HUDDLER: a multi-language compiler for automatically generated format-specific data drivers

Brian R. Calder[1], and Giuseppe Masetti
Center for Coastal and Ocean Mapping & Joint Hydrographic Center
University of New Hampshire
Durham, NH, USA

Abstract

HUDDL is a data description language, written in XML, designed to simplify and standardize the description of hydrographic data files. In addition to providing a standard way to communicate a data format specification between manufacturer and user, in computer-readable format, HUDDL descriptions can also be used to automatically generate both documentation and source code for the format from the same description document. This paper describes the structure of a HUDDL file description, and demonstrates the language for current and historical binary files, and a current ASCII file, showing both the automatically-generated source-code for the formats described, and the use of high-level (Python) code to access them in a user-friendly manner.

Introduction

The proliferation of data formats for mapping data seems to be inevitable. Trying to keep track of all of the formats in which data can be supplied, their idiosyncrasies, and variations over time can involve a significant amount of effort and developer resources. In addition, the first symptoms of a changed format can often be a data file that fails to read properly, hardly the most graceful change detection method. Further, since there is no commonality in the descriptions provided for different formats, each has to be interpreted separately, and there is no standard method to determine what documentation, or even formats, are available. At the same time, developing data parsing and writing software is both error prone and tedious; it is not even clear that all of the information required to understand a particular format is available, in general, since side information is often required to interpret particular data format revisions.

Unfortunately, attempts to develop a universal format for data files often lead to either a super-set of all possible data, and therefore a complex, difficult to implement technology; or the minimal sub-set of all of the considered data sources, and therefore a potential loss of information from the 'raw' data. The first case leads to complex and difficult data readers, reluctance to adopt the technology, and the potential for variant versions of the format as different implementations interpret the necessarily complex description. The second case harbors the potential for loss of potentially important data, precluding the

[1] Corresponding author. E-mail: brc@ccom.unh.edu, phone: +1 (603) 862-0526.

opportunity for improved processing in the future, and limited archival possibilities. Neither is ideal.

Clearly, therefore, there is potential for improvement. In this paper, in an attempt to tame the complexity of data format specification, and assist in code and documentation development, a method is proposed that allows the manufacturer (or anyone else) to provide a computer-readable (but still human-interpretable) description of a data format with sufficient detail to complete specify both the content and form of the data being written. With sufficient flexibility to encompass most common data format styles, this data description language, written in XML, allows for standardized testing for correctness of the description (i.e., by checking against a standard schema that describes the language), incorporation of arbitrary, and structured, metadata with the description (e.g., so that revision history, authors, contact information, etc. can be supplied), and supports automatic translation of the description into both system documentation and source code in a variety of languages to support data access in the format described. The method is called the Hydrographic Universal Data Description Language (HUDDL).

There are several benefits to this approach. First, automated translation of the description into source code minimizes the effort required to adopt a particular data format, avoids the tedious and error prone development process, and focuses the developers on the more important application logic for what to do with the data afterwards. Having metadata and data within the same file ensures that they cannot be separated, and therefore allows the documentation to be automatically generated from the same document as the source code, ensuring that they are always synchronized. A single-document description of the data also enables ready archival, so that the value of deep-archive data is preserved by ensuring that it can always be read in its original form, simply by providing a metadata link to the HUDDL format description (HFD). Finally, having a computer-readable description makes it easy to provide push-updates when a format changes, informing all interested parties that an update (which should consist of just a recompile) is available.

In the following, the basic structure of the HUDDL description of hydrographic data is outlined, along with a set of illustrative examples that show the use of HUDDL in parsing two contrasting data formats: Kongsberg binary EM-series datagrams [1], and HyPack HSX Format - Hysweep Text (ASCII) data[2]. In both cases, only a minimal amount of user-written code is required to extract detailed information from the files. Along the way, these examples also illustrate the ease of use of HUDDL-driven high-level language bindings (in this case with Python), which go a long way to making direct data access much more available (e.g., for plug-ins to other packages, or for lightweight web applications).

HUDDL is intended to be open and community-led; more details, the core schemas, and examples, are available from the website, https://huddl.ccom.unh.edu.

The HUDDL Description Language

HUDDL is an XML-based description language that contains sufficient information to specify in an unambiguous way, the format and content of a binary or ASCII data file. General format description languages have been attempted in the past [3-7], but have often had difficulties either with maintenance, or complexity, or both [8]. The more structured nature of hydrographic data files makes it possible for HUDDL to be relatively small and simple, while still remaining sufficiently flexible to deal with a range of different data formats. In essence, the core idea is that the manufacturer of a piece of hydrographic equipment, and the user (or, more likely, the software developer that writes the user software) share a common, computer-readable, description of the data being generated. Since this description is written to a particular set of rules (specified through a formal grammar for the language, described in the HUDDL Core Schema (HCS) document available on the project website), there should be minimal opportunity for confusion as to the data being read or written. In addition, the formal structure of the definition allows for computer-mediated manipulations, including the ability to automatically check for validity of a description, and to convert the description into the source code required to manipulate the data, at least at the most basic level. Addition of metadata to the description can allow for more human-readable documentation to be automatically generated (e.g., in the form of an HTML or PDF description after the application of an XML Stylesheet in the form of an XSLT document), which is by definition consistent with the data being generated. Furthermore, format specifications generated by applying the same document template support the retrieval of information required by the reader.

Most hydrographic data formats are similar in nature: a series of data chunks written to disk essentially asynchronously. (Some formats occasionally have a unique header block at the start of each file, which is also accommodated by HUDDL.) Consequently, the core object in an HFD is a 'block' of information, represented by the `huddl:block` element. With a `huddl:block`, each chunk of data written, e.g., a byte, floating-point number, integer of specific size, etc., is defined by a 'field', represented by a `huddl:field` element, which specifies the type of the data, and a reference name by which it is known. The fields are specified in the order in which they appear in the file; a standard set of commonly known types is provided by the HCS, and include signed and unsigned integers with different bit length (from 8-bit to 32-bit currently, with the potential for 64-bit implementations if required), single and double floating point numbers, etc. For more complex situations, a sub-block can be defined, and then used within a 1D or 2D array in another block, allowing for fairly complex data structures to be built up with minimal effort. Since a common feature of hydrographic data files is that there can be variant length arrays (e.g., one sub-record for each valid beam in a ping, which can vary from ping to ping), HUDDL has a mechanism to specify the length of an array based on data appearing previously (in a variety of ways). An example block description is shown in Figure 1.

