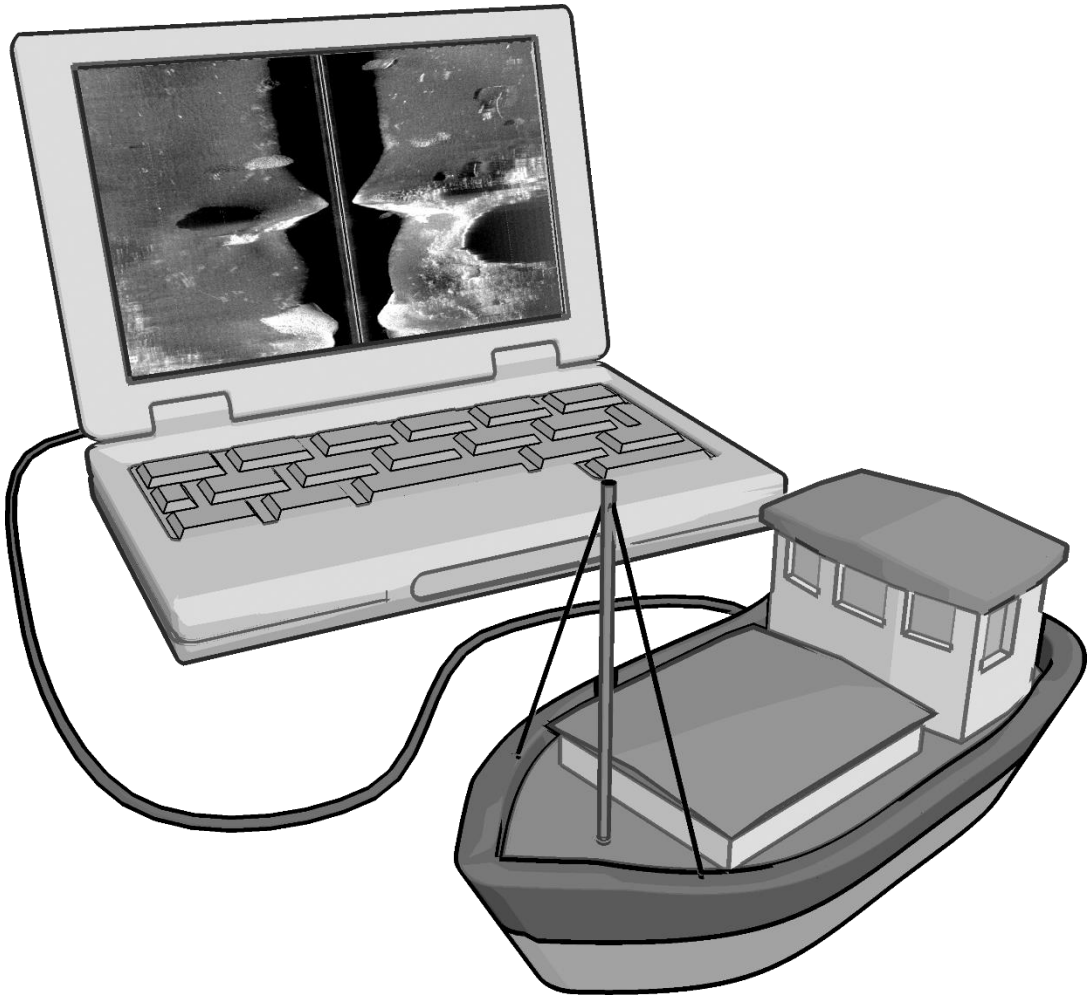
starfish

blueprint subsea



StarFish Scanline

File Formats

Contents

1. Introduction

This document is intended to provide the reader with a sufficient understanding to enable them to decode and interpret the information in the following file types produced by the StarFish Scanline software applications…

- LOGDOC – A version-2 encoded Scanline Log File.
- CSV – Comma Separated Values ASCII data exported from Scanline.
- XTF – eXtended Triton Format data exported from Scanline.

Throughout the documentation the following symbols are used to indicate special precautions or procedures:

WARNING!
This symbol indicates a warning you should follow to avoid bodily injury or damage to your equipment.

CAUTION
This symbol denotes precautions and procedures you should follow to avoid damage to your equipment.

NOTE
This symbol denotes special instructions or tips that should help you get the best performance from your system.

For technical support enquiries and for other support contact details (including website, telephone etc.), please refer to the “Product Support” section of the accompanying Starfish System Manual.

If you think you have found a bug or have any other operational issue, please report it to us at the technical support email address above, and we will endeavour to fix it in the next software release.

Where possible and appropriate, please include as much detail as you can about the issue, including…

- The current software version you’re using (displayed on the software splash screen during loading),
- The hardware details of the system you are running Scanline on
- The circumstances under which the error occurs,
- And the text of any messages displayed on screen.

2. Notices

3. LOGDOC File Format

Scanline records StarFish sonar data in its own native file format, identified with the LOGDOC filename suffix. Readers can use the following sections to help write their own parser for this file format.

For the following sections it is assumed that the reader has a reasonable level of programming expertise and is familiar with the concepts of numerical data types (bytes, integers, floating-point numbers, Boolean logic etc), ASCII/Unicode encoding as strings, and File I/O operations.

This guide starts by outlining the basic file structure and the Data Types and Enumerations that the reader should familiarise themselves with, as extensive reference is made back to these throughout the document.

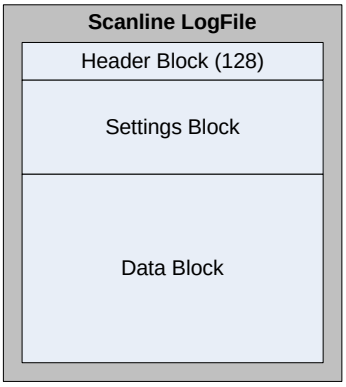
The subsequent sections cover each of the three parts of the Log File in detail (Header, Settings and Data), with descriptions of the various Data Structures that will be encountered, and the purpose of the information contained within them.

The final section looks at putting everything together to decode and subsequently interpret the Log File information.

Please note that this section is **not** aimed to be a complete specification for the Scanline Log File format. There are other undocumented settings and parameters encoded within the Log File used by Scanline for its own operational purposes, and as such these lie outside the scope of this document.

3.1. File Structure

Scanline Log Files are stored on disk as a sequential binary list (or stream) of bytes using the encoding rules and data-types described in the previous section.



The Log File is split into 3 sections…

- The **Header Block**, occupying the first 128 bytes of the file and containing information relating to the size and decoding of subsequent sections in the file.
- The **Settings Block**, which may vary in size and contains a structure similar in nature to the Windows Registry, describing the names and values of parameters related to the hardware configuration used when the data was recorded, which data should be sent to each plotter, environmental constants and display setup.
- The **Data Block** is the largest section of the Log File and contains a sequential list of Data Messages captured from each attached hardware device (sensor) when the Log File was recorded.

3.1.1. Basic Data Types

Before looking at the Scanline Log File format in depth, it is first important to understand the notation and encoding used for the various types of data that are used. The following section describes the size and format of basic data types that are then used to build up the advance (higher-level) objects described in subsequent sections.

Unless otherwise stated, assume that all multi-byte numerical values (integers, word, doubles etc.) are stored in a '**Little-Endian**' format, where the least significant byte value of the entity is at the lowest address. The other bytes follow in increasing order of significance.

To describe numerical values in this document, in addition to standard fractional and integer values, hexadecimal notation is also used. Values described this way are prefixed with '0x' to distinguish them from other numerical formats.

As a rule-of-thumb, these data-types are loosely based around the 'Microsoft C#.Net Framework V2.0' type definitions, and for further information please refer to the appropriate MSDN documentation.

DataType	Length (Bytes)	Comments
Boolean	1	Represents a logical True/False **Boolean** value in the binary list. Each **Boolean** value occupies one byte, and is encoded/decoded as… • Logical "False" as 0. • Logical "True" as 1. Other non-zero values will be interpreted as a "True" when the binary stream is read.
Byte	1	Represents an unsigned 8-bit integer (**Byte**) value in the binary list, where each value occupies one byte. Values can lie in the range 0 to 255.
ByteArray	n+4	Represents an array of 'n' **Bytes** in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **Byte** values directly representing each element in the array, in ascending order from the first array element.
DateTime		Represents a Date and Time value encoded as a structure in the binary list, as 8 consecutive bytes, and is encoded/decoded as follows… • Bytes 0-1: Year, encoded as an **Int16** value. • Byte 2: Month, encoded as a **Byte** (from 1 to 12). • Byte 3: Day, encoded as a **Byte** (from 1 to 31). • Byte 4: Hour, encoded as a **Byte** (from 0 to 23). • Byte 5: Minute, encoded as a **Byte** (from 0 to 59). • Bytes 6-7: Milliseconds, encoded as a **UInt16** (from 0 to 59999) representing the number of seconds & milliseconds in the minute. Divide the value by 1000 and convert the result to an integer to obtain the whole number of seconds. Modulo the value with 1000 to obtain the number of milliseconds remaining. For example, a value of 38250 would equate to 38.250 seconds.

