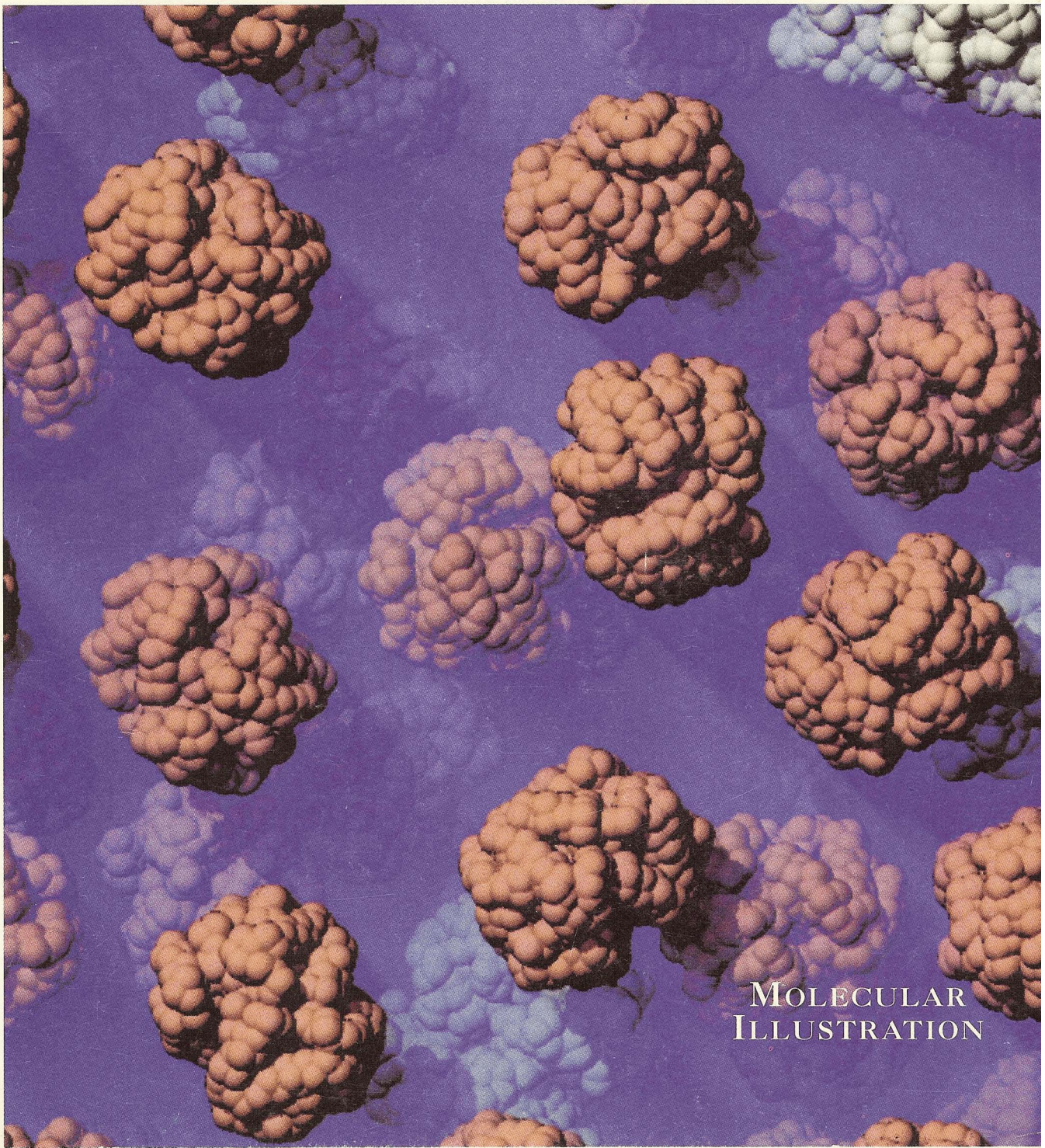
March/April 1991
$7.00

pixel

The Magazine of Scientific Visualization



Universal Data Access for Time Series Analysis

by Ben Dubrovsky

Before analysts can gain insight from recorded data, they must first gain access. They must struggle with data recorded in inconsistent formats, from different computers, in different files, and usually recorded asynchronously, at different sample rates, and fraught with dropouts and problems with time-code generators. The battle has generally been waged in one of two ways—either building custom analysis tools, or preprocessing data into a form prescribed by an off-the-shelf analysis package. Neither of these solutions, however, is optimal for today's world. What is needed is a tool that will access recorded data in whatever form it happens to exist. In order to do this, we must examine the similarities between all data formats, and exploit those similarities to develop a generic data file.

