EFFICIENT DATA STORAGE, SAMPLING, AND RETRIEVAL BY LEVERAGING OPEN SOURCE TECHNOLOGIES

Micah Ferrill
Avionics Test & Analysis Corporation
4000 Eagle Point Corporate Drive
Birmingham, AL 35242
ferrillm@avtest.com

ABSTRACT

This paper demonstrates the use of open-source software tools to manage large data sets.

Advances in technology have greatly reduced the cost of data storage and processing systems. The ability to handle large amounts of data efficiently while retaining fine-grain control of the data retrieval process becomes a challenge. In particular, traditional data processing applications are inadequate to handle the large data sets typically encountered in IRIG-106 Chapter 10[1] data files.

We answer this challenge by using readily available, open-source tools that efficiently store and retrieve IRIG-106 Chapter 10 data to/from a file-based database.

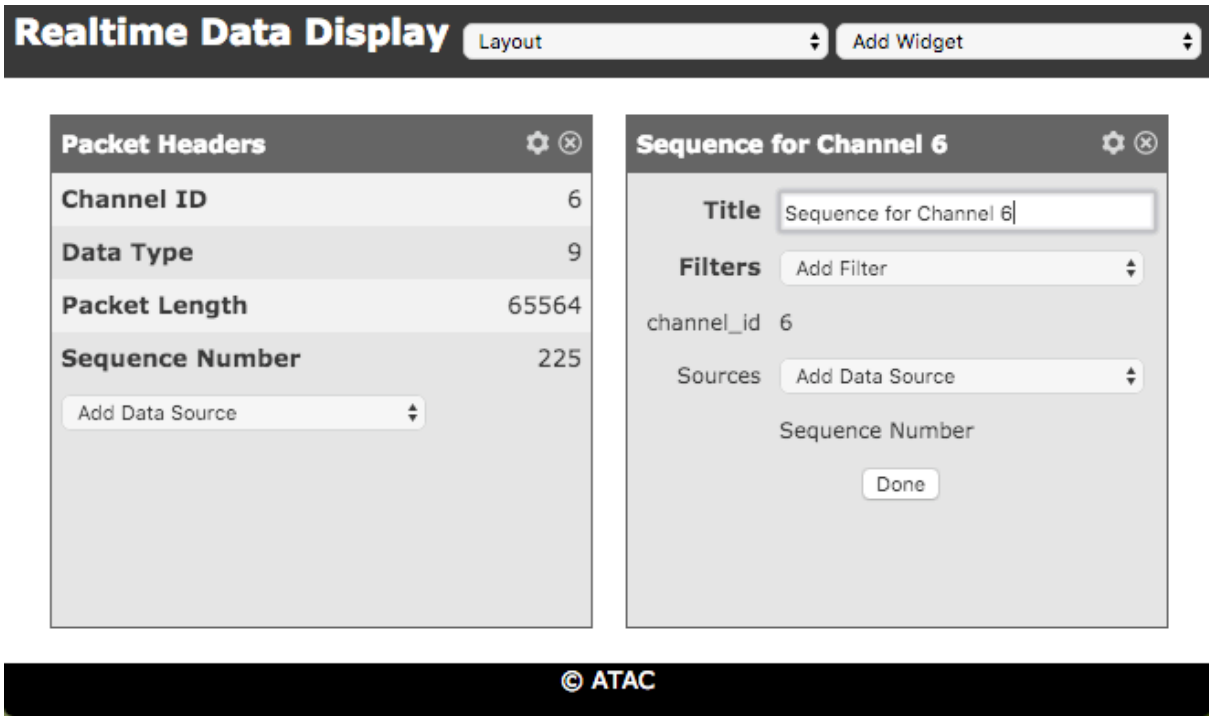
We will demonstrate a method that facilitates a separation between the parsing of raw input data and the display of desired information at a user-defined sample rate. This open-source based solution provides a low-cost, reliable, and efficient means for handling large amounts of data at a high rate of speed.

INTRODUCTION

The problem we have to solve is displaying data from large collections of IRIG 106 Chapter 10 data potentially spanning multiple files. The issue that arises, of course, is that parsing multiple large, binary data files simultaneously can be a very demanding process and potentially limits us from real-time playback. Eventually, distributed data storage and processing may hold an answer, but for some uses this is much too heavy of a solution, and a system capable of running on a laptop would be more applicable.