DataType	Length (Bytes)	Comments
		DateTime values only store local times – not UTC time-zone offset or daylight-saving information is stored.
Double	8	Represents a double-precision 64-bit number in the binary list, as 8 consecutive bytes. Each value complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic and is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range -1.79769313486232×10^{308} to 1.79769313486232e308×10^{308}, as well as positive or negative zero, Positive-Infinity, Negative-Infinity, and Not-a-Number (NaN).
DoubleArray	4+(n×8)	Represents an array of 'n' **Doubles** in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **Double** values directly representing each element in the array, in ascending order from the first array element.
Int16	2	Represents a 16-bit signed integer value in the binary list, as 2 consecutive bytes. Each value is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range -32,768 (0x8000) to 32,767 (0x7FFF).
Int32	4	Represents a 32-bit signed integer value in the binary list, as 4 consecutive bytes. Each value is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range -2,147,483,648 (0x80000000) to 2,147,483,647 (0x7FFFFFFF).
Int32Array	4+(n×4)	Represents an array of 'n' **Int32s** in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **Int32** values directly representing each element in the array, in ascending order from the first array element.
Single	4	Represents a single-precision 32-bit number (or 'Float') in the binary list, as 4 consecutive bytes. Each value complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic and is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range -3.402823×10^{38} to 3.402823×10^{38}, as well as positive or negative zero, Positive-Infinity, Negative-Infinity, and not a number (NaN).
SingleArray	4+(n×4)	Represents an array of 'n' **Singles** in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **Single** values directly representing each element in the array, in ascending order from the first array element.
StringAsciiZ	n+1	Encodes a variable length string ('n' characters long) in ASCII encoding, where each character is represented by 1 **Byte**, and terminated with a null character (value 0) **Byte**.
StringAscii	4+(n×1)	Encodes a variable length string in ASCII encoding (where each character is represented by 1 **Byte**) in the binary list.

DataType	Length (Bytes)	Comments
		The number of characters in the string (n) is stored first as an **Int32** object, followed by a sequential list of **Byte** values directly representing each ASCII character in the string, in ascending order from the first array element. If specification require that a string uses this encoding technique, and the length needs to be truncated to a certain size, the length/size value should always be interpreted as the storage space in bytes (not characters), and does not include the 4-byte overhead required to encode the length of the string – unless otherwise stated.
StringAsciiArray	8+(n×1)	Represents an array of 'n' **StringAscii**'s in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **StringAscii** values directly representing each element in the array, in ascending order from the first array element.
StringUtf16	Variable between 4+(n×2) & 4+(n×4)	Encodes a variable length string in Unicode UTF16 encoding (where each character is represented by one or two **UInt16** values) in the binary list. The number of bytes to encode the string (not to be confused with the number of characters 'n') is stored first as an **Int32** object, followed by a sequential list of **UInt16** values representing each Unicode character (or surrogate pair) in the string, in ascending order from the first array element.
StringUtf16Array	8+(n×2)	Represents an array of 'n' **StringUtf16**'s in the binary list. The length of the array (n) is stored first as an **Int32** object, followed by a sequential list of **StringUtf16** values directly representing each element in the array, in ascending order from the first array element.
UInt16	2	Represents a 16-bit unsigned integer value in the binary list, as 2 consecutive bytes. Each value is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range 0 (0x0000) to 65,535 (0xFFFF).
UInt32	4	Represents a 32-bit unsigned integer value in the binary list, as 4 consecutive bytes. Each value is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range 0 (0x00000000) to 4,294,967,295 (0xFFFFFFFF).
UInt64	8	Represents a 64-bit unsigned integer value in the binary list, as 8 consecutive bytes. Each value is encoded in a 'little endian' format, with the least significant byte being stored first. Values can lie in the range 0 (0x0000000000000000) to 18,446,744,073,709,551,615 (0xFFFFFFFFFFFFFFFF).

3.1.2. Advanced Data Types

This section describes the advanced (higher-level) objects that are built on the basic data types…

DataType	Length (Bytes)	Comments
Bearing	8	Represents a navigational 'Bearing' or heading measurement, and always stores the velocity in decimal degrees using a **Double** value. • To extract the number of minutes, multiply the fractional part of the value by 60 • To extract the number of seconds, multiply the fractional part of the above minute's value by 60 again. For example, a value of 5.106513888 represents 5°6'23.45"… • 5 degrees • (5.106513888 – 5)×60 = 6.3908333 minutes • (6.3908333 – 6)×60 = 23.45 seconds For navigational positions, such as latitude and longitude, a positive value represents a northing (N) or easting (E), while negative values represent a southing (S) or westing (W).
Colour	4	Represents a 32-bit ARGB colour value in the binary list. Each **Colour** value occupies 4 sequential bytes, and is encoded/decoded as… • Byte 0: Blue channel component (from 0 to 255 representing 0% to 100%) • Byte 1: Green channel component (from 0 to 255 representing 0% to 100%) • Byte 2: Red component (from 0 to 255 representing 0% to 100%) • Byte 3: Alpha channel component (from 0 to 255 representing 0% (transparent) to 100% (opaque)). As not all aspect of Scanline support transparent colours, this value should normally be 255. A **Colour** object can be interpreted as a **UInt32** value if a numeric representation is required.
Length	8	Represents a 'Length' measurement, and always stores the length in metres using a **Double** value. Value in metres can be converted to other units using the following multipliers… • To millimetres: ×1000.0 • To centimetres: ×100.0 • To decimetres: ×100.0 • To kilometres: ×0.001 • To inches: ×39.37008 • To feet: ×3.28084 • To yards: ×1.09361 • To miles: ×0.000621371192 • To fathoms: ×0.54681 • To nautical miles: ×0.000539956803455723542 1
PositionUtm	26	Represents a Universal-Transverse-Mercator (UTM) position encoded as a structure in the binary list, as 26 consecutive bytes, and is encoded/decoded as follows…

DataType	Length (Bytes)	Comments
		• Byte 0: Latitude Band (or Parallel), an ASCII encoded character **Byte** (capital letter) between 'C' and 'X', in accordance with the UTM specification • Each parallel represents an 8° wide band of latitude, starting at 80°S with "C", and increasing units "X" (with an extra 4°) at 84°N, omitting the letters "I" and "O". Bands "A", "B", "Y" and "Z" are not supported. • Byte 1: Longitude Band (or Meridian), a numerical **Byte** value from 1 to 60 representing 6° wide band of longitude, with zone 1 starting at 180°W, and increasing in an easterly direction. • Bytes 2-9: A **Length** value representing the UTM northing. • Bytes 10-17: A **Length** value representing the UTM easting. • Bytes 18-25: A **Length** value representing the altitude.
Satellite	16	Represents a Satellite structure in the binary list, as 16 consecutive bytes, and is encoded/decoded as follows… • Bytes 0-3: An **Int32** value representing the Satellite Id number. • Bytes 4-7: A **Single** value representing the Azimuth. • Bytes 8-11: A **Single** value representing the Elevation • Bytes 12-15: A **Single** value representing the SNR (Signal to Noise Ratio)
Scanline	20+d	Represents a structure describing a received acoustic echo (or Scanline of display data). • Bytes 0-3: A **Single** value representing the Sonar's "Contrast" control setting when the Scanline was created, in Decibels. • Bytes 4-7: A **Single** value representing the Sonar's "Offset" control setting when the Scanline was created, in Decibels. • Bytes 8-11: A **Single** value representing the Sonar's "Gain" control setting when the Scanline was created, in Decibels. • Bytes 12-15: A **Single** value representing the Sonar's "Range" control setting when the Scanline was created, in metres – and hence the distance to the target represented by the last sample of data collected. • Bytes 16 onwards: A **ByteArray** representing the signal intensities of the acoustic data as it was collected. To maximise storage space, each **Byte** value represents a signal level in Decibels multiplied by two, and should therefore be divided by 2 before use (to get a signal from 0.0dB to 127.5dB).
Velocity	8	Represents a 'Velocity' or speed measurement, and always stores the velocity in kilometres-per-hour using a **Double** value. Value in kilometres-per-hour can be converted to other units using the following multipliers… • To miles-per-hour: ×0.62137 • To metres-per-second: ×0.27778 • To knots: ×0.53996 • To feet-per-second: ×0.91135

