1) Overview

The Framer package is a set of programs for extracting and processing data from files where data is stored in concatenated frames. Specifically, files in which each frame is appended with a StorX data-logger 7 byte binary timestamp.

The programs are intended to be used together but they may be used independently and in various sequences. Except for the “framer” program which performs the initial data extraction, the programs operate as filters, passing intermediate data between each other via operating system pipes. That is: taking input from stdin and writing results to stdout.

The intermediate data is formatted as ASCII JSON so that they can be saved after any processing step for processing at a later time or by alternate programs.

Except where specified, output records are not guaranteed to be in any particular order. Each program is discussed in section 2.

The definition of each type of data frame is described in configuration files, one per type of frame. Each type of frame is called an “instrument” as in the instrument used to acquire the original data.

A package file describes a collection of configuration files which are to be applied together to process a data file.

The configuration files are in JSON format so that they may be viewed or edited in any standard text editor. There are programs included for checking the validity of the configuration files. Additionally those files can be tested via JSON validators such as JSON Lint (jsonlint.com) and jq (stedolan.github.io/jq)
The configuration files are discussed in section 5.

1.1) Usage

A typical usage involving the most general operations is:

```
framer  package.file  data.file |
fitter  package.file |
validator  package.file |
reduce-selection  package.file |
averager |
viewer > results.tsv
```

A minimal operation is:

```
framer  package.file  data.file > framer.dat
```

where that output file can be used at a later date, even skipping the extra processing steps to view its contents:

```
viewer < framer.dat > unprocessed.results.tsv
```

2) Programs

2.1) averager

Given an interval value of 1 to 60 minutes, this filter will snap every time stamp to the previous earlier multiple of that interval and output an average of the sensor values which match that snapped time stamp.

A selection of 0 or less has the meaning of "no averaging".

If field validation (validator program) is desired, that step should be performed before averaging in order to avoid including unwanted values in the averages.

Any values which are described as strings, such as some frame status values, will not be averaged and will not be output from the averager. A GPS sync quality is the example of such a string status value.

With an interval of 15, the time will fall into 4 bins: 0 <= min < 15, 15 < min <= 30, 30 < min <= 45, 45 < min < 60. with the output minutes of 0, 15, 30, 45.

This program takes only the interval value as a parameter, a single simple number.

If not specified, the default interval is 15 minutes.

An example with the default interval is like this:

```
framer package.file data.file | averager > averaged.dat
```

Using a different interval, 5 minutes is like this:

```
framer package.file data.file |
averager –interval=5 > averaged.dat
```

There may be situations where processing requires different averaging on the same data set:

```
framer package.file data.file > framer.dat
averager –interval=1  < framer.dat  > ave.1.min.dat
averager –interval=30 < framer.dat > ave.30.min.dat
```

As mentioned above, if field validation is required then it ought to be performed before averaging so that undesired/impossible values are not included in the averages:

```
framer package.file data.file |
validator package.file |
averager > ave.dat
```

2.2) check-config

This program is not a data filter. It is used to check a single configuration file for errors.

Errors generate messages on standard error and an error code returned to the operating system.

The only parameter for this program is the full path to a single configuration file.

An example operation is:

```
check-config aquadopp.7455.conf
```

If operating in a Bash shell, the output code can be checked like this:

```
if ./check-config config.file
then
     echo config ok
else
     echo error in config
fi
```

2.3) check-package

This program is not a data filter. It is used to check for errors in a package file along with each of the configuration files specified in the package.

Errors generate messages on standard error and an error code returned to the operating system.

This program takes as a parameter the full path to a package file. Also an optional configuration path, where the configuration files are expected to be located, as described in section 4.

An example operation is like this:

```
check-package package.file
```

Using the configuration path like this:

```
check-package –configpath=dir.path package.file
```

If operating in a Bash shell, the output code can be checked like this:

```
if ./check-package package-file
then
     echo ok
else
     echo error in config package
fi
```

2.4) data-join

This program acts as a filter to join multiple intermediate data files, outputting the combination. This is the only filter program which does not require input on stdin, files can be specified as options.

The order of joining may be significant since once a value for any instrument/time/field is saved it will not be replaced by a matching key from a later joined file.

An example operation is like this:

```
data-join second.dat < first.dat  > together.dat
```

Also like this:

```
data-join first.dat second.dat > together.dat
```

Or

```
data-join *.dat > all.together.dat
```

Or

```
framer package.file data.flle | data-join saved.dat > together.dat
```

But not more that one input via stdin:

```
cat *.dat | data-join → ERROR
```

2.5) data-ranges

This program will display information on the dates contained in an intermediate data file.

No parameters or options are required, only a framer style data file on standard input. The output is a human readable table showing for each instrument: the number of records and the min and max timestamps.

The output from this program is not for processing by any of the other framer programs.

Example output is shown in section 8.2

An example operation is:

```
framer package.file data.file | data-ranges
```

After an averaging operation the number of records and date ranges will be different:

```
framer package.file data.file > frames.dat
```

```
data-ranges < frames.dat
averager –interval=30 < frames.dat | data-ranges
```

2.6) data-stats

This program will display basic statistics on the values contained in an intermediate data file.
No parameters or options are required, only a framer style data file on standard input.

