Commonwealth Scientific and Industrial Research Organisation
Materials Science and Engineering

Maia XRF Imaging System
Technical Note

Processing Maia Photon Record Data

Robin Kirkham
14 January 2016

1 Introduction

Photon data from the Maia detector system, and related data acquisitions systems, are written into block-structured byte-oriented binary files, by the CSIRO 'binary logger' daemon (`csiro-blogd`). This is an application-agnostic network-to-disc logger whose purpose is to write arbitrary binary data from multiple *writer* network clients, in receipt order, to a single sequence of disc files, with high efficiency.

The writer clients include the Maia detector, various other radiation detector data acquisition systems, plus various simpler clients that log non-detector data, such as samples of EPICS process variables, or experimental metadata. The idea is that the resulting data files should contain *all* the data pertaining to the experimental run.

During data acquisition, different programs known as *reader* clients can also connect with the binary logger daemon. They can request and receive a subset of the blocks being logged in real time. This is intended for real-time data display, and works well where a useful running analysis can be developed from a sub-sampling of the data stream. This is true in the case of many photon data streams, such as from Maia. The on-line interpretation of the binary data in such clients is essentially identical to off-line interpretation from files.

The `csiro-blogd` program was used in the development of the prototype Maia detectors with the intention it would ultimately be replaced by something better, preferably using a more widely used data storage standard. This has not yet happened, and Maia continues to use the binary logger system.

For dissemination or archival storage, the binary data files could of course be expanded to a portable, self-describing format such as HDF5, netCDF or even XML. Presently, a filter to translate binary data from the binary logger container format to a HDF5 container exists but is not in wide use.

While the binary logger container format (blocks and tags) has remained constant over many years, the format of the Maia data itself has evolved over time. In early prototype Maia detectors, very little processing of photon data occurred in the detector system and programs that read the data files had to do quite a lot of work to, for instance, linearise and gain correct each photon event before further processing. The block payloads rather closely resembled the internal FPGA record structures. Later Maia systems not only perform these corrections before logging, they can also deconvolve and accumulate the events to actual elemental concentrations per spatial pixel, so the binary logger files contain fully processed data alongside raw data.

This note first describes the format of the binary logger files and blocks in general, then moves on to describing the key block types specific to photon data from Maia and other acquisition systems.

2 General

2.1 Blocks and Tags

The smallest unit of data the binary logger daemon itself deals with is a *block*. It has variable length. A block comprises a *header* (either 6 or 32 bytes), and a *payload* of 0 to 65535 bytes. The 6-byte *short* form of the header contains the block *tag* (or block type), and the *length* of the subsequent payload in bytes. This form is used only for clients writing data to the logger. The 32-byte *long* form adds time-stamp and sequence information to the tag and length contained in the short header. This form is the one logged to disc and supplied to network clients.

The logger does not interpret the payload, except for certain special logger control blocks. The format of the experimental data in the payload is defined by the application programs that write and read it. Each format is assigned a unique *tag* number. As experimental systems evolve, new tags will be assigned and others will fall out of use; old numbers are never re-used. Tags, in their symbolic form, often have a version number to distinguish improved tags over superseded versions with a similar purpose.

The tag numbers are declared in a text file called `TAGS` in the `hsi/blog` package sources in the CSIRO Subversion server. Presently over 50 tags are declared out of a possible 65536 (see Appendix A). The `TAGS` file is automatically processed into lists and lookups for different programming languages, and appears in Appendix C of this document. However, this is only so that symbolic names can be used for representing tags in code and data printouts, and is not essential to the operation of the logger or its clients.

Programs that read the data files only need interpret those blocks that are of interest to it; others may be skipped over. Network reader clients will naturally only request the blocks they are interested in.

2.2 Runs and Segments

The normal unit of data the logger deals with is a *run*. In the Maia system a run typically means a single scan over a single sample. Each run is automatically numbered sequentially by the logger *<runno>*.

In a given installation or facility (typically a beamline), run numbers are should never be re-used, in order prevent accidental overwriting during manual file manipulation. The logger preserves the run and segment numbers, and other state, across invocations.

During a run, the logger concatenates the blocks in order of receipt into files called *segments*. The logger generally starts a new segment file when the next block would cause it to exceed a particular size, typically 100 M bytes. Segments may also be used to delineate phases of a run, although this is not recommended (it is better to mark phases with an appropriate application block type). The segments are numbered sequentially *<segno>*, and the run number and segment numbers are used to name the file *<runno>*.*<segno>*. All segments for a given run are written to a run directory called *<runno>*.

2.3 Binary Formats

In the following descriptions of the binary formats of the header and payload, the following shorthand notations is used:

Type	Bytes	Description
`uint8`	1	unsigned integer, 8-bit
`int8`	1	signed integer, 8-bit
`uint16`	2	unsigned integer, 16-bit
`int16`	2	signed integer, 16-bit
`uint32`	4	unsigned integer, 32-bit
`int32`	4	signed integer, 32-bit
`uint64`	8	unsigned integer, 64-bit
`int64`	8	signed integer, 64-bit
`string0`	1+	UTF-8 string (null terminated)
`stringN`	N	fixed length UTF-8 string (null padded)
`float`	4	IEEE format floating-point
`double`	8	IEEE long format floating-point

All header data is in big-endian format: that is, the most-significant bytes in multi-byte quantities come first. This is mainly because the HYMOD PowerPC CPU runs in big-endian mode. This also matches typical Internet Protocol ordering, but is the reverse of little-endian Intel processor ordering. With care it is possible to code reading routines that are efficient but portable to any processor type: the key concept is to use byte-by-byte processing, and construct multi-byte quantities using arithmetic operators. Some hints on how to do this appear later.

2.4 Short Header Format

The short, 6-byte header used only by writer clients to send data to the logger:

`uint8`	Literal `0xaa`
`uint16`	*<tag>*
`uint8`	Literal `0xbb`
`uint16`	*<len>*

<tag> is the 16-bit (2-byte) tag number, in big-endian (most significant byte first) format. The literal marker bytes `0xaa` and `0xbb` are used to check synchronisation between client and server. The header is immediately followed by the payload of exactly *<len>* bytes. Note that *<len>* ranges from zero to 65535 (16 bits). The format of the data in the payload is determined by *<tag>*. The payload is followed immediately by the header of the next block, or the end of the file.

2.5 Long Header Format

The long, 32-byte header is found in data files, and is also received by reader clients:

`uint8`	Literal `0xaa`
`uint16`	*<tag>*
`uint8`	Literal `0xbb`
`uint16`	*<len>*
`uint16`	*<prevlen>*
`uint32`	*<runseqno>*
`uint32`	*<tagseqno>*
`uint32`	*<tv_sec>*
`uint32`	*<tv_usec>*
`uint32`	*<client>*
`uint32`	Spare

The first 6 bytes are the same as the short header. The additional fields are 2-byte or 4-byte values. *<prevlen>* is the length of the payload of the previous block, and can allow for indexing of a file in reverse, if necessary. *<runseqno>* is the sequence number of the block in the current run. It will increment once for each block, continuing across segments (files). Similarly, *<tagseqno>* increments

once for each block of that tag type. Network clients use these to determine how many blocks they are not receiving.

<tv_sec> and *<tv_usec>* are the UNIX system time when transmission of the block to the server was completed. The first is the number of seconds since 1 January 1970 UTC, and the other is the number of microseconds since that second. The timezone is recorded in the file identity block (see later), so the wall clock time of block receipt can be reconstructed.

The *<client>* field is the serial number of the client that wrote the block. It can be used to tell whether blocks were written by the same client or not. The use of this field is depreciated, and may be changed. There is one spare field in the header.

As with the short header, the payload of *<len>* bytes follows the header immediately. If *<len>* is zero, the next header follows immediately.

3 Payload Formats

The block payload formats are described generally in this section; refer to the Appendix for the more formal description of the format.

3.1 Formats generated by the Logger

Certain block types are generated by the binary logger. Blocks of these types sent to the logger by clients are ignored.

Block types 1 (`id`) and 28 (`id_2`) are identity blocks. The logger writes one of these as the first block in *every* segment file. They contain the run and segment numbers and timezone of the logger. The older version contained experimental and personnel information, but was rarely filled out correctly. The newer version contains information on the logger version, server, facility name, and so on; experimental information (sample name, beamline parameters not captured through `monitor` blocks, etc) are better logged using `comment` blocks, or a specific application-defined block.

Block types 5 (`summary`) 32 (`summary_2`), 53 (`summary_3`) or 56 (`summary_4`) are generated and logged every few seconds, and contain information about the rate of logging and accumulated size of a run. The actual block type generated depends on the version of `csiro-blogd`. They are intended only for screen display by a reader client, and are not useful when analysing data files. The logger can be configured to omit them from the data file.

3.2 Formats affecting Logger operation

Certain block tags have special effect on the binary logger.