3.1.3. Enumerations

An 'Enumeration Type' is an abstract representation of entity that has a specific number of identifiers (such as the suit of a playing card – i.e. Club, Diamond, Heart or Spade).
The Scanline Log File format uses several enumeration types to represent specific values for different interpretations for **Int32** (or **UInt32**) data values (unless stated otherwise), and these are summarised as…

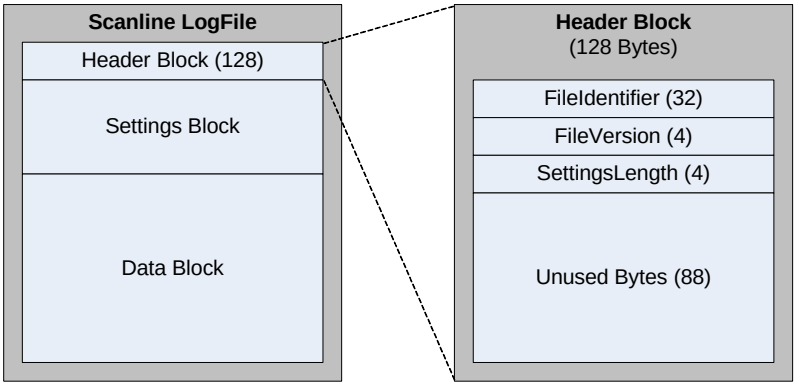
Enumeration Name	Comments & Values
BearingFormat	Used to specify the format a **Bearing** value should be displayed in… • 0 = Unsigned Degrees • 1 = Unsigned Degrees & Minutes • 2 = Unsigned Degrees, Minutes & Seconds • 3 = Signed Degrees • 4 = Signed Degrees & Minutes • 5 = Signed Degrees, Minutes & Seconds
BearingType	Used to specify the type of a **Bearing** value… • 0 = Angular (Heading) • 1 = Latitude • 2 = Longitude
DataMessageType	Used to identify the type and contents of Data Messages… • 0x00 = Unknown • 0x10 = Raw Data Message • 0x11 = Waveform Message (reserved for future use) • 0x20 = Heading Message • 0x21 = Position Message • 0x22 = Velocity Message • 0x23 = Satellite Message • 0x24 = Depth Message • 0x25 = Echosounder Message (reserved for future use) • 0x26 = Sidescan Message • 0x80 = Compiled Message (internal use only)
DeviceType	Used to specify a type of attached hardware sensor device… • 0 = Unknown • 1 = Simulator • 2 = NmeaHeading • 3 = NmeaGps • 4 = NmeaVelocity • 5 = NmeaAltimeter • 6 = Starfish450 • 7 = StarfishGps
LengthUnits	Used to specify the units that a **Length** value represents or should be displayed in…

Enumeration Name	Comments & Values
	• 0 = Millimetres • 1 = Centimetres • 2 = Decimetres • 3 = Metres • 4 = Kilometres • 5 = Inches • 6 = Feet • 7 = Yards • 8 = Miles • 9 = Fathoms • 10 = Nautical Miles
PercentageType	Used to specify how a scalar **Percentage** object should have its value displayed (when a non-magnitude value is shown, the object will compute the percentage based on its value and stored minimum and maximum limits)… • 0 = Magnitude (underlying numerical value) • 1 = Fraction (value from 0 to 1) • 2 = Percent (value from 0 to 100)
PositionType	Used to specify the type of value represented by a **Position** object… • 0 = Universal Transverse Mercator (UTM) • 1 = Latitude/Longitude
RawType	Used to specify the type of Raw Data stored in DataMessage objects • 1 = Binary Data • 2 = Comment String • 3 = NMEA String
SettingType	Used to specify the type of value stored in the **Setting Entry** object, and used the same values as the equivalent Win32 API SDK type (see 'winreg.h' and 'advapi32.dll'). • 0 = Unsupported (treat as an error) • 1 = StringUtf16 (Win32 API = REG_SZ). • 3 = ByteArray (Win32 API = REG_BINARY). • 4 = UInt32 (Win32 API = REG_DWORD). • 7 = StringUtf16Array (Win32 API = REG_MULTI_SZ). • 11 = UInt64 (Win32 API = REG_QWORD).
TemperatureUnits	Used to specify the units that a **Temperature** value represents or should be displayed in… • 0 = Celsius • 1 = Kelvin • 2 = Fahrenheit
VelocityUnits	Used to specify the units that a **Velocity** value represents or should be displayed in…

Enumeration Name	Comments & Values
	• 0 = Kilometres Per Hour • 1 = Miles Per Hour • 2 = Metres Per Second • 3 = Feet Per Second • 4 = Knots

3.2. Header Block

Before attempting the read any other parts of the Log File, the Header Block should first be processed to determine the format of the file and starting-positions of the subsequent blocks within it…



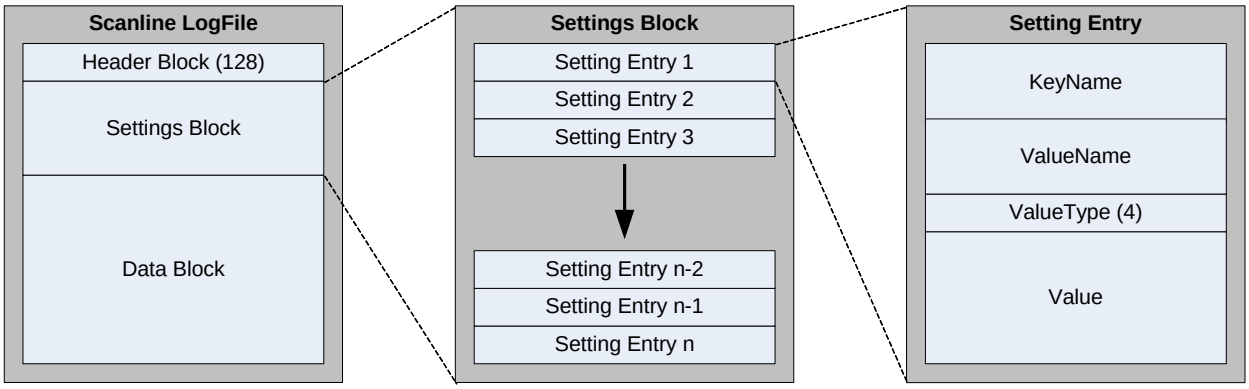
Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+0	**FileIdentifier**	StringFixedAscii	32	The first 32 bytes of the header contains a fixed length ASCII encoded string containing the value "`StarfishScanlineLog`" followed by null (value 0) bytes as padding. Any other value in this string should be treated as an invalid file.
+32	**FileVersion**	Int32	4	This numerical constant value specifies the encoding format used for the file. For Version 2 Log Files, described by this document, this value should be "`2`". Please refer to the specific documentation for other file version numbers – contact StarFish Technical Support for further details.
+36	**SettingsLength**	Int32	4	The length in bytes reserved for the subsequent Settings Block. As the settings section grows, it may be necessary to resize the settings block, and this value will change accordingly.
+40 onwards	**Unused**	Byte	88	Spare unused bytes reserved for future use. Values will be read as 0.

The Log File version number described above does not relate directly to the Scanline application version number, but is intended purely to specify the encoding scheme used for the rest of the file.

3.3. Settings Block

The Settings Block contains a data structure that stores parameterised values in a conceptually similar manor to the "Windows Registry" system, where each setting value is stored under a hierarchical tree structure representing its 'key'. Each Key may be the parent of further 'child' Keys as well as having a collection of values associated with it.

This approach allows each setting to be represented by a common "**SettingEntry**" data structure than can be stored by Scanline into the Windows Registry (i.e. when the 'Live Data Mode' settings are stored and retrieved) or serialised into the Log File binary structure.



To decode the **Settings Block**, the **Header Block** information should first have been decoded and from this the starting byte address in the file-stream for the section can be computed from…

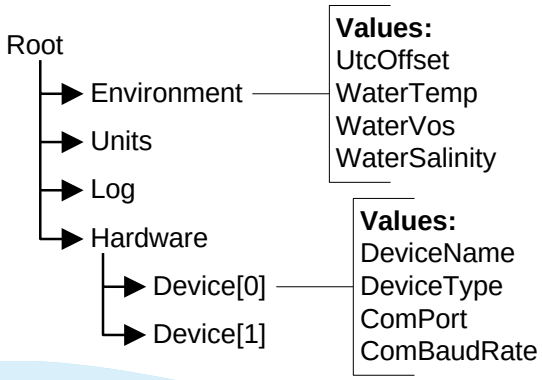
$$\text{Settings Start Address} = \text{Header Length (128 bytes)}$$

Each **SettingEntry** object has a length that can be calculated as it is decoded, the **Settings Block** can then be parsed, and a list/array of settings created.