Our intent is to showcase how we solve this problem while leveraging free and open-source software. By reusing previous work done in this space and incorporating various mature and well-vetted open source tools, we have built a solution that will facilitate real-time playback on common hardware while also architecting to facilitate the future leveraging of modern distributed systems design.

EXISTING WORK



Channel ID	6
Data Type	9
Packet Length	65564
Sequence Number	225

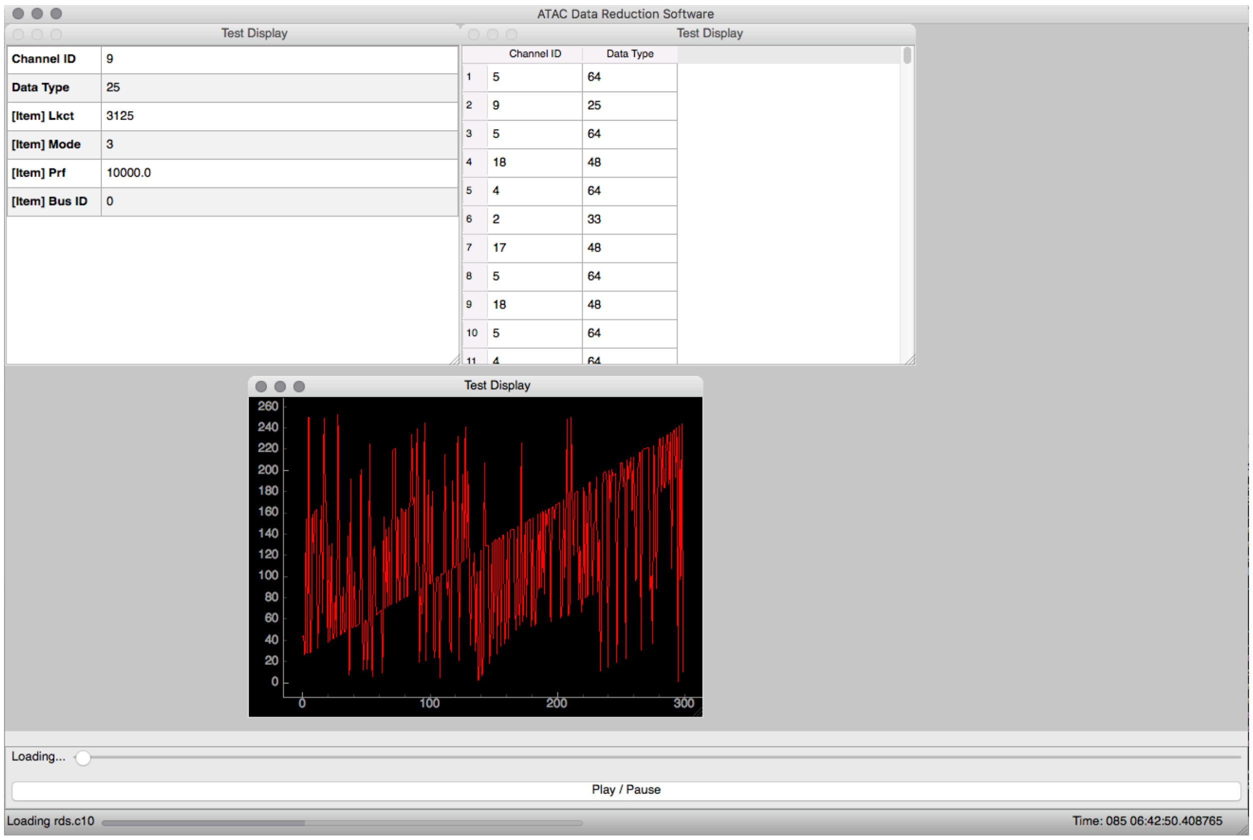
Title: Sequence for Channel 6
Filters
channel_id 6
Sources
Sequence Number

Figure 1 Lightweight Display Tool Demo

In previous years at ITC, we have demonstrated various iterations of a lightweight data display tool using a browser-based user interface. Much of the backend data processing and storage mechanism could be easily repurposed to jumpstart our new effort. One of our existing software libraries used for the data processing is a Chapter 10 parsing library written in Python[2], which was also reused for this project.