There are several basic types of recording formats to consider: time-multiplexed, messages, tag/data, telemetry frame. With time-multiplexed data the same set of data samples are recorded one after the other. Time may be recorded as a variable, or may be determined by a constant recording rate. Message data consists of several messages recorded sequentially. Each message may have a different internal structure, but all message data have a common header. The header generally includes a time stamp, an ID indicating what type of message follows, and an indication of how long the message is. With tag/data each data value is preceded by a tag number that identifies it. Time values have their own specific tags. Where any particular datum occurs, including time, it appears completely asynchronous. A telemetry frame consists of a major frame, which contains some number of minor frames. Data recorded at different sample rates may appear in different sub-sets of minor frames. Each minor frame has the same length. Major frames are recorded one after another at a constant rate.

Data analysis systems have two components—data access and data presentation. In order to break down the access process into discrete tasks, we must begin to define and exploit some similarities between recorded time-series data files:

The recorded data is time dependent. Somehow, time is encoded in the file, and there is some deterministic way of decoding it.

Data are streamed into the file. As data are gathered, they are recorded into the system. The data recorded later in the test are stored later in the file.

We define a frame as the set of data recorded at one particular point in time.

We define a variable to be a particular datum recorded inside a frame.

We can thus break the process of understanding a particular data format into the following three steps:

1. Understand the overall recording format of the data stream. Where are frame boundaries? How is time decoded? How do we navigate from one frame to the next?

2. Given a frame, understand how it is structured, and how to find and unpack specific variables from within that particular frame.

3. Given a specific variable from within a frame, calibrate or transform the variable into engineering unit (EU) form.

The first step in connecting to recorded data is the accessing of an arbitrary file structure, or recording format. The data stream is divided into several generic components—file, file header, records, record header, frame, frame header, frame information (time, length, ID, data). Note that each of these components may exist in a particular data set, but it is not necessary for all to exist. Thus, the hierarchy of the model is that a file may contain a number of records, including some header records; a record may or may not have a header and contains some number of frames; each frame may or may not have a frame header, and can be identified by a frame time, frame ID, and frame length. Once we have a model for how a generic file is organized, we can develop a software architecture to decode that structure.

When I was working at Bolt Beranek and Newman Inc. in Cambridge, Massachusetts, we did in fact develop such a program, which we called The Flexible File Server (FFS). It works in concert with a table-driven dictionary, and a general purpose function evaluator to form the basis of BBN/Probe, an advanced data visualization package. The Flexible File Server is a layer of software

that responds to requests for data frames by returning a pointer to a frame of data somewhere in the data file. The requests to the FFS may be thought of as having the form, "Find me the first frame, at or after time *t*, whose frame ID is in this list...". The list is the set of all frame ID's in which each requested variable may appear. At this point, the calling subroutine does not care how the frame is found. It is the responsibility of the FFS to find the frame that fits the given criteria. Since the FFS responds to requests for data based on a specific time, the issues of data being asynchronous and recorded at different sample rates become moot. The FFS simply finds the first frame, at or after a given time. Thus, the FFS performs its function of returning a pointer to a frame of particular ID at a particular time by making an ordered series of subroutine calls to these functions. The basic algorithm used to locate frames in a data stream is shown in Figure 1.

This algorithm is modified to recognize when a frame at a given time and of a specific ID is found, and to stop searching at that point. As records are read, information about them is kept in internal maps. This facilitates finding records and frames more quickly once they have been read the first time. The search for a frame is controlled by the algorithm shown in Figure 2.

```
open the file;
initialize the file;
for (every record in the file)
{
     read in a record from the file;
     determine the record's length;
     find the first frame in the record;
     while (more frames exist in the record)
     {
          unpack frame ID;
          unpack frame length;
          unpack frame time;
          find the next frame in the record;
     }
}
```

Figure 1.

```
if (looking for a previously seen time) {
     search known records for appropriate time;
     read in the old record;
     return pointer to known frame;
}
else {
     position to the end of file;
     while (more records exist in file) {
          read in a record from the file;
          determine the record's length;
          find the first frame in the record;
          while (more frames exist in the record) {
               unpack frame id;
               unpack frame length;
               unpack frame time;
               find the next frame in the record;
     }
}
```

Figure 2.