For storage and descriptive purposes, each Key is said to have a 'Path' that is a string representation of the sequential hierarchy of the Key with the '\' symbol used as a level separator. To give an absolute reference, the value can be appended to the end of the path…

For example…

Key/Value Hierarchy…



String representation…

```
\Environment\UtcOffset
\Environment\WaterTemp
\Environment\WaterVos
\Environment\WaterSalinity
\Units\<values>
\Log\<values>
\Hardware\<values>
\Hardware\Device[0]\DeviceName
\Hardware\Device[0]\DeviceType
\Hardware\Device[0]\ComPort
\Hardware\Device[0]\ComBaudRate
\Hardware\Device[1]\<values>
```

3.3.1. SettingEntry Object

Each setting stored in the **Settings Block** is encoded with the data structure described below.

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+0	**KeyName**	StringAsciiZ	'k'	A null terminated string containing the full Key path (without the last '\' character or value name appended).
+k	**ValueName**	StringAsciiZ	'v'	A null terminated string containing the name of the value.
+(k+v)	**ValueType**	Int32	4	An integer specifying the **DataType** used to store the Setting value in the next field. Only five DataTypes are supported; **StringUtf16**, **ByteArray**, **UInt32**, **StringUtf16Array** and **UInt64**. For values see the "**SettingType**" enumeration definition.
+(k+v+4)	**Value**	*See Comments*	'n'	This field contains the value of the setting, stored in a format defined by the previous **ValueType** field. The length of the field ('n') can be found by looking at the **DataType** encoding definition for the relevant **ValueType**.

Value Types

It is important to note that although many **DataTypes** have been previously defined (see the "Basic Data Types" section on page 6) and are used in the rest of the Log File, the **SettingEntry** data structure only supports a sub-set of 5 of these types as its "**ValueType**".

These **ValueTypes** have a direct one-to-one relationship with the equivalent Win32 API SDK type and allow a **Setting Entry** to be easily stored either in the Log File or the Windows Registry by the Scanline application.

Basic DataType To ValueType Conversions

When a new "Setting Entry" object is created, a suitable **ValueType** will chosen based on **DataType** of the value to be stored. If the **DataType** is not one of the five supported **ValueTypes**, then it will be converted as described in the table below…

If a **DataType** is not listed in the table below, then it is assumed it will be fist converted to one of these base types first…

DataType	Converts To ValueType	Comments
Boolean	UInt32	A Boolean 'True' encodes a UInt32 value of '1', while 'False' encodes as '0'
Byte	UInt32	Unused bytes as filled with 0 before storage.
ByteArray	Not Required	

DataType	Converts To ValueType	Comments
DateTime	StringUtf16	The **DateTime** value is converted to a full string in the format "DD/MM/YYYY hh:mm:ss"… • 'DD' = the numerical day value from 1 to 31. • 'MM' = the numerical month value from 1 to 12. • 'YYYY' = the numerical year. • 'hh' = the hour value from 0 to 23. • 'mm' = the minutes value from 0 to 59. • 'ss' = the seconds value from 0 to 59.
Double	StringUtf16	A floating point number is converted to a string with the maximum number of decimal places required by the its standard for accurate representation.
DoubleArray	ByteArray	See "Data Type" description for details of binary storage format.
Single	StringUtf16	A floating point number is converted to a string with the maximum number of decimal places required by the its standard for accurate representation.
SingleArray	ByteArray	See "Data Type" description for details of binary storage format.
Int16	UInt32	Unused bytes as filled with 0 before storage.
Int32	UInt32	
Int32Array	ByteArray	See "Data Type" description for details of binary storage format.
StringAscii	StringUtf16	Re-coded into the **StringUtf16** format as described in the **DataType** section. All null padding characters are ignored.
StringUtf16	Not Required	
StringAsciiArray	StringUtf16Array	
StringUtf16Array	Not Required	
StringAsciiZ	StringUtf16	Re-coded into the **StringUtf16** format as described in the **DataType** section. All null padding characters are ignored.
UInt16	UInt32	Unused bytes as filled with 0 before storage.
UInt32	Not Required	
UInt64	Not Required	

3.3.2. General Settings

The **Settings Block** may contain the following **SettingEntry** objects required to decode and interpret the **Data Block** records. Other undocumented entries may be present, but these relate to the internal operation of Scanline…

KeyPath	ValueName	ValueType	Comments
\Environment	**UtcOffset**	StringUtf16	Encodes a **Double** value that represents the hours by which local times stored in the rest of the Log File are offset from UTC time.
\Environment	**WaterSalinity**	StringUtf16	Encodes a **Double** value that represent the salinity of the water environment, in parts per thousand. This value is used to calculate to the Velocity-of-Sound value.
\Environment	**WaterTemperature**	StringUtf16	Encodes a **Temperature** value that represents the temperature of the water environment, in degrees Celsius. This value is used to calculate to the Velocity-of-Sound value.
\Environment	**WaterVosDepth**	StringUtf16	Encodes a **Length** value that represents the reference depth (in metres) required by the Velocity-of-sound calculation.
\Environment	**WaterVos**	StringUtf16	Encodes a **Velocity** value that represents the Velocity-of-Sound through water. This value can either be specified manually by a Scanline user, or computed from the Velocity-of-Sound wizard.
\Environment	**WaterVosUnits**	UInt32	Stores a value represented by the "**VelocityUnits**" enumeration type that determines what units should be used when displaying the Velocity-Of-Sound value.
\Log	**Author**	StringUtf16	A string containing the author of the Log File.
\Log	**Comments**	StringUtf16	A string containing any comments that relate to the Log File.
\Log	**Created**	StringUtf16	Encodes a **DateTime** value encoded as a string that specified when the Log File was created.
\Log	**Title**	StringUtf16	A string containing the title of the Log File.
\Units	**Depth**	UInt32	Stores a value represented by the "**LengthUnits**" enumeration type that determines how depth and range measurements (**Length** values) should be displayed.
\Units	**Distance**	UInt32	Stores a value represented by the "**LengthUnits**" enumeration type that determines how distance measurements (**Length** values) should be displayed.

KeyPath	ValueName	ValueType	Comments
\Units	**Heading**	UInt32	Stores a value represented by the "**BearingFormat**" enumeration type that determines how heading measurements should be displayed.
\Units	**PositionType**	UInt32	Stores a value represented by the "**PositionType**" enumeration type that determines how navigational positions should be displayed.
\Units	**PositionLL**	UInt32	Stores a value represented by the "**BearingFormat**" enumeration type that determines how navigational positions should be displayed when the "**PositionType**" settings specifies 'Latitude/Longitude'.
\Units	**Signal**	UInt32	Stores a value represented by the "**PercentageType**" enumeration type that determines how sonar control levels should be displayed.
\Units	**Temperature**	UInt32	Stores a value represented by the "**TemperatureUnits**" enumeration type that determines how temperature measurements should be displayed.
\Units	**Velocity**	UInt32	Stores a value represented by the "**VelocityUnits**" enumeration type that determines how velocity measurements should be displayed.

3.3.3. Hardware Settings

These settings store a list of hardware devices used by Scanline when the Log File was recorded. Each device stored its own specific settings, as well as common settings that apply to each device – only the common settings are discussed here, relating to the following "Data Filter Settings" section and the decoding of Data Messages…

KeyPath	ValueName	ValueType	Comments
\Hardware	**Devices**	UInt32	Value that specifies how many device Keys are defined in the Settings Block.
\Hardware\Device[x]	**DeviceType**	UInt32	Stores a value represented by the "**DeviceType**" enumeration type that determines the class of device, and hence the type of data it can produce. See below for a table summarising the types of data messages produced by each **DeviceType**.
\Hardware\Device[x]	**DeviceId**	UInt32	Encodes a Byte that specifies the **SourceId** that messages created from this device will use.
\Hardware\Device[x]	**DeviceName**	StringUtf16	The name of the device provided by the user.
\Hardware\Device[x]	**DeviceDescription**	StringUtf16	Any description of the device provided by the user.

In the settings above, '[x]' in the Key Path represents the Device number and may lie between 0 and the value specified in the Devices setting.

The table below summarises the types of **DataMessage** that are produced by different types of device.

DeviceType	Produces Data Message with DataMessageType values
Simulator	Depth, Heading, Position, Raw, Satellite, Sidescan, Velocity, Waveform
NmeaHeading	Heading, Raw
NmeaGps	Heading, Position, Raw, Satellite, Velocity
NmeaVelocity	Raw, Velocity
NmeaAltimeter	Depth, Raw
Starfish450	Depth, Sidescan, Waveform
StarfishGps	Heading, Position, Raw, Satellite, Velocity

3.3.4. Data Filter Settings

The Scanline Data Filter module is used when data from a Log File player or recorder is passed to the display plotter objects, to determine which messages are relevant and which should be discarded.
The Data Filter settings entries store the configuration of the Data Filter module for each type of data handled by Scanline (and represented by a corresponding value of the "DataMessageType" enumeration).