OPEN SOURCE

The Python programming language has been a favorite of ours for some time, not least because of the speed with which tools can be developed. The language also can be extended using C/C++ to enhance performance while maintaining the easy to use language characteristics of Python.



Channel ID	9
Data Type	25
[Item] Lkct	3125
[Item] Mode	3
[Item] Prf	10000.0
[Item] Bus ID	0

	Channel ID	Data Type
1	5	64
2	9	25
3	5	64
4	18	48
5	4	64
6	2	33
7	17	48
8	5	64
9	18	48
10	5	64
11	4	64

Loading rds.c10 Time: 085 06:42:50.408765

Figure 2 Draft Qt Native Interface (macOS)

We also built a cross platform OS-native user interface for the lightweight display tool by utilizing the Qt windowing toolkit[3] which has seen use in a wide array of successful projects. Qt is particularly useful to us in that it's cross-platform and written in C++, but it also has a number of Python wrappers. This allows us to prototype rapidly in Python while still having the flexibility to rewrite elements using C++ if faster performance is needed.

The SQLite database engine[4] has been the go-to embedded database option for quite some time. It is ubiquitous across all of the major browsers, smartphones, and desktop operating systems. SQLite can be used for temporary in-memory databases, or file-based data stores. With many similarities to other database engines, SQLite is also an excellent choice for prototyping with its simplicity of implementation. It also comes embedded into Python by default, making it ideal for the initial stages of this project.

THE SOLUTION

Our initial solution consists of 3 parts:

1. The loader module
2. Data displays and configuration (within the new Qt-based UI)
3. Self-adjusting update rate

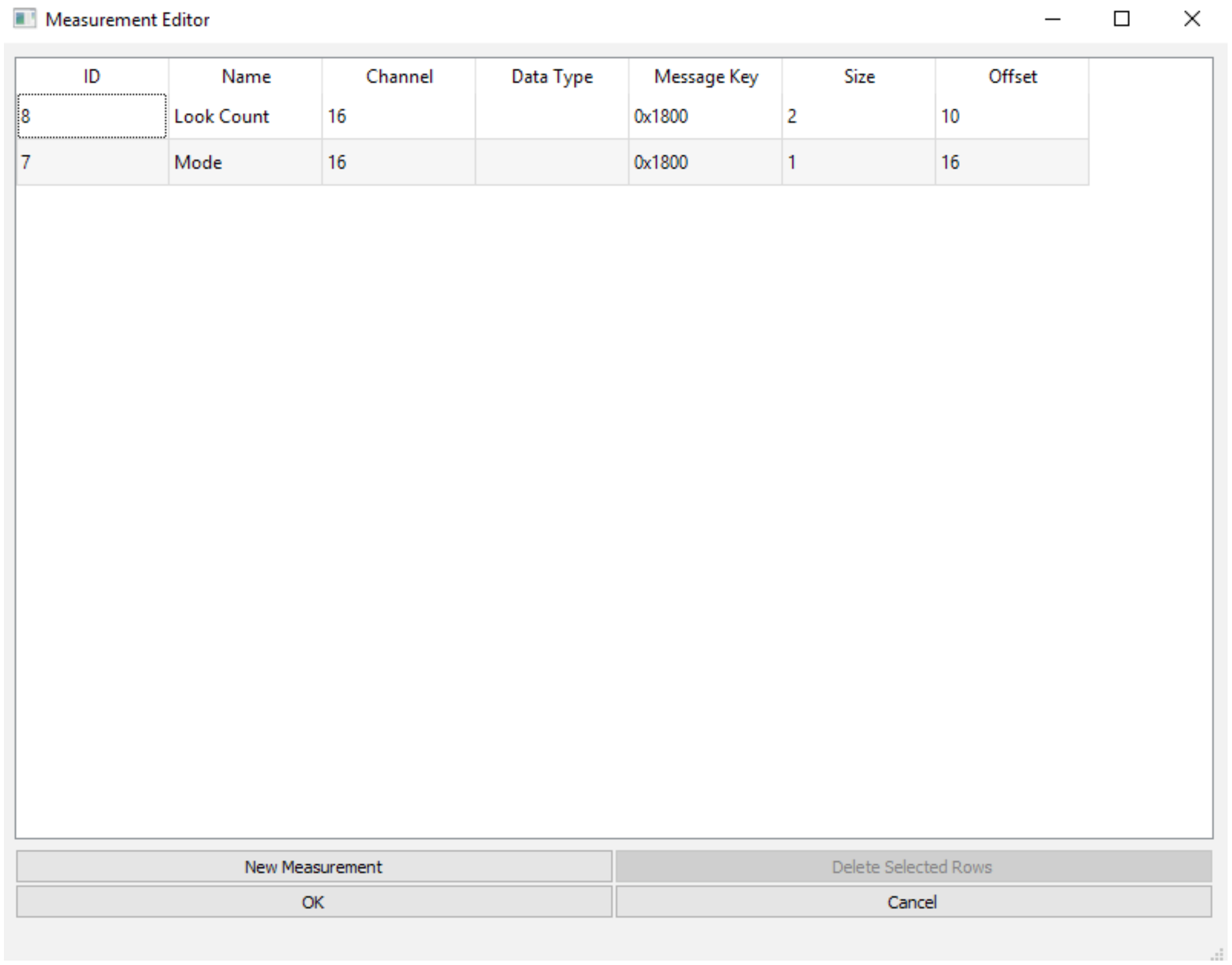
Firstly, the loader module takes chapter 10 data from file[s] or a UDP stream, parses the Chapter 10 structure and stores packets in a SQLite database along with absolute time (computed based on RTC and time packets), channel ID, and data type. This allows us to reference information within the database by time as well as filter at the packet level if desired based on measurements of interest.

