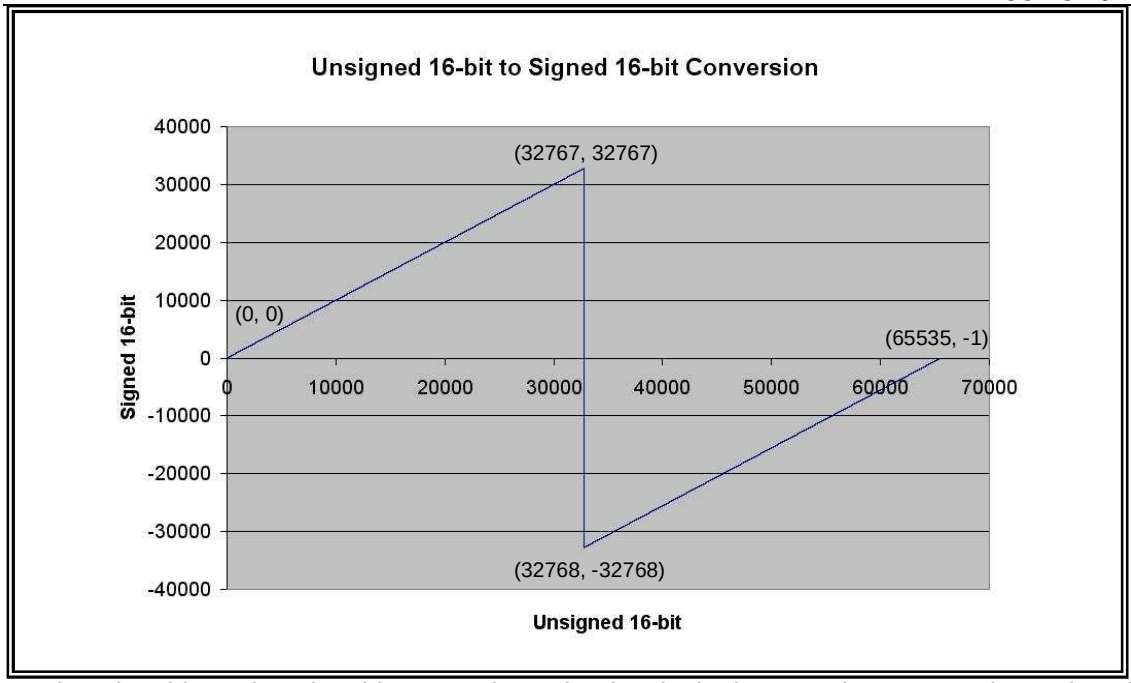
Note that 18 Breathing Data samples are written to each log record. This equates to 56ms resolution Breathing Data, allowing the PC application to calculate the Respiration Rate from the stored data.
There are also 18 R to R samples stored, however each sample contains the most recent R to R period with 1ms resolution, updated every 56ms.
When a Heart beat is detected, the indicated R to R period is updated. New R to R periods are signified by a change in the sign of the value logged, enabling new R to R values to be recognised even if the new period is the same as the old. For example, if a pulse is received by the unit and the new R to R is the same as the last one, values in the log could be:

**876, 876, 876, -876, -876, -876**

The change in sign indicates a new R to R period (old and new R to R = 876ms).

*Byte packing (16-bit signed, Little Endian format):*

| |
|---|
| LS 8 bits |
| MS 8 bits |




*Unsigned 16-bit to Signed 16-bit conversion – the data in the log must be converted to a signed 16-bit number before being processed.*





Ph: (443) 569-3603
Fax: (443) 926-9402
Web: www.zephyr-technology.com