SANDIA REPORT
SAND2023-07387
Printed August 2023

Sandia National Laboratories

Thrifty Array Format (TAF) file specifications

Daniel H. Dolan
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1189

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

Thrifty Array Foram (TAF) files store numeric data in a binary format, minimizing storage requirements while preserving quick read access. Real data of any size and dimensionality can be stored in this format at varying degrees of numeric precision. Implicit array are associated with each dimension, eliminating the need to explicitly store uniformly-spaced grid vectors. Unlimited text comments may be included with the array for user documentation, and every file begins with a text synopsis of the binary structure. The format is deliberately designed for memory mapping, where portions of the array can be read without loading the entire file at once.

ACKNOWLEDGEMENTS

Thomas Gardiner provided critical advice for the final design of the binary file format described here. Thomas Hartsfield helped prepare this report and served as a guinea pig.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Data storage is a recurring challenge in experimental science, where numerical information must be stored, transferred, and processed in a practical manner. Text files are grossly inefficient for large datasets, and the innumerable number of binary file formats are mutually incompatible. Electrical digitizers are a notorious example: each vendor uses their own binary format with extensions such as *.bin, *.trc, and *.wfm. One motivation of the SMASH toolbox [1] is to make this cacophony of binary formats accessible in MATLAB. [2]

The Hierarchical Data Format version 5 (HDF5 [3]) addresses many of the above challenges, but its flexibility is both a blessing and a curse. Virtually anything can be written to an *.h5 file, meaning that one must be prepared to read almost anything from that file. Subsets of HDF5, such as the Sandia Data Archive (SDA), [4] define specific file organization, but several problems remain.

- Direct file access is largely opaque—operations must be handled through library routines. These libraries are available for virtually all computer platforms/languages but have a reputation for being slow.
- Information cannot be efficiently removed from an HDF5 file. For example, overwriting a large data set with a smaller data set does not reduce file size. The standard trick—creating a new file and copying it over the source file—is inefficient for large data sets.
- Variable-size arrays and lossless compression—key benefits of using HDF5— require chunked datasets that cannot be easily mapped to memory.

This report describes a binary format specifically designed to avoid these shortcomings.

1.1. Motivation

Consider an 8-bit digitizer that has acquired 1 billion samples; at present, this is neither a hypothetical nor particularly extreme signal. The minimum storage requirement for that signal is 8 billon bytes (about 7.45 GB). Representing the signal as floating-point values increases storage requirements by 4–8×, or 30 GB for single precision (32 bits per point) and more than 60 GB (64 bits per point) for double precision. Storing the complete time base doubles storage requirements, with 1 billion samples requiring over 120 GB.

The irony is that many forms of signal analysis do not require the entire dataset at any one time. For example, short time Fourier transforms (STFTs) use only local regions for each calculation, but processing can be burdened by having the entire signal in memory. Memory mapping—where array elements link directly to specific file locations—is optimal for such analysis. However, that feature cannot be implemented in many HDF5 files and is difficult to manage across different vendor binary formats.

Table 2-1 Thrifty Array Format structure with decimal offsets/sizes

Byte offset	Description
0	Magic string
8	Format synopsis text
1024	Data type (8 characters)
1032	Data mapping intercept a (double)
1040	Data mapping slope b (double)
1048	Number of array dimensions $N \geq 2$ (uint64)
$Q_1 \equiv 1056$	Dimension 1 length L_1 (uint64)
$Q_1 + 8$	Dimension 1 grid start u_1 (double)
$Q_1 + 16$	Dimension 1 grid step Δ_1 (double)
...	...
$Q_1 + 24(N-1)$	Dimension N length L_N (uint64)
$Q_1 + 24(N-1) + 8$	Dimension N grid start u_N (double)
$Q_1 + 24(N-1) + 16$	Dimension N grid step Δ_N (double)
$Q_2 \equiv Q_1 + 24N$	Data array (M bytes per element)
$Q_3 = Q_1 + M\prod_{k=1}^{N} L_k$	Optional text comments

1.2. Overview

The Thrifty Array Format (TAF) was explicitly designed for memory-mapped analysis of large data sets. Efficiency and transparency are important secondary goals.