Chief among these are blocks 2 (`newrun`), 3 (`newseg`) and 29 (`endrun`). The first starts a new run, and the second starts a new segment file in the current run (and starts a new run if there is no current run). The last ends the current run, causing the logger to begin bypassing data. Early versions of the logger either absorbed these blocks, or placed them at near the start of the segment files. Later versions place them more correctly at the ends of segments.

Blocks 7 (`sendnext`), 23 (`sendprev`) and 23 (`sendprevornext`) are used by network reader clients to request the logger to send back one block of selected type. These blocks never appear in data files. The payload of these blocks is a set of 16-bit block tags, the number implied by the block length. In the case of `sendnext`, the next block written to the logger that is a member of the set will be copied to the requesting client. Tag `sendprev` sends a cached block instead. Tag `sendprevornext` sends a cached block if available, or the next member block that is written.

Blocks 51 (`setproject`) and 33 (`setgroup`) set the next run 'project' and 'group'. The logger can be configured to use these to construct the file path used to store its data files.

Block 0 (`ignore`) is for logger stress testing purposes.

3.3 Generic Blocks

Block 6 (`comment`) allows a free-form string to be logged. It may contain any characters except nul (0), except for the last character, which should be a nul (C string syntax). UTF-8 encoding should be used for non-ASCII characters.

Any information may be logged using a comment block. It is typically used to record sample information and experimental conditions. The `pddc`, `da`, `dali` and `miro` programs controlling the prototype Maia detectors also log their command-line directives using this block, thereby capturing detector configuration changes during runs.

Block 26 (`monitor`) is a more structured text block used to record sampled values of relatively slow-moving variables, in a machine and human readable form. In the Maia system, the variables are typically EPICS PVs representing incident beam flux, to allow subsequent ring current decay correction to fluorescence yields. These are logged on a constant-time basis by the `blogepics` writer client program.

3.4 Maia Blocks

As noted previously, the data produced by the Maia detector have evolved over time. New block types have been introduced and older ones retired. In this section the key blocks produced by recent Maia (`kandinski`) systems are described. Key blocks from older systems (`pddc`, `da`, `dali` and `miro`) are described in Appendix B.

Table 1 Events in `maia_events_1` blocks:

ET (Photon Energy/Time) event:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Addr							Time								Energy															

SE (Stage Encoder) event:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Axis			Encoder value (signed)																											

Axis bit-field may only be 00 (usually indicates x), 01 (y) or 10 (z).

PA (Pixel Address) event:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Axis	Pixel value (signed)																											

Axis bit-field may only be 00 (usually indicates x), 01 (y) or 10 (z).

TF (Time/Flux) counter events:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	Sel	Counter value (unsigned)																									

The selector bit-field may only be 00 (indicates block time in 100 ns units), 01 (flux counter 0) or 10 (flux counter 1).

UU Reserved for future use:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	Don't care																								

3.4.1 Event Blocks

Block 34 (`maia_events_1`) contains the basic photon (*ET*), pixel address (*PA*), and stage event (*SE*) 'list mode' data. In addition, these blocks contain incident flux counter and block time (*TF*) events, although these latter quantities apply collectively to the whole block and are not strictly 'events'.

The Maia detector logs a `maia_events_1` block when:

1. Pixel events are enabled and the pixel address changes, or
2. The number of events in the block payload is approaching the maximum (16383), or
3. The block has taken longer than a maximum time (usually about 250 ms) to accumulate.

The payload of a `maia_events_1` block comprises only 32-bit, big-endian bit-packed words, one per event. The events are identified by a hierarchical tag in the most-significant bits of the word, with the data occupying the lower order bits. These event tags should not be confused with the block tag numbers. The hierarchical tags may be extended in future using presently reserved (*UU*) tag patterns. The format of all the current event words is shown in Table 1.

ET Events. Most of the events in the payload of `maia_events_1` blocks will be photon *ET* events, which can be identified by a 0 in bit 31 (all other events have a 1 in that location).

The data comprise the address (detector channel number), and digitised time and pulse-height quantities captured from the SCEPTER chip. The former datum ranges from 0–31, 0–95 or 0–383, depending on the actual Maia detector in use. The latter two data are in ADC units. If a linearisation and gain-trim table had been loaded in the Maia system and were are enabled when the data were written, the values would have been normalised and will relate to real time and energy units.

SE Events. An SE event can be generated when any of the encoders on the three stage axes change. The data comprises two bits indicating the axis (0, 1 or 2), and the new encoder value. *SE* events are generally disabled.

PA Events. The payload is guaranteed to commence with exactly three *PA* events, indicating the spatial pixel the events in this block belong to. The *PA* events will occur in order, with axis 0 first. There will no further *PA* events in the block.

Note that while a particular `maia_events_1` block can belong to only one spatial pixel, a spatial pixel can and generally will have many `maia_events_1` blocks belonging to it, and these will obviously have the same *PA* values. These blocks will typically be consecutive in the data file, although file reading software should not rely on that.

Axes 0 and 1 are generally the x and y sample scanning axes, and axis 2 is generally unused. Ultimately axis 2 will be used for θ in tomographic data sets, or perhaps monochromator E in quick-EXAFS experiments. Axis 2 is presently ignored by GeoPIXE.

The two-dimensional (x, y) pixel array should range from (0, 0), the bottom left corner of the scan, to $(w-1, h-1)$ for the top corner, where w is the width and h the height of the scan in pixels. Pixel addresses may fall outside the formal scan area (and be negative). Imaging applications typically clip these pixels from final images.

The size of the pixels (pixel pitch) may differ in x and y, although it is frequently the same. The pitch and overall scan size (and other scan parameters) are recorded in block 42 (`maia_scan_info_1`) or 47 (`maia_scan_info_2`, see later).

TF Events. Time/flux counter events were added to the `maia_events_1` blocks in late 2011. They are not strictly 'events', as their values relate to the whole block that contains them, and there will only be one of each in a given block. They are usually located at the beginning of the block, immediately after the *PA* events, but this is only guaranteed if DMA readout mode is in use in `kandinski` (generally the case).

The data comprise two selector bits identifying the kind of event, and a 25-bit counter value. Selector 0 indicates the datum is block time, which is the real time the block was accumulated over (from the initial pixel event to the last photon). The units are 100 ns clock ticks (10 MHz reference clock).

Selectors 1 and 2 indicate flux counters 0 and 1 respectively. These count rising-edge transitions of the FC0 and FC1 digital signal inputs on the Maia processing subsystem (HYMOD), which are generally connected to voltage-to-frequency convertors driven from the incident flux ion-chamber transconductance amplifiers. The counters give accurate integrated flux over the same time frame as the photon *ET* events in the block. The counters do not wrap around, so a value of `0x1ffffff` indicates overflow.

3.4.2 Accumulator Blocks

In principle the 'raw' event data in the `maia_events_1` blocks saved in the binary logger files is, with some metadata, sufficient to generate elemental images, spectra, or other output. However, the Maia system can generate activity snapshots, spectra, deadtime data, and deconvolved elemental concentrations in its FPGA, and log these data in parallel with, or instead of, the raw event data. The blocks that contain these are known as accumulator blocks.

Accumulator blocks are written by the Maia processing subsystem when a configured *trigger* occurs. The triggers can be generated by pixel address changes, timers, software events, or by external hardware.

Accumulator sub-header. The payloads of all the accumulator blocks start with a 9-word sub-header, described in Table 2, which is followed by zero or more 32- or 64-bit data words, depending on the accumulator type. Many of the fields in the sub-header can be ignored. Only the key ones are described here.

The three pixel addresses in the sub-header is the pixel address captured when the trigger occurred. If the logging of the accumulator block was triggered by the pixel address change trigger, then the data accumulated by the accumulator relates to that pixel alone, otherwise the data may relate to more than one

Table 2 Maia accumulator block sub-header:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								Pixel address axis 0 (signed)																							
0								Pixel address axis 1 (signed)																							
0								Pixel address axis 2 (signed)																							
DT	DP	Channel group mask														Trigger source								Index				Accumulator			
MTC								0 (reserved)														O	E	Duration high bits							
Duration counter low bits (unsigned)																															
Flux counter 0 (unsigned)																															
Flux counter 1 (unsigned)																															
Accumulator word count (unsigned)																															

In the fourth word, DT indicates throttled photon events are omitted, and DP that piled-up photon events are omitted. In the fifth word, MTC stands for Missed Trigger Count, O for overflow, and E for error. The accumulation duration counter is 40 bits, of which the eight high-order bits are in the fifth word and the low order bits in the sixth word.

pixel. Note that the pixel addresses do not carry the event tag bits used in the `maia_event_1` block.