Data formats consist, of course, of many types of blocks of data, with those that appear at the top-most level having special status. To reflect this, HUDDL defines the concept of a 'stream' which represents all of the types of blocks that may repeatedly appear within a given file. Since different revisions of a data format can have different blocks, and even (unfortunately) different interpretation of the same block, a single HUDDL description can have multiple `huddl:stream` elements (which can share `huddl:block` elements if required) to describe different versions of the same format.

```
<huddl:block name="em96att">
   <huddl:field name="latency" type="huddl:u16">
      <huddl:note>Offset since start of record in ms</huddl:note>
   </huddl:field>
   <huddl:field name="status" type="huddl:u16"/>
   <huddl:field name="roll" type="huddl:s16">
      <huddl:note>In 0.01 deg. steps</huddl:note>
   </huddl:field>
   <huddl:field name="pitch" type="huddl:s16">
      <huddl:note>In 0.01 deg. steps</huddl:note>
   </huddl:field>
   <huddl:field name="heave" type="huddl:s16">
      <huddl:note>In cm steps</huddl:note>
   </huddl:field>
   <huddl:field name="heading" type="huddl:u16">
      <huddl:note>In 0.01 deg. steps</huddl:note>
   </huddl:field>
</huddl:block>

<huddl:block name="em96attitude">
   <huddl:field name="counter" type="huddl:u16"/>
   <huddl:field name="serialnum" type="huddl:u16"/>
   <huddl:field name="numentries" type="huddl:u16"/>
   <huddl:vector1d name="attitude">
      <huddl:blockType>em96att</huddl:blockType>
      <huddl:sizeField>numentries</huddl:sizeField>
   </huddl:vector1d>
   <huddl:field name="motiondesc" type="huddl:u8">
      <huddl:note>Motion sensor description byte</huddl:note>
   </huddl:field>
</huddl:block>
```

Figure 1– Example of `huddl:block` descriptions. The `huddl:block` "em96attitude" carries information related to the attitude sensors. Since a variable number of time samples is possible for each of these blocks, a nested vector of `huddl:block` "em96att" is described. The content of each of these `huddl:block` "em96att" components stores the relevant information: time latency, status, roll, pitch, heave, and heading. Additional semantic information can be added through the use of `huddl:note`, which will be reported in the automatically created documentation.

In order to recognize which `huddl:block` is to be read from the file, most data formats have a recognition number written into a 'header' prior to the data; some formats also have a 'tail' that contains, for example, a checksum. To accommodate this, each

`huddl:stream` allows the user to provide a special `huddl:block` that is to be read prior to each standard block, in which one of the `huddl:field` elements is used as a discriminator (i.e., to indicate what to expect next in the file); a special `huddl:block` to represent the 'tail' can also be provided. To complete the description, each `huddl:stream` has a list of `huddl:topBlock` elements that provide the look-up table that translates discriminator numbers from the header block into `huddl:block` names. An example `huddl:stream` description is shown in Figure 2.

```
<huddl:streams>
   [… previous format specifications releases …]
   <huddl:stream revID="revR" scope="rev19">
      <huddl:header refBlock="emhdr" discriminator="type"/>
      <huddl:topBlocks>
         <huddl:topBlock refBlock="em96attitude" alias="attitude"
                         identifier="0x41"/>
         <huddl:topBlock refBlock="em96bist" alias="bist"
                         identifier="0x42"/>
         <huddl:topBlock refBlock="em96clock" alias="clock"
                         identifier="0x43"/>
         [… additional top blocks as in format specs …]
      </huddl:topBlocks>
      <huddl:tail refBlock="emtail"/>
   </huddl:stream>
</huddl:streams>
```

Figure 2 – Example of a `huddl:stream` description, listed in the `huddl:streams` container. Each stream has an ID and a scope that are used for naming convention in the code generation. Inside the `huddl:stream`, the `huddl:header` and the `huddl:tail` elements provide information relative to the beginning and the end of each `huddl:topBlock`, respectively. The `huddl:topBlocks` section is used to list all the special blocks that can be retrieved at the stream level. Each of these list items have information about the identifier and an alias that is used for the code generation.

As described previously, HUDDL is intended to provide for documentation of the data format as well as the physical description of the contents. Consequently, the language provides a 'prolog' section, implemented by a `huddl:prolog` element, for each format, which allows for definition of various elements such as the organization sponsoring the format, the authors, contact information, a revision history, and general notes. The combination of a `huddl:prolog`, a set of `huddl:block` elements, and one or more `huddl:stream` elements (the latter two wrapped up in a `huddl:content` element) provide the basis for a 'format,' implemented by a `huddl:format` element. The translation of a format into a HUDDL description is usually straightforward; a valid (although of course not complete for the particular format described) HFD is shown on the left pane of Figure 3.

Currently, HUDDL Format Descriptions are generated by hand using an appropriate XML editor (e.g., Altova XMLSpy, oXygen XML Editor, WmHelp XmlPad) that allows for validation of the description files against the XML schemas that define the HUDDL language (the HCSs are available at the project website, https://huddl.ccom.unh.edu). (A simple text editor may also be used, but will generally lack the validation facilities of an XML editor, which are very useful and powerful.) The very structured nature of the language, however, would make it a relatively simple matter to provide custom tools to assist in the construction of HFD, potentially even as a graphical, wizard-guided, process.

Automatic Code Generation

The HUDDLER tool was developed to illustrate one of the key benefits of the HUDDL approach to data format description: automatic code generation. Since HUDDL is written in XML, there are immediate benefits in that the front-end parsing of the language is extremely simple: all that is required is a suitable XML library, which are readily available either stand-alone or within a language or framework library. In addition, since the HUDDL Core Schema is a valid XML schema, testing for syntactic integrity of a HUDDL Format Description against the HCS can be performed as a preliminary step, after which the remainder of the code can simply assume that the description is valid, reducing implementation requirements.