- Floating point numbers may be stored at different precision levels, avoiding the 2–8× size increase when stored integers are converted to single/double precision. Files can be similar in size to a vendor binary but much more accessible.

- Implicit vectors eliminate the need to store uniformly spaced grids, further reducing file size. Simple grid modifications are easily applied without loading the array into memory.

- Every file contains enough documentation for the data array to be read without additional information.

The file extension *.taf should always be used to indicate this format.

The following discussion is divided into two primary sections. Section 2 describes specifications for files using the *.taf extension. Example *.taf utilities from the SMASH toolbox are given in Section 3; similar features could be implemented in any language because the format is not specific to MATLAB. A brief report summary is given in Section 5.

```
This file contains a numeric array in little-endian binary format. Starting
at byte offset 1024 is a binary header. The first value:
    Data format (8 characters)
indicates how array elements are stored in the file. Short names and bit size
indicate standard integer (uint8, uint16, etc.) or floating point (flt32, or
flt64) data types. Null characters (ASCII 0) are used on the right side as
placeholders.

The next two values:
    Data intercept (64-bit float)
    Data slope (64-bit float)
define linear mapping of stored integers (x) to array data (y).
    y=intercept+slope*x
Infinite slope indicates data mapping is *not* used.

The next value and subsequent 3N values define the array size.
    Dimensions N >= 2 (uint64)
    Dimension 1 length (uint64)
    Dimension 1 grid start (64-bit float)
    Dimension 1 grid step (64-bit float)
    ...
Array data is stored column wise immediately after the dimension N grid step.

The remainder of the file holds optional text comments.
```

Figure 2-1 TAF opening text, which starts at byte offset 8. Long lines are wrapped here for visual clarity

2. FORMAT SPECIFICATIONS

Every *.taf file is organized into four blocks: a (mostly) text opening, a binary header, the binary data, and optional text comments. Table 2-1 summarizes the details given below.

2.1. Opening

An eight-bit magic string denotes the beginning of a *.taf file. The first four characters are "`TAF `" (ASCII characters 84, 65, 70, and 32), followed by three unsigned 8-bit integers. The first two integers are major and minor version numbers, and the third integer is the array type code.[1] Note that these three numbers are often outside the printable range of ASCII characters. The magic string ends with a newline character (ASCII 10).

Figure 2-1 shows the text starting at file byte offset 8. This synopsis contains sufficient information for the file to be read in any computer language. One important detail stated in the synopsis is that multibyte values use the IEEE little-endian convention (least significant byte first). Space characters (ASCII 32) are written after the synopsis through file byte 1023.

[1]The default type code 0 indicates a generic array. Values 1 through 255 are reserved for future applications.

2.2. Data header

The data header starting at byte 1024 describes the stored array. Every header entry uses eight byte storage as described below. There are *always* four header entries plus three additional entries per array dimension.

The first header entry indicates how array elements are stored. Data type is denoted by eight characters denoting storage type and size, padded with ASCII 0 characters on the right. Signed (`intX`) and unsigned (`uintX`) integers using X=8/16/32/64 bits consistent with the C99 standard [5] are supported; common examples are `uint8` and `uint16`. Single-precision (`flt32`) and double-precision (`flt64`) floating point values consistent with IEEE 754 [6] are also supported. Legacy *.taf files have a different convention, where the first eight header bytes are interpreted as an unsigned 64-bit integer.[2]

The next two header entries contain the intercept a and slope b needed for mapping stored values x to array elements y.

$$y = a + bx \tag{1}$$

Linear mapping is used when floating point values are stored in an integer format, saving file space for data with limited resolution. For example, a signal digitized with 8 bits might be shifted and scaled to represent voltage, but storing that information as floating point values leads to files 4–8$\times$ larger than necessary. Linear mapping is disabled for infinite intercept and slope ($a = b = \infty$), *i.e.* numeric values are read exactly as they were stored in the file. Infinity is represented by the hexadecimal value `0x7ff0000000000000` according to the IEEE 754 standard.

The fourth byte of the data header indicates the array dimensionality. This value N must be greater than one to distinguish column vectors ($L \times 1$) from row vectors ($1 \times L$); $N = 2$ in both cases. The remaining $24N$ header bytes describe each array dimensions in three-value groups.