The output is a human readable table showing for each instrument: the number of records and the min, max, median and standard deviation.
The output from this program is not for processing by any of the other framer programs.
Example output is shown in section 8.3

An example operation is:

```
framer package.file data.file | data-stats
```

After an averaging operation the number of records and date ranges will be different:

```
framer package.file data.file > frames.dat
data-stats < frames.dat
averager –interval=30 < frames.dat | data-stats
```

In combination with date selection, the time of an anomaly can be located:

```
framer package.file data.file > frames.dat
date-select –mindate=2018-05-02 –maxdate=2018-05-04 < frames.dat | data-stats
date-select –mindate=2018-05-05 –maxdate=2018-05-06 < frames.dat | data-stats
```

2.7) date-select

This program acts as a filter to extract data within a specified date range (precision at the day level rather down to an hour or smaller).
There are two optional parameters: minimum date which defaults to the year 1900, and maximum date which defaults to the year 2999. The dates must be specified in the format YYYY-MM-DD.

An example operation where nothing changes (assuming all the data is within the default dates):

```
framer package.file data.file | date-select > framer.dat
```

An example selecting data for only the year 2018:

```
framer package.file data.file |
date-select –mindate=2018-01-01 –maxdate=2018-12-31 |
viewer
```

An example of selecting everything before July 1st and everything since July 1st which are then processed separately:

```
framer package.file data.file > frames.dat
date-select –maxdate=2018-06-30 < frames.dat > before.july1.dat
date-select –mindate=2018-07-01 < frames.dat > since.july1.dat

fitter package.file < beforejuly1.dat|averager|viewer>before.tsv
fitter package.file < since.july1.dat|averager|viewer>since.tsv
```

The same example without saving the intermediate files:

```
framer package.file data.file |
date-select –maxdate=2018-06-30 |
fitter package.file | averager | viewer > before.tsv

framer package.file data.file |
date-select –mindate=2018-07-01 |
fitter package.file | averager | viewer > since.tsv
```

The date selection can be applied between any two operations:

```
framer package.file data.file |
date-select –mindate=2018-05-01 –maxdate=2018-05-31 |
averager –interval=10 | viewer
```

The date selection can be applied more than once:

```
framer package.file data.file |
date-select –mindate=2018-05-01 –maxdate=2018-05-31 |
averager –interval=15 |
date-select –mindate=2018-05-12 –maxdate=2018-05-15 | viewer
```

2.8) fitter

The output values from this program will be modified by using the fitting equations as specified in the configuration file. Fields which are not configured for fitting are unchanged.

This fitting is generally used to convert engineering values (raw counts) into scientific values by way of calibration coefficients.

The fit options are described with the configuration files in section 5.

If frame validation is desired, that step should be performed after fitting because the

validation values are usually defined in scientific units.

It probably doesn't matter if averaging occurs before or after fitting. Unlike the other filter programs, fitting should not be applied more than once on the same data set.

This programs takes as parameters a package file and an optional configuration path.

An example operation is:

```
framer package.file data.file | fitter package.file > fit.dat
```

With the optional configuration file path:

```
framer -configpath=directory.path package.file data.file |
fitter -configpath=directory.path package.file > fit.dat
```

Combined with other programs:

```
framer package.file data.file | averager |
fitter package.file | viewer > ave.and.fit.dat
```

2.9) framer

With a raw data file and a package of configuration files the output from this program will contain the data values for every frame in the raw file as matched by the definitions of the configuration files.

Note that all fields, not just those configured as report-able, are output. This is because non-report-able fields may be useful for other processing purposes.

The package files are described in section 4. The configuration files are described in section 5.

This program takes a data file name, a package file name and an optional configuration path as parameters. It does not take input via stdin. The output is to stdout.

Like the other programs it writes errors to stderr and a serious error will cause an error code to be reported to the operating system

Along with outputting frame data, it will perform individual frame validation and frame checksum tests if specified in the configuration file. Any frame which fails validation or checksum will not be copied to output.

An additional test is the validity of the 7 byte timestamp following each frame. If valid, that is the timestamp output with the frame. Otherwise a bad timestamp (date is impossible) causes the frame to be skipped.

An example operation:

```
framer package.file data.file > framer.dat
```

Specifying a location for the configuration files:

```
framer –configpath=dir.name package.file > framer.dat
```

If operating in a Bash shell, the process can be checked like this:

```
if framer package.file data.file >framer.dat
then
     echo framer ok
else
     echo framer failed
fi
```

A typical operation from start to end where the final output is a tab delimited table which is human readable and loadable into a spreadsheet or other program:

```
framer –configpath=dir.name package.file data.file |
fitter –configpath=dir.name package.file |
validator –configpath=dir.name package.file |
reduce-selection –configpath=dir.name package.file |
averager |
viewer > results.tsv
```

2.10) reduce-selection

This program will output only the data fields which are configured with the “report” option set to “true”.

Note that the configuration package specified at any step, this one in particular, need not be the exact same package used in the previous step. In this manner is it possible to create subsets of the data for further alternate processing.

This programs takes as parameters a package file and an optional configuration path.

An example operation is:

```
framer package.file data.file |
reduce-selection package.file > reduced.dat
```

An example of breaking out different data sets from the same data file by using different configuration packages:

```
framer package.file data.file > frames.dat
reduce-selection one.package.file < frames.dat > set1.dat
reduce-selection two.package.file < frames.dat > set2.dat
```

2.11) validator

This program performs validation against all of the fields which have this option set in the configuration file. The validation consists of checking against a minimum and/or maximum. Any values which fail the validation are not output.

This operation is intended to be used for removing impossible values and to prevent those values from being included in the averaging process. For example: if it is known that an instrument glitch sometimes produces a negative pH, then the field validation for pH can be configured to have a minimum of zero.