The structured nature of HUDDL, and the presence of full type information within the data format description makes it relatively easy to automatically generate code. In order to assist in adoption, HUDDLER is designed with a generic front-end that deals with parsing the description, and a configurable language-specific back-end that does the code generation for a particular language. This allows the code generator to take advantage of particular language idioms, and provide data structures that are as transparent to the user application as possible. In the current implementation native back-end generators are available for ANSI C using procedural methods, and ISO C++ 11, using object-oriented techniques. An example of C-language generated code is provided on the right panel of Figure 3. HUDDLER's code-generator modules are designed to generate, as much as possible, code that looks as if it was hand-written, relying on the compiler's optimization for cross-platform efficiency, rather than trying to generate 'optimal' code, which might only be useful on specific platforms. This also assists with readability of the resulting code.

Although C/C++ are the primary back-end targets, a prototype native Python back-end has also been developed to demonstrate the potential for other languages to be incorporated. In practice, it is unlikely that reading large hydrographic data files in a high-level language such as Python, JavaScript, Perl, or MATLAB would be efficient enough to be useful for any purpose other than prototyping, demonstration, or teaching. However, scripting languages such as Python, R and Tcl are powerful tools for the construction of flexible scientific software because they provide scientists with an interpreted problem-

solving environment and, at the same time, a modular framework for controlling software components written in C and C++. Consequently, HUDDLER also has the capability to generate SWIG [9] interface files to wrap the native back-end code in C++ so that it can be called from, e.g., Python or MATLAB. This provides the efficiency of highly optimized compiled code, and the ease of use of an interpreted language. The Python interface is illustrated extensively in the following examples.

HUDDLER is implemented as a cross-platform library, and is tested on MS Window 7, Linux Ubuntu and Fedora, and Mac OS X Mavericks. As a library, HUDDLER can be accessed in a number of ways, but to demonstrate its use, a command-line executable is provided. This executable converts the input HFD file and output language choice into appropriate source code files. Several optional parameters can also be passed to the compiler to tune the generated output (e.g., to specify the output name, the set of streams to build, the physical position of the schema to be used for validation purposes, etc.)

Examples

As a first example of the use of a HUDDLER-generated parser, consider the code of Figure 4. Here, a HFD similar to that in Figure 3 has been extended to cover all of the datagrams found in current-generation binary Kongsberg EM Series files, and then converted with HUDDLER into C source code and SWIG wrappers. SWIG is then used to generate the '`kongsberg_em_series`' Python module, which can be readily imported into the user-level Python script as shown. As can be seen in the left pane of Figure 4 the user-level code to read the file is minimal, with most of the work hidden behind the '`read_kng_revR`' method call to the '`kng`' module (which is a call-through to the automatically-generated background C-language code). As is common in this type of task, the code simply repeatedly reads blocks from the stream in a loop, using the automatically generated identification numbers (here '`KNG_REVR_ATTITUDE`') to identify the blocks of interest, and the return codes (here '`HUDDLER_TOPBLOCK_OK`') to check that the read was successful. The auxiliary code in '`read_attitude`' is used to translate the appropriate sections of the block into useful data, which can then be used; in this case simply to generate a quick visual display of the attitude data. This example clearly demonstrates the simplicity, at the user level, that a HUDDL description of a data format provides. Given an HFD for the format, the user's coding overhead to start using the data is on the order of a dozen lines of Python (i.e., not counting the application logic), most of which are formulaic and can be readily adapted from examples provided on the project's website.

Since the HFD describes all of the blocks in a format (and indicates cleanly, without failure, where blocks are not recognized so that the user can implement a forward-compatible reader), another useful but simple application is to check the entire contents of a file for expected blocks. For example, the code in Figure 5 shows how to generate a

histogram of the data blocks in a Kongsberg EM Series data file. The overhead code here is almost identical to that of Figure 4, with the application logic implemented by a simple, dictionary-based, accumulator. Again, the power of the HUDDL approach is evident: the user overhead is minimal, and the application logic very cleanly expressed with respect to the data structures, which mimic the logical description of the file.

HUDDL also provides an inexpensive but robust way to deal with many legacy data formats. When access to old datasets in an arbitrary binary format is required, an *ad hoc* HFD may be created that can deal with the particular vagueness of some legacy formats or some rare variant implementations. As an example, the code in Figure 6 illustrates recovery of position and sound speed profiles from data collected in 1996 with a Kongsberg EM 950 MBES. In this situation, only as much of the data format as required needs to be implemented; data blocks not described will be cleanly ignored by the code generated, with appropriate reporting to the user.

Although HUDDL is expected to be used most often with binary data formats, the language is quite flexible, and can also be used to access ASCII data formats. For example, Figure 7 shows the result of coding a description of the Hypack HSX Format - Hysweep Text (ASCII) data format, where the code is used to parse out the information about slant-range, beam intensity, and data quality indicators for a Reson 8101 system (Figure 8). Of particular interest here is the uniformity of the application logic with respect to the binary file examples shown previously: from the user's point of view, all of the details of how to get the data into the data structures from the file is abstracted by the HFD. All the programmer sees is the converted and filled data structures, which allow for idiomatic language structures to be applied (here to convert from ASCII and generate lists for further processing).

All the above examples have been structured as IPython notebooks, then published (together with their full code content) as blog posts on the HUDDL project website: https://huddl.ccom.unh.edu/?q=blog.

Discussion

The current implementation of HUDDL is capable of supporting a number of different data formats commonly used for hydrographic data. As manufacturers continue to develop new formats, however, there may obviously be some additional features that may need to be added. The HUDDL Core Schema being available from the project website, it is entirely possible that equipment manufacturers themselves could do this, although it is expected that this is likely to be a community-led effort, as indeed it really has to be if the modifications are to be understood sufficiently widely to be useful. There is, however, significant advantage in equipment manufacturers providing HUDDL descriptions of their data formats, both for their own benefit, and that of their customers, as outlined previously.

It is hoped that the HUDDL description of data formats is sufficiently simple to encourage equipment manufacturers to adopt the method natively.