The header also includes counters for the flux 0 and flux 1 inputs, and a duration time counter, which increments in 100 ns (10 MHz) units. These counters have more bits than those used for the event blocks, because the accumulation time may be much longer than an event block. The maximum accumulation time is $(2^{40} - 1) \times 100 \times 10^{-9} = 109,951$ seconds, or about 30.5 hours. The counters do not wrap around, so overflows are indicated by the counters having maximal values.

Accumulator data. The last word in the sub-header indicates the number of accumulator data words that follow. The format of these depends on the accumulator.

The key accumulator block types are block 39 (`maia_activity_accum_1`), a histogram of 32-bit photon counts against channel number, and channel group; block 40 (`maia_energy_spectrum_accum_1`), a histogram (spectrum) of 32-bit photon counts against energy; and block 35 (`maia_da_spectrum_accum_1`), a vector of 28.24-bit fixed-point values representing elemental concentrations.

Detailed description of all the accumulator blocks is beyond the scope of this note.

3.4.3 Metadata Blocks

The key scan metadata block is known as the scan record. This exists in two versions, block 42 (`maia_scan_info_1`) or block 47 (`maia_scan_info_2`), in data written after some point in 2012.

The scan record is supposed to precede a sample scan, furnishing information about it, including the dimensions in pixels (in all three axes), the order the axes will be scanned, the positional origin of the scan and the size of the pixels. Version 2 of the block includes the physical units of the length measurements; in version 1 these are assumed to be millimetres. These scan data are used by GeoPIXE and other analysis programs.

The scan record also includes an 'information' string used for the identity of the sample being scanned, and related information. A number of structured and unstructured sub-formats have been used for this string in attempts to add important values (such as synchrotron beam energy) to the metadata. An improved version of this block is expected before long.

The other main metadata blocks are block 45 (`var_list_1`) and block 46 (`var_value_1`), and the 'new' metadata block block 55 (`metadata`).

Block 46 is intended to be for dumps of *all* Maia control variable settings, in the same string format as the Maia initialisation files, and also the `varsh` control program. The idea is that these values can be extracted from the blog files of a run, and the detector system later set up in exactly the same way, or, at least, the settings on a particular run examined.

Block 45 is intended as dump of declarations of variables that will appear in subsequent block 46s. Until late 2014 only a small subset of Maia control variables were dumped using these blocks, but now the complete detector state is captured in the data files.

A new block 55 (`metadata`) is now defined. This has a plain-text, key-value format and will ultimately replace a number of the other metadata blocks. Key names will be drawn from a defined list but the new scheme is more easily extended than the binary blocks that will be replaced.

3.5 Beamline Data Acquisition Unit Blocks

In parallel with the Maia system, a simpler data acquisition unit was developed for ion beam systems. This general-purpose unit employs the same SCEPTER chip used in Maia for pulse capture and readout of 32 detector shaped pulse inputs. In addition it can acquire from four traditional NIM analogue-to-digital convertors, accumulate charge through digital pulse counting inputs, and control a 4-axis stepper motor sample positioner and 2-axis analogue beam deflection.

These systems save data using the binary logger. The system's control program `klee` writes photon event data using either block 48 (`pm_event_ts_1`) for time-stamped events (64 bits or 8 bytes per event), or block 49 (`pm_event_nots_1`) for non-time-stamped events (32 bits or 4 bytes per event). These are blocks very similar in concept to the Maia photon event blocks, although Maia does not support photon time-stamping at present.

Table 3 describes the events that may appear in block 48 (`pm_event_ts_1`). Each event, whether it results from a photon event, and pixel address change event, or a counter readout, is recorded by a 64-bit (8-byte) structure.

Table 3 Events in `pm_event_ts_1` blocks:

ET (Photon Energy/Time) event:

Byte 0								Byte 1								Byte 2								Byte 3							
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	Addr							Time								Energy															
Byte 4								Byte 5								Byte 6								Byte 7							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timestamp																															

PA (Pixel Address) event:

Byte 0								Byte 1								Byte 2								Byte 3							
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	Axis			Pixel value (signed)																											
Byte 4								Byte 5								Byte 6								Byte 7							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timestamp																															

Axis bit-field may have values between binary 000 (axis 0) and binary 110 (axis 6), indicating which axis the pixel value refers to. Axes 0 and 1 are the two analogue beam deflection axes, axes 2 to 5 are mechanical stepper axes, and axis 6 is reserved.

TC (Time/Charge) counter readout:

Byte 0								Byte 1								Byte 2								Byte 3							
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	1	1	1	Sel	Counter value (unsigned)																										
Byte 4								Byte 5								Byte 6								Byte 7							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timestamp																															

Select bit-field may be 00, indicating the counter is the block time counter (increments of 100 ns), or 01, indicating the counter is the charge accumulation counter.

UU Reserved for future use:

Byte 0								Byte 1								Byte 2								Byte 3							
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	1	1	1	1	1	Don't care																									
Byte 4								Byte 5								Byte 6								Byte 7							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timestamp																															

3.5.1 Photon Events

Most events will be from pulse acquisition in either the SCEPTER chip or the traditional NIM hardware. These are the photon/time or *ET* events, which contain the address (channel), energy (amplitude), and time of the event, and its absolute timestamp. Addresses 0 to 31 indicate the event was generated by SCEPTER while addresses 32 to 35 indicate it was generated by the NIM hardware.

For SCEPTER events, energy is digitised to 13-bit precision and time to 11 bits. Both quantities are unsigned. The time field generally records time-over-threshold (or dead-time), but the SCEPTER may be configured to capture the time of threshold-to-peak, peak-to-tail, or peak-to-readout clock (absolute timestamp). The time units are also configurable. Refer to the SCEPTER chip manual for more details.

For NIM events, the energy field is the amplitude acquired by the NIM device, of up to 13 bits. The time field is the time the device indicates it is 'busy', usually the pulse time-over-threshold plus a small readout period. The quanta are 100 ns.

The absolute timestamps are the time of acquisition of the event, also in quanta of 100 ns, and are sampled from a counter clocked at 10 MHz.

3.5.2 Pixel Events

Pixel events indicate motion of either the sample (or at least, one of the four stepper motor outputs that generally are used to position the sample), or of the ion beam (one of the two analogue outputs that generally drive beam deflection). Motion is quantised into *pixels* which have a configurable pitch in each axis dimension. The axis field of the event indicates which axis has moved, and the value is the new value of that pixel. Like the photon events, pixel events are time-stamped using the same counter.

3.5.3 Time/Charge Events

These are not strictly events but indicate the readout of the block timer and the block charge counter at the end of event accumulation for that block. The time is in 100 ns quanta, and charge is the count of digital pulses received on the 'charge' input to the data acquisition system.

Table 4 Events in `pm_event_nots_1` blocks:

ET (Photon Energy/Time) event:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Addr							Time								Energy															

PA (Pixel Address) event:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Axis			Pixel value (signed)																											

TC (Time/Charge) counter readout:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Sel	Counter value (unsigned)																										

UU Reserved for future use:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	Don't care																									

Table 4 describes the events that may appear in block 49 (`pm_event_nots_1`). Each event is recorded as a 32-bit (4-byte) structure. The structures are identical to those in block 48 except that the time-stamp is omitted.

4 Reading Applications

A number of applications exist for reading the Maia data files.

4.1 GeoPIXE

The GeoPIXE application, originally developed by Chris Ryan for PIXE (Proton Induced X-ray Emission) imaging on a nuclear microprobe, has been adapted to support SXRF imaging, and to support for the Maia detector system. It is the primary tool for reading, calibrating, fitting and elemental imaging of raw Maia data. It also generates the configuration files (DA matrix, energy calibration, pileup and throttle) that are loaded into the detector system FPGA for real-time elemental analysis. It is also capable of processing blog files written by the Beamline Data Acquisition Units for PIXE imaging.

4.2 Maia Control

The main graphical user interface for Maia is known as Maia Control, and is included in the GeoPIXE suite, from which it draws display modules for spectra and elemental images.

4.3 Blog Browser

A subsidiary application in the GeoPIXE suite, also implemented in IDL, is the Blog Browser (on Linux, this can be launched by typing `blogbrowse`, if the execution path is correctly set up). It is a graphical tool for inspecting blog files.

4.4 Blog Utilities

The binary logger suite itself has a number of UNIX command-line utilities for manipulating blog data. Program `blogread` connects to the server as a reader client, and prints selected blocks in real time. Program `blogfile` can read saved data files and print them in the same way. Presently these programs print the payloads for most blocks in hexadecimal.

4.5 Mondriaan

This program was written as a simple and fast alternative to GeoPIXE to investigate stage motion and software bugs that caused heavy black vertical and horizontal lines on images (hence the choice of name). It remains a quick-and-dirty way of processing blog files into images, particularly if an uncorrected photon count image is all that is required. It produces floating-point, high dynamic-range TIF files which can be examined using an image viewer such as Image J.