If the validation operation is desired, the fitting step should be performed first because the validation values are usually specified in scientific units.

This operation, “field validation” is different from the “frame validation” which is performed by the framer program. This “field validation” is for individual fields, whereas the “frame validation” affects a whole frame regardless of all the included fields.

This programs takes as parameters a package file and an optional configuration path.

An example operation is:

```
framer package.file data.file |
fitter package.file |
validator package.file | viewer > results.tsv
```

2.12) viewer

This program will convert an intermediate data file on standard input to human readable tab delimited format on standard output. Or the output can be saved for viewing in a spreadsheet or other program.

The output has column names on the first line, followed by one line for the data at each timestamp in ascending timestamp order.

This program can be applied after any processing step, but always becomes the final step.

This program takes one optional parameter to set the format of the output. By default the format has the data separated by instrument (--style=separate), with the other option of combining all instrument data at the same timestamp into a single line (--style=combined).

The combined option should only be used when there is no possibility of field name duplication between instruments, since that condition causes an undefined result. For example: if two instruments produce a temperature at the same timestamp, is it not defined which temperature will be output.

Example output is shown in section 8.1

An example operation is:

```
framer package.file data.file | viewer
```

Example operation after any step:

```
framer package.file data.file > frames.dat
viewer < frames.dat > everything.tsv
fitter package.file < frames.dat | viewer > fit.tsv
fitter package.file < frames.dat | averager | viewer > ave.tsv
```

3) Common Options and Parameters

Note that these options begin with double dashes.

3.1) --configpath

This option sets the path where each program will look for the configuration files named in the package file. All of the programs which use a package file will use this parameter.

The default is ".", the current directory.

Example: --configpath=data/setups

3.2) --enable-tests=yes or no

This option is used by the programs framer and validator.

By default (“yes”) the testing for check sums, frame validity, field transform and field validity is enabled, and that should be the standard operating setting.

There are possible conditions when those tests need to be disabled without having to edit any of the associated instrument configuration files.

Note that individual tests cannot be disabled with this flag, all of the tests for all the package files will be disabled.

To disable use: --enable-tests=no

3.3) --verbose

This option sets the level of program information displayed to standard error. When set to zero a program is "silent", excepting fatal errors. The default level is one which causes a minimum of information to the displayed. The maximum level is 9 for the maximum output.

Example: --verbose=2

3.4) --version

This option causes any program to display its version number then exit.

3.5) package file

This is the full path to the package file. Description of the contents is described in section 4.

3.6) data file

The data file name is required by the fitter. The file's full path is required, i.e. there is no relation to the configpath option.

4) Package Files

This is a simple text file which names the instruments used to process a data file.

A comment in this file is any text following a "#" character. Comments are ignored during processing and so may be used to hide an instrument from being used.

Each line of the package file represents a single instrument with two parts separated by a space. The first part is the instrument name, second is the instrument serial number. The instrument name will be converted to lower case and leading zeros will be removed from the serial number.

Each processing program will expect to find a file matching a name of instrument + dot + serial number + ".conf" suffix in the location specified by the configpath parameter.

A missing configuration file will cause a program to exit with an error.

Strictly speaking, the two parts of the name need not match the instrument and serial number defined in the configuration file, but it is a useful technique.

An example of a package file is:

```
StorX 129
Suna 0052
WQMX 1050
```

which will cause the program to look in configpath for three files:

```
storx.129.conf
suna.52.conf
wqmx.1050.conf
```

5) Configuration Files

The contents of data frames and details on how the values should be extracted from data files are described in the framer configuration files. These are JSON formatted files which may be reformatted or validated by any number of on-line JSON validation tools. As described in the Package Files section, each configuration file is for a single instrument with a name format of "instrument dot serial-number dot conf" all lower case.

All items within the file should be double quoted strings and non-printable characters must be stored by their UNICODE values: CR = "\u000D", LF= "\u000A", tab = “\u0009” or standard escape codes: CR = “\r”, LF = “\n”, TAB = “\t”,

Character case is not important, except for the "frame_header" where the characters must match exactly the sync word in the data file. String values used in validation testing are also to be stored in exact character case.

Ordering of the items within the file is not important with the one major exception that the fields section must be an ordered array (denoted by JSON square brackets rather than curly brackets) in the same sequence as the order in which fields appear in the instrument frame.

5.1) Common Keywords

5.1.1) comment - can be placed in any section to aid in human understanding of the selection or chosen options.

Example usage:

```
"frame_validity": [
               { "name": "status", "comment": "require verified GPS sync",
                 "op": "=", "value": "A", "enable": "true" }
               ],
```

5.1.2) enable - several optional configuration items make use of this keyword to mark the item as turned on. The "on" value is one of "true", “t”, "yes", “y”, “on”, “ok” or "1". All other values or if the keyword is missing is considered as "off".

This keyword provides a method of disabling an operation without removing the section from the configuration file or setting to a non-true value.

5.1.3) report - several configuration items make use of this keyword to mark the value as "to be output". Similar to the "enable" keyword, it takes "on" values of "true", “t”, "yes", “y”, “on”, “ok”, or "1". All other values or a missing pair is considered "off".

5.2) Keyword Definitions

5.2.1) instrument - which is typically the instrument name. This pair is informational and not used as part of any computation. But it is required for identification in the output data files.

Example usage is:

```
"instrument": "SUNA"
```

5.2.2) serial - the serial number of the instrument. Also required for identification in the output data files.

Example usage is:

```
"serial": "567"
```

5.2.3) frame_header - defines the sync word of the instrument frame which is used to locate each frame within the data file.