When a Log File is played back…

1. The **PayloadType** field of the **DataMessage** is read (and must be a value of the "**DataMessageType**" enumeration).
2. The corresponding Filter entry for the **PayloadType** is retrieved.
3. The **SourceId** of the Filter entry is compared to the **SourceId** in the **DataMessage** (indicating which hardware device generated the message).
4. Only if the two values match or the Filter **SourceId** is set to 0xFF, will the **DataMessage** then be passed to the list of registered display data plotters.

KeyPath	ValueName	ValueType	Comments
\Filter\Filter[<type>]	**SourceId**	UInt32	The Data Source Id number of the hardware sensor device that should have its data used in preference to other data. Values from 0x01 to 0xFE represent the **SourceId** of **DataMessages** that will be shown on the relevant Display Plotter. A value of 0x00 indicates that no source should be used for this type of data. A value of 0xFF indicates that any source should be used for this type of data.

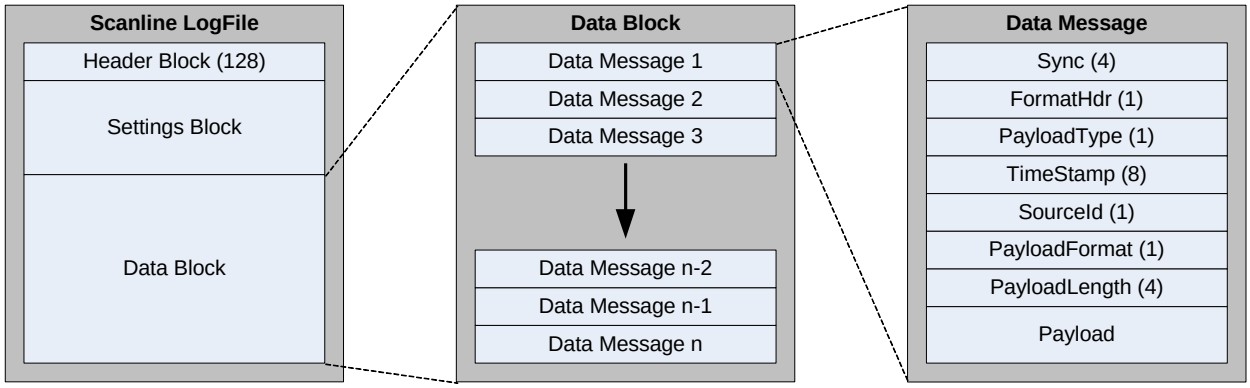
In the Key Path '[<type>]' represents the **name** one of the values in the "**DataMessageType**" enumeration, encoded as a string. For example...

- "\Filter[Depth]",
- "\Filter[Heading]",
- "\Filter[Position]",
- "\Filter[Sidescan]" etc.

3.4. Data Block

The **Data Block** of a Scanline Log File stores each sensor "**DataMessage**" as a binary record sequentially appended to the previous one for the remainder of the file after the Header and Settings blocks.

Each type of **Data Message** has a standard header containing the type, time stamp, length and formatting information that allows the remaining data payload to be decoded.



To decode the data messages, the **Header Block** information should first have been decoded and from this the starting byte address in the file-stream for the data section can be computed from…

$$\text{Data Start Address} = \text{Header Length (128 bytes)} + \text{Settings Length}$$

When opening a Log File, Scanline reads the header information, then from the 'Data Start Address', quickly parses through the Data Block, building an index of the starting position of each data message.
As each data message header has a known fixed length, and contains the length of the subsequent payload, this operation allows any formatting errors in the file to be detected, as the SYNC field of each message would be misaligned, and if the end of the file is encountered before the end of a message is expected, decoding is aborted and the incomplete message ignored.

3.4.1. DataMessage Object

Each data message starts with a standard header that describes its common properties and treatment for subsequent decoding. The formatting of this header takes the form…

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+0	**Sync**	UInt32	4	Synchronisation bytes that should be read as 0xFFFFFFFF. A different value indicates a misalignment error has occurred when parsing through the Data Block.
+4	**MessageFormat**	Byte	1	A numerical value that describes the formatting that should be used for the remainder of the Data Message Header. Currently, this value should always be a '1', indicating the format described here.

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
				Future file versions may require this number to be incremented to describe a header with additional fields.
+5	**PayloadType**	Byte	1	A numerical value that described the type of message and how the payload contents should be decoded. For values, refer to the "**DataMessageType**" enumeration definition.
+6	**TimeStamp**	DateTime	8	The date and time when the message was created by the recording hardware sensor device.
+14	**SourceId**	Byte	1	The Data Source Id number of the hardware sensor device that generated the message. • A value of 0x00 represents an unknown source. • A value of 0xFF is reserved for internal use.
+15	**PayloadFormat**	Byte	1	A numerical value that describes the specific decoding formatting/algorithm that should be applied to the payload data, based on the message **PayloadType** field.
+16	**PayloadLength**	Int32	4	The number of bytes that follow the message header, in the payload section.
+17 onwards	**Payload**	*See Comments*		An array of bytes that describe the information content of the message. This data should be decoded based on the value in the message **PayloadType** and **PayloadFormat** fields.
	Checksum	Byte	1	The last byte of the data message is a checksum field that is the logical XOR of all preceding bytes in the message. By logically XOR'ing all messages bytes, including the checksum, the result should be zero. This can be quickly used to check for an invalid message, should the file be incomplete or damaged.

3.4.2. DataMessage Payloads

Depth Message (PayloadFormat 1)

A depth message is usually produced when a NMEA message is read by an attached sensor device that contains a positional fix (i.e. 'DPT), or computed by a sonar device.

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**Depth**	**Length**	4	A **Length** value (see "Basic Data Types" section) that specifies the depth below the transducer.

Heading Message (PayloadFormat 1)

A heading message is usually produced when a NMEA message is read by an attached sensor device that contains a heading (i.e. 'HDT' or 'RMC' string).

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**Heading**	**Bearing**	8	A **Bearing** value (see "Basic Data Types" section).

Position Message (PayloadFormat 1)

A positional message is usually produced when a NMEA message is read by an attached sensor device that contains a positional fix (i.e. 'RMC' string).
This message format is now obsolete and will not be encountered in V2 Log Files.

Position Message (PayloadFormat 2)

A positional message is usually produced when a NMEA message is read by an attached sensor device that contains a positional fix (i.e. 'GLL' or 'RMC' string).

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**Position**	**PositionUTM**	26	A **PositionUTM** value (see "Basic Data Types" section).

Raw Message (PayloadFormat 1)

A raw-data message that stores either system console/debugging messages or raw NMEA messages that can be subsequently decoded by third-party applications to extract data not provided in other Scanline data messages.

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**RawType**	**Byte**		Stores a value represented by the "**RawType**" enumeration type that determines how the **RawData** field should be interpreted.
+18	**RawData**	*See Comments*		A Binary array that stores the Raw data associated with the message, and should be decoded depending on the value specified in **RawType**… • **Binary Data**, decode this value as a **ByteArray**. • **Comment String** or **NMEA String**, decode this value as a **StringAscii**.

Satellite Message (PayloadFormat 1)

A satellite message contains details about a GPS positional fix and the make-up of the current GPS satellite constellation.

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**FixValid**	**Boolean**	1	A **Boolean** value indicating if the GPS fix is valid.
+18	**Fix3D**	**Boolean**	1	A **Boolean** value indicating if the GPS fix is a 3D one (or 2D if false).
+19	**FixPDOP**	**Single**	4	A Single value describing the Positional Dilution Of Position (PDOP) of the fix.
+23	**FixHDOP**	**Single**	4	A Single value describing the Horizontal Dilution Of Position (HDOP) of the fix.
+27	**FixVDOP**	**Single**	4	A Single value describing the Vertical Dilution Of Position (VDOP) of the fix.
+31	**FixIDs**	**Int32Array**	n+4	An array of Satellite ID's that are used to describe which Satellites have contributed to the GPS fix.
	Satellites	**Int32**	4	A value specifying how many satellites ('s') are currently available in the above constellation.
	SatelliteArray	**Satellite**	(16×s)	An array of 's' **Satellite** object, of length specified in the **Satellites** field above, where each entry describes a single satellite in the constellation.

Sidescan Message (PayloadFormat 1)

A Sidescan message is produced by Sidescan sonar devises…

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**Port**	**Scanline**	p+20	The Port channel data, where 'p' is the number of acoustic data samples collected.
+(p+37)	**Starboard**	**Scanline**	s+20	The Starboard channel data, where 's' is the number of acoustic data samples collected.