A primary concern in native adoption is, of course, the efficiency of the code generated by HUDDLER, or some other custom application that implements the same idea. The current back-end code generators in HUDDLER are relatively efficient, having been derived from a cruder application that has been in active use for over a decade, but there are inevitably efficiencies that could be derived from a more careful analysis of usage and read/write patterns, and the code generator could be extended to index files and/or be multi-threaded for more modern architectures. Since the specification is open, it is entirely possible that individual manufacturers could build their own applications with these benefits, but there is significant advantage to doing this within a community that orbits around a common code generator.

As described previously, generating HUDDL description files by hand can be a little time consuming. (Although not, of course, on a par with developing all of the software to access a data format by hand!) Since the HUDDL format XML is very heavily structured, however, it would be readily possible to develop an application to assist in the construction method, for example parsing the core schemas and automatically populating a graphical tool that provides all of the appropriate XML elements, in the right formats, at the right locations within the file.

Finally, the current implementation of HUDDL provides what might be described as a 'stone knives and loincloths' description of a data format; that is, it focuses on the lowest level description of the data, and does not provide any higher abstraction mechanism [8]. While this still removes an enormous amount of the tedium implicit in generating the software library to access the data, it still requires the user to provide a significant amount of application logic to translate the data into hydrographically useful information. At the simplest level, for example, a backscatter level coded as an integer in 0.5dB steps would need to be scaled in order to be used for mosaic construction. In many cases, it may be possible to provide relatively simple translation services, with user-specified behaviors, as part of the description of the data (e.g., the provision of simple scale factors for unit conversion), but the general problem is complex. Specifically, it is not entirely clear that the general translation problem can be approached without an equivalent amount of user-specified application logic or without running into the super-set/sub-set problems that have afflicted universal data formats. The extent to which this can be done, and the appropriate methods for the implementation of this idea, are currently under investigation.

Conclusions

The HUDDL data format description language was designed to simplify the issues involved in low-level access to binary (and ASCII) data files. Written in XML, and describing only the

format of the file, rather than the contents, a HUDDL file description provides a stable, standardized, computer-readable but human-understandable mechanism to documenting a file format, which also allows for documentation and supports strong syntax verification against the HUDDL Core Schemas, and automatic code generation for I/O library construction. Automatic code generation provides an economical means to provide new data formats for an application, allowing the programmer time to focus on what to do with the data, rather than how to access it. Multi-language output support and high-level language bindings make HUDDL suitable for both large-data processing codes, and for end-user scientific investigation of new and archive data.

A project like HUDDL works best when supported within a user community. The HUDDL Core Schemas, example format descriptions, and examples of use are provided through the project's website at https://huddl.ccom.unh.edu, with the aim of lowering the barrier to adoption of the technology, and to make it more available. The source code for HUDDLER is also available, although other methods of translating the XML source of a HFD into other forms are certainly possible. The project website can also be used to host HFDs developed by other parties (for example, sonar hardware and software manufacturers). Having a large collection of HFDs in a single location has a significant benefit of simplicity for users attempting to access a particular format. It also allows aggregation of services to provide, for example, push notifications (perhaps via RSS or other services) when an HFD changes.

Possibly the most useful aspect of a HUDDL-based description is its ability to clearly communicate at least the syntax of the format specification between vendors and users in a readily useful form. This allows them to be used as a "trusted" reference (particularly if they derive from a manufacturer), especially for archival data, where they can be attached as metadata. In the future, it might also be possible to include the URI for an HFD (or at least a UUID) as part of a data file so that the format description could be automatically retrieved, translated, and compiled (or, more likely, selected from a pre-compiled cache) when a HUDDL-aware application encounters a format release for the first time.

HUDDL file descriptions have a relatively simple structure and syntax, which can readily be translated from current format documents using nothing more than a standard text editor (although an XML-aware editor is a significant advantage). The description language is, however, flexible, allowing it to be readily extended for new, or variant old, formats – a process that is actively encouraged.

```
<?xml version="1.0" encoding="UTF-8"?>
<huddl:schema version="1.0.0">
  <huddl:format name="Kongsberg EM Series" scope="kng">

    <huddl:prolog>
      <huddl:title>Kongsberg EM Series formats</huddl:title>
      <huddl:organization>
        <huddl:name>Kongsberg Maritime AS</huddl:name>
      </huddl:organization>
      [… additional metadata as history and contact info …]
    </huddl:prolog>

    <huddl:content>
      <huddl:blocks>
        <huddl:block name="em96att">
          <huddl:field name="latency" type="huddl:u16"/>
          <huddl:field name="status" type="huddl:u16"/>
          <huddl:field name="roll" type="huddl:s16">
          [… additional fields as in format specs …]
        </huddl:block>
        <huddl:block name="em96attitude">
          <huddl:field name="counter" type="huddl:u16"/>
          <huddl:field name="serialnum" type="huddl:u16"/>
          <huddl:field name="numentries" type="huddl:u16"/>
          <huddl:vector1d name="attitude">
            <huddl:blockType>em96att</huddl:blockType>
            <huddl:sizeField>numentries</huddl:sizeField>
          </huddl:vector1d>
          [… additional fields as in format specs …]
        </huddl:block>
      </huddl:blocks>

      <huddl:streams>
        <huddl:stream revID="revR" scope="rev19">
          <huddl:header refBlock="emhdr" discriminator="type"/>
          <huddl:topBlocks>
            <huddl:topBlock refBlock="em96attitude"
                            alias="attitude" identifier="0x41"/>
            [… additional top blocks as in format specs …]
          </huddl:topBlocks>
          <huddl:tail refBlock="emtail"/>
        </huddl:stream>
      </huddl:streams>
    </huddl:content>

  </huddl:format>
</huddl:schema>
```

```
/* Huddler - Automatically generated code. NOT manually modify! */
#include "kongsberg_em_series.h"

static bool read_em96attitude( FILE* f, em96attitude_t* data,
                               int* tot_read, int* tot_bytes )
{
  int bytes_read = 0;

  if(fread(&(data->counter), 2, 1, f) != 1) return false;
  bytes_read += 2;
  […additional code to read block fields …]
  for(i = 0; i < (int) data->num_entries; ++i)
    if( !read_em96att(f, data->attitude+i, &bytes_read, tot_bytes) )
      return false;

  bytes_read++;
  *tot_read += bytes_read;
  return true;
}

HdlrErrorType read_kng_revR( FILE* f, kng_revR_t* data,
                             u32* n_read, kng_validator_t vldtr )
{
  bool valid = false;
  int bytes_read = 0, bytes_tot = -1, resynch_distance = 0;
  HdlrErrorType rc;

  while( !valid && (resynch_distance < 1024) ){
    bytes_read = 0;
    if(!read_emhdr(f, &(data->header), &bytes_read, &bytes_tot ))
      if( feof(f) ) { rc = HUDDLER_EOF_WHILE; }
      else { rc = HUDDLER_READ_ERROR; }
      return rc;
  }

  switch( data->header.type ){
    case 0x41:
      if( !read_em96attitude( f, &(data->datagram.attitude),
                              &bytes_read, &bytes_total ) )
         return HUDDLER_TOPBLOCK_READFAIL;
      data->id = (kng_revR_id)0x41;
      break;
    […additional code to read all the topblocks …]
    default:
      data->id = (kng_revR_id)data->header.type;
      return HUDDLER_TOPBLOCK_UNKNOWN;
  }
  *n_read = bytes_read;
  return HUDDLER_TOPBLOCK_OK;
}
```

