Copernicus Climate Change Service Global Land and Marine Observations Database

Data reader. Developer manual

**Issued by:** NUIM / Peter Thorne

**Date:** 31/12/2018

**Ref:**C3S\_D311a\_Lot2.3.4.4-2018\_201812\_ Marine\_code\_v1

**Official reference number service contract:** 2017/C3S\_311a\_Lot2\_NUIM/SC1

Contributors

natural environment research council, National oceanography centre (NERC noc)

1. David I. Berry

2. Elizabeth C. Kent

natIONAL UNIVERSITY OF IRELAND MAYNOOTH

1. Peter Thorne

2. Corinne Voces

National oceanic and atmospheric administration’s national centers for environmental information

1. Eric Freeman

# Table of Contents

[1. Table of Contents 3](#_Toc4758881)

[2. Introduction 6](#_Toc4758882)

[3. Tool overview 7](#_Toc4758883)

[4. Time stamps management 8](#_Toc4758884)

[5. How to register a new data file format (in the application) 9](#_Toc4758885)

[6. References 10](#_Toc4758886)

[7. Annex 11](#_Toc4758887)

[7.1 Time stamps management 11](#_Toc4758888)

[7.2 File format models 13](#_Toc4758889)

[7.2.1 Fixed width simple models 13](#_Toc4758890)

[7.2.1.1 1 section 13](#_Toc4758891)

[7.2.1.2 Multiple sections 13](#_Toc4758892)

[7.2.2 Fixed width complex models 15](#_Toc4758893)

[7.2.2.1 IMMA1 model – multiple and optional sections 15](#_Toc4758894)

[7.2.2.2 TDF-11 model – multiple and exclusive sections 15](#_Toc4758895)

[7.2.2.3 TD14 model – multiple reports per record with common sections ( or header) 15](#_Toc4758896)

[7.2.2.4 TD14 model – multiple reports per record 16](#_Toc4758897)

[7.2.3 Field delimited simple models 16](#_Toc4758898)

[7.2.3.1 1 section 16](#_Toc4758899)

[7.2.3.2 Multiple sections 16](#_Toc4758900)

[7.2.4 Field delimited complex models 16](#_Toc4758901)

[7.2.4.1 Multiple reports per record: have to include schema models 16](#_Toc4758902)

[8. Annex – development 17](#_Toc4758903)

[8.1 Tool architecture 17](#_Toc4758904)

[8.2 Main changes: discussion 17](#_Toc4758905)

[8.2.1 Working with missing values 17](#_Toc4758906)

[8.2.2 File column converters 18](#_Toc4758907)

[8.2.3 Rebuilding architecture 19](#_Toc4758908)

[8.2.4 Some tips 19](#_Toc4758909)

Executive Summary

The C3S 311a Lot 2 (Global Land and Marine Observations Database) service is concerned with the provision of globally available land and marine surface meteorological records. The service includes inventorying of, and brokering access to, data sources, their harmonization (via conversion to a Common Data Model (CDM), merging, and quality assurance) and their provision via the Copernicus Climate Change Service (C3S) Climate Data Store (CDS).

The version history is given below:

|  |  |  |
| --- | --- | --- |
| **Version** | **Release Date** | **Release notes** |
| 1.0 |  |  |
| 2.0 |  |  |

# Introduction

The Copernicus Climate Change Service (C3S) Global Land and Marine Observations Database service provides brokered access to global historical holdings of surface meteorological observations. It builds upon existing national, regional and global efforts to create an augmented set of quality assured holdings that can be used to create datasets, products and services.

**This document will contain relevant detailed information on the software and configuration used to produce the marine data holdings for service providers to be able to access, use, modify and/or update where necessary. The current version of this document is focused on the data flow leading to the December 2018 beta release (header and observations tables) and the scripts and software tools derived from it.**

The document is ordered as follows:

* Section 2 summarizes the marine code requirements.
* Section 3 describes the marine main data flows.
* The Annex to this document provides further insight into scripts and tools.

# Tool overview

The mdf\_reader is a python3 tool designed to read (|and decode? ) *meteorological data formats* where:

* Data are stored in a human-readable manner: “ASCII” format
* Data are organized in single line reports
* Reports have an internal structure that can be modelized in a *schema* and have the schema defined (and code tables)
* Reports may be fixed width or field delimited

The mdf\_reader uses the information provided in a *datamodel*[[1]](#footnote-1) to read its data to a *pandas* DataFrame. The resulting DataFrame has its columns names and data types set according to the element’s description in the *schema*. The values in the columns are the original data file values, where the following transformations can be applied to numeric elements:

* Numeric data decoding to base 10: standard numeric encodings supported (see…)
* Numeric data conversion to parameter units: scale | offset

Additionally, the mdf\_reader provides a framework to store and access the code tables that different data file formats use.

