Raw OLAP data formats

OLAP produces several data formats, which are intended to be replaced by their final format, such as HDF5. The formats below are not officially supported and subject to change without notice.

Data can be recorded as either complex voltages (yielding X and Y polarisations) or one or more stokes. In either case, a sequence of blocks will be stored, each of which consists of a header and data. The header is defined as:

```
struct header {
  uint32 sequence_number; /* big endian */
  char padding[508];
};

/*
// Proposed: no header. Missing data is replaced by zeros.
struct header {
};
*/
```

in which sequence_number starts at 0, and is increased by 1 for every block. Missing sequence numbers implies missing data. The padding can have any value and is to be ignored.

Complex Voltages

Each (pencil) beam produces two files: one containing the X polarisation, and one containing the Y polarisation. The names of these files adhere to the following scheme:

Lxxxxx_Byyy_S0_bf.raw	X polarisations of beam yyy of observation xxxxx
Lxxxxx_Byyy_S1_bf.raw	Y polarisations of beam yyy of observation xxxxx

Each file is a sequence of blocks of the following structure:

```
struct block {
  struct header header;

  /* big endian */
  // 2010-09-20 release and later:
  fcomplex voltages[SAMPLES|2][SUBBANDS][CHANNELS];

  /*
  // 2010-06-29 release and earlier stored data per subband instead of per
beam:
  fcomplex voltages[BEAMS][CHANNELS][SAMPLES|2][POLARIZATIONS];
  */

  /*
  // Proposed:
```

```
  float voltages[SAMPLES][SUBBANDS][CHANNELS];
  // Note that because the header will also be empty, the file is
essentially
  // a seamless stream of
  //   float voltages[SUBBANDS][CHANNELS];
  // If the subbands are chosen seamless as well, the data reduces to a
stream of
  //   float voltages[CHANNELS];
  // with simply a larger number of channels.
  */
};
```

Older releases: 2010-09-20:

1. filenames ended in -bf.raw instead of _bf.raw

Coherent Stokes

Each (pencil) beam produces one or four files: one containing the Stokes I (power) values, and optionally three files for Stokes Q, U, and V, respectively. The names of these files adhere to the following scheme:

Lxxxxx_Byyy_S0_bf.raw	Stokes I of beam yyy of observation xxxxx
Lxxxxx_Byyy_S1_bf.raw	Stokes Q of beam yyy of observation xxxxx
Lxxxxx_Byyy_S2_bf.raw	Stokes U of beam yyy of observation xxxxx
Lxxxxx_Byyy_S3_bf.raw	Stokes V of beam yyy of observation xxxxx

Each file is a sequence of blocks of the following structure:

```
struct block {
  struct header header;

  /* big endian */
  // 2010-09-20 release and later:
  float stokes[SAMPLES|2][SUBBANDS][CHANNELS];

  /*
  // 2010-06-29 release and earlier stored data per subband instead of per
beam:
  fcomplex voltages[BEAMS][CHANNELS][SAMPLES|2][STOKES];
  */

  /*
  // Proposed:
  float stokes[SUBBANDS][CHANNELS];
  // Note that this format is exactly the same as for complex voltages
  */
};
```

Older releases: 2010-09-20:

1. Values of Stokes U and V are multiplied by 1/2
2. filenames ended in -bf.raw instead of _bf.raw

Incoherent Stokes

Incoherent stokes are stored per subband, with one or four stokes per file, using the following naming convention:

Lxxxxx_SByyy_bf.incoherentstokes	Stokes of subband yyy of observation xxxxx

Each file is a sequence of blocks of the following structure:

```
struct block {
  struct header header;

  /* big endian */
  // 2010-10-25 release and later:
  float stokes[STOKES][CHANNELS][SAMPLES|2];

  /*
  // 2010-09-20 release:
  float stokes[STOKES][SAMPLES|2][CHANNELS];

  // 2010-06-29 release and earlier:
  float stokes[CHANNELS][SAMPLES|2][STOKES];
  */
};
```

The order in which the Stokes values are stored is: I, Q, U, V.

Older releases: 2010-09-20:

1. Values of Stokes U and V are multiplied by 1/2
2. filenames ended in -bf.raw instead of _bf.raw
3. data order changed

BFRaw format

Raw station data can be stored in a format called BFRaw. This format is used for debugging purposes and is not a regular observation mode, it takes more manpower to record it. The BFRaw format is recorded below for those who need to access it.