Figure 3– In the left pane, the part of a HUDDL Format Description that describes the specific blocks containing attitude data and the stream that provides access to them as top-blocks. In the right pane, the code snippets created by HUDDLER to read the specific format release and the top-block containing the attitude measurements.

```python
#!/usr/local/bin/python
import kongsberg_em_series as kng
import numpy as np
import matplotlib

class KngData:
    """ container for Kongsberg attitude data """
    roll = np.zeros(0, dtype=np.float)
    [… create all the required empty numpy arrays …]

def read_attitude(data_header, data_content, kng_data):
    """ read and store a Kongsberg attitude datagram """
    nr_entries = data_content.attitude.numentries
    loc_roll = np.zeros(nr_entries, dtype=np.float)

    for i in range(nr_entries):
        loc_att = kng.em96att_getitem(data_content.attitude.att, i)
        loc_roll[i] = loc_att.roll / 100.0
        [… read the other attitude measures: pitch, heave, etc. …]

    kng_data.roll = np.append(kng_data.roll, loc_roll)

def plot_attitude(kng_data):
    """ plot and save the read Kongsberg attitude data """
    fig, (ax0, ax1, ax2, ax3) = plt.subplots(ncols=1, nrows=4)
    plot0 = ax0.fill_between(kng_data.time, kng_data.roll,
                             linewidth=0.3, facecolor='green')
    ax0.set_title('Roll')
    [… use Matplotlib to plot all the attitude arrays …]

def main():
  kng_data = KngData()
  testfile_path = '0001_20120606_175918_ShipName.all'
  ip = kng.fopen(testfile_path, 'rb')
  [… check if the file was successfully open …]

  abort = False
  while not abort:
     rc, n_read = kng.read_kng_revR(ip, data, n_read,
                                    kng.validate_em96)
     if rc == kng.HUDDLER_TOPBLOCK_OK:
        if data.header.type == kng.KNG_REVR_ATTITUDE:
           read_attitude(data.header, data.datagram, kng_data)
     else:
        abort = True

  plot_attitude(kng_data)
```

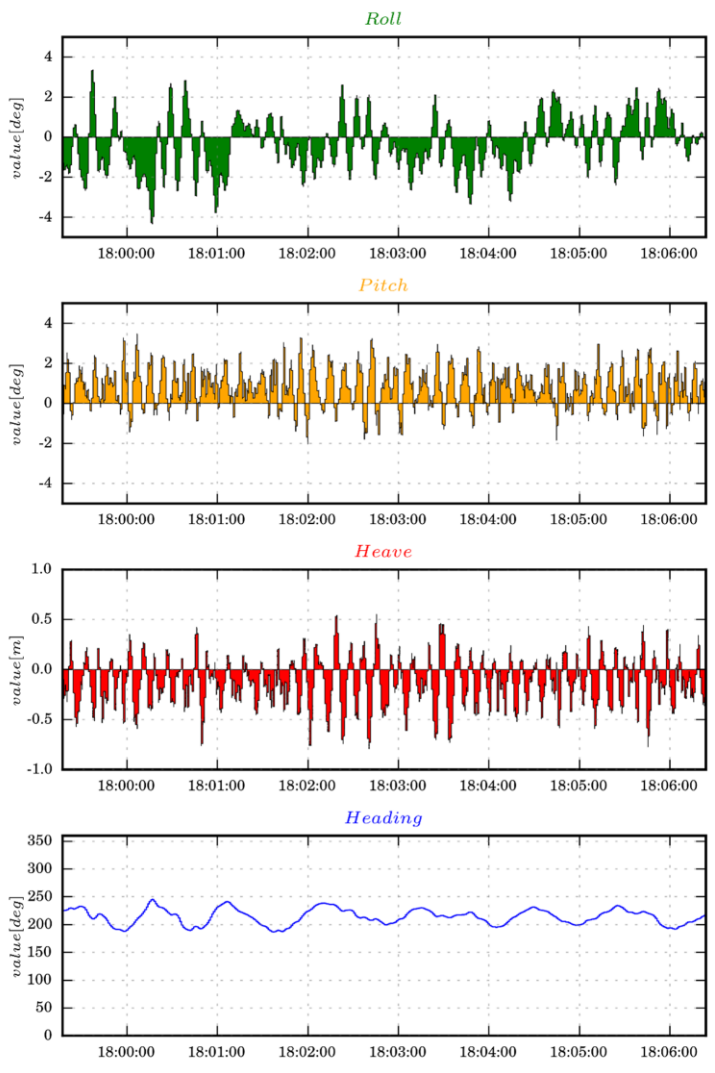

Figure 4 - In the left pane, a simple script that imports the HUDDLER-generated library that provides all the methods to open and access the attitude data. In the right pane, the output generated by the Python script that can be used to quickly inspect the attitude data before manipulation and/or using it in processing algorithms. The full code for this example is accessible online at: https://huddl.ccom.unh.edu/?q=blogs/gmas/plotting-attitude-data-kongsberg-file-python.

```
[… same code skeleton described for the Attitude code …]
datagram_dict = dict()
datagram_nr = 0

def main():
  [… code to open the file and looping through the content …]

  abort = False
  while not abort:
     rc, n_read = kng.read_kng_revR(ip, data, n_read,
                                    kng.validate_em96)
     if rc == kng.HUDDLER_TOPBLOCK_OK:
        datagram_nr += 1
        datagram_dict[data.header.type] =
            datagram_dict.get(data.header.type, 0) + 1)
     else:
        abort = True

  [… plotting code …]
```

