hermes_core

HERMES SOC Team

Mar 26, 2024

CONTENTS

This is the documentation for the hermes_core Python Package.

CHAPTER

ONE

RELEASE HISTORY

1.1 Full Changelog

This project uses semantic versioning.

1.1.1 Latest

- Added data class to hold measurements and to save to CDF files

1.1.2 0.2.0 (2023-03-22)

This release includes improvements tested in the second dataflow test. Since the last release, the improvements are as follows:

- Uses fstrings instead of the format syntax
- Adds a log message on import to show the version number of the package
- Documentation content and styling improvements
- Switches from using `setup.py` to `pyproject.toml` for package
- Bug fixes for supporting `pathlib`'s Path objects, and a permissions bug to the devcontainer
- Using f-strings and log on import by @ehsteve in https://github.com/HERMES-SOC/hermes_core/pull/28
- Added latest versions of python to testing by @ehsteve in https://github.com/HERMES-SOC/hermes_core/pull/29
- Add to the Documentation the location for where config files should be store (dynamically) by @dbarrous in https://github.com/HERMES-SOC/hermes_core/pull/35
- Fix devcontainer config to use new vscode user by @dbarrous in https://github.com/HERMES-SOC/hermes_core/pull/32
- Update to docs, added logo, updated theme colors and favicon by @ehsteve in https://github.com/HERMES-SOC/hermes_core/pull/37
- Fix to version number and move to pyproject.toml usage by @ehsteve in https://github.com/HERMES-SOC/hermes_core/pull/40
- Bug fix by @ehsteve in https://github.com/HERMES-SOC/hermes_core/pull/41

1.1.3 0.1.0 (2022-10-05)

This version release was tested in the first HERMES Ground System data flow test.

- First draft of python packaging including sphinx documentation based on the sunpy package template
- First draft of the documentation including coding standards for the HERMES ecosystem
- Automated testing and coverage using GitHub actions
- Logging support
- Configuration support
- Utilities parsing compliant filenames for level 0 binary files and creating and parsing higher level filenames

CHAPTER

TWO

CALIBRATION AND MEASUREMENT ALGORITHM DOCUMENT (CMAD)

HERMES consists of multiple instruments, each of which hosts its Calibration and Measurement Algortihm Document. See the documentation for each instrument.

CHAPTER

THREE

USER'S GUIDE

Welcome to our User guide. For more details checkout the *API Reference*.

3.1 A Brief Tour

Insert a tour here.

3.2 Opening and Writing HERMES Data

3.2.1 Overview

The `HermesData` class provides a convenient and efficient way to work with HERMES science CDF data files. The point of this class is to simplify data management, enhances data discovery, and facilitates adherence to CDF standards.

CDF (Common Data Format) files are a binary file format commonly used by NASA scientific research to store and exchange data. They provide a flexible structure for organizing and representing multidimensional datasets along with associated metadata. CDF files are widely used in space physics. Because of their versatility, CDF files can be complex. CDF standards exist to make it easier to work with these files. International Solar-Terrestrial Physics (ISTP) compliance is a set of standards defined by the Space Physics Data Facility (SPDF). ISTP compliance ensures that the data adheres to specific formatting requirements, quality control measures, and documentation standards. Uploading CDF files to the NASA SPDF archive requires conforming to the ISTP guidelines. In addition, HERMES maintains it's own standards in the CDF guide.

The CDF C library must be properly installed in order to use this package to save and load CDF files. The CDF library can be downloaded from the SPDF CDF Page to use the CDF libraries in your local environment. Alternatively, the CDF library is installed and available through the HERMES development Docker container environment. For more information on the HERMES Docker container please see our *Development Environment Page*.

To make it easier to work with HERMES data, the `HermesData` class facilitates the abstraction of HERMES CDF files. It allows users to read and write HERMES data and is compliant with PyHC expectations. The data is stored in a `TimeSeries` table while the metadata is stored in dictionaries. `TimeSeries` is a Python class for handling scientific time series data that provides a convenient and familiar interface for working with tabular data. By loading the contents of a CDF file into a `TimeSeries` table, it becomes easier to manipulate, analyze, and visualize the data. Additionally, metadata attributes can be associated with the table, allowing for enhanced documentation and data discovery. The `HermesData` class aims to provide a simplified interface to reading and writing HERMES data and metadata to CDF files while automatically handling the complexities of the underlying CDF file format.

3.2.2 Creating a `HermesData` object

Creating a *`HermesData`* data container from scratch involves four pieces of data:

- **`timeseries` (required) - an `TimeSeries` containing the time dimension of**
  the data as well as at least one other measurement. This data structure must be used for all scalar time-varying measurement data.
- **`spectra` (optional) - an `NDCollection` containing one or more `NDCube` objects**
  representing higher-dimensional measurements and spectral data. This data must should be used for all vector or tensor-based measurement data.
- **`support` (optional) - a `dict[astropy.nddata.NDdata | astropy.units.Quantity]` containing one**
  or more non-time-varying (time invariant) measurements, time-invariant support or metadata variables.
- **`meta` (optional) - a `dict` containing global metadata information about the CDF. This data**
  structure must be used for all global metadata required for ISTP compliance.

Alternatively, a *`HermesData`* data container can be loaded from an existing CDF file using the *`load()`* function.

Creating a `TimeSeries` for `HermesData` `timeseries`

A *`HermesData`* must be initialized by providing a `TimeSeries` object with at least one measurement. There are many ways to initialize one but here is one example:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries(
...     time_start='2016-03-22T12:30:31',
...     time_delta=3 * u.s,
...     data={'Bx': u.Quantity(
...         value=[1, 2, 3, 4],
...         unit='nanoTesla',
...         dtype=np.uint16
...     )}
... )
```

Be mindful to set the right number of bits per measurement, in this case 16 bits. If you do not, it will likely default to float64 and if you write a CDF file, it will be larger than expected or needed. The valid `dtype` choices are uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, float164. You can also create your time array directly

```
>>> from astropy.time import Time, TimeDelta
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> times = Time('2010-01-01 00:00:00', scale='utc') + TimeDelta(np.arange(100) * u.s)
>>> ts = TimeSeries(
...     time=times,
...     data={'diff_e_flux': u.Quantity(
...         value=np.arange(100) * 1e-3,
...         unit='1/(cm**2 * s * eV * steradian)',
...         dtype=np.float32
...     )}
... )
```

Note the use of `time` and `astropy.units` which provide several advantages over using arrays of numbers and are required by *`HermesData`*.

Creating a `NDCollection` for `HermesData` spectra

The *`HermesData`* object leverages API functionality of the `ndcube` package to enable easier analysis of higher-dimensional and spectral data measurements. The main advantage that this package provides in in it's handling of coordinate transformations and slicing in real-world-coordinates compared to using index-based slicing for higher-dimensional data. For more information about the `ndcube` package and its API functionality please read the SunPy NDCube documentation.

You can create a `NDCollection` object using an approach similar to the following example:

```
>>> import numpy as np
>>> from astropy.wcs import WCS
>>> from ndcube import NDCube, NDCollection
>>> spectra = NDCollection(
...     [
...         (
...             "example_spectra",
...             NDCube(
...                 data=np.random.random(size=(4, 10)),
...                 wcs=WCS(naxis=2),
...                 meta={"CATDESC": "Example Spectra Variable"},
...                 unit="eV",
...             ),
...         )
...     ]
... )
```

The `NDCollection` is created using a list of `tuple` containing named `(str, NDCube)` pairs. Each `NDCube` contains the required data array, a `WCS` object responsible for the coordinate transformations, optional metadata attributes as a `dict`, and an `units` unit that is used to treat the data array as an `Quantity`.

Creating a `dict` for `HermesData` support

The *`HermesData`* object also accepts additional arbitrary data arrays, so-called non-record-varying (NRV) data, which is frequently support data. These data are required to be a `dict` of `NDData` or `Quantity` objects which are data containers for physical data. The *`HermesData`* class supports both `Quantity` and `NDData` objects since one may have advantages for the type of data being represented: `Quantity` objects in this support `dict` may be more advantageous for scalar or 1D-vector data while `NDData` objects in this support `dict` may be more advantageous for higher-dimensional vector data. A guide to the `nddata` package is available in the astropy documentation.

```
>>> from astropy.nddata import NDData
>>> support_data = {
...     "const_param": u.Quantity(value=[1e-3], unit="keV", dtype=np.uint16),
...     "data_mask": NDData(data=np.eye(100, 100, dtype=np.uint16))
... }
```

Metadata passed in through the `NDData` object is used by *`HermesData`* as variable metadata attributes required for ISTP compliance.

Creating a `dict` for `HermesData` `meta`

You must create a `dict` or `OrderedDict` containing the required CDF global metadata. The class function *`global_attribute_template()`* will provide you an empty version that you can fill in. Here is an example with filled in values.

```
>>> input_attrs = {
...     "DOI": "https://doi.org/<PREFIX>/<SUFFIX>",
...     "Data_level": "L1>Level 1",  # NOT AN ISTP ATTR
...     "Data_version": "0.0.1",
...     "Descriptor": "EEA>Electron Electrostatic Analyzer",
...     "Data_product_descriptor": "odpd",
...     "HTTP_LINK": [
...         "https://spdf.gsfc.nasa.gov/istp_guide/istp_guide.html",
...         "https://spdf.gsfc.nasa.gov/istp_guide/gattributes.html",
...         "https://spdf.gsfc.nasa.gov/istp_guide/vattributes.html"
...     ],
...     "Instrument_mode": "default",  # NOT AN ISTP ATTR
...     "Instrument_type": "Electric Fields (space)",
...     "LINK_TEXT": [
...         "ISTP Guide",
...         "Global Attrs",
...         "Variable Attrs"
...     ],
...     "LINK_TITLE": [
...         "ISTP Guide",
...         "Global Attrs",
...         "Variable Attrs"
...     ],
...     "MODS": [
...         "v0.0.0 - Original version.",
...         "v1.0.0 - Include trajectory vectors and optics state.",
...         "v1.1.0 - Update metadata: counts -> flux.",
...         "v1.2.0 - Added flux error.",
...         "v1.3.0 - Trajectory vector errors are now deltas."
...     ],
...     "PI_affiliation": "HERMES",
...     "PI_name": "HERMES SOC",
...     "TEXT": "Valid Test Case",
... }
```

Here is an example using the *`global_attribute_template()`* function to create a minimal subset of global metadata attributes:

```
>>> from hermes_core.timedata import HermesData
>>> input_attrs = HermesData.global_attribute_template("eea", "l1", "1.0.0")
```

Using Defined Elements to create a `HermesData` Data Container

Putting it all together here is instantiation of a *HermesData* object:

```
>>> from hermes_core.timedata import HermesData
>>> hermes_data = HermesData(
...     timeseries=ts,
...     support=support_data,
...     spectra=spectra,
...     meta=input_attrs
... )
```

For a complete example with instantiation of all objects in one code example:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> from ndcube import NDCube, NDCollection
>>> from astropy.nddata import NDData
>>> from hermes_core.timedata import HermesData
>>> # Create a TimeSeries structure
>>> data = u.Quantity([1, 2, 3, 4], "gauss", dtype=np.uint16)
>>> ts = TimeSeries(time_start="2016-03-22T12:30:31", time_delta=3 * u.s, data={"Bx":␣
↪data})
>>> # Create a Spectra structure
>>> spectra = NDCollection(
...     [
...         (
...             "example_spectra",
...             NDCube(
...                 data=np.random.random(size=(4, 10)),
...                 wcs=WCS(naxis=2),
...                 meta={"CATDESC": "Example Spectra Variable"},
...                 unit="eV",
...             ),
...         )
...     ]
... )
>>> # Create a Support Structure
>>> support_data = {
...     "data_mask": NDData(data=np.eye(100, 100, dtype=np.uint16))
... }
>>> # Create Global Metadata Attributes
>>> input_attrs = HermesData.global_attribute_template("eea", "l1", "1.0.0")
>>> # Create HermesData Object
>>> hermes_data = HermesData(
...     timeseries=ts,
...     support=support_data,
...     spectra=spectra,
...     meta=input_attrs
... )
```

The *HermesData* is mutable so you can edit it, add another measurement column or edit the metadata after the fact. Your variable metadata can be found by querying the measurement column directly.

```
>>> hermes_data.timeseries['Bx'].meta.update(
...     {"CATDESC": "X component of the Magnetic field measured by HERMES"}
... )
>>> hermes_data.timeseries['Bx'].meta
```

The class does its best to fill in metadata fields if it can and leaves others blank that it cannot. Those should be filled in manually. Be careful when editing metadata that was automatically generated as you might make the resulting CDF file non-compliant.

3.2.3 Creating a `HermesData` from an existing CDF File

Given a current CDF File you can load it into a *`HermesData`* by providing a path to the CDF file:

```
>>> from hermes_core.timedata import HermesData
>>> hermes_data = HermesData.load("hermes_eea_default_ql_20240406T120621_v0.0.1.cdf")
```

The *`HermesData`* can the be updated, measurements added, metadata added, and written to a new CDF file.

3.2.4 Adding data to a `HermesData` Container

A new set of measurements or support data can be added to an existing instance. Remember that new measurements must have the same time stamps as the existing ones and therefore the same number of entries. Support data can be added as needed. You can add the new measurements in one of two ways.