Example usage is:

```
"frame_header": "SATSLB0512"
```

5.2.4) frame_type - either "ASCII" or "BINARY" as required by the instrument configuration.

Example usage is:

```
"frame_type": "ascii"
```

and

```
"frame_type": "binary"
```

5.2.5) frame_terminator - the characters which match the end of an ASCII frame. This is required for ASCII frames, optional/ignored in binary configurations.

Example usage is:

```
"frame_terminator": "\u000d\u000a"
```

5.2.6) checksum - an optional array of items which describe how the frame checksum is computed.

Each item within the array consists of the key-value pairs "name", "type", "enable" and "skip".

The value of the "name" item is the field which contains the instrument's checksum for each frame.

The value of the "skip" item is the number of characters to at the beginning of the frame which will not be included in the checksum calculations. For example: GPS frames often begin with a “$” which should be ignored, in which case the skip value should be “1”.

These checks are performed by the framer program. A bad checksum will cause the data frame to not be output.

Though unlikey, it is possible to specify multiple checksums.

The only supported type is "NMEA".

Example usage is:

```
"checksum": [ { "name": "checksum_field",
                "type": "NMEA",
                "enable": "true" } ]
```

5.2.7) constants - an optional array of field values which do not appear in the instrument data frame but will be output as if they are data values. Such values may be useful for further data processing or frame identification.

Placing calibration coefficients here is a reasonable usage. Another use might be to place the instrument serial number here so that the instrument can be identified in processing by alternate programs.

Each item within the array consists of the key-value pairs "name", "value", and "report" which are described further in the fields section.

Values of the constants are always taken to be numeric, i.e. they cannot be string types.

Example usage is:

```
"constants": [ { "name": "offset",
                 "value": "3.14",
                 "report": "true" } ]
```

or

```
"constants": [ { "name": "slope",
                 "value": "42.0",
                 "report": "true" },
               { "name": "sn",
                 "value": "411",
                 "report": "false" } ]
```

5.2.8) duplicates - an optional array allows a method of outputting a second copy of any of the data values. In this way multiple fit operations can be performed on the same value, or output of a fit and unfit value at the same time.

A value can have any number of duplicates and the list can contain any set of fields. A duplicate takes many of the same options as a field: report, fit, validate, and transform. But a duplicate cannot be used in a checksum or field validator. Length and type options are not needed. Only numeric fields can be duplicated.

The additional item is the “of” option which specifies the name of the field which is being duplicated.

The field being duplicated must be report-able for the duplicate to be output.

Example usage is:

```
"duplicates": [
     { "name": "DOUBLE_VOLT", "report": "true", "of": "VOLTAGE",
          "fit": { "type": "POLYU",
          "coeffs": [{ "name": "a0", "value": "0" },
                       { "name": "a1", "value": "2" }] }
     },
     { "name": "HALF_VOLT", "report": "true", "of": "VOLTAGE",
          "fit": { "type": "POLYU",
          "coeffs": [{ "name": "a0", "value": "0" },
                       { "name": "a1", "value": "0.5" }] }
     },
     { "name": "RAWPUMP", "report": "true", "of": "PUMP"
     }
]
```

5.2.9) frame_validity - an optional array of items which describe how to verify if a whole frame is considered valid based on the sensor scientific units rather than a check sum.

For example: a GPS frame is valid only if the status field equals "A". Another example is that a SUNA frame is valid only if the RMSe is > 0.01

Each item within the array consists of the key-value pairs "name", "type", "op", "value", "enable". The value of the name item is the sensor field to which the test is applied. The value of the op (operator) item is the comparison used in the test: "=", "<", "<=", ">", ">=", and for not equal either "<>" or "!=". The value item is used in the test against the sensor field.

The test is performed as if by reading the items in this order: name op value; with the validation passing the test if the comparison is true. I.E. "rmse < 0.01" will pass only when the RMSe is less than 0.01

These checks are performed by the framer program. If one of the validations fails the frame in question is not output.

Example usage is:

```
"frame_validity": [ { "name": "status",
          "op": "=", "value": "A", "enable": "true" } ],
```

or

```
"frame_validity": [ { "name": "nitrate_RMSe",
      "op": "<", "value": "0.01", "enable": "true" } ],
```

5.2.10) fields - an array of items describing each field in the data frame.

Each item within the array consists of the key-value pairs "name", "type", "delimiter", "length", "report" and the objects "fit", "transform" and "validity".

A simple example field section is:

```
{ "name": "PUMP", "report": "true",
      "type": "AI", "delimiter": "," },
```

5.2.10.1) fields/type - The type describes the format of the value within the data frame.

For ASCII frames the type may be AS = text, AI = integer, AU = integer, AF = float, AD = double/float, AX = skip/ignore.

For binary frames the type may be BS = signed integer, BF = float, BD = double, BU = unsigned integer, BB = binary bytes/ignore, BSLE = signed integer little endian, BULE = unsigned integer little endian.

The skip/ignore fields are not output by the framer program, i.e. are not report-able.

5.2.10.2) fields/length - The length is optional/ignored for ASCII frames but is required for binary frames, specifying the number of bytes for the field. The integer types can take lengths of 1, 2, 4 or 8. The BF type must take a length of 4. The BD type must take a length of 8. The BB type can take any length and so is useful for skipping a section of bytes. The BB type cannot be set as report-able.