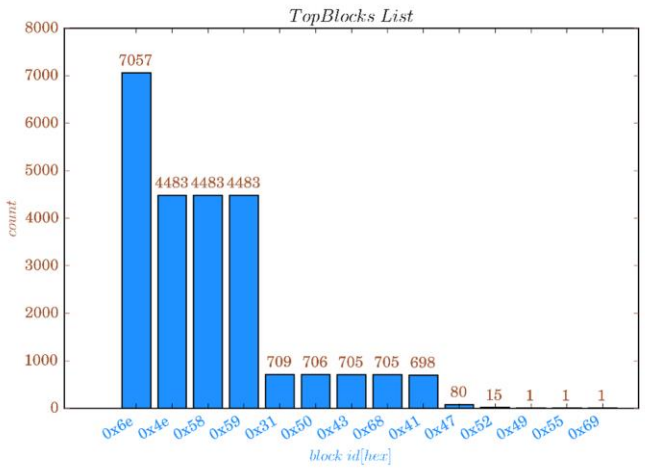

Figure 5 - In the left pane, the code snippets specific to creating a dictionary of `huddl:topBlocks` used to count the total number of datagrams of each type. In the right pane, a bar chart generated to visually display the distribution of `huddl:topBlocks` for the processed file (the x-axis is labeled using the hexadecimal ID value that uniquely identify each Kongsberg datagram). The full code for this example is accessible online at: https://huddl.ccom.unh.edu/?q=blogs/gmas/stream-and-topblocks-exploring-content-data-file.

```python
import kongsberg_em1000 as em1k
import numpy as np
[… some boiler-plate code as import of commonly-used modules …]

class Em1kSsp:
  depths = np.zeros(0, dtype=np.float)
  values = np.zeros(0, dtype=np.float)
class Em1kPos:
  times = np.zeros(0, dtype='datetime64[us]')
  lats = np.zeros(0, dtype=np.float)
  longs = np.zeros(0, dtype=np.float)
class Em1kData:
  ssp_list = list()
  nav = Em1kPos()

def make_time(date, time):
  [… helper function to retrieve date and time …]
def make_fix(lng, ltd):
  [… helper function to retrieve position in degree …]

def read_nav(data_header, data_content, em1k_data):
  """ read and store a Kongsberg nav datagram """

  dt = make_time(data_content.position.date,
                 data_content.position.time)
  long, lat = make_fix(data_content.position.longitude,
                       data_content.position.latitude)
  em1k_data.nav.times = np.append(em1k_data.nav.times, dt)
  em1k_data.nav.longs = np.append(em1k_data.nav.longs, long)
  em1k_data.nav.lats = np.append(em1k_data.nav.lats, lat)

def read_svp(data_header, data_content, em1k_data):
  """ read and store a Kongsberg svp datagram """
  nr_entries = data_content.svp.n_values
  svp = Em1kSsp()
  svp.depths = np.zeros(nr_entries, dtype=np.float)
  svp.values = np.zeros(nr_entries, dtype=np.float)

  for i in range(nr_entries):
    loc_svp = em1k.em91svp_array_getitem(data_content.svp.svp, i)
    svp.depths[i] = loc_svp.depth
    svp.values[i] = loc_svp.speed / 10.0

    em1k_data.ssp_list.append(svp)

[… looping code that calls the helper functions by type …]
[… plotting code …]
```