- The first value indicates the array length L_k along dimension k.
- The next value is the implicit grid start u_k.
- The final value is the implicit grid step Δ_k.

The implicit uniform grids for each dimension span:

$$v_{ki} = u_k + (i-1)\Delta_k \tag{2}$$

where $i = 1 \ldots L_k$.

[2] Only four data types are present in legacy files: value=8 denotes uint8; value=16 denotes uint16; value=32 denotes single precision; and value=64 denotes double precision. This obsolete convention should not be used in new *.taf files.

2.3. Binary data

The data array is stored immediately after the data header, starting at byte offset $Q_1 = 1056 + 24N$. Array elements use the binary format described in the previous section, mapping floating values to integers as needed. The type size X (`intX`, `uintX`, or `fltX`) dictates the number of bytes $M = X/8$ per data element; for example, eight bytes are used for double precision (64 bit) arrays. A total of:

$$M \prod_{k=1}^{N} L_k \tag{3}$$

bytes are used to store the complete array.

The array is written in column-wise manner, following subscript conventions used by MATLAB (and Fortran). A $L_1 \times L_2$ matrix is stored in sequential groups of L_1 values: first column 1, then column 2, and so forth through column L_2. In this two-dimensional example, array element (m,n) is binary value $(m-1)L_2 + n$. The 3×2 array:

```
1 2 3
4 5 6
```

would be stored as the column vector $[1\ 4\ 2\ 5\ 3\ 6]^T$ within a *.taf file. Array transposes may be needed when this array is read into a row-wise language such as C.

2.4. Optional text comments

The remainder of the file is reserved for optional text comments. Starting at byte offset:

$$Q_2 = 1024 + 24N + M \prod_{k=1}^{N} L_k \tag{4}$$

ASCII characters may be used to document the file's content. Newline characters (ASCII 10) are the recommended separator between comments.

3. WORKING WITH *.TAF FILES

Figures 3-1–3-2 show minimal TAF file readers for MATLAB and Python; the former uses C-syntax and can be easily converted to C/C++ by the reader. In both causes, TAF files are opened with read access, moving the position indicator to byte offset 1024. Eight characters are read to determine data type, stripping off white space and converting floating point abbreviations to MATLAB convention. Two 64-bit reads are then performed for linear mapping intercept and slope; recall that these may be NaN values.

The next 64-bit read establishes array dimensionality N. Three values (size, start, and step) are then read for each dimension. A cell array of grid values is generated from this information:

Table 3-1 TAF file operations

Operation	Notes
addComment	Add new file comment
adjust	Adjust an implicit grid
append	Append data to the array
backup	Create backup file
convert	Convert vendor binary
create	Create file from numeric array
crop	Crop array using grid bounds
export	Export data to text file
map	Map file to memory
permute	Permute array dimensions
plot	Plot array columns
probe	Query file contents
read	Read data/grid from file
replace	Replace numeric array in existing file
setComment	Revise file comment(s)
summarize	Statistically summarize file
transpose	Swap rows/columns in an existing file

`vector{1}` are grid points for the first dimension, `vector{2}` are grid points for the second dimension, and so forth.

Stored data is initially read as a one-dimensional array based on the established data type; all remaining bytes (if any) are read as comment characters. Linear mapping is applied to the stored data as needed, and the array is reshaped to the stored dimension size.

The `ArrayFile` class of the `SMASH.ThriftyAnalysis` package provides more general support for *.taf files. Table 3-1 summarizes operations provided by that class, some of which are described in the remainder of this section.

3.1. Creating files

The abstract `ArrayFile` class provides several ways of creating *.taf files. The static create method writes a specified data array to file.

```
object=ArrayFile.create(file,data,format);
```

The optional input “format” defaults to ’single’, *i.e.* data is written as floating point values with 32 bits. Higher resolution s