4.6 Monet

This is an on-line monitoring tool for Maia ("monitor ET" or `monet`), implemented in C++ and ROOT. It exists in numerous versions tailored for particular situations; `monet2` seems to be the current version. From a subset of photon record data obtained over the network from the logger it accumulates and displays histograms of energy, time over threshold, detector address, and spatial pixel, in various useful combinations. `monet` and `monet2` have been made obsolete by Maia Control.

4.7 Rolling Your Own

Reading the Maia files should be implemented as the three-layer process. The top (run) level should iterate over all the segment files in a run. Note the segment structure in a run bears no relationship to image scan lines. Be sure to read the segments in numeric order, not alphabetical order of filename.

The next level should iterate over all blocks in the segment, and interpret the block headers. The payloads of blocks that are of interest (a minimal set would be `maia_events_1`, `maia_scan_2` or `maia_scan_3`) can then be passed to the next level for interpretation.

The current pixel address *PA* should be cached from the first three 32-bit words of each `maia_events_1` block, and all following *ET* events in the block decoded. Typically a spectrum would be accreted from the energy fields, although an energy calibration may be applied first. Note that the pixels will not necessarily start from (0, 0, 0) or proceed in any particular order.

A `maia_scan_2` or `maia_scan_3` block should appear toward the beginning of the run to indicate the raster size, although pixels outside the raster may possibly appear. It may be useful to search through the data file for the scan record, then return to the beginning of the run and process the event data.

Once blogging starts, all events are logged tagged by XYZ pixel address (PA). As the stage moves, the PA is calculated as it goes and used to tag these records. If the stage moves outside the indicated scan area it may produce PA that go negative or larger than the size of the scan. This is beyond the control of Maia, as the stage (and blog enable/disable) are controlled at the beamline. When processing data, all XY must be filtered against the scan bounds.

A language compiled to machine code, such as C or C++, is recommended over scripting languages. Execution will be much faster (runs typically comprise gigabytes of data) and low-level binary operations can be awkward in scripting languages. As mentioned, care must be exercised so the reading application is at least source portable. Byte-ordering ('endian') issues can be avoided by using byte-by-byte processing and arithmetic reconstruction of the bit-packed quantities in *ET* and *PA* events.

The following C++ fragment illustrates decoding a *ET* from a `maia_events_1` block, and can be read in conjunction with Table 1:

```
// pointer into buffer containing block payload
unsigned char *recvbuffer;
unsigned int word;
unsigned int i;
. . .

// decode one ET event, advance pointer
word =
    (recvbuffer[i+0] << 24) |
    (recvbuffer[i+1] << 16) |
    (recvbuffer[i+2] <<  8) |
    (recvbuffer[i+3] <<  0);
i += 4;

tag = (word >> 31) & ((1 <<  1) - 1);
if (tag == 0) {
    adr = (word >> 22) & ((1 <<  9) - 1);
    dt  = (word >> 12) & ((1 << 10) - 1);
    de  = (word >>  0) & ((1 << 12) - 1);
    . . .
}
. . .
```

A Tag Numbers

No	Label	Description
0	ignore	Ignore
1	id	Identity
2	newrun	New Run
3	newseg	New Segment
4	tod	Time-of-Day
5	summary	Activity Summary
6	comment	Comment
7	sendnext	Request Next Block
8	maia_et_events_1	Energy/Time Events (32 element)
9	maia_xy_events_1	XY Stage Events
10	maia_pa_events_1	Pixel Address Events
11	maia_da_put_1	DA analysis pileup table initialisation data
12	maia_da_calibration_1	DA analysis detector calibration coefficients
13	maia_da_caltable_1	DA analysis detector calibration table
14	maia_da_matrix_1	DA matrix values for per element
15	maia_da_pixel_1	DA analysis result for a pixel
16	maia_da_init_file_1	DA analysis initialisation file name entry
17	maia_da_element_1	DA analysis initialisation element string
18	maia_da_params_1	DA analysis general parameters
19	maia_da_matrix_raw_1	DA matrix table entries per element
20	maia_da_cal_1	DA analysis DA calibration coefficients
21	maia_da_throttle_1	throttle table
22	maia_enable_1	DALI enable bits
23	sendprev	Request Previous Block
24	sendprevornext	Request Previous or Next Block
25	maia_et_events_2	Energy/Time Events (96 element)
26	monitor	Monitor
27	pm_etrr_1	Pulse Mezz SCEPTER energy/time/rrclock event
28	id_2	Identity
29	endrun	End Run
30	maia_rexec_1	MIRO rexec parameters
31	maia_et_events_3	Energy/Time/Position Events (384 element)
32	summary_2	Activity Summary
33	setgroup	Set Run Group
34	maia_events_1	Energy/Time/Position Events (384 element)
35	maia_da_accum_1	DA/DT accumulator output
36	maia_roi_accum_1	region of interest accumulator output
37	maia_deadtime_accum_1	dead time accumulator output
38	maia_dtpp_accum_1	dead time per pixel accumulator output
39	maia_activity_accum_1	detector activity accumulator output
40	maia_energy_spectrum_accum_1	energy spectrum accumulators output
41	maia_et2d_accum_1	ET 2D accumulator output
42	maia_scan_info_1	scan parameters
43	maia_time_spectrum_accum_1	time spectrum accumulators output
44	maia_da_info_1	DA/DT info block
45	var_list_1	Common library var list
46	var_value_1	Common library var values
47	maia_scan_info_2	scan parameters
48	pm_event_ts_1	Pulse Mezz time-stamped events
49	pm_event_nots_1	Pulse Mezz non-time-stamped events
50	pm_activity_1	Pulse Mezz activity record
51	setproject	Set Run Project
52	clientaction	Client connect/disconnect log
53	summary_3	Activity Summary
54	setclient	Set client name
55	metadata	Metadata

B Old Maia photon data

This appendix described the key blocks logged by older Maia systems (`pddc`, `da`, `dali` and `miro`).

B.0.1 Photon Blocks and Sample Scanning

There are a fairly large number of Maia system blocks, but only one (block 25) actually needs to be interpreted in order to read the majority of data files produced by the older systems.

The photon record blocks are number 8 (`maia_et_events_1`), number 25 (`maia_et_events_2`), 31 (`maia_et_events_3`) and 34 (`maia_events_1`). The first is associated with the 32-element Maia prototype, the second with the 96-element prototypes, and the third is a simplified, interim format used by the 384-element detector. The last one is the format to be used long-term by the final 384-element detector.

Tags 8, 25 and 31 contains one 32-bit (4 byte) *PA event* (pixel address) word (Table 5), followed by a sequence of zero or more 32-bit *ET event* (energy-time, or photon) words (Table 7), up to the maximum block size. The top 1 or 2 bits of the event words indicates the event type, and the remaining 30 or 31 bits contain the event data, which has a bit-packed format.

The *PA* event indicates the spatial pixel that the subsequent *ET* (photon) events in that block belong to. Several blocks of events may belong to the one spatial pixel. They will typically be consecutive, although file reading software should not rely on this. The 30-bit data field in the *PA* event contains two 15-bit signed integer fields, the x and y pixel addresses.

The 384-element system supports three scanning axes (z may represent tomographic θ or monochromator E, or be unused). Three *PA* words are therefore used in tag 34, representing x, y and z. The *axis* indicates which.

The pixel array ranges from (0, 0), the bottom left corner of the scan, to $(w-1, h-1)$ for the top corner, where w is the width and h the height of the scan in pixels. The size of the pixels (pixel pitch) may differ in x and y, although it is frequently the same. The pitch and overall scan size (and other scan parameters) are recorded in block 30 (see below), although this is a recent addition; previously this data was recorded only as `miro` commands in the comment blocks. Pixel

Table 5 *PA* (pixel address) bit-field structure.

In `maia_et_events_1`, `maia_et_events_2` and `maia_et_events_3` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	Pixel y														Pixel x															

In `maia_events_1` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Axis	Pixel value (signed)																											

Axis bit-field may be 00 (indicates x), 01 (y) or 10 (z) only.

Table 6 *SE* (stage encoder) event bit-field structure.

In `maia_events_1` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Axis	Encoder value (signed)																													

Axis bit-field may be 00 (indicates x), 01 (y) or 10 (z) only.

addresses may fall outside the formal scan area (and be negative). The Maia scan generator ensures pixels inside the scan area have constant *transit* time (by maintaining constant fly-scan velocity). The pixels outside are where the stage is accelerated and decelerated, and so have variable dwell. Imaging applications typically clip these pixels from final images.

The photon (*ET*) events also have a bit-field structure. Its format varies a little between the different detector prototypes, but the essential elements are the *energy*, *time* and *address* fields. The first is digitised photon energy, or pulse height (12 bits), the second digitised pulse time over discriminator threshold (12 or 10 bits), and the third the detector element or pad address (0–31, 0–95, or 0–383). Strictly, these events represent records read from the SCEPTER pulse capture/derandomiser ASICs and their attendant ADCs, and will include some proportion of multi-photon (piled up) events. The F bit in some of the *ET* events indicates the state of the SCEPTER PDF flag, and indicates the PD/TAC array was full.