The more explicit approach is to use *`add_measurement()`* function:

```
>>> data = u.Quantity(np.arange(len(hermes_data.timeseries['Bx'])), 'Gauss', dtype=np.
↪uint16)
>>> hermes_data.add_measurement(measure_name="By", data=data, meta={"CATDESC": "Test␣
↪Metadata"})
```

To add non-time-varying support data use the *`add_support()`* function:

```
>>> hermes_data.add_support(
...     name="Calibration_const",
...     data=u.Quantity(value=[1e-1], unit="keV", dtype=np.uint16),
...     meta={"CATDESC": "Calibration Factor", "VAR_TYPE": "support_data"},
... )
>>> hermes_data.add_support(
...     name="Data Mask",
...     data=NDData(data=np.eye(5, 5, dtype=np.uint16)),
...     meta={"CATDESC": "Diagonal Data Mask", "VAR_TYPE": "support_data"},
... )
```

3.2.5 Adding metadata attributes

Additional CDF file global metadata and variable metadata can be easily added to a *HermesData* data container. For more information about the required metadata attributes please see the *HERMES CDF Format Guide*

Global Metadata Attributes

Global metadata attributes can be updated for a *HermesData* object using the object's *meta* parameter which is an `OrderedDict` containing all attributes.

Required Global Attributes

The *HermesData* class requires several global metadata attributes to be provided upon instantiation:

- `Descriptor`
- `Data_level`
- `Data_version`

A *HermesData* container cannot be created without supplying at lest this subset of global metadata attributes. For assistance in defining required global attributes, please see the *global_attribute_template()* function.

Derived Global Attributes

The *HermesDataSchema* class derives several global metadata attributes required for ISTP compliance. The following global attributes are derived:

- `CDF_Lib_version`
- `Data_type`
- `Generation_date`
- `HERMES_version`
- `Logical_file_id`
- `Logical_source`
- `Logical_source_description`
- `Start_time`

For more information about each of these attributes please see the *HERMES CDF Format Guide*

Using a Template for Global Metadata Attributes

A template of the required metadata can be obtained using the *global_attribute_template()* function:

```
>>> from collections import OrderedDict
>>> from hermes_core.timedata import HermesData
>>> HermesData.global_attribute_template()
OrderedDict([('DOI', None),
         ('Data_level', None),
         ('Data_version', None),
```

(continues on next page)

(continued from previous page)

```
         ('Descriptor', None),
         ('HTTP_LINK', None),
         ('Instrument_mode', None),
         ('Instrument_type', None),
         ('LINK_TEXT', None),
         ('LINK_TITLE', None),
         ('MODS', None),
         ('PI_affiliation', None),
         ('PI_name', None),
         ('TEXT', None)])
```

You can also pass arguments into the function to get a partially populated template:

```
>>> from collections import OrderedDict
>>> from hermes_core.timedata import HermesData
>>> HermesData.global_attribute_template(
...     instr_name='eea',
...     data_level='l1',
...     version='0.1.0'
... )
OrderedDict([('DOI', None),
         ('Data_level', 'L1>Level 1'),
         ('Data_version', '0.1.0'),
         ('Descriptor', 'EEA>Electron Electrostatic Analyzer'),
         ('HTTP_LINK', None),
         ('Instrument_mode', None),
         ('Instrument_type', None),
         ('LINK_TEXT', None),
         ('LINK_TITLE', None),
         ('MODS', None),
         ('PI_affiliation', None),
         ('PI_name', None),
         ('TEXT', None)])
```

This can make the definition of global metadata easier since instrument teams or users only need to supply pieces of metadata that are in this template. Additional metadata items can be added if desired. Once the template is instantiated and all attributes have been filled out, you can use this during instantiation of your `HermesData` container.

Variable Metadata Attributes

Variable metadata requirements can be updated for a `HermesData` variable using the variable's `meta` property which is an `OrderedDict` of all attributes.

Required Variable Attributes

The *HermesData* class requires one variable metadata attribute to be provided upon instantiation:

- CATDESC : (Catalogue Description) This is a human readable description of the data variable.

Derived Variable Attributes

The *HermesDataSchema* class derives several variable metadata attributes required for ISTP compliance.

- TIME_BASE
- RESOLUTION
- TIME_SCALE
- REFERENCE_POSITION
- DEPEND_0
- DISPLAY_TYPE
- FIELDNAM
- FILLVAL
- FORMAT
- LABLAXIS
- SI_CONVERSION
- UNITS
- VALIDMIN
- VALIDMAX
- VAR_TYPE

For more information about each of these attributes please see the *HERMES CDF Format Guide*

Using a Template for Variable Metadata Attributes

A template of the required metadata can be obtained using the *measurement_attribute_template()* function:

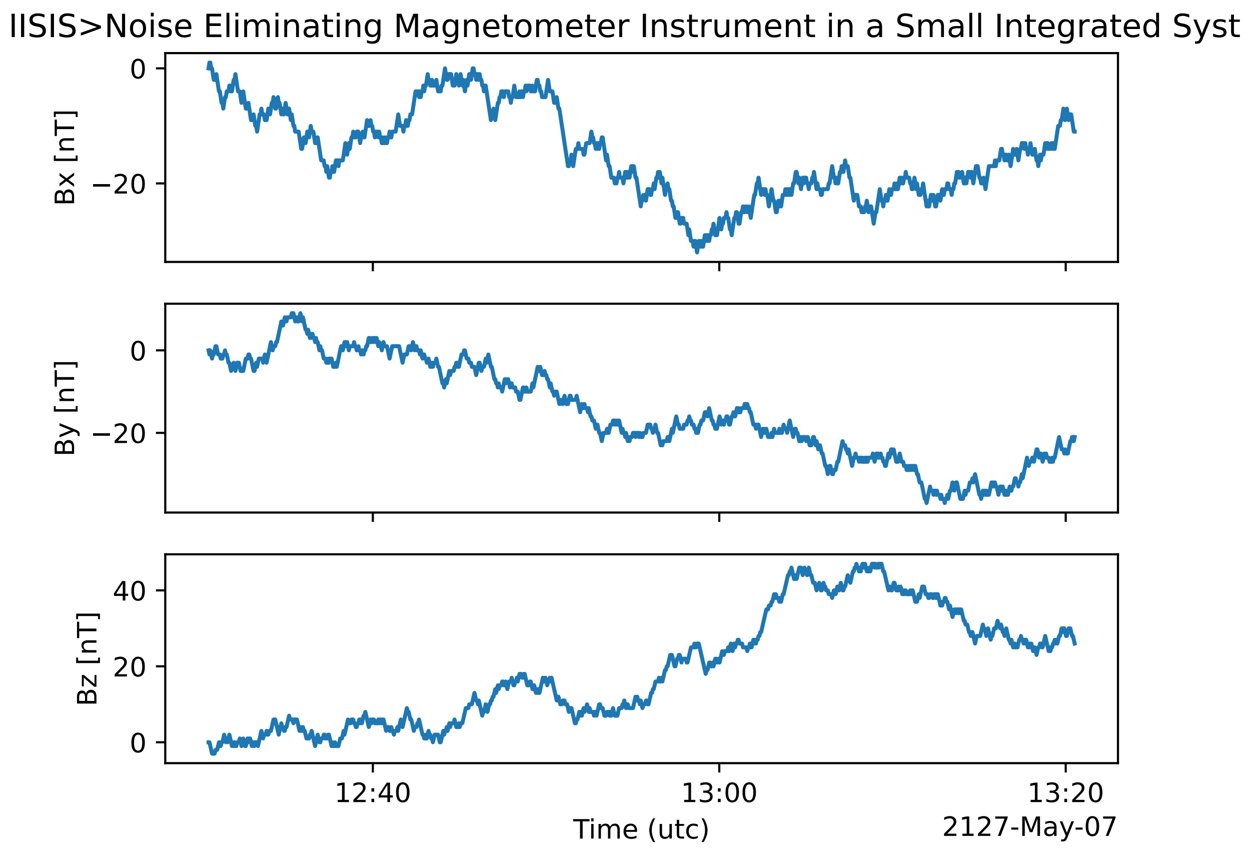
```
>>> from collections import OrderedDict
>>> from hermes_core.timedata import HermesData
>>> HermesData.measurement_attribute_template()
OrderedDict([('CATDESC', None)])
```

If you use the *add_measurement()* function, it will automatically fill most of them in for you. Additional pieces of metadata can be added if desired.

