Binary file structure for PAMGUARD detector output.

Version 4.0

D. Gillespie & M. Oswald, February 2017

# Introduction

This document describes the binary file storage structures used by PAMGuard. Prior to 2010 the primary storage site for PAMGUARD output data was a relational database (MS Access, MySQL, SQLite). The function and capabilities of the binary storage system is fundamentally different to the storage of data in the database. Most importantly, a relational database is not suitable for storage of variable record length data (e.g. a short clip of click waveform from the click detector or the time/amplitude/frequency contour of a dolphin whistle). Furthermore, the database interface is slow and some databases have limited size.

The binary storage module is designed to handle any type of data, particularly data having a variable record length. All data stored in binary files have a common overarching structure, but each PAMGuard module contains bespoke functions for writing their particular types of data.

Although PAMGuard is written in Java, the binary data format does not use any Java specific formats (i.e. Java serialisation). This means that PAMGuard binary files may be read by any program capable of opening a file and reading data from it (e.g. C, C++, Matlab, R, etc.).

Binary storage is enabled in PAMGuard by adding a “Binary Storage” module from the main file menu. Only one Binary Storage module is allowed in a PAMGuard configuration. Each PAMGuard output stream (or PamDataBlock) that is to write binary data (which is by no means all of them) will automatically connect to the binary store and its data will be saved. When configuring the binary store, the user specifies a folder or directory on their computer for data output and has the option to place each day’s data in a separate folder (these sub folders switch at midnight GMT, not local time). The user can also specify how long each file should be. The default setting for this is one hour.

Binary files end with .pgdf for PAMGuard Data File. A common file name format is used for all PAMGuard module output: file names are made up from the module type, the module name and the data stream name plus the data in a YYYYMMDD\_HHMMSS time format. For example a click detector file name might read Click\_Detector\_Beaked\_Whales\_Clicks\_20150825\_032012.pgdf, i.e. a PAMGuard “Click Detector” module, called “Beaked Whales” with an output data stream “Clicks” which was created at 03:20:12 on 25 August, 2015.

Each pgdf file contains the following blocks of data:

1. A general header which has the same format for all data streams.
2. A module specific header (optional) which may contain module specific configuration data.
3. Data objects. These can be of more than one type and there may be any number of them.
4. A module specific footer (perhaps giving summary data for that module over the duration of the file)
5. A general footer with information such as the data end time. This is the same for all modules.

Both the general and the module header contain version numbers which enable us to change the format over time. There will always be backwards compatibility with older data types, however new data created with new PAMGuard versions may not open with older versions. From PAMGuard version 1.15.04 a warning will be issued if you attempt to open binary files created with a later PAMGuard version. For example, if you collected data with version 2.00.00 (which does not exist yet, but may use a slightly different file format) and attempt to open those files with PAMGuard 1.15.03 uncontrolled errors will occur. If you attempt to open those files with version 1.15.04, PAMGuard may not be able to read the files, but warnings will be issued telling you to upgrade your PAMGuard version.

Output (from PAMGuard) uses only sequential file access (rather than random access), although other programs could of course open the files in any way they wish. This means that the file headers contain the file start time, but not the file end time, length or number of data objects which are only encoded in the file footer. To speed up data indexing when dealing with large data sets, the headers and footers (items 1,2 4 and 5 in the above list) are also written into files ending with .pgdx which have the same name as the .pgdf files. These are used for mapping and finding data when using the PAMGuard viewer. During data analysis using the PAMGuard Viewer, a data structure known as a “datagram” may also be added to the pgdx index file.

# File Format

Java writes data using a big endian format irrespective of the platform it’s running on (see <http://en.wikipedia.org/wiki/Endianness>). This is different to the standard format for the Windows system which is little endian. This means that if you’re reading the data with C, you’ll have to do a lot of byte swapping to make sense of the data coming in. Also note that Java does not support unsigned integer values. For the purposes of this document we will use the terms int8, int16, int32 and int64 to describe the various formats (see Table 1). Strings, which may have a variable length are generally written with the Java DataOutputStream.writeUTF() function. For standard ASCII characters, this will be two bytes (written as an int16) giving the length of the string followed by one byte per character. Unicode characters are also supported in this format, but are not used within PAMGuard – for details see the [JAVA Help](http://en.wikipedia.org/wiki/UTF-8#Modified_UTF-8) and [Wikipedia](http://en.wikipedia.org/wiki/UTF-8#Modified_UTF-8).