The values for energy and time are raw, uncalibrated ADC counts. The prototype detectors do not log normalised, linearised or energy-corrected values, even though these are actually computed in the FPGA when in DA deconvolution is enabled. The software reading the files must do this. GeoPIXE includes numerous functions for normalising, linearising and energy calibrating Maia detector data.

The pulse width value may be used with the pulse height to reject most piled-up events, because these will have longer pulse widths than those from single photon events. The FPGA can reject piled-up events, but only if it is loaded with a pileup table.

Blocks of type 34 (`maia_events_1`) may contain *SE* events (Table 6). These mark changes in the sample stage encoders or interferometers, which are the underlying source for the coarser *PA* values. They will be interspersed with *ET* events. *SE* events are emitted by the detector only when tracking fine motion of the sample stage.

B.0.2 Other Blocks

Earlier versions of the detector also wrote blocks 9 (`maia_xy_events_1`) and 10 (`maia_pa_events_1`) which had a similar format to the photon blocks, but contained XY and PA events respectively only. These blocks can be safely ignored.

Table 7 *ET* (energy/time, i.e., photon) event bit-field structures.

In `maia_et_events_1` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	F	Time													Energy								Addr							

In `maia_et_events_2` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	F	Time													Energy								Addr							

In `maia_et_events_3` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Time										Energy													Addr							

In `maia_events_1` blocks:

Byte 0								Byte 1								Byte 2								Byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Addr										Time													Energy							

Block 30 (`maia_rexec_1`) was added recently to record information about a stage scan, particularly overall scan size, pixel pitch, fly-scan speed and (redundantly) pixel dwell. Scanning has to date always used x is the fast axis in a bottom-to-top raster, but this should not be relied upon.

When real-time DA deconvolution is enabled, a host of other blocks are logged; a full description of these is beyond the scope of this note. Many of these blocks record the values in the DA matrix and attendant tables (pileup, energy calibration) in the FPGA. Block 15 (`maia_da_pixel_1`) is of note, because contains it contains the DA elemental concentrations for each chemical element (up to 32) in each spatial pixel.