3.2.6 Visualizing data in a `HermesData` Container

The *`HermesData`* provides a quick way to visualize its data through *`plot`*. By default, a plot will be generated with each measurement in its own plot panel.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> from hermes_core.timedata import HermesData
>>> bx = np.concatenate([[0], np.random.choice(a=[-1, 0, 1], size=1000)]).cumsum(0)
>>> by = np.concatenate([[0], np.random.choice(a=[-1, 0, 1], size=1000)]).cumsum(0)
>>> bz = np.concatenate([[0], np.random.choice(a=[-1, 0, 1], size=1000)]).cumsum(0)
>>> ts = TimeSeries(time_start="2016-03-22T12:30:31", time_delta=3 * u.s, data={"Bx": u.
↪Quantity(bx, "nanoTesla", dtype=np.int16)})
>>> input_attrs = HermesData.global_attribute_template("nemisis", "l1", "1.0.0")
>>> hermes_data = HermesData(timeseries=ts, meta=input_attrs)
>>> hermes_data.add_measurement(measure_name=f"By", data=u.Quantity(by, 'nanoTesla',␣
↪dtype=np.int16))
>>> hermes_data.add_measurement(measure_name=f"Bz", data=u.Quantity(bz, 'nanoTesla',␣
↪dtype=np.int16))
>>> fig = plt.figure()
>>> hermes_data.plot()
>>> plt.show()
```



3.2.7 Writing a CDF File

The *HermesData* class writes CDF files using the `pycdf` module. This can be done using the *save()* method which only requires a path to the folder where the CDF file should be saved. The filename is automatically derived consistent with HERMES file naming requirements. If no path is provided it writes the file to the current directory. This function returns the full path to the CDF file that was generated. From this you can validate and distribute your CDF file.

3.2.8 Validating a CDF File

The *HermesData* uses the `istp` module for CDF validation, in addition to custom tests for additional metadata. A CDF file can be validated using the *validate()* method and by passing, as a parameter, the full path to the CDF file to be validated:

```
>>> from hermes_core.util.validation import validate
>>> validation_errors = validate(cdf_file_path)
```

This returns a `list[str]` that contains any validation errors that were encountered when examining the CDF file. If no validation errors were found the method will return an empty list.

3.3 HERMES CDF Format Guide

3.3.1 1. Introduction

The *HermesDataSchema* class provides an interface to examine the HERMES CDF Format Guide.

1.1 Purpose and Scope

This document is provided as a reference for construction of HERMES standard CDF files. It is intended to complement information available from the Space Physics Data Facility (listed in Sec. 1.2). It lays down REQUIREMENTS and RECOMMENDATIONS for Level 2 (and above) CDF files that are intended for public access, and should be taken as RECOMMENDATIONs for all other mission CDFs. This document is based on discussions within the HERMES Science Data Working Group (HSDWG) and personnel at NASA's Space Physics Data Facility (SPDF). It is intended to provide sufficient reference material to understand CDF files and the requirements for creating HERMES CDF files, and to understand the structure and contents of the resulting CDF files.

1.2 References

Relevant documents that provide background material and support details provided in this guide are listed below:

- SPDF CDF User's Guide
- SKTEditor
- ISTP Guidelines
- ISTP/IACG Global Attributes

3.3.2 2. HERMES Science Investigations

The HERMES Instrument Suite will make high-time resolution measurements of plasmas (ions and electrons) and magnetic fields. The HERMES Instrument Suite consists of the following complement of instruments:

- **Electron Electrostatic Analyzer (EEA): The EEA provides measurements of**
  low-energy electrons in the solar wind and in Earth's deep magnetotail by measuring electron flux as functions of energy and direction.
- **Miniaturized Electron pRoton Telescope (MERIT): The MERiT instrument**
  measures the flux of high-energy electrons and ions with two telescopes pointing in opposite directions and nominally spanning the forward and reverse Parker Spiral.
- **Noise Eliminating Magnetometer In a Small Integrated System (NEMISIS):**
  NEMISIS is comprised of a fluxgate magnetometer (M0) at the end of a deployable boom and two inductive magnetometers (M1, M2) mounted on the HERMES platform. Each sensor measures the vector magnetic field at its location. Measurements from the 3 sensors are combined to reduce the contribution to the local field due to Gateway.
- **Solar Probe Analyzer for Ions (SPAN-I): The SPAN-i ion sensor measures**
  Interplanetary and Magnetotail ion flux as functions of direction and energy/charge from several eV/q to 20 keV/q. A time-of-flight section enables it to sort particles by their mass/charge ratio, permitting differentiation of ion species.

HERMES Instrument Team Facilities (ITFs) are the principal institutions associated with each of the HERMES science investigations. These facilities and their personnel provide support to the operation of their instruments and the overall data processing and distribution effort for HERMES science data products. The institutions listed in Table 2-1 have responsibility for each of the investigations and their corresponding instruments.

Table 1: Table 2-1 HERMES ITF Summary

HERMES Investigation	Managing Institution	Principal Investigator
Electron Electrostatic Analyzer (EEA)	Goddard Space Flight Center (GSFC)	D. Gershman
Miniaturized Electron pRoton Telescope (MERIT)	Goddard Space Flight Center (GSFC)	S. Kanekal
Noise Eliminating Magnetometer In a Small Integrated System (NEMISIS)	Goddard Space Flight Center (GSFC), University of Michigan	E. Zesta, M. Moldwin (Co-I),
Solar Probe Analyzer for Ions (SPAN-I)	University of California, Berkeley (UCB), Space Sciences Laboratory (SSL)	R. Livi

3.3.3 3. Conventions

All HERMES scientific data products that will be shared between HERMES entities (e.g. ITFs, IDS groups) or made available to the general research community will be stored as CDF data files and are expected to be compatible with CDF version 3.5. Data that will not be shared beyond an individual team may be stored in any format that is convenient for that team.

3.1 Science Product Naming Conventions

The HERMES data products will be produced with the following filename format where the individual identifying components are described in Table 3-1. Additionally, to ensure software compatibility between disparate systems, filenames will consist of all lowercase characters. Filenames are used as a system identifier for a logical grouping of data and are also stored in the `Logical_file_id` global attribute field (see Section 4.1). It is expected that filenames will be created dynamically from the attributes identified in Section 4 of this document.

Filename Format