Smaller bit values (8 or 16) yield smaller files at the expense of resolution; double precision is maintained for for 64 bits. Non-finite values, such as `inf` and `nan`, are preserved for 32/64 bits and clipped for 8/16 bits.

```
function [data,vector,comment]=readTAF(file)

% open file and start reading the header
fid=fopen(file,'r');
fseek(fid,1024,'bof');
atype=transpose(fread(fid,8,'*char'));
atype=deblank(atype);
switch atype
   case 'flt32'
      atype='single';
   case 'flt64'
      atype='double';
end

intercept=fread(fid,1,'double');
slope=fread(fid,1,'double');

% determine array size and create grid vectors
N=fread(fid,1,'uint64');
L=nan(1,N);
vector=cell(1,N);
for n=1:N
   L(n)=fread(fid,1,'uint64');
   start=fread(fid,1,'double');
   step=fread(fid,1,'double');
   vector{n}=start+step*(0:L(n)-1);
end

% read array
LT=prod(L);
data=fread(fid,LT,atype);
comment=transpose(fread(fid,inf','*char'));
fclose(fid);

% convert array type and size
if slope ~= 0
   data=intercept+slope*cast(data,'double');
end
data=reshape(data,L);

end
```

Figure 3-1 Minimal TAF file reader for MATLAB

```
# basic TAF access
def probe(file): # returns header information as dict
   import numpy as np
   report=dict();
   dt=np.dtype([('Format','S8'),('Intercept','<d'),('Slope','<d'),\
            ('Dimensions',np.uint64)]);
   k=1024;
   buffer=np.fromfile(file,offset=k,count=1,dtype=dt);
   DataFormat=buffer['Format'].tobytes().decode();
   DataFormat=DataFormat.replace('\x00','');
   if 'flt' in DataFormat:
      DataFormat=DataFormat.replace('flt','float');
   report['Format']=DataFormat;
   report['Intercept']=buffer['Intercept'].item();
   report['Slope']=buffer['Slope'].item();
   report['Dimensions']=buffer['Dimensions'].item();
   dt=np.dtype([('Length',np.uint64),('Start','<d'),('Step','<d')]);
   k=k+32;
   buffer=np.fromfile(file,offset=k,count=report['Dimensions'],dtype=dt);
   report['Length']=buffer['Length'];
   report['Start']=buffer['Start'];
   report['Step']=buffer['Step'];
   report['DataOffset']=np.uint64(k+3*report['Dimensions']*8);
   report['Points']=np.uint64(report['Length'].prod());
   return report

def read(file): # returns data as numpy array
   import numpy as np
   report=probe(file);
   dt=np.dtype([('Raw',report['Format'])]);
   buffer=np.fromfile(file,offset=report['DataOffset'],dtype=dt,\
                  count=report['Points']);
   buffer=buffer['Raw'];
   if np.isfinite(report['Intercept']) & np.isfinite(report['Slope']):
      buffer=buffer.astype(np.float64);
      buffer=report['Intercept']+report['Slope']*buffer;
   buffer=buffer.reshape(report['Length'],order='F'); # Fortran order
   return buffer
```

Figure 3-2 Minimal TAF file reader for python

The create method also creates several example *.taf files.

```
ArrayFile.create(2); % 2D example
ArrayFile.create(3); % 3D example
```

The two-dimensional example contains a three-column matrix of values that increase linearly, quadratically, and cubically in time. The three-dimensional example contains the same information along with three sinusoids evaluated on the same time base. The example array sizes are 101×3 and $101 \times 3 \times 2$, respectively.

The static convert method transforms vendor binary files to TAF.

```
ArrayFile.convert(source,format);
```

The input "source" can be an individual file or wild card pattern for batch conversion. Converted files use the same base name as their source, *e.g.* name.wfm becomes name.taf. Valid binary formats include:

- 'column' for text files, where numeric data is delimited by white space with an optional text header.
- 'dig' for NTS *.dig files.
- 'lecroy' for Lecroy *.trc files.
- 'keysight' for Keysight/Agilent *.bin and *.h5 files. Multi-record files are automatically converted to individual signal files: name-1.taf, name-2.taf, and so forth.
- 'tektronix' for Tektronix *.isf and *.wfm files.

Source files are *not* modified or deleted by conversion.

3.2. Reading files

The static probe method accesses information about an existing *.taf file.

```
info=ArrayFile.probe(file)
```

The output structure contains information from the data header, all text comments, and various byte offsets. The static summarize method:

```
report=ArrayFile.summarize(files);
```

provides a statistical summary of the data array.

The static read method reads the data array of an existing file.

```
[data,grid]=ArrayFile.read(file);
```

The stored data is returned as a double precision array. The second output is cell array of uniformly-spaced grid vectors.