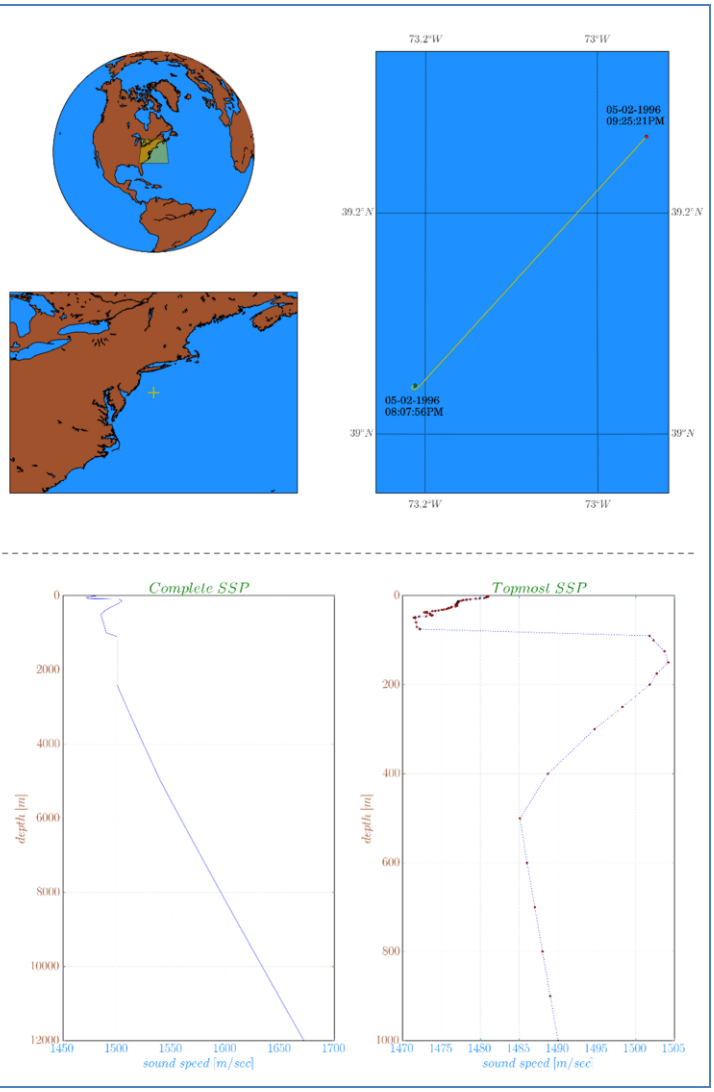

Figure 6 - The left pane shows some code snippets with the data structures and the helper functions used to retrieve both the navigation fixes and the sound speed profile from a legacy Kongsberg data format. In the right pane, several maps showing the retrieved navigation at different scales (upper sub-panel), and the sound speed profile contained in the data file (lower sub panel) plotting all the samples (left) and just the first 1000 meters (right) (The sample density is much higher in the first few meters; as required by the acquisition software, the profile has been artificially extended to 12000 meters). The full code for this example is accessible online at: https://huddl.ccom.unh.edu/?q=blogs/gmas/digging-past-huddl-legacy-data-formats.

```
FTP NEW 2
HSX 6
TND 01:23:45 01/01/2000
INF "J. Doe" "RV Quicky" "Full CVRG" "OpAREA" 0.72 0.00 1423.11
HSP 0.0 70.0 105.0 105.0 65 65 3 1 1.6 0.0 1 1
DEV 0 20 "Hypack Navigation"
DV2 0 14 0 1
OF2 0 0 -5.1 -3.9 -22.3 0.00 0.00 0.00 0.80
PRI 0
DEV 1 32784 "Reson Seabat 81xx (Network)"
DV2 1 9 0 1
OF2 1 3 -4.3 0.0 2.7 3.00 0.20 -2.00 0.00
MBI 1 1 0 1801 101 0 75.000 -1.500

[… hidden ASCII data …]

GYR 2 31988.966 90.75
HCP 3 31988.903 -0.07 -0.24 0.77
HCP 3 31988.936 -0.07 -0.25 0.78
SNR 1 31988.802 11820233 1 4 50 6 15 1
RMB 1 31988.802 1 0 1801 101 1499.00 11820233
78.4 76.6 74.9 75.1 74.9 72.7 67.4 64.2 61.4 59.1 57.3 54.5
52.4 50.4 48.7 47.1 45.8 44.6 43.6 42.8 42.2 41.5 40.9 40.5
40.1 39.7 39.2 39.0 38.7 38.3 38.3 38.0 37.9 37.7 37.6 37.8
37.4 37.4 37.4 37.3 37.4 37.4 37.5 37.6 37.7 38.0 38.2 38.3
38.3 38.6 38.7 38.9 39.3 39.7 39.7 39.9 40.4 40.6 40.9 41.2
41.7 41.8 42.1 42.5 42.7 43.4 44.0 44.9 45.4 46.3 47.2 48.3
49.4 50.6 51.7 53.1 54.5 55.8 57.3 58.9 60.6 62.4 64.1 66.1
68.4 71.0 73.9 76.9 79.7 82.8 86.9 91.4 96.1 100.9 105.1 111.0
118.3 127.2 139.2 150.3 162.8
30720 1072 928 1088 1520 1424 1904 2048 2240 2192 4656 2256
2864 4736 3280 3008 4640 3152 2400 2752 3472 3888 5936 3312
4256 5792 1920 3504 4432 4304 10032 7696 4112 5376 6080 5632
1616 2560 3424 9376 3696 10544 10672 6400 6256 3744 2784 4640
3376 4960 6512 4592 3792 3024 2976 3840 3008 2736 4656 3360
2272 4288 4960 4464 3472 3488 3136 2784 2912 3584 6656 4032
4976 2528 2864 4016 4016 2688 2176 1936 1184 1552 1696 688 608
784 1104 864 800 768 912 1136 1168 1152 1024 960 576 736 784
1648 1136
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 3 3 2 2 3 3 3 3 3 3 2
2 2 3 3 3
RSS 1 31989.027 0 1000 1000 1499.00 11820234 36.7 14990 0 4096
0 0

[… hidden ASCII data …]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<huddl:schema version="1.0.0">
  <huddl:format name="HYPACK HSX" scope="hsx">

    <huddl:prolog>
      <huddl:title>HYPACK, Inc. HSX Format</huddl:title>
      <huddl:organization>
        <huddl:name>HYPACK, Inc.</huddl:name>
      </huddl:organization>
      […additional metadata as history and contact info …]
    </huddl:prolog>

    <huddl:content>
      <huddl:blocks>
        <huddl:block name="hsx_rmb_content">
          <huddl:delimited1d name="data" type="huddl:u8"
                             delimiter="\n"/>
          […additional fields as in format specs …]
        </huddl:block>
        <huddl:block name="hsx_rmb">
          <huddl:delimited1d name="dn" type="huddl:u8"
                             delimiter=" "/>
          <huddl:delimited1d name="t" type="huddl:u8"
                             delimiter=" "/>
          […additional fields as in format specs …]
          <huddl:vector1d name="content">
            <huddl:blockType>hsx_rmb_content</huddl:blockType>
            <huddl:sizeField>bd</huddl:sizeField>
            <huddl:hexStringBitCount/>
          </huddl:vector1d>
        </huddl:block>
      </huddl:blocks>

      <huddl:streams>
        <huddl:stream revID="Rev8" scope="rev8">
          <huddl:header refBlock="hdx_hdr" discriminator="type"/>
          <huddl:topBlocks>
            <huddl:topBlock refBlock="hsx_rmb"
                            alias="RMB" identifier="0x20424D52"/>
            […additional top blocks as in format specs …]
          </huddl:topBlocks>
        </huddl:stream>
      </huddl:streams>
    </huddl:content>

  </huddl:format>
</huddl:schema>
```