The database interface structure, while simplistic in appearance, has a number of interesting benefits. For one thing, the interface (as the loader or display see it) has been abstracted away from the specific database back-end. By having a module that provides a common interface to the various components we can centralize any database optimizations to that module as well as having the ability to change database back-ends without having to rewrite the entire application.

This technique has already seen a lot of use in Python and web development in the ORM (object-relational mapper) world. In some cases, not only the database interface, but the schema definition is abstracted away from the database engine into code. Using this technique one can quickly develop a solution on a lightweight database such as SQLite, and then deploy the same code to a production service using a more complex database system such as MySQL or Postgres by simply changing the database configuration. Part of the reason we borrowed from this approach is to facilitate future support for distributed data storage and computing.

Secondly, in a module separate from the loader is the new Qt display interface. From here, the user may configure and manage measurement definitions consisting of:

1. Measurement name – for human-readable reference
2. Packet description - channel ID and/or data type if appropriate
3. Message description – Mil-Std-1553 command word, ARINC 429 label, etc.
4. Size and offset of measurement data within specified message



ID	Name	Channel	Data Type	Message Key	Size	Offset
8	Look Count	16		0x1800	2	10
7	Mode	16		0x1800	1	16

Figure 3 Measurement Editor (Windows 10)

Finally, the last piece of the puzzle is an auto-adjusting sample rate for display updates. When playback starts, the first packet is loaded and its absolute time tracked. The next update asks for the first packet later than that timestamp. To ensure real-time performance of the overall process we then take the duration of an update, and add that to the time offset for the next update. Thus, when an update is of a duration that would put us out of sync we “skip” a few packets to remain synced to the clock. By utilizing this method, we can then focus on optimizing individual displays without having to worry about the overall application falling out of sync.

FUTURE DEVELOPMENT PLANS

Eventually, the measurement editor interface will be expanded with import functions for TMATS or CSV. Other formats may be considered in the future. File-level filtering may also be a consideration to handle conflicting channel IDs.

Derived parameters are also an important consideration, and we plan to solve that problem in two ways: firstly, by making common transforms readily available through the measurement editor, and secondly by providing a scripting engine for complex derived measurements.

As referred to above, we are also exploring distributed computing and storage options for the future. By architecting a certain level of abstraction away from the specific database engine used, and having a distinction between the loading and displaying processes we have striven for a future-leaning design.

We hope to use these techniques and technologies well into the future to provide efficient, low-cost solutions with rapid development turn-around.

REFERENCES

[1] IRIG 106 Chapter 10 – irig106.org
[2] Python - python.org
[3] Qt - qt.io
[4] SQLite – sqlite.org