Several data file formats have been added to the tool, meaning that its *schema* is internally available to the tool and thus need not be provided as an input file. Also, their code tables are included. Data file formats already included are listed in table xxx.

It is, however, possible to read other formats, as long as a valid *schema* can be built for it and provided it to the tools as input. See schemas to build your schema….

# Time stamps management

# How to register a new data file format (in the application)

To register a new data format in the application, a series of files and parameters need to be included in the mdf\_reader. To map the new data file format correctly to its configuration files, it is important, firstly, to keep consistency in the data file format name we use across them. We will here refer to it as ***dff\_name***.

1. **Schema file:**

* Build schema file following section 0
* Validate schema json format (read it with schema\_manager.py or https://jsonlint.com)
* Archive schema as ./schemas/***dff\_name***.json

1. **Register data file format in the application:**

Append ***dff\_name*** to ﻿*\_format\_list* in properties.py

1. **Code tables**

* Build code tables json files following section 6
* Validate code tables format (read with code\_table\_manager.py)
* Archive in as ./code\_tables/***dff\_name***/

# References

# Annex

## Time stamps management

For data sets with time versions of code tables, we need to access the year of the observation. This must be accomplished in a unique way, regardless of the different ways data formats report dates.

The objective is to have a column with a pandas datetime object, that we can then use in a unique manner (Say: pd.DatetimeIndex(df[datetime column]).year). Time conversion is only optional as in some cases, it can make the required user input more complicated (decimal hours in imma1, just as a first example) and it is not that necessary at this stage in which we just mean to read data, not to map it.

How we create this column depends on how the date (and time information) is reported in the data:

1. Data has date (|datetime) information in a single element time stamp: convert that element to datetime object

* User declares the element as a ‘datetime’ type in the schema *column\_type* descriptor
* User informs of the date (|datetime) format in the schema *datetime\_format* descriptor
* The data element is converted to datetime object with the converters.object\_to\_datetime class (pd.to\_datetime())

1. Data has date and time information split in different elements: add a \_datetime column

* User declares the columns where date can be built from in the schema header *date\_parser* descriptor
* Currently: this is passed to a special function that prepends a \_datetime column to the data frame

(Change to: prepend the \_datetime column. Use the converter..? no, looks quite different….is’s not converting, it is creating a new thing….)

Problem is that we doing this in chunks, writing this to buffer and then reading: because how pandas does this, the final data frame can have its datetime column as an object if a single record is not convertible!!! how about declaring not valid records as NaT before passing to the buffer?:

* When we read from the buffer, we cannot specify out datetime columns as ‘datetime’: that option is not supported in pandas
* We have to tell pandas to try parse the dates in that column and that data element will be parsed ( and converted to datetime object) by pandas on reading
* Pandas will return the column as an “object” if a single record is not convertible!!!

How about declaring not valid records as NaT before passing to the buffer?:

* pd.to\_datetime used in the converter.object\_to\_datetime() and in functions.df\_prepend\_datetime:

*If ‘coerce’, then invalid parsing will be set as NaT*

This means that, hopefully, a record that is not valid is written as a NaT to the buffer, and hopefully converted by pandas parser as a NaT, instead of setting the full thing to an object….

Yes!! Test:

2010 7**99** 0 ………..,20100701,0000, ………

2010**13** 1 0 …………,20100701,0000, ……

2010 7 1 0 ………….,201007**99**,0000, …….

2010 7 1 0 ………….,2010**13**01,0000. ……….

﻿ \_datetime c99

\_datetime Odate

0 NaT 2010-07-01

1 NaT 2010-07-01

2 2010-07-01 NaT

3 2010-07-01 NaT

4 2010-07-01 2010-07-01

Notes: no time zone / time zone awareness is currently added

***Applied solution as proposal:***

e.g. imma1 data with cisdm\_dbo supplemental data.

Date definition in schemas:

* imma1: in schema header:

"date\_parser":{"section":"core","elements":["YR","MO","DY"],"format":"%Y-%m-%d"}

* cisdm\_dbo: in schema elements

"Odate": {"description":"Observation date (yyyymmdd)", "field\_length":8,"column\_type":"datetime", "scale":"%Y%m%d"},

Resulting \_datetime elements:

﻿ \_datetime c99

\_datetime Odate

0 2010-07-01 2010-07-01

1 2010-07-01 2010-07-01

2 2010-07-01 2010-07-01

How datetime columns are parsed and passed to pd.to\_datetime():

1. Columns declared in the date\_parser[‘elements’] are joined with a hyphen
2. Then the resulting string column is passed to a pd.to\_datetime() to create the datetime object

Other options include passing directly the columns containing the datetime information in the original data frame, but, pd.to\_datetime needs these columns to be named with: *keys can be common abbreviations like [‘year’, ‘month’, ‘day’, ‘minute’, ‘second’, ‘ms’, ‘us’, ‘ns’]) or plurals of the same*. That would make the definition of the date\_parser in the schema more complicated, as specific names would have to be assigned. Some data formats have the date in a single field. Additionally, if the day field is not available, the current approach, with a format like "%Y%m” would still be able to create a datetime object, whereas passing the individual columns would mean having to fake a day column or pd.to\_datetime failing the create the object.