Figure 7 – In the left pane, a (partial) example of the content of an Hypack HSX Format - Hysweep Text (ASCII) format, with the specific ping (section starting with '`RMB`' id) that will be read and plotted in this example. In the right pane, the specific HFD code snippet that describes the corresponding `huddl:block` '`hsx_rmb`'. This block lists several fields using the `huddl:delimited1d` data structure (used to read delimited text), and a variable-length vector of nested `huddl:blocks` '`hsx_rmb_content`'.

```
import hypack_hsx as hsx
import numpy as np
[… some boiler-plate code as import of commonly-used modules …]

class Em1kSsp:
  depths = np.zeros(0, dtype=np.float)
  values = np.zeros(0, dtype=np.float)
class Em1kPos:
  times = np.zeros(0, dtype='datetime64[us]')
  lats = np.zeros(0, dtype=np.float)
  longs = np.zeros(0, dtype=np.float)
class Em1kData:
  ssp_list = list()
  nav = Em1kPos()

def read_beams(data_header, content, hsx_d):
    """ read and store a Hypack RMB beams data """
    new_ping = HsxMbPing()
    slr_str = hsx.content_getitem(content.RMB.content, 0).data
    new_ping.slr = np.array(list(map(float, slr_str.split())))
    lvl_str = hsx.content_getitem(content.RMB.content, 1).data
    new_ping.lvl = np.array(list(map(int, lvl_str.split())))
    qlt_str = hsx.content_getitem(content.RMB.content, 2).data
    new_ping.qlt = np.array(list(map(int, qlt_str.split())))
    hsx_d.mb.pings.append(new_ping)

def read_rmb(data_header, content, hsx_d):
    """ read and store a Hypack RMB topblock """
    time_mn = hsx_data.info.date +
              datetime.timedelta(seconds=float(content.RMB.t))
    hsx_d.mb.time = np.append(hsx_data.mb.time, time_mn)
    hsx_d.mb.dn = np.append(hsx_d.mb.dn, int(content.RMB.dn))
    hsx_d.mb.st = np.append(hsx_d.mb.st, int(content.RMB.st))
    hsx_d.mb.sf = np.append(hsx_d.mb.sf, int(content.RMB.sf))
    hsx_d.mb.bd = np.append(hsx_d.mb.bd, int(content.RMB.bd))
    hsx_d.mb.n = np.append(hsx_data.mb.n, int(content.RMB.n))
    hsx_d.mb.sv = np.append(hsx_d.mb.sv, float(content.RMB.sv))
    hsx_d.mb.pn = np.append(hsx_d.mb.pn, int(content.RMB.pn))
    read_beams(data_header, content, hsx_d)

[… looping code that calls the helper functions by type …]
[… plotting code …]
```

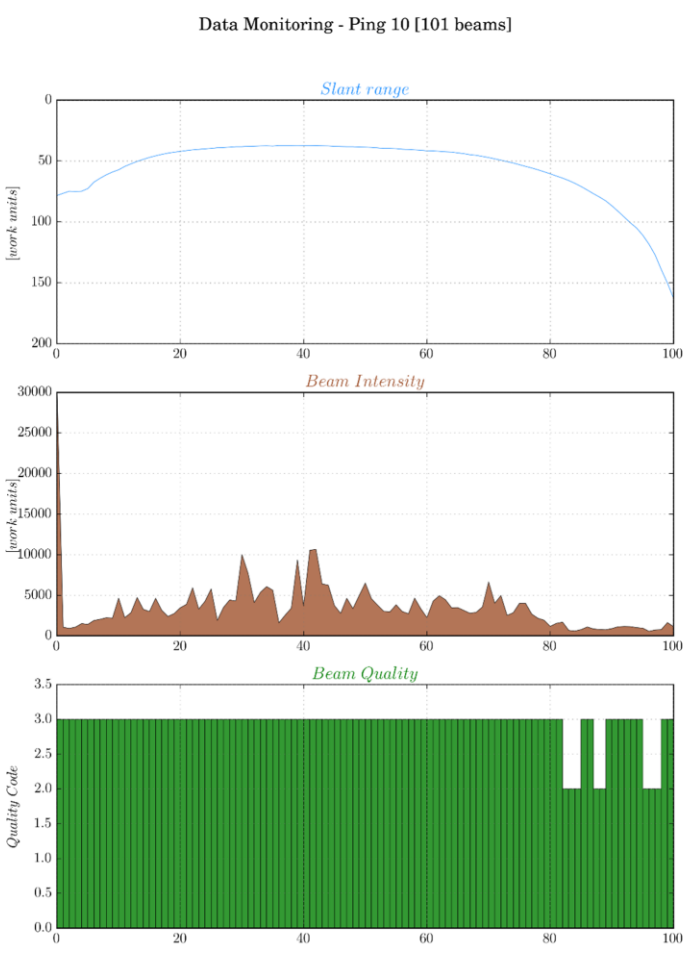

Figure 8 - The left pane shows some code snippets with the data structures and the helper functions used to retrieve the bathymetric content from the RMB (Raw Range Multibeam) top-blocks in a survey file stored in Hypack HSX Format - Hysweep Text (ASCII) data. Since the slant range, the intensity, and the quality information are stored as nested vectors in a top-block, we create a helper function '`read_beams`' that is called from the parent method '`read_rmb`' to read this information for all the beams in each ping. In the right pane, the slant range (top), the intensity level (middle), and the quality flag (bottom) for a given ping are visualized. The full code for this example is accessible online at: https://huddl.ccom.unh.edu/?q=blogs/gmas/dealing-ascii-data-formats.

References

[1] Kongsberg Maritime AS, "EM Series Datagram formats - Instruction Manual," October 2013 ed, 2013, p. 128.

[2] HYPACK, "Hypack Hydrographic Display Software - User Manual," January 2013 ed: HYPACK, Inc., 2013.

[3] R. Ramachandran, S. Graves, H. Conover, and K. Moe, "Earth Science Markup Language (ESML): a solution for scientific data-application interoperability problem," *Computers & Geosciences,* vol. 30, pp. 117-124, 2004.

[4] A. W. Powell, M. J. Beckerle, and S. M. Hanson, "Data Format Description Language (DFDL) v1. 0 Specification," 2011.

[5] M. Westhead and M. Bull, "Representing Scientific Data on the Grid with BinX–Binary XML Description Language," *EPCC, University of Edinburgh,* 2003.

[6] K. Varda. (2008, 02/16/2014). *Protocol Buffers: Google's Data Interchange Format*. Available: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[7] V. Y. Lum, N. C. Shu, and B. C. Housel, "A General Methodology for Data Conversion and Restructuring," *IBM Journal of Research and Development,* vol. 20, pp. 483-497, 1976.

[8] G. Masetti and B. Calder, "HUDDL for description and archive of hydrographic binary data," in *Canadian Hydrographic Conference*, St. John's, NL (Canada), 2014, p. 24.

[9] D. M. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems,* vol. 19, pp. 599-609, 2003.