```
scId_instrumentId_mode_dataLevel_optionalDataProductDescriptor_startTime_vX.Y.Z.ext
```

Table 2: Table 3-1: Filename Component Description

Short Name	Description	Valid Options
scID	Spacecraft ID	`hermes`
instrumentId	Instrument or investigation identifier shortened to three letter acronym.	`eea`, `mrt`, `nms`, `spn`
mode	*TBS*	*TBS*
dataLevel	The level to which the data product has been processed	`l0`, `l1`, `ql`, `l2`, `l3`, `l4`
optionalDataProductDescriptor	This is an optional field that may not be needed for all products. Where it is used, identifier should be short (e.q. 3-8 characters) descriptors that are helpful to end-users. If a descriptor contains multiple components, underscores are used to separate those components.	An optional time span may be specified as "2s" to represent a data file that spans two seconds. In this case, "10s" and "5m" are other expected values that correspond with ten seconds and 5 minutes respectively.
startTime	The start time of the contained data given in ISO 8601 format.	`20230519T000003`
vX.Y.Z	The 3-part version number of the data product. Full description of this identifier is provided in Section 3.1.1 of this document.	`v0.0.0`, `v`
.ext	The required file extension, where CDF is required.	`cdf`

3.1.1 Version Numbering Guidelines

The three-part version number contains the interface number, quality number, and bug fix/revision number. The initial release of CDF data that is suitable for scientific publication should begin with "v1.Y.Z". Each component of the version number is incremented in integer steps, as needed, and Table 3-2 describes the instances in which the value should be incremented. Release "v0.Y.Z" may be used for early development purposes.

Table 3: Table 3-2: Version Numbering Guidelines

Part	Name	Description
X	Interface Number	Increments in this number represent a significant change to the processing software and/or to the content/structure of the file. These changes may be incompatible with existing code. Increments in this number may require code changes to software.
Y	Quality Number	This number represents a change in the quality of the data in the file, such as change in calibration or increase in fidelity. Changes should not impact software but may require consideration when processing data.
Z	Bug Fix / Revision Number	This number changes to indicate minor changes to the contents of the file due to reprocessing of missing data. Any dependent data products should generally be reprocessed if this value changes.

3.3.4 4. Global Attributes

Global attributes are used to provide information about the data set as an entity. Together with variables and variable attributes, the global attributes make the data correctly and independently usable by someone not connected with the instrument team, and hence, a good archive product.

The required, recommended, and optional global attributes that have been identified for use with HERMES data products are listed below. Additional global attributes can be defined but they must start with a letter and can otherwise contain letters, numbers, and the underscore character (no other special characters allowed). Note that CDF attributes are case-sensitive and must exactly follow what is shown here.

Detailed descriptions of the attributes listed below are available at the ISTP/IACG Global Attributes Webpage.

4.1 Required Global Attributes

The following global attributes shown in Table 4-1 are required with HERMES data products. HERMES-specific values are provided where applicable. For each attribute the following information is provided:

- description: (`str`) A brief description of the attribute
- default: (`str`) The default value used if none is provided
- derived: (`bool`) Whether the attibute can be derived by the HERMES *`HermesDataSchema`* class
- required: (`bool`) Whether the attribute is required by HERMES standards
- validate: (`bool`) Whether the attribute is included in the *`validate()`* checks (Note, not all attributes that are required are validated)
- overwrite: (`bool`) Whether the *`HermesDataSchema`* attribute derivations will overwrite an existing attribute value with an updated attribute value from the derivation process.

Note that this table is derived from `hermes_core/data/hermes_default_global_cdf_attrs_schema.yaml`

Table 4: Table 4-1: Required Global Attributes

Attribute	description	default	derived	required	validate	overwrite
CDF_Lib_ve	Version of the CDF Binaries library used to generate the CDF File		True	True	False	False
DOI	DOI is a persistent Unique Digital Identifier with the form https://doi.org/<PREFIX>/<SUFFIX> with the <PREFIX> identifying the DOI registration authority and the <SUFFIX> identifying the dataset. The DOI should point to a landing page for additional information about the dataset. DOIs are typically created by the SPASE naming authority or archive.		False	True	True	False
Data_level	This attribute is used in file name creation and records the level of processsing done on the dataset. For HERMES the following are valid values: - l0>Level 0 - l1>Level 1 - l2>Level 2 - l3>Level 3 - l4>Level 4 - ql>Quicklook		False	True	False	True
Data_produc	This is an optional field that may not be needed for all products. Where it is used, identifier should be short (e.q. 3-8 characters) descriptors that are helpful to end- users. If a descriptor contains multiple components, underscores are used to separate those components.		False	False	False	True
Data_type	This attribute is used by CDF file writing software to create a filename. It is a combination of the following filename components: mode, data level, and optional data product descriptor.		True	True	False	True
Data_versior	This attribute identifies the version (vX.Y.Z) of a particular CDF data file.		False	True	True	False
Descriptor	This attribute identifies the name of the instrument or sensor that collected the data. Both a long name and a short name are given. For any data file, only a single value is allowed. For HERMES, the following are valid values: - EEA>Electron Electrostatic Analyzer - MERIT>Miniaturized Electron pRoton Telescope		False	True	True	False

4.2 Recommended Attributes

The following global attributes are recommended but not required with HERMES data products. HERMES-specific values are provided where applicable.

Table 5: Table 4-2: Recommended Attributes

Attribute	Description
Acknowledgement	This field indicates how the data should be cited.
Generated_by	This attribute indicates where users can get more information about this data and/or check for new versions.

4.3 Optional Attributes

Table 6: Table 4-2: Optional Attributes

Attribute	Description
Parents	This attribute lists the parent data files for files of derived and merged data sets. The syntax for a CDF parent is: "CDF>logical_file_id". Multiple entry values are used for multiple parents. This attribute is required for any HERMES data products that are derived from 2 or more data sources and the file names of parent data should be clearly identified. CDF parents may include source files with non-cdf extensions.
Skeleton_version	This is a text attribute containing the skeleton file version number.