An example with a length is:

```
{ "name": "PUMP", "report": "true", "type": "BU", "length": 2 },
```

5.2.10.3) fields/delimiter - The delimiter item is required for ASCII frames, signifying a character string which separates each field from the previous field.

A non-printable character should be defined as a UNICODE character or escape character such as tab = “\u0009” or “\t”.

A simple example field section is:

```
{ "name": "PUMP", "report": "true",
      "type": "AI", "delimiter": "," },
```

5.2.10.4) fields/fit - The fit item is used for applying an equation to a data value, typically for converting raw counts to scientific units.

The fit section requires a "type" and "coeffs" which is an array of "name" and "value" pairs. The fit types are:

"POLYF" which is y = a0 * (x – a1) * (x – a2) ... where the coeff names are “a0”, “a1”, “a2”, etc.

"POLYU" which is y = a0 + a1 * x + a2 * x^2 ... where the coeff names are “a0”, “a1”, “a2”, etc.

"OPTIC2" which is y = Im * a1 * (x – a0) where the coeff names are “im”, “a0”, “a1”

“OPTIC3” which is y = Im * a1 * (x – a0) * Cint / Aint where the coeff names are “im”, “a0”, “a1”, “cint”, “aint”.

“POW10” which is y = Im * 10^(x – a0)/a1 where the coeff names are “im”, “a0”, “a1”.

Only one fit can be applied to a field. The fitting operations are performed by the fitter program.

An example field with a fit is:

```
{ "name": "VOLTAGE", "report": "true",
     "type": "BU", "length": "2",
     "fit": { "type": "POLYU",
          "coeffs": [{ "name": "a0", "value": "0.25" },
                     { "name": "a1", "value": "0.000382698" }] }
},
```

5.2.10.5) fields/validity - The validity item is for checking that the field value is within specified limits. The value will not be output if it is outside the validation limits. More than one validation can be applied to a field, so the definitions are included in a square bracket array.

Note that the field validator is for the specific field, whereas the frame validator defined at the frame level applies to the whole frame.

The definition is a set of "enable", "op" and "value" items. The “op” (operator) is one of "=", ">", ">=", "<", "<=", and for not equal "<>" or "!=".

The validation checks are applied by the validator program. The comparison values are usually given in scientific units, so that the fitter program should be run before the validator program.

An example of a field with a validity check:

```
{ "name": "pH", "report": "true",
     "type": "AF", "delimiter": ",",
     "validity": [
          { "op": ">=", "value": "0", "enable": "true" },
          { "op": "<=", "value": "14", "enable": "true" } ]
 },
```

5.2.10.6) fields/transform -The transform is an array of items which will cause a transformation on the field value based on the value of another field. For example the NORTH-SOUTH transform can be used to change a GPS longitude to a negative number if the hemisphere field equals "S".

Each item within the array consists of the key-value pairs "name", "type", and "enable".

The value of the name item is the sensor field which is tested in order to apply the transformation to the current field.

Currently only "NORTH-SOUTH" and "EAST-WEST" transforms are available. EAST-WEST is similar to the NORTH-SOUTH transform for the other geographic direction.

These operations are performed by the framer program.

Example usage is:

```
"transform": [{ "name": "lathemi", "enable": "true",
                "type": "north-south" }],
```

Example within a field description:

```
 "fields": [
{ "name": "UTCPOS",
    "type": "AF", "delimiter": "," },
{ "name": "STATUS",
    "type": "AS", "delimiter": "," },
{ "name": "LATPOS", "report": "true",
    "type": "AF", "delimiter": ",",
    "transform": [{ "name": "LATHEMI", "enable": "yes",
                    "type": "north-south",
                     "comment": "ensure correct +/-" }] },
{ "name": "LATHEMI",
    "type": "AS", "delimiter": "," },
 …
 ]
```

6) Advanced Usage

6.1) reduce-selection

Its almost always a good idea to apply the reduce-selection as early as possible in order to make the intermediate data as small as possible.

The reduce-selection acts as a filter that outputs only those fields which are configured as report-able.

Immediately after running the framer is best.

```
framer package.file data.file | reduce-selection package.file > framer.dat
```

6.2) Binary integers

There are rare cases where data consists of 3 byte binary integers. Framer does not handle this size.

A possible solution is to extract the data in 3 x 1 byte unsigned integers then perform the reassembly in an alternate program.

6.3) Large data sets

There are cases where a large data set needs to undergo multiple processing options. First extract all the data using a package file which defines all possible included frames:

```
framer everything.package largedata.file > frames.dat
```

then separate out the individual sets by using package files which specify the wanted sets:

```
reduce-selection set1.package < frames.dat > set1.dat
reduce-selection set2.package < frames.dat > set2.dat
reduce-selection set3.package < frames.dat > set3.dat
```

followed by further selection by date-select and the remainder of the processing.

6.4) Joining

Intermediate data files which have been split can be joined back together in any sequence.

For example, a large data set which spans a whole year which needs to have the month of May removed can be performed like this:

data-select --maxdate=2018-04-30 < whole-year.dat > before.may.dat
data-select --mindate=2018-06-01 < whole-year.dat > after.may.dat
data-join before.may.dat after.may.dat > without.may.dat

Joining can be performed as part of a sequence:

```
framer package.file data.file |
data-join other.set.dat |
averager > combined.dat
```

Joining data sets which have been processed differently may produce undesired results.

For instance, joining fitted with unfitted will result in values together in different units.

6.5) JSON arrays

The configuration files and intermediate data files are written in JSON (JavaScript Object Notation) format.

Note the difference between JSON sets (within {}) vs JSON arrays (within []).