Table . Data formats used in this document and common programming languages.

|  |  |  |  |
| --- | --- | --- | --- |
| This document | Length in bytes | Java | C / C++ (32 bit platforms) |
| int8 | 1 | byte | char or int8\_t |
| int16 | 2 | short | short or int16\_t |
| int32 | 4 | int | int, long or int32\_t |
| float | 4 | float | float |
| int64 | 8 | long | long long, int64\_t |
| double | 8 | double | double |
| char[n] | Fixed length string of n characters | | |
| charUTF | Variable length string (see text) | | |

Reading PAMGuard files with Matlab it’s relatively straight forward since you can specify endianness as you open the file, e.g.

f = fopen(fileName, 'r', 'ieee-be.l64');

## File Structures

Each pgdf file contains a series of binary objects. Every object will start with an int32 giving the size of that object in bytes. This number includes itself in the size calculation. So it will always be possible to skip through the file using the following pseudocode:

While not eof

objectSize = ReadInt32()

SkipForwardBytes(objectSize-4)

Next

The number immediately after the objectSize is another int32 giving the object type. This will be a negative number for the header/footer/datagram objects, and a positive number for any data in the file. Object identifiers used by the file management system are:

-1 = File Header

-2 = File Footer

-3 = Module header

-4 = Module footer

-5 = Datagram

The objects for the file header, footer and datagram are fixed across all data streams. Module headers and footers have a fixed section (e.g. containing the module version number) and then a variable length section for module specific data (e.g. detector or process configuration values such as thresholds, FFT lengths, etc.). Data objects have a fixed header containing the time of the data object and a variable length section which can be in any format (it being the responsibility of individual module developers to ensure backwards compatibility should anything change).

The main data format for files is shown in Table 2.