C TAGS File

```
# This is part of BLOG, a Binary Logger for experimental data
# (c) 2006 CSIRO Manufacturing and Infrastructure Technology
# $Id: TAGS 6805 2015-10-15 01:00:45Z jen117 $
#
# TAGS -- authorative block tag numbers and descriptions
#
# This is the authorative list of block tag numbers. It is designed to be
# automaticially transformed into header files and code in different languages,
# as well as being checked for consistency.
#
# Each block tag has a single line in this file:
#
#       <number> <symbolic-name> <short-description>
#       # <long description ...>
#
# Numbers should be assigned in order, without gaps, although they need not
# appear in this file in that order. The symbolic name should follow C
# identifier syntax, and begin with a project acronym, and optionally end
# with a version number. The short description should not exceed about 40
# characters. A longer description should follow on the next line, as a
# comment, and can be of any length. It should describe the format of the
# data in the payload part of the block.
#
# The following abbrevitions are used:
#
#       uint8/uint16/uint32     unsigned byte/word/long
#       int8/int16/int32        signed byte/word/long
#       string0                 null-terminated string of any length
#       string16 (etc)          string of given length


# These blocks belong to BLOG

0       ignore                  Ignore
        # Any data will be ignored

1       id                      Identity
        # tag at the beginning of each segment (obsolete since SVN 2617)
        #
        #       uint4           BLOG format version number
        #       uint4           run number
        #       uint4           segment number
        #       uint4           (zero)
        #       uint4           file ctime, seconds since 1/1/1970 UTC
        #       string0         local timezone name (not abbreviation)
        #       string0         reference string
        #       string0         experiment string
        #       string0         equipment string
        #       string0         location string
        #       string0         personnel string

28      id_2                    Identity
        # blogd writes this tag at the beginning of each segment (file).
        #
        #       uint4           BLOG format version number
        #       uint4           run number
        #       uint4           segment number
        #       uint4           (zero)
        #       uint4           file ctime, seconds since 1/1/1970 UTC
        #       string0         local timezone name (not abbreviation)
        #       string0         blogd subversion revision string
        #       string0         logger hostname string
        #       string0         facility identifier string
```

```
        #       string0         blogd working directory string
        #       string0         blogd data path string

2       newrun                  New Run
3       newseg                  New Segment
        # Trigger blogd to begin a new run (increment run number, start new
        # data file) or begin a new segment (start new data file only).
        # Any data will be ignored

29      endrun                  End Run
        # Trigger blogd to enter discard mode. No more data is logged to disc
        # until the next newrun or newseg; either of these start a new run.
        # However, data is still served up to reader clients as before.

33      setgroup                Set Run Group
        # Sets the name of the next run group. This name is substituted for
        # &g when the data path (directories runs are written to) is generated
        #
        #       string0         new run group name

51      setproject              Set Run Project
        # Sets the name of the next run project. This name is substituted for
        # &p when the data path (directories runs are written to) is generated
        #
        #       string0         new run project name

4       tod                     Time-of-Day
        # Means by which clients can log their idea of time-of-day. Note that
        # blogd writes its idea of time-of-day (but not timezone) in each
        # block.
        #
        #       uint4           (zero)
        #       uint4           seconds since 1/1/1970 UTC
        #       uint4           mictoseconds since last second
        #       string0         local timezone name (Australia/Victoria, etc)

5       summary                 Activity Summary
        # Summary of blogd activity (obsolete since SVN 3584)
        #
        #       uint4           run number
        #       uint4           segment number
        #       uint4           run time (seconds)
        #       uint4           current number of clients
        #       uint4           total number of clients
        #       uint4           total number of blocks written
        #       double          rate of blocks being written
        #       uint4           total number of bytes
        #       double          rate of bytes being written

32      summary_2                       Activity Summary
        # Summary of blogd activity (version 2).
        #
        #       uint4           run number
        #       uint4           segment number
        #       uint4           discard mode flag
        #       uint4           run time (seconds)
        #       uint4           current number of clients
        #       uint4           total number of clients
        #       uint4           total number of blocks written
        #       uint8           total number of bytes written
        #       uint4           rate of blocks being written per sec
        #       uint4           rate of bytes being written per sec
        #       string0         next run group name

53      summary_3                       Activity Summary
```

```
        # Summary of blogd activity (version 3).
        #
        #       uint4      run number
        #       uint4      segment number
        #       uint4      discard mode flag
        #       uint4      run time (seconds)
        #       uint4      current number of clients
        #       uint4      total number of clients
        #       uint4      total number of blocks written
        #       uint8      total number of bytes written this run
        #       uint4      rate of blocks being written per sec
        #       uint4      rate of bytes being written per sec
        #       string0    next run group name
        #       string0    current run group name
        #       string0    next run project name
        #       string0    current run project name

56      summary_4                       Activity Summary
        # Summary of blogd activity (version 4).
        #
        #       uint32     run number
        #       uint32     segment number
        #       uint32     discard mode flag
        #       uint32     run time (seconds)
        #       uint32     current number of clients
        #       uint32     total number of clients
        #       uint32     total number of blocks written
        #       uint64     total number of bytes written this run
        #       uint32     rate of blocks being written per sec
        #       uint32     rate of bytes being written per sec
        #       string0    next run group name
        #       string0    current run group name
        #       string0    next run project name
        #       string0    current run project name
        #       string0    current blog server error message
        #       uint64     total size of dump filesystem in bytes
        #       uint64     total number of bytes free on the dump filesystem
        #       uint64     (est) hours left before dump filsystem is full

57      report                  Blog Server Status Report
        # Report on blog server status.
        #
        #       uint64     total memory in host system
        #       uint64     free memory in host system
        #       uint64     total swap in host system
        #       uint64     free swap in host system
        #       uint32     host cpu user time (in "jiffies")
        #       uint32     host cpu nice time (in "jiffies")
        #       uint32     host cpu system time (in "jiffies")
        #       uint32     host cpu idle time (in "jiffies")
        #       uint32     host cpu iowait time (in "jiffies")
        #       uint32     host cpu irq time (in "jiffies")
        #       uint32     host cpu softirq time (in "jiffies")
        #       uint32     host interrupt count
        #       uint32     host context switch count
        #       uint32     time at which host booted (secs since epoch)
        #       uint32     total number of processes on host
        #       uint32     total number of running processes on host
        #       uint32     total number of blocked processes on host
        #       float      host load average over last minute
        #       float      host load average over last 5 minutes
        #       float      host load average over last 15 minutes
        #       double     host uptime
        #       double     host idle time
        #       uint32     dump fs major number (all zero if NFS)
```

```
        #       uint32     dump fs minor number
        #       string0    dump fs device name
        #       uint32     dump fs reads completed
        #       uint32     dump fs reads merged
        #       uint32     dump fs sectors read
        #       uint32     dump fs time reading (ms)
        #       uint32     dump fs writes completed
        #       uint32     dump fs writes merged
        #       uint32     dump fs sectors written
        #       uint32     dump fs time writing (ms)
        #       uint32     dump fs I/Os in progress
        #       uint32     dump fs time doing I/O (ms)
        #       uint32     dump fs weighted time doing I/O (ms)
        #       uint32     blogd vm size
        #       uint32     blogd "resident set size"
        #       uint32     blogd "proportional set size"
        #       uint32     blogd shared clean pages
        #       uint32     blogd shared dirty pages
        #       uint32     blogd private clean pages
        #       uint32     blogd private dirty pages
        #       uint32     blogd referenced
        #       double     blogd user CPU time used
        #       double     blogd system CPU time used
        #       uint32     blogd maximum resident set size
        #       uint32     blogd number of swaps
        #       uint32     blogd block input operations
        #       uint32     blogd block output operations
        #       uint32     blogd signals received
        #       uint32     blogd voluntary context switches
        #       uint32     blogd involuntary context switches
        #       uint32     number of client entries below (i.e. a variable
        #                  length array of variable size entries (as many
        #                  complete entries as will fit) with each entry
        #                  consisting of:
        #
        #               uint32          client serial number
        #               string0         hostname of client connection
        #               string0         port (service)
        #               string0         name
        #               uint64          total number of blocks written by client
        #               uint64          total number of bytes written by client
        #               uint64          delta number of blocks written by client
        #               uint64          delta number of bytes written by client
        #               uint64          total number of blocks read by client
        #               uint64          total number of bytes read by client
        #               uint64          delta number of blocks read by client
        #               uint64          delta number of bytes read by client

52      clientaction            Client connect/disconnect log
        #
        #       uint4           client serial number
        #       uint4           connection action:
        #                           0 already connected
        #                           1 newly connected
        #                           2 newly disconnected
        #                               (more may be added)
        #       uint4           current number of clients
        #       uint4           total number of clients
        #       string0         hostname of client connection
        #       string0         port (service)
        #       string0         name

54      setclient               Set client name
        # A name that will be used to identify this client connection.
        #
```

```
        #       string0         name

6       comment                 Comment
        # A block of text. The format is entirely up to the user.
        #
        #       string0         text

7       sendnext                Request Next Block
        # Used by "reader" clients. Comprises a set of (len/2) 16-bit
        # BLOG tags. Following receipt, blogd will send back to the client
        # the *next* block received of any of the listed tag types, whenever
        # that is. The request is then discarded.
        #
        #       uint16          block tag number
        #       uint16          block tag number
        #       ...

23      sendprev                Request Previous Block
        # Used by "reader" clients. Comprises a set of (len/2) 16-bit
        # BLOG tags. Following receipt, blogd will send back to the client
        # the *most recently received* (cached) block of any of the listed
        # tag types, starting from the first listed. If there are no blocks
        # of any of the types, a 0 (ignore) block is returned to the client.
        #
        #       uint16          block tag number
        #       uint16          block tag number
        #       ...

24      sendprevornext          Request Previous or Next Block
        # Used by "reader" clients. Comprises a set of (len/2) 16-bit
        # BLOG tags. Following receipt, blogd will send back to the client
        # the *most recently received* (cached) block of any of the listed
        # tag types, starting from the first listed. If there are no cached
        # blocks, the request is atomically converted to a sendnext request
        #
        #       uint16          block tag number
        #       uint16          block tag number
        #       ...

26      monitor                 Monitor
        # Produced by "blogepics" and other clients which monitor relatively
        # slow-moving process variables. The block is a null-terminated single
        # string, with structured text information within.
        #
        #       string0         text
        #
        # Each line of the text string contains information about one variable,
        # and has the format
        #
        #       <name>  <state> <value>
        #
        # where <name> is usually the EPICS PV name, <state> is a string
        # indicating its state (reliability), and <value> is one (or more,
        # in the case of arrays) values

55      metadata                Metadata
        # Contains key-value pairs, one per line
        #
        #       string0         text
        #
        #       <key> <value>


# ------------------------------------------------------------------ OLD MAIA
```

```
# These blocks belong to the MAIA project (pddc, dali, miro systems)

8       maia_et_events_1        Energy/Time Events (32 element)
        # A set of (len/4) 32-bit ET event words, big endian order,
        # bitfield defined by maia/pddc/event.3pl or maia/dali/event.3pl
        #
        #       struct pa               most recent PA event
        #       struct et               ET event
        #       struct et               ET event
        #       ...
        #
        #   struct {
        #       unsigned tag: 2;        (must be 10)
        #       int y: 15;
        #       int x: 15;
        #   } pa;
        #
        #   struct {
        #       unsigned tag: 2;        (must be 00)
        #       unsigned pdf: 1;
        #       unsigned dt: 12;
        #       unsigned de: 12;
        #       unsigned adr: 5;
        #   } et;

9       maia_xy_events_1        XY Stage Events
        # A set of (len/4) 32-bit XY event words, big endian order,
        # bitfield defined by maia/pddc/event.3pl and event.h
        #
        #       struct pa               most recent PA event
        #       struct xy               XY event
        #       struct xy               XY event
        #       ...
        #
        #   struct {
        #       unsigned tag: 2;        (must be 01)
        #       int y: 15;
        #       int x: 15;
        #   } xy;

10      maia_pa_events_1        Pixel Address Events
        # A set of (len/4) 32-bit XY event words, big endian order,
        # bitfield defined by maia/pddc/event.3pl and event.h
        #
        #       struct pa               most recent PA event
        #       struct pa               PA event
        #       struct pa               PA event
        #       ...
        #
        #   struct {
        #       unsigned tag: 2;        (must be 10)
        #       int y: 15;
        #       int x: 15;
        #   } pa;

25      maia_et_events_2        Energy/Time Events (96 element)
        # A set of (len/4) 32-bit ET event words, big endian order,
        # bitfield defined by maia/miro/event.3pl.
        # Note there are no XY events
        #
        #       struct pa               most recent PA event
        #       struct et               ET event
        #       struct et               ET event
        #       ...
        #
```

```
#   struct {
#       unsigned tag: 2;        (must be 10)
#       int y: 15;
#       int x: 15;
#   } pa;
#
#   struct {
#       unsigned tag: 2;        (must be 00)
#       unsigned pdf: 1;
#       unsigned dt: 10;
#       unsigned de: 12;
#       unsigned adr: 7;
#   } et;

16      maia_da_init_file_1     DA analysis initialisation file name entry
        # This contains a single string which is the file identification entry
        # from the initialisation file - it is usually, but not necessarily, the
        # file name.
        #
        #       string of arbitrary length

17      maia_da_element_1       DA analysis initialisation element string
        # A uint16 giving the element number followed by a string providing
        # a label for the element.
        #
        #       uint16    channel (element) number
        #       string of arbitrary length

11      maia_da_put_1      DA analysis pileup table initialisation data
        # Pairs, one for each pulse height. Each pair is a minimum time over
        # threshold and a maximum time over threshold. Each entry is a uint16.
        #
        #       uint16    minimum time-over-threshold for pulse height 0
        #       uint16    maximum time-over-threshold for pulse height 0
        #       uint16    minimum time-over-threshold for pulse height 1
        #       uint16    maximum time-over-threshold for pulse height 1
        #       ...          etc.

12      maia_da_calibration_1   DA analysis detector calibration coefficients
        # Triples, one for each detector. Each triple consists of offset, gain
        # and quadratic terms in that order and each term is a 32 bit float.
        #
        #       float      offset for detector 0
        #       float      gain for detector 0
        #       float      quad term for detector 0
        #       float      offset for detector 1
        #       ...          etc.

13      maia_da_caltable_1      DA analysis detector calibration table
        # The full calibration calculated from the detector calibration
        # coefficients for each detector and the DA calibration coefficients.
        # The first entry is a uint16 giving the detector number. The remaining
        # entries map the pulse height to DA bin. Each of these entries is a
        # uint16.
        #
        #       uint16    detector number
        #       uint16    bin number for pulse height 0
        #       uint16    bin number for pulse height 1
        #       ...          etc.

14      maia_da_matrix_1        DA matrix values for per element
        # The first 32-bit entry is the channel (element) number, 0 to 31.
        # Following this there are 2048 or 4096 32 bit floats.
        #
        #       uint32    channel (element) number
```

```
        #       float      DA value for bin 0
        #       float      DA value for bin 1
        #       ...          etc.

19      maia_da_matrix_raw_1    DA matrix table entries per element
        # The first 32-bit entry is the channel (element) number, 0 to 31.
        # Following this there are 2048 or 4096 signed 32 bit integers.
        #
        #       uint32    channel (element) number
        #       int32      DA entry for bin 0
        #       int32      DA entry for bin 1
        #       ...          etc.

20      maia_da_cal_1      DA analysis DA calibration coefficients
        # A triple of offset, gain and quadratic terms in that order, each
        # term being a 32 bit float.
        #
        #       float      offset
        #       float      gain
        #       float      quad term

18      maia_da_params_1        DA analysis general parameters
        #       uint16    number of channels implemented
        #       uint16    number of channels active (contiguous from 0)
        #       uint16    number of detectors implemented
        #       uint16    number of bits for pulse height (12)
        #       uint16    number of bits for time over threshold (12)
        #       uint16    number of bits for DA bins (11 or 12)

21      maia_da_throttle_1      throttle table
        # The throttle does not control event input to the DA processing pipeline
        # however it is convenient to load the throttle table from the
        # DA initialisation file, so this block is included here.
        # The table consists of one entry for each pulse height, the entry
        # being a scaling factor. Entries are uint16.
        #
        #       uint16    scale factor for pulse height 0
        #       uint16    scale factor for pulse height 1
        #       ...          etc.

15      maia_da_pixel_1         DA analysis result for a pixel
        # The first 32 bit entry is the pixel address in the same format as
        # maia_pa_events_1 entries. The following float entries are the results
        # for each channel (element) for the number of active channels in
        # channel order.
        #
        #       uint32          PA event
        #       float      DA result for channel (element) 0
        #       float      DA result for channel (element) 1
        #       ...          etc.
        #
        #   struct {
        #       unsigned tag: 2;        (must be 2)
        #       int y: 15;
        #       int x: 15;
        #   } pa;

22      maia_enable_1      DALI enable bits
        # This block is logged when any enable bits are changed.
        #
        #       uint32    enable word

30      maia_rexec_1            MIRO rexec parameters
        # Contains the parameters of the scan that is about to commence. The
        # parameters will match those available from the miro rinfo command
```

```
        #
        #       uint8           scan type (0 = normal raster)
        #       uint8           scan major direction (0 = x, 1 = y)
        #       uint8           (spare)
        #       uint8           (spare)
        #       uint32          raster size x (pixels)
        #       uint32          raster size y (pixels)
        #       float           origin x (mm)
        #       float           origin y (mm)
        #       float           pixel pitch x (mm)
        #       float           pixel pitch y (mm)
        #       float           velocity (mm/s)
        #       float           time per pixel (s)
        #       float           edge overrun (mm)
        #       uint32          estimated total time (s)

31      maia_et_events_3        Energy/Time/Position Events (384 element)
        # A set of (len/4) 32-bit ETE event words, big endian order,
        # bitfield defined by maia/kandinski/event.3pl. This is a transitional
        # tag and will eventually be replaced.
        #
        # The first word is the current (or new) pixel address (PA). Subsequent
        # words are either photon (ET) or encoder position (XY) events. The
        # tags bits are used to disambiguate them. These have a hierarchical
        # structure now. XY events may or may not be present.
        #
        #       struct pa               most recent PA event
        #       struct et OR xy         ET event or XY event
        #       struct et OR xy         ET event or XY event
        #       ...
        #
        #   struct {
        #       unsigned tag: 2;        (must be 10)
        #       int y: 15;
        #       int x: 15;
        #   } pa;
        #
        #   struct {
        #       unsigned tag: 1;        (must be 0)
        #       unsigned dt: 10;
        #       unsigned de: 12;
        #       unsigned adr: 9;
        #   } et;
        #
        #   struct {
        #       unsigned tag: 3;        (must be 110)
        #       unsigned axis: 2;       (0 = x, 1 = y, 2 = z)
        #       unsigned pos: 27;
        #   } xy;


# ------------------------------------------------------------------ PULSE MEZZ

# These blocks belong to projects using the HYMOD2 Pulse Mezzanine

27      pm_etrr_1               Pulse Mezz SCEPTER energy/time/rrclock event
        # A block containing only generic 64-bit etrr_1 structures:
        #
        #   struct {
        #       unsigned _pad: 2;
        #       unsigned conseq: 1;
        #       unsigned pdf: 1;
        #       unsigned de: 12;
        #       unsigned dt: 12;
        #       unsigned rr: 24;
```

```
        #       unsigned adr: 12;
        #   } etrr_1;

48      pm_event_ts_1           Pulse Mezz time-stamped events
        # Successor to block 27, with 64-bit photon events, but also a
        # block/pixel structure similar to block 34.
        # Details to follow

49      pm_event_nots_1         Pulse Mezz non-time-stamped events
        # As above, but with photon timestamps omitted.
        # Details to follow

50      pm_activity_1           Pulse Mezz activity record
        #
        #       uint32          number of photon channels (N)
        #       uint32          counts channel 0
        #       uint32          counts channel 1
        #       ...
        #       uint32          counts channel N-1
        #       uint32          time (units of 100e-9 s)
        #       uint32          charge (counts)
        #       uint32          total counts channels 0 to N-1
        #
        # N is usually 36 but may change


# ------------------------------------------------------------- CURRENT MAIA

# These blocks are produced by the current (kandinski) Maia system

34      maia_events_1           Energy/Time/Position Events (384 element)
        # A set of (len/4) 32-bit raw event words, big endian order,
        # bitfield defined by maia/kandinski/event_buffers.3pl
        #
        # The first three words are the current (or new) pixel address (PA),
        # the first word is X, second word is Y and third word is Z.
        #
        # Subsequent words are either photon (ET) or stage encoder (SE)
        # or TF (time/flux) events. The tags bits are used to disambiguate
        # them. These have a hierarchical structure.
        #
        #       struct pa               X of most recent PA event
        #       struct pa               Y of most recent PA event
        #       struct pa               Z of most recent PA event
        #       struct et OR se         ET  SE event
        #       struct et OR se         ET, SE event
        #       ...
        #
        #   // energy/time (photon) events
        #   struct {
        #       unsigned tag: 1;        (must be 0)
        #       unsigned adr: 9;
        #       unsigned dt: 10;
        #       unsigned de: 12;
        #   } et;
        #
        #   // stage encoder events (generally disabled)
        #   struct {
        #       unsigned tag: 1;        (must be 1)
        #       unsigned axis: 2;       (00 = x, 01 = y, 10 = z)
        #       signed coord: 29;
        #   } se;
        #
        #   // pixel address events (always first in the block)
        #   struct {
```