Rules_of_use	Text containing information on citability and/or PI access restrictions. This may point to a World Wide Web page specifying the rules of use. Rules of Use are determined on both a mission and instrument basis, at the discretion of the PI.
Time_resolution	Specifies time resolution of the file, e.g., "3 seconds".

3.3.5 5. Variables

There are three types of variables that should be included in CDF files: * data, * support data, * metadata.

Additionally, required attributes are listed with each variable type listed below.

To facilitate data exchange and software development, variable names should be consistent across the HERMES instruments and four spacecraft. Additionally, it is preferable that data types are consistent throughout all HERMES data products (e.g. all real variables are CDF_REAL4, all integer variables are CDF_INT2, and flag/status variables are UINT2). This is not to imply that only these data types are allowable within HERMES CDF files. All CDF supported data types are available for use by HERMES.

For detailed information and examples, please see the `ISTP/IACG Webpage`

5.1 Data

These are variables of primary importance (e.g., density, magnetic field, particle flux). Data is always time (record) varying but can be of any dimensionality or CDF supported data type. Real or Integer data are always defined as having one element.

5.1.1 Naming

HERMES data variables must adhere to the following naming convention * `scId_instrumentId_paramName`

An underscore is used to separate different fields in the variable name. It is strongly recommended that variable names employ further fields, qualifiers and information designed to identify unambiguously the nature of the variable, instrument mode and data processing level, with sufficient detail to lead the user to the unique source file which contains the variable. It is recommended that these follow the order shown below.

- `scId_instrumentId_paramName[_coordSys][_paramQualifier][_subModeLevel][_mode][_dataLevel]`

where the required fields are described in Table 5-1 and the optional fields are described in Table 5-2. An example data variable would be `hermes_eea_n_gse_l2`.

Table 7: Table 5-1: Required Data Variable Fields

Required Field Name	Description
scId	Spacecraft identifier, see Table 3-1 for acceptable values
instrumentId	Instrument or investigation identifier, see Table 3-1 for acceptable values and note the caveats listed in Section 5.1.1.1.
paramName	Data parameter identifier, a short (a few letters) representation of the physical parameter held in the variable.

Table 8: Table 5-2: Optional Data Variable Fields

Optional Field Name	Description
coordSys	An acronym for the coordinate system in which the parameter is cast.
paramQualifier	Parameter descriptor, which may include multiple components separated by a "_" as needed (e.g. "pa_0" indicates a pitch angle of 0).
subModeLevel	Qualifier(s) to include mode and data level information supplementary to the following two fields.
mode	See Table 3-1 for acceptable values.
dataLevel	See Table 3-1 for acceptable values.

5.1.1.1 Caveats

Note the following caveats in the variable naming conventions:

- CDF variable names must begin with a letter and can contain numbers and underscores, but no other special characters.
- In general, the instrumentId field follows the convention used for file names as defined in Section 3.1. However, since variable names cannot contain a hyphen, an underscore should be used instead of a hyphen when needing to separate instrument components. For instance, "eea-ion" is a valid instrumentId in a filename but when used in a variable name, "eea_ion" should be used instead.

- To ensure software compatibility between disparate systems, parameter names will consist of all lowercase characters.

5.1.2 Required Epoch Variable

All HERMES CDF data files must contain at least one variable of data type CDF_TIME_TT2000, typically named "Epoch". This variable should normally be the first variable in each CDF data set. All time varying variables in the CDF data set will depend on either this "epoch" variable or on another variable of type CDF_TIME_TT2000 (e.g. hermes_eea_epoch). More than one CDF_TIME_TT2000 type variable is allowed in a data set to allow for more than one time resolution, using the required DEPEND_0 attribute (see Section 5.5) to associate a time variable to a given data variable. It is recommended that all such time variables use "epoch" within their variable name.

For ISTP, but not necessarily for all HERMES data, the time value of a record refers to the center of the accumulation period for the record if the measurement is not an instantaneous one. All HERMES time variables used as DEPEND_0 are strongly recommended to have DELTA_PLUS_VAR and DELTA_MINUS_VAR attributes which delineate the time interval over which the data was sampled, integrated, or otherwise representative of. This also locates the timetag within that interval.

The epoch datatype, CDF_TIME_TT2000, is defined as an 8-byte signed integer with the characteristics shown in Table 5-3.

Table 9: Table 5-3: Characteristics of CDF_TIME_TT2000

Name	Example
time_base	J2000 (Julian date 2451545.0 TT or 2000 January 1, 12h TT)
resolution	nanoseconds
time_scale	Terrestrial Time (TT)
units	nanoseconds
reference_position	rotating Earth Geoid

Given a current list of leap seconds, conversion between TT and UTC is straightforward (TT = TAI + 32.184s; TT = UTC + deltaAT + 32.184s, where deltaAT is the sum of the leap seconds since 1960; for example, for 2009, deltaAT = 34s). Pad values of - 9223372036854775808 (0x8000000000000000) which corresponds to 1707-09-22T12:13:15.145224192; recommended FILLVAL is same.

It is proposed that the required data variables VALIDMIN and VALIDMAX are given values corresponding to the dates 1990-01-01T00:00:00 and 2100-01-01T00:00:00 as these are well outside any expected valid times.

5.1.3 Required Attributes: Data Variables

Data variables require the following attributes:

- CATDESC
- DEPEND_0
- DEPEND_i [for dimensional data variables]
- DISPLAY_TYPE
- FIELDNAM
- FILLVAL
- FORMAT or FORM_PTR
- LABLAXIS or LABL_PTR_i

- SI_CONVERSION
- UNITS or UNIT_PTR
- VALIDMIN and VALIDMAX
- VAR_TYPE

In addition, the following attributes are strongly recommended for vectors, tensors and quaternions which are held in or relate to a particular coordinate system:

- COORDINATE_SYSTEM
- TENSOR_ORDER
- REPRESENTATION_i
- OPERATOR_TYPE [for quaternions]

5.1.4 Attributes for DEPEND_i Variables

Variables appearing in a data variable's DEPEND_i attribute require a minimal set of their own attributes to fulfill their role in supporting the data variable. The standard SUPPORT_DATA variable attributes are listed in Section 5.3.2. Other standard variable attributes are optional.

5.2 Quaternions

HERMES mec files contain unit quaternions which can be employed to rotate from one coordinate system to the other. For an arbitrary rotation, that rotational information can expressed as a rotation through an angle about a unit vector u. The Wikipedia page on "Quaternions and Spatial Rotation" provides details and the relationship between the quaternion and a 3x3 rotation matrix. In the mec files, quaternions are represented by:

```
q = (qx, qy, qz, qw)
```

in which qw (also known elsewhere as qc) = cos (/2) and (qx, qy, qz) = u sin (/2). Extensions of existing attribute standards are strongly recommended to be used to describe such quaternions. The following attributes serve this purpose:

- OPERATOR_TYPE=UNIT_QUATERNION
- REPRESENTATION_1 = "x", "y", "z", "c" [in the right order; the "c" denotes the cosineterm]
- COORDINATE_SYSTEM=XXX [standard syntax, as for vectors; the FROM frame]
- TO_COORDINATE_SYSTEM=YYY [same syntax; the TO frame]

Such a quaternion will take a vector given in the XXX coordinate system and generate its components in the YYY coordinate system.

5.3 Support Data

These are variables of secondary importance employed as DEPEND_i variables as described in section 5.1.3 (e.g., time, energy_bands associated with particle flux), but they may also be used for housekeeping or other information not normally used for scientific analysis.

5.3.1 Naming

Support data variable names must begin with a letter and can contain numbers and underscores, but no other special characters. Support data variable names need not follow the same naming convention as Data Variables (5.1.1) but may be shortened for convenience.

5.3.2 Required Attributes: Support Variables

- CATDESC
- DEPEND_0 (if time varying)
- FIELDNAM
- FILLVAL (if time varying)
- FORMAT/FORM_PTR
- LABLAXIS or LABL_PTR_i
- SI_CONVERSION
- UNITS/UNIT_PTR
- VALIDMIN (if time varying)
- VALIDMAX (if time varying)
- VAR_TYPE = “support_data”

Other attributes may also be present.

5.4 Metadata

These are variables of secondary importance (e.g. a variable holding “Bx”, “By”, “Bz” to label magnetic field). Metadata are usually text strings as opposed to the numerical values held in DEPEND_i support data.

5.4.1 Naming

Metadata variable names must begin with a letter and can contain numbers and underscores, but no other special characters. Metadata variable names need not follow the same naming convention as Data Variables (5.1.1) but may be shortened for convenience.

5.4.2 Required Attributes: Metadata Variables

- CATDESC
- DEPEND_0 (if time varying, this value must be “Epoch”)
- FIELDNAM
- FILLVAL (if time varying)
- FORMAT/FORM_PTR
- VAR_TYPE = metadata

5.5 Variable Attribute Schema

The following variable attributes shown in Table 5-4 are required with HERMES data products. HERMES-specific values are provided where applicable. For each attribute the following information is provided:

- description: (`str`) A brief description of the attribute
- derived: (`bool`) Whether the attibute can be derived by the HERMES *HermesDataSchema* class
- required: (`bool`) Whether the attribute is required by HERMES standards
- overwrite: (`bool`) Whether the *HermesDataSchema* attribute derivations will overwrite an existing attribute value with an updated attribute value from the derivation process.
- valid_values: (`list`) List of allowed values the attribute can take for HERMES products, if applicable
- alternate: (`str`) An additional attribute name that can be treated as an alternative of the given attribute. Not all attributes have an alternative and only one of a given attribute or its alternate are required.
- var_types: (`str`) A list of the variable types that require the given attribute to be present.

Note that this table is derived from `hermes_core/data/hermes_default_variable_cdf_attrs_schema.yaml`

Table 10: Table 5-4 HERMES Variable Attribute Schema

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
TIMI	fixed (0AD, 1900, 1970 (POSIX), J2000 (used by CDF_TIME_TT2000), 4714 BC (Julian)) or flexible (provider-defined)	True	True	False			
RES-O-LU-TIOI	Using ISO8601 relative time format, for example: “1s” = 1 second. Resolution provides the smallest change in time that is measured.	True	True	False			
TIMI	TT (same as TDT, used by CDF_TIME_TT2000), TAI (same as IAT, TT-32.184s), UTC (includes leap seconds), TDB (same as SPICE ET), EME1950 [default: UTC]	True	True	False			

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
REF-ER-ENC	Topocenter (local), Geocenter , rotating Earth geoid (used by CDF_TIME_TT2000). Reference_Position is optional metadata to account for time variance with position in the gravity wells and with relative velocity. While we could use a combined TimeSystem attribute that defines mission-specific time scales where needed, such as UTC-at-STEREO-B, it's cleaner to keep them separate as Time_Scale=UTC and Reference_Position=STEREO-B.	True	True	False			
LEA	comma-delimited list (within brackets) of leap seconds included in the form of a lists of ISO8601 times when each leap second was added, appended with the size of the leap second in ISO8601 relative time (+/- time, most commonly: "+1s") [default: standard list of leap seconds up to time of data]. Leap_Seconds_Included is needed to account for time scales that don't have all 34 (in 2009) leap seconds and for the clocks in various countries that started using leap seconds at different times. The full list is required to handle the equally or more common case where a time scale starts at a pecific UTC but continues on without leap seconds in TAI mode; this is basically what missions that don't add leap seconds are doing. $ cat tai-utc.dat | awk 'ORS=","' { val = $7 - prev } {prev = $7} { print $1$2"01+" val "s" }'	False	False	False			

continues on next page

Table 10 – continued from previous page

At- tribu	description	de- rivec	re- quire	over- write	valid_values	alternate	var_types
AB- SO- LUT	Absolute or systematic error, in same units as Units attribute.	False	False	False			
REL- A- TIVE	Relative or random error, in same units as Units attribute - to specify the accuracy of the time stamps relative to each other. This is usually much smaller than Absolute_Error.	False	False	False			
BIN_	relative position of time stamp to the data measurement bin, with 0.0 at the beginning of time bin and 1.0 at the end. Default is 0.5 for the time at the center of the data measurement. Since clock readings are usually truncated, the real value may be closer to 0.0.	False	False	False			
CAT- DES	This is a human readable description of the data variable. Generally, this is an 80- character string which describes the variable and what it depends on.	False	True	False			data support_data metadata
DEL	DEPEND_i variables are typically physical values along the corresponding i-th dimension of the parent data variable, such as energy levels or spectral frequencies. The discreet set of values are located with respect to the sampling bin by DELTA_PLUS_VAR and DELTA_MINUS_VAR, which hold the variable name containing the distance from the value to the bin edge. It is strongly recommended that HERMES DEPEND_i variables include DELTA_PLUS_VAR and DELTA_MINUS_VAR attributes that point to the appropriate variable(s) located elsewhere in the CDF file.	False	False	False			

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
DEL	DEPEND_i variables are typically physical values along the corresponding i-th dimension of the parent data variable, such as energy levels or spectral frequencies. The discreet set of values are located with respect to the sampling bin by DELTA_PLUS_VAR and DELTA_MINUS_VAR, which hold the variable name containing the distance from the value to the bin edge. It is strongly recommended that HERMES DEPEND_i variables include DELTA_PLUS_VAR and DELTA_MINUS_VAR attributes that point to the appropriate variable(s) located elsewhere in the CDF file.	False	False	False			
DE-PEN	Explicitly ties a data variable to the time variable on which it depends. All variables which change with time must have a DEPEND_0 attribute defined. See section 5.2.1 which specifies the HERMES usage of DEPEND_0.	True	True	False			data

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
DE-PEN	Ties a dimensional data variable to a SUPPORT_DATA variable on which the i-th dimension of the data variable depends. The number of DEPEND attributes must match the dimensionality of the variable, i.e., a one-dimensional variable must have a DEPEND_1, a two-dimensional variable must have a DEPEND_1 and a DEPEND_2 attribute, etc. The value of the attribute must be a variable in the same CDF data set. It is strongly recommended that DEPEND_i variables hold values in physical units. DEPEND_i variables also require their own attributes, as described in section 5.1.4.	False	False	False			
DIS-PLA	This tells automated software, such as CDAWeb, how the data should be displayed.	True	True	False	time_series time_series>noerr spectrogram stack_plot image		data
FIEL NAM	A shortened version of CATDESC which can be used to label a plot axis or as a data listing heading. This is a string, up to ~30 characters in length.	True	True	False			data support_data metadata
FIL-L-VAL	Identifies the fill value used where data values are known to be bad or missing. FILLVAL is required for time-varying variables. Fill data are always non-valid data. The ISTP standard fill values are listed in Table 5-4.	True	True	False			data support_data metadata