There are configuration options which are specified as arrays because they can have more than one setting. Also the array specifies an ordering.

And they can be empty ([]) if the option is not used.

6.6) Alternate Fitting

There are conditions under which the same data value can represent more than one scientific type with the use of alternate fitting coefficients. This can be performed with “duplicates” on the original data file.

Without the use of “duplicates”, there is a method of applying alternate fits.

For each alternate fitting, the field name requires a unique name if the data is to be later joined together as a single data set.

The following contrived example shows how to apply two fits to the same original raw data field to produce two output fields.

The first configuration file might have the following definition which is included in “package1”:

```
"fields": [
{ "name": "VOLT", "report": "y", "type": "BU", "length": "2" },

{ "name": "ALPHA", "report": "y", "type": "BU", "length": "4",
        "fit": { "type": "POLYU",
             "coeffs": [{ "name": "a0", "value": "0.25" },
                        { "name": "a1", "value": "0.000382698" }] }
},

{ "name": "TEMP", "report": "y", "type": "BU", "length": "2" }
]
```

Another configuration file in “package2” with a different name and set of coefficients for the second field:

```
"fields": [
{ "name": "VOLT", "report": "y", "type": "BU", "length": "2" },

{ "name": "BETA", "report": "y", "type": "BU", "length": "4",
        "fit": { "type": "POLYU",
             "coeffs": [{ "name": "a0", "value": "42" },
                        { "name": "a1", "value": "3.0" },
                        { "name": "a2", "value": "0.14" }] }
},

{ "name": "TEMP", "report": "y", "type": "BU", "length": "2" }
]
```

Extract the data and fit with the two separate configurations, then join to have both fields together at the same timestamps.

```
framer package1.file data.file |
fitter package1.file > with-alpha.dat

framer package2.file data.file |
fitter package2.file > with-beta.dat

data-join with-alpha.dat with-beta.dat > together.dat
```

Because of the field name difference between the two configurations, its probably best to perform all the required operations before combining the two data files.

The data selection, averaging, stats and viewing can be performed safely after joining.

```
framer package1.file data.file |
fitter package1.file |
validator package1.file |
reduce-selection package1.file > with-alpha.dat

framer package2.file data.file |
fitter package2.file |
validator package2.file |
reduce-selection package2.file > with-beta.dat

data-join with-alpha.dat with-beta.dat > together.dat
```

7) Intermediate File Format

Following is an example of an intermediate data file produced by the framer program; read by the other programs, and output again by the filter programs.

Ordering is not important.

The header section is required along with the item "extracted by": "Framer".

The other sections contain the data: instrument identifier → timestamp → multiple sensor values.


```
{
 "header info": {
      "extracted by": "Framer",
      "extractor version": "1.0",
      "extracted on": "2019-02-16 09:44",
      "data file": "test.data"
 },
 "adcp/723": {
      "2019-01-05 07:02:02": {
           "pitch": -34,
           "roll": 257,
           "east": -291,
           "north": -285,
           "up": -25
      },
      "2019-01-05 07:02:03": {
           "pitch": 20,
           "roll": 293,
           "east": -192,
           "north": -209,
           "up": -73
      }
 },
 "suna/217": {
      "2019-01-05 07:03:31": {
           "nitrate_um": 23.507,
           "nitrate_mg": 0.3292
      },
      "2019-01-05 07:03:34": {
           "nitrate_um": 15.420,
           "nitrate_mg": 0.2159
      }
 }
}
```

8) Output

Using the above example data file, following is the viewable output.

8.1) viewer

These lines are tab separated. The actual output may appear less aligned.
The output is ordered by the time value.

```
Time                instrument east nitrate_mg nitrate_um north pitch roll up
2019-01-05 07:02:02 adcp/723   -291                       -285  -34   257  -25
2019-01-05 07:02:03 adcp/723   -192                       -209   20   293  -73
2019-01-05 07:03:31 suna/217        0.3292     23.507
2019-01-05 07:03:34 suna/217        0.2159     15.42
```

8.2) data-range

```
instrument records min date            max date
---------- ------- ------------------- -------------------
  adcp/723       2 2019-01-05 07:02:02 2019-01-05 07:02:03
  suna/217       2 2019-01-05 07:03:31 2019-01-05 07:03:34
---------- ------- ------------------- -------------------
     total       4 2019-01-05 07:02:02 2019-01-05 07:03:34
```

8.3) data-stats

```
instrument      field records    min    max    mean std dev
---------- ---------- ------- ------ ------ ------- -------
  adcp/723       east       2   -291   -192  -241.5    49.5
  adcp/723      north       2   -285   -209    -247      38
  adcp/723      pitch       2    -34     20      -7      27
  adcp/723       roll       2    257    293     275      18
  adcp/723         up       2    -73    -25     -49      24
  suna/217 nitrate_mg       2 0.2159 0.3292 0.27255 0.05665
  suna/217 nitrate_um       2  15.42 23.507 19.4635  4.0435
```

9) Example Frame Definitions

Supposing an instrument called a Reader S/N 76 has an ASCII data frame like this:

```
Sync123 TAB date COMMA counter COMMA temp COMMA salinity CR LF
```

The configuration would be:

```
{
"instrument": "Reader",
"serial": "76",
"frame_type": "ascii",
"frame_header": "Sync123",
"frame_terminator": "\u000D\u000A",
"fields": [
   { "name": "date", "type": "as", "delimiter": "\u0009" },
   { "name": "count", "type": "ai", "delimiter": "," },
   { "name": "temp", "report": "yes", "type": "af", "delimiter": "," },
   { "name": "salinity", "report": "yes", "type": "af", "delimiter": "," }
]
}
```