Table . Data format for PAMGuard binary files.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Header Version** | **Item** | **Format** | **Notes** |
| File Header | 0+ | Length of file header in bytes | Int32 | Every object will start with this number. |
| 0+ | Object Identifier | Int32 | Always -1 |
| 0+ | Header / general file Version | Int32 | Currently 4 |
| 0+ | “PAMGUARDDATA” | Char(12) | Just so it’s obvious that this really is a P file |
| 0+ | PAMGUARD Version | CharUTF\* | e.g. 1.8.00 |
| 0+ | PAMGUARD Branch | CharUTF\* | e.g. Core, Beta, etc. |
| 0+ | Data Date | Int64 | Data time at start of file in Java millis |
| 0+ | Analysis Date | Int64 | Time at which analysis started (same as data time for real time data, or the analysis date for data processed offline) |
| 0+ | File Start Sample | Int64 | Current sample number for this data stream |
| 0+ | Module type | CharUTF\* | Module type |
| 0+ | Module Name | CharUTF\* | Module name |
| 0+ | Stream Name | CharUTF\* | Data stream name |
| 0+ | Extra info length | Int32 | Length of additional data |
| 0+ | Extra info | byte[] | Additional data (not currently read back in) |
| Module Header | 0+ | Length in File | Int32 | Length of this object = 16 + object binary length |
| 0+ | Object Identifier | Int32 | Always -3 |
| 0+ | Module version Info | Int32 | Version info specific to the pamguard module writing data to this stream. This is more likely to change than the general file format in the main file header and reflects small changes in the structure of this specific module. |
| 0+ | Object binary Length | Int32 | = Length in File – 16 (a bit of redundancy) ! Will be zero if there is no additional data |
| 0+ | Object Data | Byte[] | Length = Object binary Length. This data will be module specific and should contain essential configuration data such as the frequency bands of a noise measurement, the FFT length used for whistle detection, etc. See Section 3 for information on specific module headers |
| Data Object 1 (See Table 3 for general data structure, and Section 4 for module-specific data format) | | | | |
| Data Object 2 (See Table 3 for general data structure, and Section 4 for module-specific data format) | | | | |
| Data Object 3 (See Table 3 for general data structure, and Section 4 for module-specific data format) | | | | |
| etc. … | | | | |
| Module footer | 0+ | Length in File | Int32 | Length of this object = 12 + object binary length |
| 0+ | Object Identifier | Int32 | Always -4 |
| 0+ | Object binary Length | Int32 | = Length in File – 12 (a bit of redundancy) ! Will be zero if there is no additional data |
| 0+ | Object Data | Byte[] | Length = Object binary Length. See Section 5 for information on specific module footers |
| File Footer | 0+ | Length in File | Int32 | 48 (version <3) or 64 (version 3+) |
| 0+ | Object identifier | Int32 | Always -2 |
| 0+ | Total number of objects in file | Int32 | Not counting header, control struct and footer (i.e. can be = 0) |
| 0+ | Data Date | Int64 | Data time at end of file in Java millis |
| 0+ | Analysis Date | Int64 | Time at which analysis ended (same as data time for real time) |
| 0+ | File End Sample | Int64 | Sample number at end of file |
| 3+ | Preceeding UID | Int64 | The UID of the last object in the previous file |
| 3+ | Highest UID | Int64 | The UID of the last object in the file |
| 0+ | File length | Long (Int64) | Total length of the file (will be more use when this is repeated in an index file) |
| 0+ | File End Reason | Int32 | Reason file ended |
| Datagram | ? | Length in File | Int32 | 24 + Length of data (see note below) |
| ? | Object Identifier | Int32 | Always -5 |
| ? | Time Interval in seconds | Int32 |  |
| ? | Datagram format | Int32 | Currently 0 |
| ? | Number of data points in data gram *ndp* | Int32 |  |
| ? | Number of data units in each data point *ndu* | Int32 |  |
| Then follows a double nested loop, the outer loop over the objects and the inner loop over the data for each object. The outer loop does not write any data. The inner loop writes the following: | | | |
| ? | Start Time | Int64 | Length of data = *ndp* \* (8 + 8 + 4 \* *ndu*) |
| ? | End Time | Int64 |
| ? | Data Units | Float[] |
| EOF |  |  |  |  |

Table 3 shows the format of data objects within a binary file. A file can contain objects of more than one type with each object being uniquely identified by it’s object identifier. These need to be unique within a file, but do not need to be unique across PAMGuard. Note that as of version 3, certain variables (marked optional in table) will only be written to the binary file if they have been set by the data object.

Table . General Data Structure.

|  |  |  |  |
| --- | --- | --- | --- |
| **Header Version** | **Item** | **Format** | **Notes** |
| 0+ | Length in File | Int32 | Length of this object |
| 0+ | Object Identifier | Int32 | Positive integer (can be 0) which must be unique within this data stream, not across PAMGuard |
| 0+ | Time milliseconds | Int64 | Timestamp in milliseconds |
| 3+ | Contents of flag bitmap | Int16 | A set of flags indicating which optional variables are included in the data object |
| 2+ | Time nanoseconds | Int64 | Additional nanosecond resolution time stamp (optional in version 3+) |
| 2+ | Channel bitmap | Int32 | Channel bitmap (optional in version 3+) |
| 3+ | UID | Int64 | Object UID (optional) |
| 3+ | Start Sample | Int64 | Start sample of the object (optional) |
| 3+ | Sample Duration | Int32 | Duration of object in samples (optional) |
| 4+ | Frequency limits | 2\*Float | min and max frequency limits, in that order. Either both values are stored, or nothing is stored (i.e. there will never be an instance of storing only the max frequency) (optional) |
| 4+ | Millisecond Duration | Float | Duration of object in milliseconds (optional) |
| 4+ | Num of Time Delays *nD* | Int16 | Number of time delays (optional) |
| 4+ | Time Delays | Float[*nD*] | *nD* Float values (optional) |
| 0+ | Object binary length ‘l’ | Int32 | = Length in File – 20 (version < 2)  = Length in File – 32 (version 2)  = version 3+ object length cannot be equated to Length in File because variables are only written if the values are set. |
| 0+ | Object Data | Byte[l] | Length = Object binary Length. Need not be same as Object 1. See Section 4 for information on specific module data |