Velocity Message (PayloadFormat 1)

A velocity message is usually produced when a NMEA message is read by an attached sensor device that contains a velocity (i.e. 'VTG' or 'RMC' string).

Byte Offset	Field Name	Data Type	Length (Bytes)	Comments
+17	**Velocity**	**Velocity**	8	A **Velocity** value (see "Basic Data Types" section).

3.5. Using Log File Data

3.5.1. Recommended Log File Parsing Method

The sequence of events below is provided as a summary of the previously discussed sections and objects, showing the recommended method for extracting and processing the data stored in a Scanline Log File…

1. Open a suitable read-only file-stream object to access the Log File data on your hard-disk
2. Read the **Header Block** and validate the file is of the correct type and version.
3. Compute the starting addresses for the subsequent Settings and Data blocks.
4. Read the **Settings Block** into a list (or searchable 'dictionary') of **SettingsEntry** objects.
5. Unless you wish to process all data in the Log File, including multiple sources of the same types of data, either…
    - Build a list of Data Source ID's for each **DataMessageType** and select the appropriate source in each **DataMessageType** category for your own Data Filter.
    - Use the Log File Data Filter settings to determine which Data Source ID's for each category of **DataMessageType** was chosen by the user when the Log File was recorded/edited.
6. Move the file-stream read pointer to the start of the **Data Block**.
7. Start parsing through the sequential list of **DataMessage** objects in the Data Block. A class-based approach in an object-oriented programming language can simplify the decoding process.
8. Reject any messages whose **SourceId** field do not match the **SourceId** previously selected in the Data Filter and proceed to the next message.
9. Store the last decoded **DataMessage** objects of type Position, Heading, Velocity and Depth in temporary variables.
10. When decoding a Scanline message, combine this data with the previously stored Position, Heading, Velocity and Depth messages to create an object that allows access to all synchronised data, and is suitable for your display/processing algorithm.
11. When the end of the file-stream is reached, close the stream and tidy up any objects in memory.

3.5.2. Interpreting Sidescan Data

Having extracted Sidescan data messages from the Data Block of the Log File, some post-processing may be required before display of the acoustic data can occur.
As discussed in the **Scanline** definition section, the raw data is stored in bytes from 0 to 255, representing signal levels from 0dB to 127.5dB (a division of 2 is required on each raw data value).

To convert the signal levels into a display colour, compute a normalised target intensity using the stored control values from the Scanline (or current global display settings), and apply the following formula to the raw sample data values…

$$\mathrm{EchoStrength(x)} = \frac{\left(\frac{\mathrm{RawData(x)[dB]}}{2} - \mathrm{Offset[dB]} + \mathrm{Gain[dB]}\right)}{\mathrm{Contrast[dB]}}$$

Clip this value to the interval $0 \le EchoStrength(x) \le 1$ to determine the index position of the colour to use from the display palette.

The acoustic data collection starts when the sonar transmission begins and ends when echoes from targets at the furthest range have been collected. This implies that sound has to travel to and from the target, so all range values used in decoding equations should be doubled.
Based on the stored Scanline "Range" value and the Velocity-Of-Sound used by the Log File, the sampling duration can be computed from…

$$\mathrm{Duration[s]} = \frac{(\mathrm{Range[m]} \times 2)}{\mathrm{VOS[ms\ -1]}}$$

And the sample period is therefore…

$$\mathrm{SamplePeriod[s]} = \frac{\mathrm{Duration[s]}}{\mathrm{Samples}}$$

While the range resolution represented by each sample is…

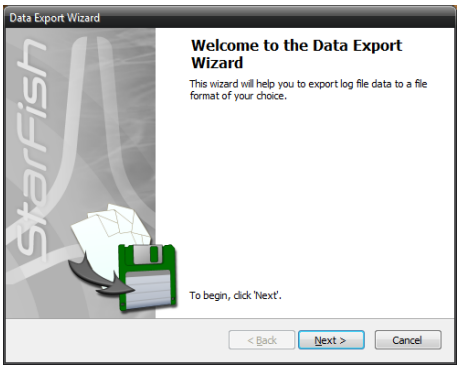
$$\mathrm{Resolution[m]} = \frac{\mathrm{Range[m]}}{\mathrm{Samples}}$$

4. Exporting Data from Scanline

When you have recorded a log-file from a StarFish sidescan sonar, you can use one of the Scanline export converters to output the data in different formats, suitable for use in third party analysis software.

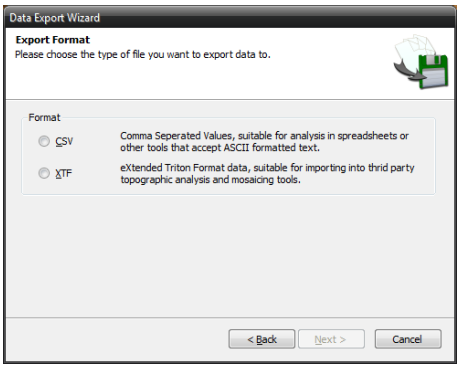
Please refer to the Scanline User Manual for details on installing the Scanline software application.

- Start the "Data Export Wizard" by clicking '*Log : Export Numerical Data*' or using the System button 'Export' menu.



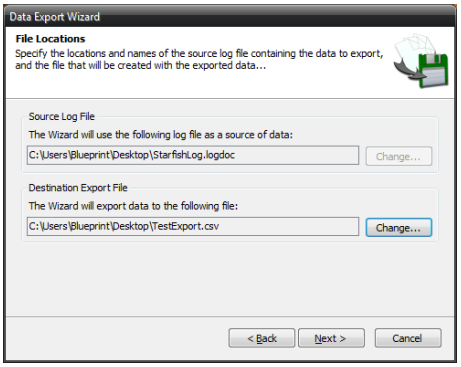
- When the Wizard starts, click "Next" to proceed to the format selection page.

- Choose the type of file you want to export to from…
  - **CSV** – Comma Separated Values. An ASCII text file that can be easily imported into spreadsheets, or custom software data processing.
  - **XTF** – Extended Triton Format. A binary file commonly used for topographic analysis.

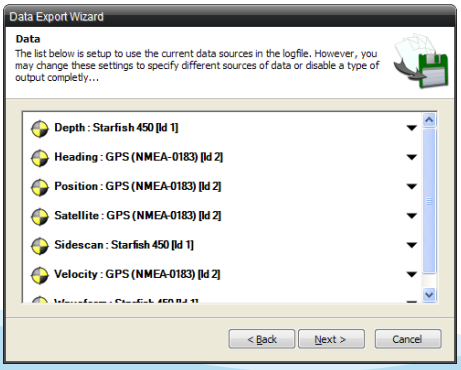


- Click "Next" to proceed after making your selection.

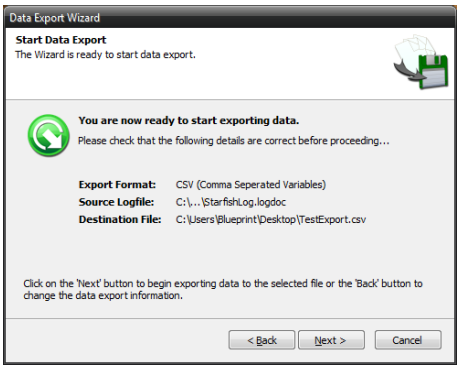
- Choose the source log-file (if one isn't already open), and the file to export the data into, by clicking on the appropriate "Change…" buttons.



- Click "Next" to proceed and if required, adjust the sources of data that will be exported.

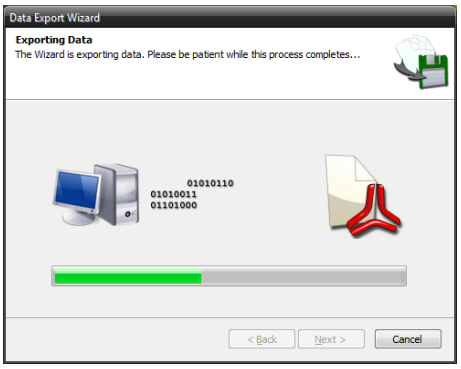


- Click “Next” again to review your choices.



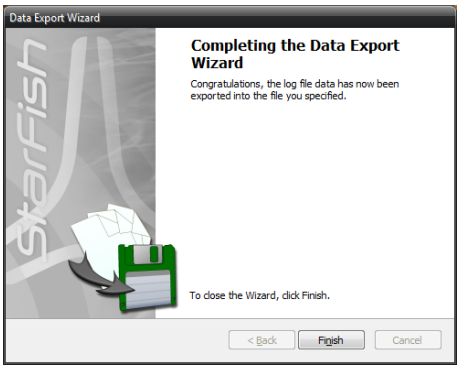
- If you need to make any changes before starting the export, use the “Back” button to go back through the Wizard – otherwise click “Next” to start the export.