The approach chosen has been thought to be the most widely applicable.

## File format models

As pointed out in section 3, the mdf\_reader is a python tool designed to read (|and decode? ) *meteorological data files* where:

* Data are stored in a human-readable manner: “ASCII” format
* Data are organized in reports
* Reports have an internal structure that can be modelized in a *schema*
* Reports (1 or multiple) are stored in a single line ( data file record)
* Reports may be fixed width or field delimited

### Fixed width simple models

#### 1 section

Alias: fw\_simple\_1



#### Multiple sections

Alias: fw\_simple\_x



### Fixed width complex models

#### IMMA1 model – multiple and optional sections

Alias: fw\_complex\_opt





#### TDF-11 model – multiple and exclusive sections

Alias: fw\_complex\_exc





#### TD14 model – multiple reports per record with common sections ( or header)

Alias: fw\_x\_multi



#### TD14 model – multiple reports per record

Alias: fw\_1\_multi



### Field delimited simple models

#### 1 section

Alias: dl\_simple\_1

See section 9.1.1.1

#### Multiple sections

Alias: dl\_simple\_x

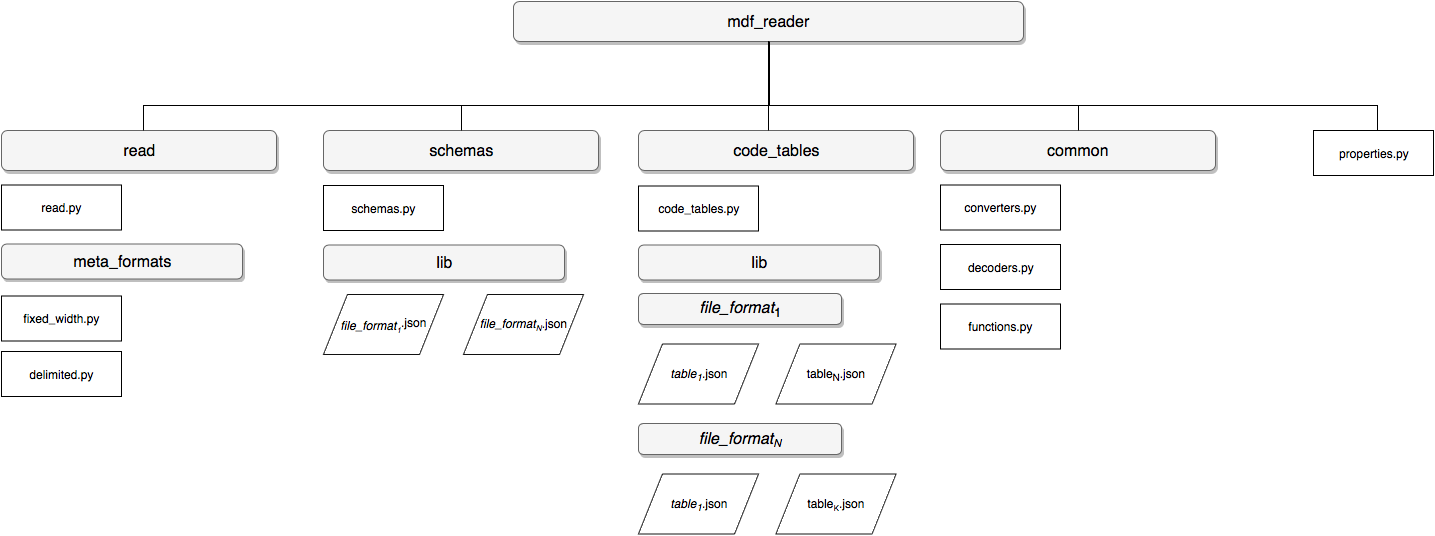
See section 9.1.1.2

### Field delimited complex models

#### Multiple reports per record: have to include schema models

# Annex – development

## Tool architecture



## Main changes: discussion

### Working with missing values

***What:***

In v0, global variables controlled the value of the missing value within the read DF, depending on the declared element data type (imiss, omiss, etc…). This value was assign during the reading, using the data type specific converters passed to the reader.