# Header formats for specific modules

## LTSA Module

Table 4. LTSA Module header information

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 0+ | FFT Length | Int32 |  |
| 0+ | FFT Hop | Int32 |  |
| 0+ | Interval (seconds) | Int32 |  |

## Noise Monitor

Table 5. Noise Monitor header information

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1+ | Number of Bands *nb* | Int16 |  |
| 1+ | Statistic Types | Int16 | Bitmap of which bands are used |
| 1+ | Low Freq Edges | Float[*nb*] | List of low frequency edges |
| 1+ | High Freq Edges | Float[*nb*] | List of high frequency edges |

## Whistle and Moan Detector

Table 6. Whistle and Moan Detector header information

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1+ | Delay Scale | Int32 |  |

# Data formats for specific modules

## AIS Processing Module

Table 7. Format for AIS Processing. Object Identifier = 0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1+ | MMSI Number | Int32 |  |
| 1+ | Fill Bits | Int16 | Number of "fill-bits" added to complete the last six-bit character |
| 1+ | Character Data | CharUTF | Contents of the M.1371 radio message using the six-bit field type |
| 1+ | AIS Channel | CharUTF | Either 1 or 2, or null if not provided |

## Click Detector Version 0

Table 8. Format for click detector data V0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 0 | Time millis | Int64 |  |
| 0 | Start sample | Int64 |  |
| 0 | Channel map | Int32 |  |
| 0 | Triggered channels | Int32 |  |
| 0 | Num delay measurements, *nd* | Int16 | Usually nChan\*(nChan-1)/2 |
| 0 | Delay measurements | Float[*nd*] |  |
| 0 | Num angle measurements *na* | Int16 | (0, 1 or 2) |
| 0 | Angle measurements | Float[*na*] |  |
| 0 | Duration (samples) | Int16 |  |
| 0 | WaveData | Int16[][] |  |

## Click Detector Version 1 and above

Table 9. format for click detector data. Click Object Identifier = 1000.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1-3 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 4+ |
| 1-3 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 4+ |
| 1+ | Triggered channels | Int32 |  |
| 1+ | Click Type | Int16 |  |
| 2+ | Click flags | Int32 |  |
| 1-3 | Num delay measurements *nd* | Int16 | Usually nChan\*(nChan-1)/2. As of version 4, this is now saved to the general data structure (Table 3) |
| 1-3 | Delay measurements | Float[*nd*] | only written if *nd* <> 0. As of version 4, this is now saved to the general data structure (Table 3) |
| 1+ | Num angle measurements *na* | Int16 | (0, 1 or 2) |
| 1+ | Angle measurements | Float[*na*] | only written if *na* <> 0 |
| 3+ | Num angle measurement errors *ne* | Int16 |  |
| 3+ | Angle measurement errors | Float[*ne*] | only written if *ne* <> 0 |
| 1-3 | Duration (samples) | Int16 | Removed from here and saved to general data structure (Table 3) version 4+ |
| 1+ | Wave Max Amplitude | float | Max amplitude of wave data. |
| 1+ | Then follows a double nested loop, the outer loop over the channels and the inner loop over the duration. | | |
| WaveData | Int8[][] | Wavedata scaled by 127/max amplitude so it uses full dynamic range of 8 bit data. |

## Clip Generator

Table 10. Format for clip generator. Clip Generator Object Identifier = 1 (store basic data only) or 2 (store basic data and wave data).

