W-2 PSEIS, Meta-Data Dictionaries

*RANDALL L. SELZLER
RSelzler@Data-Warp.com

Abstract: The Parallel Seismic Earth Imaging System (PSEIS[1]) is a software package that is envisioned as a successor to DDS[2]. Like DDS, meta-data is kept in a plain text dictionary. PSEIS dictionaries also support functionality similar to a Unix file system, but without the overhead. It leverages a familiar and powerful concept for parameter management. A simple, intuitive and familiar syntax is used to encode and decode nested dictionaries within a regular Unix file. This capabilities makes the script interface for power users more convenient and flexible.

Introduction: Dictionaries are used for meta-data, such as processing histories, parameters and binary format descriptions. Dictionaries are part of the interface for end users and programmers. They are encouraged to examine the contents using their favorite text editor. Power users can manually change the contents, if they are careful. This is practical because the text is in a separate file, isolated from the binary data and because it uses a simple syntax. This dictionary concept was borrowed from SEPlib[3] (history files).

Syntax: The dictionary syntax used by DDS only supports a *flat* structure, which makes it difficult to encode and decode some relationships. The syntax used by PSEIS (Figure 1) supports a *hierarchy*. Dictionaries and definitions are analogous to directories and regular files. The new syntax also improves support for comments and definition histories.

The syntax for a definition is a *name*, followed by "=" and then the *value*. The syntax is intuitive, simple and flexible. It is not cluttered with elaborate markup constructs (XML).

3 Way View:

Regular file named "test".

Definition of "test ", value is Vienna.

Dictionary named "test", containing Name, Muf, Bar, Ding, Dong.

Contents of regular file "/home/joe/**test**".

```
Vienna { # Open brace begins "test" Dictionary.
  Name= Value # Optional comment begins with #.
  . . .
  Muf= Austria { # Begin "Muf" dictionary.
    Foo= 1 # Nested definition.
    Bar= 2 # Relative name is "Muf/Bar and
           # full is "/home/joe/test/Muf/Bar".
  } # End Muf dictionary.
  . . .
  Bar= 3.14 # Same name, different name space.
  . . .
  Ding= EAGE 2006 # The one and only Ding.
  . . .
  Dong= oldest value # Dong[3] value.
  Dong= old value    # Dong[2] value.
  Dong= new value    # Dong[1] value.
  Dong= newest value # Dong[0] and Dong value.
  . . .
}  # Close brace ends "test" dictionary.
```

Figure 1: Example dictionary illustrating new syntax.

The value is any sequence of printable characters (except for special ones). It continues across lines until the next definition begins, a nested dictionary is encountered or the enclosing dictionary ends. Special characters within the value must be escaped by prefixing them with "\". These include =, #, $, {, },", [,] and \.

Comments begin with "#" and continue until the end of line. Nothing within them needs to be escaped. They are primarily intended for people, not software. Comments and escape sequences are normally removed from the text before the value is seen by an application.

Applications are free to interpret the value any way they choose. By convention, all white space is equivalent, unless enclosed in double quotes. Quotes are needed for strings that contain white space. New lines within quotes may be ignored by prefixing them with "\".

Nested Dictionary: A definition's value may be followed by one brace pair ("{…}"). If present, it opens the scope for a *nested dictionary*. Each dictionary has a unique name space.

Dictionaries may be nested to any level. The *base dictionary* (which may be a regular file or directory) and its parent are identified by the reserved names "." and "..". The path through Unix directories, dictionary files and nested dictionaries is delimited by "/".

Definition names are relative to a base dictionary, unless they begin with "/". A base is analogous to a current working directory. The base and name are passed to dictionary functions. An absolute base and/or name begins with "/".

```
. . .
axis= x  y  { # Value is list of axis names.
  x= { # Attributes for x axis.
    size= 96
    origin= 0.0
    delta= 50.0
    units= ft
  }
  y= { # Attributes for y axis.
    size= 24
    origin= 1000.0
    Delta= 75.0
    Units= ft
  }
} # End of axis dictionary.
. . .
```

Figure 2: Example value with attributes.

Unlike Unix file systems, each definition can have a value (analogous to a regular file) *and/or* a nested dictionary (analogous to a directory). Nested dictionaries can be used for attributes that further qualify the value. Figure 2 illustrates how they are used to describe data set dimensions and their attributes. Nesting is also a natural way to describe the structure of fields within I/O records.

Names: Definition names may be any sequence of non-white space characters, except =, #, $, {, }, ", [,] and \. By convention, an "=" *immediately* follows the definition name.

History: Processing histories (Figure 3) remember all jobs and steps that have been applied to a data set. The details of each step are isolated in their own nested dictionary. Each is named "`step`". The most recent version can be reference using the name "`step`" or "`step[0]`". Older versions can be referenced using successively larger index values in the C array notation. For example, "`step[0]/args/out`" yields a value of "`bar`", an index of 1 yields "`foo`" and a larger index would yield an empty value.

Functions are provided to count the number of definitions within a dictionary and retrieve their names. Applications can use this information to coordinate work with previous processing steps. For example, the "`equalize`" step could retrieve the "`gain`" used by the "`agc`" step and take appropriate action.

```
step= agc { date=… cwd=… args=… {
    gain= 2
    . . .
  }
}
step= filt { date=… cwd=… args=… {
    out= foo
    . . .
  }
}
step= equalize { date=… cwd=… args=… {
    in= foo
    out= bar
    . . .
  }
}
. . .
```

Figure 3: Illustration of processing histories.

Discussion: Unix file systems do not have a mechanism to retain previous versions of directories and regular files.

Dictionaries do not support the Unix concept of binary data, user IDs, permissions, dates, links and special devices. Dictionaries are kept in plain text within one file. They leverage the concept of a Unix file system, while avoiding overhead that would be required to keep small bits of meta-data in separate regular files.

Dictionary files are convenient for organizing related information. Power users can cut definitions from old histories and paste them into new job scripts. Parameter files can be created or dissected with standard Unix tools such as grep, gawk and sed. This leverages existing software and investments in training.

Conclusion: Data handling is fundamental. Get it right and other issues will fall into place!

Acknowledgment: I'm grateful to the DDS user community within BP and many colleagues from the Amoco Research days for their support. Thanks are also due BP America for releasing the DDS software and for permission to publish this paper.

References:

[1] PSEIS http://PSEIS.org
[2] DDS http://FreeUSP.org/DDS/index.html
[3] SEPlib http://sepwww.stanford.edu