Supposing a data file has frames from an instrument, Collector S/N 8932, which produces binary output in this format:

```
START8932 speed/2-byte-integer direction/4-byte-float status/1-byte-integer
```

The corresponding configuration is:

```
{
"instrument": "Collector",
"serial": "8932",
"frame_type": "binary",
"frame_header": "START8932",
"fields": [
   { "name": "speed", "report": "yes", "type": "bu", "length": "2" },
   { "name": "dir", "report": "yes", "type": "bf", "length": "4" },
   { "name": "status", "report": "no", "type": "bu", "length": "1" }
]
}
```

10) Example Configurations

10.1) Example of a binary StorX frame. The only report-able field is "voltage" which has a POLYU fit to convert from engineering units to scientific units of volts.

```
{
"instrument": "STORX",
"serial": "129",
"frame_header": "SATSTX0129",
"frame_type": "binary",
"comment": "created by conversion program",
"fields": [
    { "name": "AUX1",
        "type": "BU", "length": "2" },
    { "name": "AUX2",
        "type": "BU", "length": "2" },
    { "name": "AUX3",
        "type": "BU", "length": "2" },
    { "name": "AUX4",
        "type": "BU", "length": "2" },
    { "name": "AUX5",
        "type": "BU", "length": "2" },
    { "name": "AUX6",
        "type": "BU", "length": "2" },
    { "name": "AUX7",
        "type": "BU", "length": "2" },
    { "name": "VOLTAGE", "report": "true",
        "type": "BU", "length": "2",
        "fit": { "type": "POLYU",
             "coeffs": [{ "name": "a0", "value": "0.25" },
                    { "name": "a1", "value": "0.000382698" }] } },
    { "name": "CRLFTERMINATOR",
        "type": "BU", "length": "2" }
]
}
```

10.2) Example of a WQMX ASCII frame. Several of the fields are report-able and a comment is included within the "P_OXSAT" field.

```
{
"instrument": "WQMX",
"serial": "1042",
"frame_header": "WQMX,1042",
"frame_type": "ascii",
"frame_terminator": "\u000D\u000A",
"fields": [
```

```
    { "name": "WQMXHDR",
        "type": "AS", "delimiter": "\u0009" },
    { "name": "WQMXSN",
        "type": "AS", "delimiter": "," },
    { "name": "WQMDATE",
        "type": "AS", "delimiter": "," },
    { "name": "WQMTIME",
        "type": "AS", "delimiter": "," },
    { "name": "PUMP", "report": "true",
        "type": "AI", "delimiter": "," },
    { "name": "COND", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "TW", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "PRESS", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "SALINITY", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "DOFREQ",
        "type": "AF", "delimiter": "," },
    { "name": "OXSAT", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "DO", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "P_OXSAT", "report": "true",
        "type": "AF", "delimiter": ",",
        "comment": "probably should be multiplied by 100" },
    { "name": "TURB_RAW700",
        "type": "AI", "delimiter": "," },
    { "name": "NTU700",
        "type": "AF", "delimiter": "," },
    { "name": "FL_RAW",
        "type": "AI", "delimiter": "," },
    { "name": "FLUOR", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "CDOM_RAW",
        "type": "AI", "delimiter": "," },
    { "name": "CDOM", "report": "true",
        "type": "AF", "delimiter": "," }
]
}
```

10.3) Example of a modem's GPS frame using check sum and frame validity. Reporting only lat and lon fields.

```
{
"instrument": "MODEM",
"serial": "183",
"frame_header": "$GPRMC",
"frame_type": "ascii",
"frame_terminator": "\u000D\u000A",
"checksum": [
 { "name": "CHECKSUM", "type": "NMEA", "skip": 1, "enable": "true" }
],
"frame_validity": [
{ "name": "status", "op": "=", "value": "A", "enable": "true" }
],
"fields": [
    { "name": "UTCPOS",
        "type": "AF", "delimiter": "," },
    { "name": "STATUS",
        "type": "AS", "delimiter": "," },
    { "name": "LATPOS", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "LATHEMI",
        "type": "AS", "delimiter": "," },
    { "name": "LONPOS", "report": "true",
        "type": "AF", "delimiter": "," },
    { "name": "LONHEMI",
        "type": "AS", "delimiter": "," },
    { "name": "SPEED",
        "type": "AF", "delimiter": "," },
    { "name": "COURSETRUE",
        "type": "AF", "delimiter": "," },
    { "name": "DATE",
        "type": "AI", "delimiter": "," },
    { "name": "MAGVAR",
        "type": "AF", "delimiter": "," },
    { "name": "MAGHEMI",
        "type": "AS", "delimiter": "," },
    { "name": "MODEGPS",
        "type": "AS", "delimiter": "," },
    { "name": "CHECKSUM",
        "type": "AI", "delimiter": "*" }
]
}
```