ET, SE, PA, and TF are to be dissected in bit form. See page 5.

```
        #       unsigned tag: 3;        (must be 111)
        #       unsigned axis: 2;       (00 = x, 01 = y, 10 = z)
        #       signed value: 27;
        #   } pa;
        #
        #   // block time and flux accumulator
        #   struct {
        #       unsigned tag: 5;        (must be 11111)
        #       unsigned axis: 2;       (00 = bt, 01 = fc0, 10 = fc1)
        #       signed value: 25;
        #   } tf;
        #
        # Tags of 7 bits or more are not yet used in data files, but may be
        # used in future.
        #
        #   struct {
        #       unsigned tag: 7;        (must be 1111111)
        #       signed value: 23;
        #   } unused;


# The next set of blocks are accumulator output blocks
#
# Each block will have a fixed header of 9 words, followed by a variable
# number of words depending on the block type (or "subject"). The header is:
#
#       uint32_t        pixel X coordinate
#       uint32_t        pixel Y coordinate
#       uint32_t        pixel Z coordinate
#       uint32_t        trigger (see below)
#       uint32_t        dwell/readout (see below)
#       uint32_t        duration (low 32 bits)
#       uint32_t        flux1
#       uint32_t        flux2
#       uint32_t        word count
#       uint32_t        data ...
#
# The "trigger" word is packed with bitfields for a number of things:
#
#         1    1     16      6       3      5
#       +---------------------------------------+
#       | D | PU | groups | TRIG | INDEX | SUBJ |
#       +---------------------------------------+
#
#       D       reject events flagged with discard bit from throttle
#       PU      reject events flagged with pileup bit
#       groups  group set for the accumulator<--------The group set for the accumulator... is this the channel number?
#       TRIG    the trigger causing the readout -
#                       0nnnnn  CPU trigger 0 to 31
#                       10.nnn  timer trigger 0 to 7
#                       1100..  PA entry trigger
#                       1101..  PA exit trigger
#                       1110..  PA transition trigger
#       INDEX   the index of the accumulator
#       SUBJ    the module type
#
# The duration/readout word is
#
#          5       17       1     1     8
#       +-----------------------------------+
#       | MTC | reserved | OFV | ERR |  DU |
#       +-----------------------------------+
#
#       MTC     Missed trigger count
#       OFV     Accumulator overflowed (not implemented on many accums)
```

```
#       ERR     Error flag for accumulator
#       DU      Upper 8 bits of duration
#
#       Duration is in units of 100ns

35      maia_da_accum_1         DA/DT accumulator output
        # Contains the output from the DA and deadtime accumulators.
        # The number of DA accumulators E is set in kandinski.h (usually 32)
        # but can be inferred from the sub-header word count (count-3)/2.
        #
        #     uint32    total event count
        #     uint32    pileup event count
        #     uint32    total time over threshold (dead time)
        #     int64     (fixed 28.24) DA accumulator 0
        #     int64     (fixed 28.24) DA accumulator 1
        #     ...
        #     int64     (fixed 28.24) DA accumulator E-1

36      maia_roi_accum_1        region of interest accumulator output
        # The word count is 2 * NROI (defined in kandinski.h - currently 0!).
        # Each datum is format "uint:32" and is the count of events whose
        # energy falls within the region of interest.

37      maia_deadtime_accum_1   dead time accumulator output
        # The word count is 3 * NDETS (defined in kandinski.h - currently 384).
        # Each datum is format "uint:32". Three data are produced for each of
        # NDETS detectors. The data triples are -
        #     total event count per detector
        #     pileup event count per detector
        #     total time over threshold per detector

38      maia_dtpp_accum_1       dead time per pixel accumulator output
        # ABOLISHED

39      maia_activity_accum_1   detector activity accumulator output
        # The word count is NDETS + GROUPS (each defined in kandinski.h -
        # currently 384 and 16 respectively).
        # Each datum is format "uint:32".
        # The first NDETS words are the total event count per detector.
        # The final GROUPS words are the total event count per group.

40      maia_energy_spectrum_accum_1    energy spectrum accumulators output
        # The word count is (1 << SAABITS) (defined in kandinski.h - currently
        # 12 i.e. 4096 words).
        # Each datum is format "uint:32".
        # Each word is the number of events at the energy indicated by the
        # index into the block.

43      maia_time_spectrum_accum_1      time spectrum accumulators output
        # The word count is (1 << DTBITS) (defined in kandinski.h - currently
        # 10 i.e. 1024 words).
        # Each datum is format "uint:32".
        # Each word is the number of events of the time indicated by the
        # index into the block.

41      maia_et2d_accum_1       ET 2D accumulator output
        # The word count is (1 << ETABITS) (defined in kandinski.h - currently
        # 14 i.e. 16384 words).
        # Each datum is format "uint:32".
        # The index into the block is a 14 bit quantity constructed as -
        #   7 bits 7 bits
        #   -------------
        #   |  E  |  T  |
        #   -------------
        #  where    E is the top 7 bits of the energy
```