The static map method provides direct memory access to an existing file.

```
[map,scaleFcn]=ArrayFile.map(file);
```

Raw data can be read from the file `map.Data.Raw` property, which accepts array index or subscript arguments. Values returned by that property have the same format as the stored data. The `scaleFcn` function converts stored data to double precision format.

3.3. Modifying files

The most basic modification to a *.taf file is addition of a text comment.

```
ArrayFile.addComment(file,entry)
```

New entries are automatically appended with a newline character to distinguish them from previous comments. The setComment method:

```
ArrayFile.setComment(file,value)
```

overwrites all exiting comments with the specified value.

The static adjust method provides several types of file modification.

```
ArrayFile.adjust(file,dimension,operation,value);
```

Modifications are performed one dimension (1..*N*) at a time. Valid operations include:

- 'shift' adds the specified value to the implicit grid.
- 'scale' multiples the input grid by the specified value.
- 'start' replaces implicit grid start with the specified value.
- 'step' replaces the implicit grid spacing by the specified value.
- 'span' makes the implicit grid cover the specified [min max] range.

Arrays can be cropped one dimension at a time.

```
ArrayFile.crop(file,dimension,grid,index);
```

The inputs “grid” and “index” indicate cropping bounds. The former references the implicit grid for the requested dimension, *e.g.* a grid bound of [-inf 0] retains array elements with grid values less than or equal to zero. Index bounds explicitly specify elements to be retained, *e.g.* [1 10] keeps the first ten values. Empty bounds may be used as placeholders to retain all array elements.

4. OTHER OPERATIONS

Users may find the following methods useful. Refer to the class help for additional details.

- The backup method creates copies of an existing file. This is usually done prior to irreversible modifications such as crop.
- The append method adds data along the *final* array dimension. For example, a $M \times L$ array ($L \geq q$) can be appended to an $M \times N$ array, resulting in a $M \times (N+L)$ array. Array dimensionality is not changed in the process.
- The permute and transpose methods alter the shape an array without change the total number of elements. These methods can be used in conjunction with the append method to control where new data is added to an existing array.
- The export method generates a text file from an existing TAF file.
- The plot method generates lines from each column of a TAF file.
- The replace method overwrites an existing array with new data of the same size, leaving implicit grid values and comments unchanged.

Note that the transpose, export, and plot methods only support two-dimensional arrays.

5. SUMMARY AND WORKAROUNDS

Thrifty Array Format (*.taf) files store numerical data in a compact, accessible manner. Using a simple data header, arrays of arbitrary dimensionality and size are stored in a binary format amenable to memory mapping. Partial information can be rapidly extracted from the file without reading the entire array into memory. These features are designed for experimental data, such as digitizer measurements, but can useful for any large dataset.

Although TAF does not provide explicit compression, type mapping and implicit gridding allow the format to rival or exceed HDF5 deflate performance. Digitizer signals, for example, start out as integers but are almost immediately converted to floating point values for analysis. However, measured data rarely has more than 8–16 effective bits, so floating-point storage is 2–8$\times$ larger than the underlying precision supports; naively storing uniformly-spaced time values increase storage size by another factor of two. HDF5 deflate can partially compensate for the 4–16$\times$ increase, but only with appropriate chunking and significant write speed reduction.

Complex arrays are not directly supported in TAF, but real and imaginary components can always be written as separate elements, *e.g.* a $M \times 1$ complex vector would be stored as $M \times 2$ with real values in the first column and imaginary values in the second column. Sparse matrices must also be converted for use in TAF, either by conversion to a full matrix or a three-column array ([row column value]).

REFERENCES

[1] D.H. Dolan and T. Ao. The Sandia Matlab AnalysiS Hierarchy (SMASH) package. Technical Report SAND2016-6848, Sandia National Laboratories, (2016).

[2] MATLAB software, The Mathworks Inc., Massachusetts, United States.

[3] The HDF Group. Hierarchical Data Format, version 5, (1997-2022). https://www.hdfgroup.org/HDF5/.

[4] D.H. Dolan and T. Ao. Sandia Data Archive (SDA) file specifications. Technical Report SAND2015-1118, Sandia National Laboratories, (2015).

[5] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December (2011).

[6] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* , 1–84 (2019).

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.