- The progress bar will show the state of the export.



- Depending on the size of the source Log File and the format you are exporting to, this process may take several minutes. If you wish to abort click the “Cancel” button, otherwise wait for the wizards “Complete” page to be displayed.

- Click the “Finish” button to close the wizard.



5. XTF File Format

Scanline supports exporting data in the industry standard XTF (eXtended Triton Format) file format – specifically to the revision X24 standard.

Further details of the format can be found on the internet, but some useful resources are (please note the following links are given as examples only, and Blueprint Subsea accepts no liability or responsibility for their content):

- Wikipedia Definition and Resources - *https://en.wikipedia.org/wiki/EXtended_Triton_Format*
- SourceForge Open Source Project for rendering XTF files - *https://sourceforge.net/projects/imagejforxtf/*
- GitHub Python library for reading XTF files - *https://github.com/oysstu/pyxtf*

Please note that technical support provided by Blueprint Subsea for the XTF file format is very limited. The structure definitions below are provided only as means of example only.

5.1. Structures

The following notes should be read alongside the XTF file specification documentation for structure and data type names.

In the following structure definitions, these abbreviations are used:

- M – MANDATORY, must be filled in or set to default value.
- R – RECOMMENDED, set to value if not used or if no value given set to 0.
- O – OPTIONAL, set to value if not used or if no value given set to 0.
- U – UNUSED, reserved for future use.

5.1.1. XtfFileHeader Structure

Each XTF file begins with a file header record. This record is given by the **XTFFILEHEADER** structure and is at least 1024 bytes in length.

It can be larger than 1024 bytes when the total number of channels to be stored in the file is greater than six (the number of **XTFCHANINFO** structures contained in the **XTFFILEHEADER** structure causes the **XTFFILEHEADER** structure to grow larger than 1024 bytes).

In this event, the total size of the file header record grows in increments of 1024 bytes until there is enough room to hold all of the **XTFCHANINFO** structures. Two important elements of the file header are:

- Number of sonar channels
- Number of bathymetry channels

These are used to determine how many **XTFCHANINFO** structures will be in the header record. The **XTFCHANINFO** structures for all of the sonar channels will always precede the structures for the bathymetry channels.

```
struct XtfFileHeader
{
    byte FileFormat;                        //M, 50 for Q-MIPS file format, 51 for Isis format
    byte SystemType;                        //M, Type of system used to record this file. 202=Isis
    char[8] RecordingProgramName;           //M, Example: "Isis"
    char[8] RecordingProgramVersion;
    char[16] SonarName;                     //R, Name of server used to access sonar
    ushort SonarType;                       //M, i.e. K2000=5, DF1000=7, SEABAT=8
    char[64] NoteString;                    //R
    char[64] ThisFileName;                  //R, Name of this file. Example:"LINE12-B.XTF"
    ushort NavUnits;                        //M, 0=METERS or 3=DEGREES
    ushort NumberofSonarChannels;           //M, if > 60, header goes to 8K in size
    ushort NumberOfBathymetryChannels;      //M
    byte NumberOfSnippetChannels;           //M
    byte NumberOfForwardLookArrays;         //M
    ushort NumberOfEchoStrengthChannels;    //M
    byte NumberOfInterferometryChannels;    //M
    byte Reserved1;                         //U
    ushort Reserved2;                       //U
    float ReferencePointHeight;             //O, Height of reference point above water line (m)
    byte[12] ProjectionType;                //U
    byte[10] SpheroidType;                  //U
    int NavigationLatency;                  //O
    float OriginY;                          //U
    float OriginX;                          //U
    float NavOffsetY;                       //O
    float NavOffsetX;                       //O
    float NavOffsetZ;                       //O
    float NavOffsetYaw;                     //O
    float MRUOffsetY;                       //O
    float MRUOffsetX;                       //O
    float MRUOffsetZ;                       //O
    float MRUOffsetYaw;                     //O
    float MRUOffsetPitch;                   //O
    float MRUOffsetRoll;                    //O
    XtfChanInfo[6] ChanInfo;                //M
}
```

5.1.2. XtfChanInfo Structure

Channel information structure (contained in the file header). One-time information describing each channel, 128 bytes long. This is data relating to each channel that will not change during the course of a run.

```
struct XtfChanInfo
{
    byte TypeOfChannel;                     //M, PORT=1, STBD=2, SBOT=3 or BATH=4
    byte SubChannelNumber;                  //O
    ushort CorrectionFlags;                 //O, 1=raw, 2=Corrected
    ushort UniPolar;                        //O, 0=data is bipolar, 1=data is unipolar
    ushort BytesPerSample;                  //M, 1 or 2
    uint Reserved;                          //U
    char[16] ChannelName;                   //O, Text describing channel. i.e., "Port 500"
    float VoltScale;                        //O, How many volts is represented by max sample value. Example: 5.0.
    float Frequency;                        //O, Centre transmit frequency
    float HorizBeamAngle;                   //O, Typically 1 degree or so
    float TiltAngle;                        //O, Typically 30 degrees
    float BeamWidth;                        //O, 3dB beam width, Typically 50 degrees
    float OffsetX;                          //O
    float OffsetY;                          //O
    float OffsetZ;                          //O
    float OffsetYaw;                        //O
    float OffsetPitch;                      //O
    float OffsetRoll;                       //O
    ushort BeamsPerArray;                   //O
    byte SampleFormat;                      //R
    byte[53] ReservedArea2;                 //U
}
```

5.1.3. XtfPingHeader

After the header, data packets may follow in any order. Each packet must contain a packet header, which identifies the type of packet and the number of bytes in the packet. If the software does not recognize the packet type, it can skip forward the given number of bytes to the next packet. Thus, the packets can be thought of as a linked list of data elements which vary in size. To make file searching easier, all data packets must be an even multiple of 64 bytes in length. A data packet typically has the following format:

- Packet Header (usually 256 bytes). Identifies number of channels in this packet and total size of the packet. Each packet begins with a key pattern of bytes, called the "magic number", which can be used to align the data stream to the start of a packet.

And for each channel,

- Channel header (optional, usually 64 bytes)
- Channel data (optional, byte count varies)

Sonar (or Bathy) Ping header - The data here can change from ping to ping but will pertain to all channels that are at the same time as this ping. 256 bytes in length.

```
struct XtfPingHeader
{
    ushort MagicNumber;                 //M, Must be set to 0xFACE (hex)
    byte HeaderType;                    //M, 0 = XTF_HEADER_SONAR (sidescan data)
    byte SubChannelNumber;              //M
    ushort NumChansToFollow;            //M, If HeaderType is sonar, number of channels to follow
    uint Reserved1;                     //U
    uint NumBytesThisRecord;            //M, Total byte count for this ping including this ping header
    ushort Year;                        //M, Computer date when this record was saved
    byte Month;                         //M
    byte Day;                           //M
    byte Hour;                          //M, Computer time when this record was saved
    byte Minute;                        //M
    byte Second;                        //M
    byte HSeconds;                      //M, hundredths of seconds (0-99)
    ushort JulianDay;                   //O, Number of days since January 1
    uint EventNumber;                   //O, Last logged event number
    uint PingNumber;                    //M, Counts consecutively from 0 and increments for each update.
    float SoundVelocity;                //M, m/s, Round trip, defaults to 750.
    float OceanTide;                    //O, Ocean tide in meters.
    uint Reserved2;                     //U
    float ConductivityFreq;             //O, Conductivity frequency in Hz
    float TemperatureFreq;              //O, Temperature frequency in Hz
    float PressureFreq;                 //O, Pressure frequency in Hz
    float PressureTemp;                 //O, Pressure Temperature (Degrees C)
    float Conductivity;                 //O, Conductivity in S/m
    float WaterTemperature;             //O, Water temperature in C
    float Pressure;                     //O, Water pressure in psia
    float ComputedSoundVelocity;        //O
    float MagX;                         //O, X-axis magnetometer data, mgauss
    float MagY;                         //O, Y-axis magnetometer data, mgauss
    float MagZ;                         //O, Z-axis magnetometer data, mgauss
    float AuxVal1;                      //O
    float AuxVal2;                      //O
    float AuxVal3;                      //O
    float AuxVal4;                      //O
    float AuxVal5;                      //O
    float AuxVal6;                      //O
    float SpeedLog;                     //O, Speed log sensor on towfish - knots. This isn't fish speed!
    float Turbidity;                    //O, turbidity sensor (0 to  volts) stored times 10000
    float ShipSpeed;                    //O, Speed of ship in knots
    float ShipGyro;                     //O, Ship gyro in degrees
    double ShipYcoordinate;             //O, Ship latitude or northing
    double ShipXcoordinate;             //O, Ship longitude or easting
    ushort ShipAltitude;                //O, Decimeters (meters*10, stored only)
    ushort ShipDepth;                   //O, Decimeters (meters*10, stored only)
    byte FixTimeHour;                   //R, Hour of most recent nav update
    byte FixTimeMinute;                 //R, Minute of most recent nav update
    byte FixTimeSecond;                 //R, Second of most recent nav update
    byte FixTimeHsecond;                //R, HSeconds of most recent nav update
    float SensorSpeed;                  //R, Speed of the in knots.
    float KP;                           //O, Kilometers Pipe
    double SensorYcoordinate;           //R, Sensor latitude or northing
    double SensorXcoordinate;           //R, Sensor longitude or easting
    ushort SonarStatus;                 //O
    ushort RangeToFish;                 //O, Slant range to fish * 10 (Not currently used)
    ushort BearingToFish;               //O, Bearing to towfish from ship * 100 (Not currently used)
    ushort CableOut;                    //O, Amount of cable payed out in meters (Not currently used)
    float Layback;                      //O, Distance over ground from ship to fish.
    float CableTension;                 //O, Cable tension from serial port. Stored only.
    float SensorDepth;                  //R, Distance from sea surface to sensor.
    float SensorPrimaryAltitude;        //R, Distance from towfish to the sea floor.
    float SensorAuxAltitude;            //O, Auxillary altitude
    float SensorPitch;                  //R, Pitch in degrees (positive=nose up)
```

```
    float SensorRoll;                   //R, Roll in degrees (positive=roll to stbd)
    float SensorHeading;                //R, Fish heading in degrees
    float Heave;                        //O, Sensor heave at start of ping. Positive value means sensor moved up.
    float Yaw;                          //O, Sensor yaw. Positive means turn to right.
    uint AttitudeTimeTag;               //R, milliseconds - used to coordinate with millisecond time value
                                        //   in Attitude packet.
    float DOT;                          //O, Distance Off Track
    uint NavFixMilliseconds;            //R, millisecond clock value when nav received
    byte ComputerClockHour;             //O
    byte ComputerClockMinute;           //O
    byte ComputerClockSecond;           //O
    byte ComputerClockHsec;             //O
    short FishPositionDeltaX;           //O, Stored as meters*3.0, supporting  10000.0m
    short FishPositionDeltaY;           //O, X,Y offsets can be used instead of logged layback.
    byte FishPositionErrorCode;         //O, Error code for FishPosition delta x,y
    byte[11] ReservedSpace2;            //U
}
```