I don't understand how to convert the 28.24 fixed-point numbers.

```
        #           T is the top 7 bits of the time over threshold
        # Each word is the number of events at the energy and
        # time-over-threshold indicated by the index into the block.

# ------------------------------------------------------------ MAIA INFO BLOCKS
# these are not accumulator blocks, so they don't have the accumulator header

42      maia_scan_info_1                scan parameters
        # Contains the parameters of the scan that is about to commence.
        # May mark multiple scans in the one run. Much of this is advisory.
        #
        #       uint32          scan sequence number in this run
        #       uint32          scan reference number (typically from EPICS)
        #       uint8           scan raster order*
        #       uint8           (spare)
        #       uint8           (spare)
        #       uint8           (spare)
        #       uint32          raster size 0/x (pixels, min 1)
        #       uint32          raster size 1/y (pixels, min 1)
        #       uint32          raster size 2/z (pixels, min 1)
        #       float           origin 0/x (mm)
        #       float           origin 1/y (mm)
        #       float           origin 2/x (mm)
        #       float           pixel pitch 0/x (mm, strictly positive)
        #       float           pixel pitch 1/y (mm, strictly positive)
        #       float           pixel pitch 2/z (mm, strictly positive)
        #       float           time per pixel (s, strictly positive)
        #       string0         user scan info (e.g, sample name, multiline)
        #
        # * The scan raster order should be enumerative, and may inform
        # pixelisation or image processing steps of the scan sequence.
        # Axes are listed from fastest to slowest. Sequence 1 is most common.
        #
        #       0       no particular order, unknown etc
        #       1       XYZ
        #       2       XZY
        #       3       YXZ
        #       4       YZX
        #       5       ZXY
        #       6       ZYX

47      maia_scan_info_2                scan parameters
        # Contains the parameters of the scan that is about to commence.
        # May mark multiple scans in the one run. Much of this is advisory.
        # (Same is maia_scan_info_1 with the addition of unit strings).
        #
        #       uint32          scan sequence number in this run
        #       uint32          scan reference number (typically from EPICS)
        #       uint8           scan raster order*
        #       uint8           (spare)
        #       uint8           (spare)
        #       uint8           (spare)
        #       uint32          raster size 0/x (pixels, min 1)
        #       uint32          raster size 1/y (pixels, min 1)
        #       uint32          raster size 2/z (pixels, min 1)
        #       float           origin 0/x (mm)
        #       float           origin 1/y (mm)
        #       float           origin 2/z (mm)
        #       float           pixel pitch 0/x (mm, strictly positive)
        #       float           pixel pitch 1/y (mm, strictly positive)
        #       float           pixel pitch 2/z (mm, strictly positive)
        #       float           time per pixel (s, strictly positive)
        #       string0         user scan info (e.g, sample name, multiline)
        #       string0         units 0/x
        #       string0         units 1/y
```

```
        #       string0         units 2/z

44      maia_da_info_1  DA/DT info block
        # Contains settings needed to interpret maia_da_accum_1 blocks
        #
        #       uint32          number of elements E (kandinski.h, usually 32)
        #       uint32          number of active elements
        #       float           deadtime scale 0 (offset) (s/ADC unit)
        #       float           deadtime scale 1 (rate)
        #       float           element scale factor 0
        #       ...
        #       float           element scale factor E-1
        #       string0         element name 0 (~20 char max)
        #       ...
        #       string0         element name E-1
        #       string0         element concentration unit  0 (~20 char max)
        #       ...
        #       string0         element concentration unit  E-1
        #       string0         DA provenance information string


# ------------------------------------------------------------ NETWORK VARIABLES

# These blocks are for text dumps of the network control variables implemented
# by the Common Library (var module). These blocks are single text strings,
# formatted identically to the .var file format read by var_parse_...()
# functions, and the varsh program.

45      var_list_1      Common library var list
        # A single null-terminated string containing zero or more lines of
        # plain text. Each line is either blank, a comment (#), or a line
        # declaring a variable. These lines contain space-separated tokens:
        #
        #       <name> <type> <unit> <flags>
        #
        # where <name> is the variable name (including all array dimensions),
        # <type> is the parse type (int, float, ...), units is the measurement
        # units (string, possibly ""), and flags are s|g|c (others may be
        # added). Additional optional tokens may be present.

46      var_value_1     Common library var values
        # A single null-terminated string containing zero or more lines of
        # plain text. Each line is either blank, a comment (#), or a line
        # showing a variable value using space-separated tokens:
        #
        #       <name> <value> [value] ...
        #
        # The <name> should be one declared in a previous var_list_1 block.
        # Array subscripts follow the rules implemented in the common library
        # var protocol. The <value> must be able to be parsed as an item of the
        # declared type. There must be exactly one, or as many values implied
        # by <name>.


# ------------------------------------------------------- MASTER/SLAVE VARIABLES

# These blocks are for implementing a master/slave relationship between two
# blogd instances. From the JIRA issue:
#
# 1. by default, an instance of the logger will assume it is stand-alone and
# will manage its own run number (initialised from the statefile) - as it
# currently does
#
# 2. a new config param will specify the ip-address/port-number of another
# blog server which will cause the logger to request a run number from that
```

```
# server whenever it needs a new one rather than manage the run number itself;
# it will still store the run number into its statefile; THE FACILITY OF THE
# LOCAL AND REMOTE LOGGERS MUST BE THE SAME (anything else such as path, group,
# project, etc can be different)
#
# Note that if a file-system is to be shared by multiple instances of the
# logger, then all of those loggers must use the same "master" logger for
# their run numbers (not sure how to enforce this)

58      run_number_request      Client (slave) request for new run number
        # Trigger the receiving blogd to create a new run number for the
        # client (by incrementing its run number and storing it in its
        # state file)
        #
        # the run number will be returned in a response block (see below)
        #
        # the payload of the request contains:
        #
        #       uint32          a "unique" number identifying this request
        #       uint32          the client blogd's current run number
        #       string0         the facility string of the client blogd
        #                       (this must match the facility string
        #                       configured for the receiving blogd)

59      run_number_response     Server (master) response with new run number
        # the payload of the request contains:
        #
        #       uint32          the "unique" number identifying the request
        #                       this block is in response to
        #       uint32          the new run number for the client
        #       string0         the facility string of the server blogd
        #                       (this must match the facility string
        #                       configured for the receiving blogd)

# end of TAGS file
```

Document SVN revision 6889