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1+ | Trigger (milliseconds) | Int64 |  |
| 1 | Duration (samples) | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1+ | Filename | CharUTF |  |
| 1+ | Trigger name | CharUTF |  |
| The following audio data is only written when the Object Identifier = 2 | | | |
| 1+ | Number of channels of data *nc* | Int16 |  |
| 1+ | Number of samples *ns* | Int32 | Number of samples of data/channel |
| 1+ | Scaling factor | Float |  |
| 1+ | Then follows a double nested loop, the outer loop over the channels *nc* and the inner loop over the number of samples *ns*. | | |
| WaveData | Int8[*nc*][*ns*] | Wavedata scaled by 127/max amplitude so it uses full dynamic range of 8 bit data. |

## dBHt Measurement Module

Table 11. Format for dBHt Module. Object Identifier = 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 1 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1+ | RMS | Int16 | RMS value scaled up by 100 |
| 1+ | Zero Peak | Int16 | Zero peak scaled up by 100 |
| 1+ | Peak Peak | Int16 | Peak-Peak value scaled up by 100 |

## Difar Module

Table 12. Format for Difar module. Object Identifier = 0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 0-1 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | Clip Start (milliseconds) | Int64 | Start of clip, in milliseconds |
| 0-1 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | Display Sample Rate | Float |  |
| 0+ | Number of samples of demuxed data *ns* | Int32 |  |
| 0-1 | Lower Frequency Limit | Float | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0-1 | Upper Frequency Limit | Float | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | Amplitude (dB) | Float | Calculated amplitude, in dB |
| 0+ | Gain | Float | Gain value, or -9999 if there is no gain |
| 0+ | Selected Angle | Float | Angle selected from the Difar-gram |
| 0+ | Selected Frequency | Float | Frequency selected from the Difar-gram |
| 0+ | Species Code | CharUTF |  |
| 1+ | Tracked Group Code | CharUTF |  |
| 0+ | Max Demux Data *max* | Float | Maximum value of the demuxed data |
| 0+ | Then follows a double nested loop, the outer loop over the 3 data types (Omni, EW and NS respectively) and the inner loop over the number of samples *ns*. | | |
| Demuxed data | Int16[*3*][*ns*] | Demuxed data scaled by 32767/*max* so it uses full dynamic range of 16 bit data. |
| 0+ | Number of Matched Units *nmu* | Int16 |  |
| The following are only written if the number of matched units *nmu* > 0. | | | |
| 0+ | Cross Location Latitutde | Float |  |
| 0+ | Cross Location Longitude | Float |  |
| 1+ | Cartesian X Error | Float |  |
| 1+ | Cartesian Y Error | Float |  |
| 0+ | Then follows a loop over the number of matched units *nmu*. For each pass through the loop, the following two data points are written: | | |
| Channel Bitmap | Int16 |  |
| Time (milliseconds) | Int64 |  |

## LTSA Module

Table 13. Format for the LTSA module. Object Identifier = 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module Version** | **Item** | | **Format** | **Notes** |
| 0-1 | Start sample | | Int64 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0 | Duration | | Int64 | Removed after version 0 |
| 0-1 | Channel map | | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | End Time (milliseconds) | | Int64 |  |
| 0+ | nFFT | | Int32 |  |
| 0+ | Max Amplitude *max* | | Float |  |
| 0+ | Then follows the LTSA data, the format of which depends on the version. Version 0 scaled the data by 32767/*max* and stored as Int16. Versions 1+ also scaled, but then converted to log scale and stored as Int8. The size of the array is ½ \* FFT Length (as read from the module-specific header) | | | |
| Ver 0 | LTSA Data | Int16[.5\*FFT] | Data scaled by 32767/*max* so it uses full dynamic range of 16 bit data. |
| --- or --- | | | |
| Ver 1+ | LTSA Data | Int8[.5\*FFT] | Data scaled by 32767/*max* and then converted to log scale |

## Noise Monitor

Table 14. Format for the Noise Monitor module. Object Identifier = 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Module Version** | **Item** | | **Format** | | **Notes** |
| 0+ | Number of Channels *nc* | | Int16 | |  |
| 0+ | Number of Bands *nb* | | Int16 | |  |
| 1+ | Number of Measures *nm* | | Int16 | | *nm* = 4 for Version 0 |
| 0+ | Then follows a double nested loop, the outer loop over the bands *nb* and the inner loop over the number of measures *nm*. Version 0 saved the data as float values, but Version 1+ scaled the values up by a factor of 100 and saved as Int16 | | | | |
| Version 0 | Noise Data | Float[*nb*][*nm*] |  | |
| --- or --- | | | | |
| Version 1+ | Noise Data | Int16[*nb*][*nm*] | Noise data scaled by 100. | |