5.1.4. XtfPingChanHeader Structure

Ping Channel header. This is data that can be unique to each channel from ping-to-ping and is stored at the front of each channel of sonar data. 64 bytes in length.

```
    struct XtfPingChanHeader
    {
        ushort ChannelNumber;           //M, 0=port(lf) 1=stbd(lf) 2=port(hf) 3=stbd(hf)
        ushort DownsampleMethod;        //O, 2=MAX, 4=RMS
        float SlantRange;               //M, Slant range of the data in meters
        float GroundRange;              //O, Ground range of the data in meters (SlantRange^2 - Altitude^2)
        float TimeDelay;                //O, Amount of time (in seconds) to the start of recorded data, almost always 0.0
        float TimeDuration;             //R, Amount of time (in seconds) recorded
        float SecondsPerPing;           //R, Amount of time (in seconds) from ping to ping
        ushort ProcessingFlags;         //O, 4=TVG, 8=BAC&GAC, 16=Filter, etc... almost always 0
        ushort Frequency;               //R, Centre transmit frequency for this channel
        ushort InitialGainCode;         //O, Settings as transmitted by sonar
        ushort GainCode;                //O
        ushort BandWidth;               //O
        uint ContactNumber;             //U
        ushort ContactClassification;   //U
        byte ContactSubNumber;          //U
        byte ContactType;               //U
        uint NumSamples;                //M, Number of samples that will follow this structure. The
                                        //   number of bytes will be this value multiplied by the
                                        //   number of bytes per sample (given in the file header)
        ushort MillivoltScale;          //O
        float ContactTimeOffTrack;      //U
        byte ContactCloseNumber;        //U
        byte Reserved2;                 //U
        float FixedVSOP;                //O, Fixed along-track size of each ping, stored in cm.
        short Weight;                   //O, Weighting factor passed by some sonars
        byte[4] ReservedSpace;          //U
}
```

7. CSV File Format

The "Comma Separated Values" file exported from Scanline contains one data message per line, where each field is separated by a comma and the line is terminated by a Carriage Return and Line Feed character.

During the export process Scanline Log Files are processed and each recorded message type is sequentially converted into an ASCII text CSV equivalent line.
The first field of each line contains a string describing the type of data that follows on that line, allowing different parsers to be written to handle each one, or filtering/sorting to be performed in applications such as a spreadsheet.

The following sections describe how to parse different message types…

7.1. Depth Data

Field	Description	Comments
1	Literal String "**DPT**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3	Depth	The depth in metres.

7.2. Heading Data

Field	Description	Comments
1	Literal String "**HDG**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3	Heading	The 'true' heading in degrees.

7.3. Position Data

Field	Description	Comments
1	Literal String "**POS**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3	Latitude	The latitude of the positional fix in decimal degrees. A positive value represents a 'northing' fix.
4	Longitude	The longitude of the positional fix in decimal degrees. A positiive value represents an 'easting' fix.
5	Zone	The UTM Grid zone identifier.
6	Northing	The UTM northing value (in metres).
7	Easting	The UTM easting value (in metres).

8	Altitude	The altitude of the positional fix, if available from the source (in metres).

7.4. Sidescan Data

Field	Description	Comments
1	Literal String "**SSS**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3	Literal String "**P**"	Identifies the following data is from the Port Channel scanline
4	Contrast	The contrast value recorded with the scanline - in decibels. The contrast represents the span of the palette that the data values will be shown over (before the gain and offset are applied).
5	Offset	The offset value recorded with the scanline - in decibels.
6	Gain	The gain value recorded with the scanline - in decibels.
7	Range	The range the scanline data represents - in metres.
8	Samples	The number of data points in the scanline, that will follow this value. The range resolution of each data sample can be found by dividing the range by the number of samples.
9 to (9+Samples-1)	Data	An array of data points representing each sample on the scanline. Values are in raw decibels from the sonar receiver. For displaying, the gain and offset can be added to each value, then the contrast used when mapping to a display palette.
…	Literal String "**S**"	Identifies the following data is from the Starboard Channel scanline
…	…	The above data structure repeats with the contrast, offset, gain, range, samples and data array.

7.5. Velocity Data

Field	Description	Comments
1	Literal String "**VEL**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3	Velocity	The velocity in kilometres-per-hour.

7.6. Waveform Data

Field	Description	Comments
1	Literal String "**WAV**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.

3	Trace Color	The color of the waveform trace, expressed as a 32-bit ARGB integer (0x<aa><rr><gg><bb>) where <aa> is the alpha component and <rr>, <gg> and <bb> are the red, green and blue colour components.
4	Sample Period	The timer period (in seconds) of each sample
5	YMax	The maximum value that should be shown on the Y-axis.
6	YMin	The minimum value that should be shown on the Y-axis.
7	Title	The title of the waveform
8	Samples	The number of data samples in the waveform
9 to (9+Samples-1)	Data	An array of data points, representing the magnitude of each sample in the waveform.

7.7. Compiled Data

A compiled data message differs slightly from previous message types in that it contains the last preceding content of each message type all sequentially appended into a single CSV line.

Messages are appended to the CSV record in the order Position, Heading, Velocity, Depth, Sidescan – other record types are not added. If a particular type of message is not being collected, then that message content is omitted from the compiled data record and an empty CSV field is inserted (i.e. two field separating commas next to each other with not content between them).

This message type has been added to allow easy processing of sidescan data where the position and heading of each scanline is bound to the sonar data – applications such as a spreadsheet can be used to filter/remove all non-CPL lines from the CSV file, and just parse the CPL records to extract georeferenced sidescan data.

Once a CPL message has been identified with the following initial fields shown in the table below, the subsequent fields are decoded using the previously described formatting – i.e. when a field containing SSS is encountered, the next fields are decoded as Sidescan data etc.

Field	Description	Comments
1	Literal String "**CPL**"	Identifies the format of the data on the rest of the line.
2	Time Stamp	The time and date when the data message was entered into the log-file.
3 onwards	Position Message	First field in group starts with “POS”, or empty field if not used.
…	Heading Message	First field in group starts with “HDG”, or empty field if not used.
…	Velocity Message	First field in group starts with “VEL”, or empty field if not used.
…	Depth Message	First field in group starts with “DPT”, or empty field if not used.
…	Sidescan Message	First field in group starts with “SSS”, or empty field if not used.

Distributor…

P/N: BP01588-1

UM-125-P01588-01
July 2018