A BFRaw file starts with a file header containing the configuration:

```
struct file_header
{
  // 0x3F8304EC, also determines endianness
```

```
  uint32_t    magic;
  // The number of bits per sample (16)
  uint8_t     bitsPerSample;
  // The number of polarizations (2)
  uint8_t     nrPolarizations;
  // Number of subbands, maximum of 62
  uint16_t    nrSubbands;
  // 155648 (160Mhz) or 196608 (200Mhz)
  uint32_t    nrSamplesPerSubband;
  // Name of the station
  char      station[20];
  // The sample rate: 156250.0 or 195312.5 .. double (number of samples per
second for each subband)
  double   sampleRate;
  // The frequencies within a subband
  double   subbandFrequencies[62];
  // The beam pointing directions (RA, DEC in J2000)
  double   beamDirections[8][2];
  // mapping from subbands to beams (SAPs)
  int16_t     subbandToSAPmapping[62];
  // Padding to circumvent 8-byte alignment
  uint32_t    padding;
};
```

After the file header, there is a series of blocks until the end of file, configured using values from the file header:

```
struct block
  // 0x2913D852
  uint32_t      magic;

  // per-SAP information (up to 8 SAPs can be defined, but typically only 1
is used)

  // number of samples the signal is shifted to align the station beam to
the reference
  // phase center (=Observation.referencePhaseCenter in the parset)
  int32_t       coarseDelayApplied[8];
  // Padding to circumvent 8-byte alignment
  uint8_t       padding[4];

  // the sub-sample delay which still has to be compensated for (in
seconds),
  // at the beginning and at the end of the block
  double     fineDelayRemainingAtBegin[8];
  double     fineDelayRemainingAfterEnd[8];
  // Compatible with TimeStamp class (see below)
  int64_t      time[8];


```

```
  struct marshalledFlags
  {
    // up to 16 ranges of flagged samples within this block
    uint32_t      nrFlagsRanges;
    struct range
    {
      uint32_t    begin; // inclusive
      uint32_t    end;   // exclusive
    } flagsRanges[16];
  } flags[8];

  std::complex<int16_t>
samples[fileHeader.nrSubbands][fileHeader.nrSamplesPerSubband][fileHeader.nr
Polarizations];
};
```

To convert a TimeStamp-compatible int64_t to a C-readable timestamp, use

```
/* clockspeed is in Hz */
int64 nanoseconds = (int64) (timestamp * 1024 * 1e9 / clockspeed);

struct timespec ts;
ts.tv_sec  = nanoseconds / 1000000000ULL;
ts.tv_nsec = nanoseconds % 1000000000ULL;
```

Types and constants

Types

A 'float' is a 32-bit IEEE floating point number. An 'fcomplex' is a complex number defined as

```
struct fcomplex {
  float real;
  float imag;
};
```

Constants

Constants can be computed using the parset file. Below is a translation between the C constants used above and their respective parset keys:

SAMPLES	The number of time samples in a block	OLAP.CNProc.integrationSteps / OLAP.Stokes.integrationSteps
SUBBANDS	The number of subbands (beamlets) specified	len(Oberservation.subbandList)
CHANNELS	The number of channels per subband	Observation.channelsPerSubband

STOKES	The number of stokes calculated (1 or 4)	len(OLAP.Stokes.which)

Useful routines

The following routines might be useful when reading raw OLAP data.

Byte swapping

Needed if you read data on a machine which used a different endianness. Typically, x86 machines (intel, amd) are little-endian, while the rest (sparc, powerpc, including the BlueGene/P) is big-endian.

```
#include <stdint.h> // for uint32_t. On Windows, use UINT32.

uint32_t swap_uint32( uint32_t x )
{
  union {
    char c[4];
    uint32_t i;
  } src,dst;

  src.i = x;
  dst.c[0] = src.c[3];
  dst.c[1] = src.c[2];
  dst.c[2] = src.c[1];
  dst.c[3] = src.c[0];

  return dst.i;
}

/* Do NOT take a float as an argument. An incorrectly read float
   (because it has the wrong endianness) is subject to modification
   by the platform/compiler (normalisation etc). */
float swap_float( char *x )
{
  union {
    char c[4];
    float f;
  } dst;

  dst.c[0] = x[3];
  dst.c[1] = x[2];
  dst.c[2] = x[1];
  dst.c[3] = x[0];

  return dst.f;
}
```

Variable-sized arrays

Since the dimensions of the arrays produced by OLAP depend on the parset, it's handy to have access to arrays with variable size. The easiest way is to use C++ and the boost library (which is often installed by default):

```
#include "boost/multi_array.hpp"

int main() {
  /* create an array of floats with 2 dimensions, and initialise it to have
dimensions [2][3] */
  boost::multi_array<float,2> myarray(boost::extents[2][3]);

  /* getting and setting is the same as with regular C arrays */
  myarray[1][2] = 1.0;

  /* note: &myarray[0][0] (or myarray.origin()) is the address of the first
element, which can be
     used if the full array needs to be read from disk. */

  return 0;
}
```