10.4) Example of a SUNA binary frame with frame validation and field validation. The large middle section of the spectrum is not displayed here.

```
{
"instrument": "SUNA",
"serial": "0649",
"frame_header": "SATSLB0649",
"frame_type": "binary",
"frame_validity": [
   { "name": "RMSe", "op": "<", "value": "0.01", "enable": "y" }
],
"fields": [
    { "name": "DATEFIELD",
        "type": "BS", "length": "4" },
    { "name": "TIMEFIELD",
        "type": "BD", "length": "8" },
    { "name": "NITRATE_UM", "report": "true",
        "type": "BF", "length": "4",
        "validity": [
               { "op": ">", "value": "-10", "enable": "true" },
               { "op": "<", "value": "100", "enable": "true" }] },
    { "name": "NITRATE_MG", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "ABS_254",
        "type": "BF", "length": "4" },
    { "name": "ABS_350",
        "type": "BF", "length": "4" },
    { "name": "BR_TRACE",
        "type": "BF", "length": "4" },
    { "name": "NITRATE_SPEC_AVG",
        "type": "BU", "length": "2" },
    { "name": "DARK_AVG",
        "type": "BU", "length": "2" },
    { "name": "INT_FACTOR",
        "type": "BU", "length": "1" },
    { "name": "UV187_06",
        "type": "BU", "length": "2",
        "fit": { "type": "OPTIC2",
              "coeffs": [{ "name": "a0", "value": "0.0" },
                         { "name": "a1", "value": "1.0" },
                         { "name": "Im", "value": "1.0" }] } },
…
    { "name": "UV392_29",
        "type": "BU", "length": "2",
        "fit": { "type": "OPTIC2",
              "coeffs": [{ "name": "a0", "value": "0.0" },
                         { "name": "a1", "value": "1.0" },
                         { "name": "Im", "value": "1.0" }] } },
```

```
    { "name": "T_INT",
        "type": "BF", "length": "4" },
    { "name": "T_SPEC",
        "type": "BF", "length": "4" },
    { "name": "T_LAMP",
        "type": "BF", "length": "4" },
    { "name": "LAMP_TIME",
        "type": "BU", "length": "4" },
    { "name": "HUMIDITY", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "VOLT_MAIN",
        "type": "BF", "length": "4" },
    { "name": "VOLT_12",
        "type": "BF", "length": "4" },
    { "name": "VOLT_5",
        "type": "BF", "length": "4" },
    { "name": "CURRENT",
        "type": "BF", "length": "4" },
    { "name": "FIT_S2",
        "type": "BF", "length": "4" },
    { "name": "FIT_S3",
        "type": "BF", "length": "4" },
    { "name": "FIT_B0",
        "type": "BF", "length": "4" },
    { "name": "FIT_B1",
        "type": "BF", "length": "4" },
    { "name": "RMSe", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "CTD_TIME",
        "type": "BU", "length": "4" },
    { "name": "CTD_SAL",
        "type": "BF", "length": "4" },
    { "name": "CTD_TEMP",
        "type": "BF", "length": "4" },
    { "name": "CTD_DEPTH",
        "type": "BF", "length": "4" },
    { "name": "CHECK_SUM",
        "type": "BU", "length": "1" }
]
}
```

10.5) Similar to the previous SUNA example but using the "BB" field type to skip the large set of spectrum fields.

```
{
"instrument": "SUNA",
"serial": "0649",
"frame_header": "SATSLB0649",
"frame_type": "binary",
"constants": [
],
"frame_validity": [
   { "name": "RMSe", "op": "<", "value": "0.01", "enable": "y" }
],
"fields": [
    { "name": "DATEFIELD",
        "type": "BS", "length": "4" },
    { "name": "TIMEFIELD",
        "type": "BD", "length": "8" },
    { "name": "NITRATE_UM", "report": "true",
        "type": "BF", "length": "4",
        "validity": [
               { "op": ">", "value": "-10", "enable": "true" },
               { "op": "<", "value": "100", "enable": true" }] },
    { "name": "NITRATE_MG", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "ABS_254",
        "type": "BF", "length": "4" },
    { "name": "ABS_350",
        "type": "BF", "length": "4" },
    { "name": "BR_TRACE",
        "type": "BF", "length": "4" },
    { "name": "NITRATE_SPEC_AVG",
        "type": "BU", "length": "2" },
    { "name": "DARK_AVG",
        "type": "BU", "length": "2" },
    { "name": "INT_FACTOR",
        "type": "BU", "length": "1" },
    { "name": "SPECTRUM",
        "type": "BB", "length": "512",
        "comment": "skip the 256 spectrum fields"},
    { "name": "T_INT",
        "type": "BF", "length": "4" },
    { "name": "T_SPEC",
        "type": "BF", "length": "4" },
    { "name": "T_LAMP",
        "type": "BF", "length": "4" },
    { "name": "LAMP_TIME",
        "type": "BU", "length": "4" },
```

```
    { "name": "HUMIDITY", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "VOLT_MAIN",
        "type": "BF", "length": "4" },
    { "name": "VOLT_12",
        "type": "BF", "length": "4" },
    { "name": "VOLT_5",
        "type": "BF", "length": "4" },
    { "name": "CURRENT",
        "type": "BF", "length": "4" },
    { "name": "FIT_S2",
        "type": "BF", "length": "4" },
    { "name": "FIT_S3",
        "type": "BF", "length": "4" },
    { "name": "FIT_B0",
        "type": "BF", "length": "4" },
    { "name": "FIT_B1",
        "type": "BF", "length": "4" },
    { "name": "RMSe", "report": "true",
        "type": "BF", "length": "4" },
    { "name": "CTD_TIME",
        "type": "BU", "length": "4" },
    { "name": "CTD_SAL",
        "type": "BF", "length": "4" },
    { "name": "CTD_TEMP",
        "type": "BF", "length": "4" },
    { "name": "CTD_DEPTH",
        "type": "BF", "length": "4" },
    { "name": "CHECK_SUM",
        "type": "BU", "length": "1" }
]
}
```