This was to prevent pandas promoting integer types to floating when there’s a missing value in the column, as NaN in floating.

**Why change:**

Seems artificial to set an imiss, fmiss: it is “hard” to maintain, and depending on the ie. Int type (int8, int16…) one or other can be used. The use of uint as an supported data type is, additionally, not possible, as usually the imiss < 0.

Depending on the parameter, its units, scale factor, this set imiss might get into the “plausible” limits of it…..

It is hard to maintain!

***Ok, we would lose control on the actual dtype of the column, but then on writing to output we can always go back to the schema, and configuration gets simplified***

Also, the data type promotion is sufficiently reported and justified:

<https://pandas.pydata.org/pandas-docs/version/0.22/gotchas.html#nan-integer-na-values-and-na-type-promotions>: “*This trade-off is made largely for memory and performance reasons, and also so that the resulting Series continues to be “numeric”. One possibility is to use dtype=object arrays instead”*

<https://jakevdp.github.io/PythonDataScienceHandbook/03.04-missing-values.html>

**Applied solution:**

* Converters are removed from initial file|buffer reading (no more filling with set missing\_values)
* Initial read from buffer is performed with dtypes for all elements as “object” (see below 10.2.2)
* Element actual dtype attribution is performed decoding and conversion afterwards
* Missing vaues are detected and flagged (python defaults) also during decoding and conversion

**Side effects:**

* Integer elements will be automoatically promoted to floating point dtype if missing value is (not) present, to be able to accommodate the floating NaN

### File column converters

***What:***

In the earlier version, file columns are parsed on reading and converted to its declared data type, adding also the corresponding missing value tag. This is done with the *converters* option in read\_fwf. After reading, elements to with a scaling factors are scaled and reconverted to the declared column data type.

***Why change:***

* Converters performance is questionable, see link below, plus does not seem flexible enough to handle conversion and scaling.

https://stackoverflow.com/questions/42462906/pandas-read-csv-converters-performance-issue

* Additionally, in between formatting in the initial reading and scaling, there can be a mismatching between the column\_type and the column\_value. It is hard to clearly define to what stage of the data ingestion the data format (column\_format) refers to.

***Proposal:***

1. Read columns as objects (mind gaps and meaningful blanks!!!)
2. Apply dtypes and scales/offsets afterwards
3. Also mind how we build/call the conversion functions:

<https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6>

***Applied solution as proposal:***

Test of core section of imma file (no. records = 487613, chunksize = 500000)

Initial approach timings (convert and flag on reading, then apply scale and re-convert types):

﻿INFO [20190225 08:24:17](data\_reader.py) Reading section

INFO [20190225 08:25:05](data\_reader.py) done reading

INFO [20190225 08:25:05](data\_reader.py) Scaling

HR, LAT, LON, D, W, SLP, PPP, AT, WBT, DPT, WH, SD, SP, SH

INFO [20190225 08:25:05](data\_reader.py) done scaling

New approach timings (read as object, then scale-convert-flag):

﻿INFO [20190225 08:23:15](data\_reader\_vect.py) Reading section elements

INFO [20190225 08:23:15](data\_reader\_vect.py) core ...

INFO [20190225 08:23:28](data\_reader\_vect.py) done

INFO [20190225 08:23:28](data\_reader\_vect.py) Scaling

YR,MO,DY,HR,LAT,LON,IM,ATTC,TI,LI,DS,VS,NID,II,ID,C1,DI,D,WI,W,VI,VV,WW,W1,SLP,A,PPP,IT,AT,WBTI,WBT,DPTI,DPT,SI,SST,N,NH,CL,HI,H,CM,CH,WD,WP,WH,SD,SP,SH

INFO [20190225 08:23:52](data\_reader\_vect.py) Scaling done

|  |  |  |  |
| --- | --- | --- | --- |
|  | Read | Scale | Total |
| Initial approach | 48 s | < 1 s | 48 s |
| New approach | 13 s | 24 s | 37 s |

New approach is 11 seconds faster, not that much!!! ☺ . We anyways stick to it, as there is more consistency of data types this way than the other.

We could still improve the scaling step by vectorizing with numpy arrays the conversion functions, but some features as “error coercing” seem not to be supported in np.astype(), and this is important for this application: would have to look for ways around in numpy to handle situations like a letter in a numeric field.

### Rebuilding architecture

### Some tips

See <https://docs.python-guide.org/writing/structure/> for modules (best importing practices) and packages overall.

1. Wiki: A **data model** (or **datamodel**) is an abstract model that organizes elements of [data](https://en.wikipedia.org/wiki/Data) and standardizes how they relate to one another and to properties of the real world entities [↑](#footnote-ref-1)