Notice that we have defined an approach, and a common data file format, without making specific references to any one format—the algorithms above are generic. We define specific software responsibilities and interfaces for each of the above subroutines. A suite of subroutines is provided to handle the most common cases needed for each of these functions—for example, decoding time tags written in VAX floating point form, or decoding frame IDs that are recorded as integers inside a frame.

It is impossible to provide a subroutine (or subroutines) general enough to handle all of the possible cases for all formats. We permit the system, therefore, to be extended by dynamically linking in, at run time, subroutines to replace one or many of the built-in functions. As an example, if a particular data format contains a time tag that is not understood by the suite of built-in subroutines, then a separate module may be written to perform just the task of decoding that time tag. In this way, the system becomes extensible enough to accomodate virtually any recording format.

Dynamic linking of subroutines also allows data-specific checks for time clitches, bad data frames, or data out of synch. Since we are under program control when inside these subroutines, we can tailor them to do highly specific checking that would not be done inside a general-purpose software system. For instance, we can check to see if time tags are out of order. If they are, we can choose to treat the errant frame as an anomaly,

and discard it, or, we can choose to adjust the time for purposes of analysis, providing that necessary adjustment is smaller than some set value. Again, once we are under program control, the choices become virtually endless.

DATA DICTIONARY

Once we are able to access and navigate through a raw data file, the next step is to access and recover individual recorded variables from frames of data. This can be accomplished by using a table-driven dictionary. It allows access to variables by symbolic name, relieving end users from having to know where particular data may be recorded. The data dictionary stores translations between symbolic names and positions inside particular data frames. The dictionary maintains the following major categories of information for referencing the recorded data: frame ID and mask, position (which set of bits—identified by start word, start bit, and field length—in the frame must be recovered to generate the particular data channel, allowing for the datum to be split into two pieces. Also indicate whether the datum is bit-, byte- or word-reversed), storage type, data class, array information, conditional information, scaling information.

Using the information stored in such a dictionary, we can unpack any data recorded in a frame in a known position in a given format.

ENGINEERING UNIT CONVERSION

Once data are found in a given data stream, and unpacked, they often must be converted to engineering unit form to be useful. One way of accomplishing this is to use a general purpose function evaluator built in to the visual data analysis package. The function evaluator has the capability to transform a particular variable by an interpreted mathematical function. The evaluator should be able to deal with asynchronous and aperiodic signals, automatically resampling data when necessary.

For instance, if a dictionary entry exists for a variable named `X_RAW`, we can calibrate that variable by defining a function, such as:

$$X = a * X\_RAW^5 + b * X\_RAW^4 + c * X\_RAW^3 + d * X\_RAW^2 + e * X\_RAW + f$$

The function `X` may now be used in all standard analyses. In addition, a wide variety of built-in functions is provided representing trigonometric, signal processing, and mathematical operations. The function evaluator may also be extended by providing a capability to manipulate signals in an external function written in a higher-level language. This facility allows for different types of calibration such as table look-up, and for custom tailored analysis.

Using a general-purpose architecture, like the one described in this article, minimizes the amount of time necessary to get a new data set online for analysis. The FFS and dictionary are the only components that need to be re-configured for new formats. None of the analysis software itself needs to be changed. And, as there is usually no software coding involved in the process, programmer time can be minimized.

The difficulty of dealing with multiple recording formats will never go away completely. The savings that come from providing universal access to data, however, make the search for solutions worthwhile.

Ben Dubrovsky has an S. M. in computer science from Harvard University. He wrote the Flexible File Server program when he was at Bolt Beranek and Newman, Inc. He is now developing multi-media and interactive computer video systems at The Chedd-Angier Production Co. in Watertown, Massachusetts.