## Noise Band Monitor

Table 15. Format for the Noise Band Monitor module. Object Identifier = 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 0-2 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 3+ |
| 0-2 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 3+ |
| 0+ | RMS | Int16 | RMS value, scaled up by a factor of 100 |
| 0+ | Zero Peak | Int16 | Zero Peak value, scaled up by a factor of 100 |
| 0+ | Peak-Peak | Int16 | Peak-Peak value, scaled up by a factor of 100 |
| 2+ | SEL | Int16 | SEL value, scaled up by a factor of 100 |
| 2+ | SEL Integration Time | Int16 |  |

## Right Whale Edge Detector

Table 16. Format for the Right Whale Edge Detector module. Object Identifier = 0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module Version** | **Item** | **Format** | **Notes** |
| 0 | Start sample | Int64 | Removed from here and saved to general data structure (Table 3) version 1+ |
| 0 | Channel map | Int32 | Removed from here and saved to general data structure (Table 3) version 1+ |
| 0+ | Sound Type | Int16 |  |
| 0+ | Signal | Float |  |
| 0+ | Noise | Float |  |
| 0+ | Number of Slices *ns* | Int16 |  |
| 0+ | Then follows a loop over the number of slices *ns*. For each pass through the loop, the following data points are written: | | |
| Slice Number | Int16 |  |
| Low Frequency | Int16 | Freqnecy in FFT slices. To convert to Hz, multiple this value by (Sampling Rate / FFT Length) |
| Peak Frequency | Int16 | Freqnecy in FFT slices. To convert to Hz, multiple this value by (Sampling Rate / FFT Length) |
| High Frequency | Int16 | Freqnecy in FFT slices. To convert to Hz, multiple this value by (Sampling Rate / FFT Length) |
| Peak Amplitude | Float |  |

## Whistle and Moan Detector

Table 17. Format for Whistle and Moan Detector data. Whistle/Moan Object Identifier = 2000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module Version** | **Item** | | **Format** | **Notes** |
| 0-1 | Start sample | | Int64 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0-1 | Channel map | | Int32 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | Number of FFT slices | | Int16 |  |
| 1+ | Amplitude in dB | | Int16 |  |
| 1 | Number of time delays *nd* | | Int8 | Removed from here and saved to general data structure (Table 3) version 2+ |
| 1 | Time delays in samples | | Int16[*nd*] | only written if *nd* <> 0. Removed from here and saved to general data structure (Table 3) version 2+ |
| 0+ | Then follows a double nested loop, the outer loop over the number of fft slices and the inner loop over the number of peaks within each slice. The outer loop writes the following: | | | |
| Slice Number | | Int32 |  |
| Number of peaks | | Int8 |  |
|  | The inner loop writes the following: | | |
| Low frequency | Int16 | Low edge of sound in FFT bins |
| Peak frequency | Int16 | Peak (loudest) FFT bin |
| High frequency | Int16 | High edge of sound in FFT bins |
| Link peak from previous slice | Int16 | Link to peak in previous slice. |

# Footer formats for specific modules

## Click Detector

Table 18. format for click detector footer. Note that if no click types have been defined, this information is not written to the module footer and the Object Binary Length (from the Module Footer section of Table 2) will equal 0.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module Version** | **Item** |  | **Format** | **Notes** |
| 1+ | Number of click types *nt* |  | Int16 | Number of different click types |
| 1+ | Number of clicks of each type |  | Int32[*nt*] | Number of clicks of each different click type |

# PAMGUARD Settings

PAMGUARD Settings are written to the binary store whenever PAMGUARD starts from the Start menu or from the Network controller. They are not written when PAMGuard restarts due to a buffer overflow in acquisition or when starting to process a new file during offline data analysis. Settings are written to .psfx files. These encapsulate the current psf format used for more general settings, but individual serialised Java objects are wrapped up in a similar way to other binary data so that other programmes (e.g. Matlab) can at least read a list of modules.

Note that datagrams (see Table 2) are stored in the psfx files.