See also http://www.boost.org/doc/libs/1_43_0/libs/multi_array/doc/user.html

If you need to use C, things become a bit more cumbersome. You need to roll out your own multi-dimensional array, although you'll have to customise your code for each number of dimensions in order to keep your code readable. For example:

```
/* create an array of floats with 2 dimensions, max1 and max2 in size
respectively */
struct myarray {
  float *data;
  unsigned max1,max2;
};

/* return myarray[one][two] */
float get( struct myarray *array, unsigned one, unsigned two )
{
  return *(myarray.data + one * myarray.max2 + two);
}

/* set myarray[one][two] to value */
void set( struct myarray *array, unsigned one, unsigned two, float value )
{
  *(myarray.data + one * myarray.max2 + two) = value;
}

int main() {
  /* create an array of floats */
```

```
  struct array myarray;

  /* allocate the array with dimensions [2][3] */
  myarray.max1 = 2;
  myarray.max2 = 3;
  myarray.data = malloc( myarray.max1 * myarray.max2 * sizeof *myarray );

  /* emulate myarray[1][2] = 1.0 */
  set(&myarray,1,2,1.0);

  /* note: myarray.data is the address of the first element, which can be
used if the full
     array needs to be read from disk. */

  /* free the array */
  free( myarray.data );

  return 0;
}
```

Keep in mind that if you need to switch endianness as well, you first need to read into a char array, and convert it to a float array after reading from disk. This is included in the example below.

Example reading of OLAP data using (minimal) C++ and Boost

The following code reads raw complex voltages from disk.

```
#include "boost/multi_array.hpp"
#include <cstdio>
#include <stdint.h> // for uint32_t. On Windows, use UINT32.

struct header {
  uint32_t sequence_number;
  char padding[508];
};

int is_bigendian() {
  union {
    char c[4];
    uint32_t i;
  } u;

  u.i = 0x12345678;
  return u.c[0] == 0x12;
}

uint32_t swap_uint32( uint32_t x )
{
```

```
  union {
    char c[4];
    uint32_t i;
  } src,dst;

  src.i = x;
  dst.c[0] = src.c[3];
  dst.c[1] = src.c[2];
  dst.c[2] = src.c[1];
  dst.c[3] = src.c[0];

  return dst.i;
}

float swap_float( char *x )
{
  union {
    char c[4];
    float f;
  } dst;

  dst.c[0] = x[3];
  dst.c[1] = x[2];
  dst.c[2] = x[1];
  dst.c[3] = x[0];

  return dst.f;
}

int main()
{
  // example file (60MB!) is available at
  //
http://www.astron.nl/~mol/L09330_B000_S0-example-stokes-I-248-subbands-16-ch
annels-763-samples.raw

  unsigned SUBBANDS = 248;        // |Observation.subbandList|
  unsigned CHANNELS = 16;         // Observation.channelsPerSubband
  unsigned SAMPLES  = 12208 / 16; // OLAP.CNProc.integrationSteps /
OLAP.Stokes.integrationSteps
  unsigned FLOATSPERSAMPLE = 1;   // 1 for Stokes, 2 for Complex Voltages
(real and imaginary parts)

  struct header header;
  int swap_endian = !is_bigendian();

  // the raw_array is read from disk and converted to the float_array
  // the extra dimension [4] covers the size of a float in chars in the
raw_array
  boost::multi_array<char,5>
raw_array(boost::extents[SAMPLES|2][SUBBANDS][CHANNELS][FLOATSPERSAMPLE][4])
```

```
;
  boost::multi_array<float,4>
float_array(boost::extents[SAMPLES|2][SUBBANDS][CHANNELS][FLOATSPERSAMPLE]);

  FILE *f = fopen( "L09330_B000_S0-example-stokes-I-248-subbands-16-
channels-763-samples.raw", "rb" );
  if (!f) {
    puts( "Could not open input file." );
    return 1;
  }

  while( !feof(f) ) {
    // read header
    if( fread( f, &header, sizeof header, 1 ) < 1 )
      break;

    if( swap_endian )
      header.sequence_number = swap_uint32( header.sequence_number );

    printf( "Reading block %u...\n", header.sequence_number );

    // read data
    if( swap_endian ) {
      if( fread( f, raw_array.origin(), raw_array.num_elements(), 1 ) < 1 )
        break;

      // swap all data regardless of array dimensions
      char  *src = raw_array.origin();
      float *dst = float_array.origin();

      for( unsigned i = 0; i < float_array.num_elements(); i++ ) {
        *dst = swap_float( src );
        dst++; src += 4;
      }
    } else
      if( fread( f, float_array.origin(), float_array.num_elements(), 1 ) <
1 )
        break;

    // process block here
  }

  fclose( f );
  return 0;
}
```

Changelog for each release

2010-10-25	Incoherent Stokes data order changed

	File naming scheme changed (-bf → _bf)
	Stokes U and V are no longer multiplied by 1/2
2010-09-20	First release documented