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
FOR-MAT	This field allows software to properly format the associated data when displayed on a screen or output to a file. Format can be specified using either Fortran or C format codes. For instance, "F10.3" indicates that the data should be displayed across 10 characters where 3 of those characters are to the right of the decimal. For a description of FORTRAN formatting codes see the docs here: https://docs.oracle.com/cd/E19957-01/805-4939/z40007437a2e/index.html	True	True	False		FORM_PTR	data support_data metadata
FOR	The value of this field is a variable which stores the character string that represents the desired output format for the associated data.	False	False	False		FORMAT	
LAB	Used to label a plot axis or to provide a heading for a data listing. This field is generally 6-10 characters. Only one of LABLAXIS or LABL_PTR_i should be present.	True	True	False		LABL_PTR_1	data support_data

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
LAB	Used to label a dimensional variable when one value of LABLAXIS is not sufficient to describe the variable or to label all the axes. LABL_PTR_i is used instead of LABLAXIS, where i can take on any value from 1 to n where n is the total number of dimensions of the original variable. The value of LABL_PTR_1 is a variable which will contain the short character strings which describe the first dimension of the original variable. The value of the attribute must be a variable in the same CDF data set and is generally 6-10 characters. Only one of LABLAXIS or LABL_PTR_i should be present.	False	False	False		LABLAXIS	
SI_C	The conversion factor to SI units. This is the factor that the variable must be multiplied by in order to convert it to generic SI units. This parameter contains two text fields separated by the ">" delimiter. The first component is the conversion factor and the second is the standard SI unit. Units are defined according to their standard SI symbols (ie. Tesla = T, Newtons = N, Meters = m, etc.) For data variables that are inherently unitless, and thus lack a conversion factor, this data attribute will be " > " where ' ' is a blank space and the quotation marks are not included. Units which are not conveniently transformed into SI should follow the blank syntax " > " described above.	True	True	False			data support_data

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
UNI	A 6-20 character string that identifies the units of the variable (e.g. nT for magnetic field). Use a blank character, rather than “None” or “unitless”, for variables that have no units (e.g., a ratio or a direction cosine). An active list of HERMES standard UNITS and their SI_CONVERSIONs is maintained on the mission web-pages at https://lasp.colorado.edu/galaxy/display/HERMES/Units+of+Measure, accessible via the HERMES Science Working Team pages. Those pages also lay out the rules for formatting the UNITS string.	True	True	False		UNIT_PTR	data support_data
UNI	The value of this field is a variable which stores short character strings which identify the units of the variable. Use a blank character, rather than “None” or “unitless”, for variables that have no units (e.g., a ratio or a direction cosine). The value of this attribute must be a variable in the same CDF data set.	False	False	False		UNITS	
VAL MIN	The minimum value for a particular variable that is expected over the lifetime of the mission. Used by application software to filter out values that are out of range. The value must match the data type of the variable.	True	True	False			data support_data
VAL MAX	The maximum value for a particular variable that is expected over the lifetime of the mission. Used by application software to filter out values that are out of range. The value must match the data type of the variable.	True	True	False			data support_data

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
VAR	Used in CDAWeb to indicate if the data should be used directly by users.	True	True	False	data support_data metadata ignore_data		data support_data metadata
CO-OR-DI-NAT	All variables for which the values are dependent on the system of coordinates are strongly recommended to have this attribute. This includes both full vectors, tensors, etc. or individual values, e.g. of an angle with respect to some axis. The attribute is a text string which takes the form: "XXX[>optional long name]"	False	False	False			
TEN-SOR	All variables which hold physical vectors, tensors, etc., or sub-parts thereof, are strongly recommended to have their tensorial properties held by this numerical value. Vectors have TENSOR_ORDER=1, pressure tensors have TENSOR_ORDER=2, etc. Variables which hold single components or sub-parts of a vector or tensor, e.g., the x-component of velocity or the three diagonal elements of a tensor, use this attribute to establish the underlying object from which they are extracted. TENSOR_ORDER is a number, usually held as a CDF_INT4, rather than a character string.	False	False	False			

continues on next page

Table 10 – continued from previous page

Attribu	description	derived	required	overwrite	valid_values	alternate	var_types
REPRESENTATION	This strongly recommended attribute holds the way vector or tensor variables are held, e.g., as Cartesian or polar forms, and their sequence order in the dimension i in which they are held. Cartesians are indicated by x,y,z; polar coordinates by r (magnitude), t (theta - from z-axis), p (phi - longitude or azimuth around z-axis from x axis), l (lambda = latitude). Examples follow.	False	False	False			
OPERATOR	This has been introduced to describe HERMES quaternions (see Section 5.2 below). It has allowed values "UNIT_QUATERNION" or "ROTATION_MATRIX" although other values could be added. Unit quaternions correspond to pure spatial rotations.	False	False	False			
WCSAX	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. The value field shall contain a non-negative integer no greater than 999, representing the number of axes in the associated data array.	True	False	False			
MJDRE	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. The value shall contain a floating point number representing the reference time position of the time stamps along the 0'th axis of the measurement.	True	False	False			

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
TIMI U-NIT	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. The value shall contain a character string giving the units of the time stamps along the 0'th axis of the measurement. The TIMEUNIT should match the CUNITi along the time axis of the measurement	True	False	False			
TIMI	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. The value shall contain a floating point number representing the resolution of the time stamps along the 0'th axis of the measurement. The TIMEDEL should match the CRDELi along the time axis of the measurement.	True	False	False			
CNA	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value shall contain a charachter string represnting the name of the i'th axis. The name is used for comment/documentation purposes only and is not used as a part of the i'th axis coordinate transformations.	True	False	False			

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
CTY	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value field shall contain a character string, giving the name of the coordinate represented by axis i.	True	False	False			
CU-NITi	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value shall be the units along axis i, compatible with CTYPEi to be used for scaling and coordinate transformations along the i'th axis.	True	False	False			

continues on next page

Table 10 – continued from previous page

At-tribu	description	de-rivec	re-quire	over-write	valid_values	alternate	var_types
CR-PIXi	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value field shall contain a floating point number, identifying the location of a reference point along axis i, in units of the axis index. This value is based upon a counter that runs from 1 to NAXISn with an increment of 1 per pixel. The reference point value need not be that for the center of a pixel nor lie within the actual data array. Use comments to indicate the location of the index point relative to the pixel.	True	False	False			
CR-VAL	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value field shall contain a floating point number, giving the value of the coordinate specified by the CTYPEn keyword at the reference point CRPIXi.	True	False	False			

continues on next page

Table 10 – continued from previous page

Attribu	description	derived	require	overwrite	valid_values	alternate	var_types
CDE	This is a FITS WCS Keyword being repurposed for handling WCS transformations with high-dimensional or spectral CDF data variables. This metadata attribte should be used for the i'th dimension (1-based) and reapeated for all WCSAXES dimensions. The value field shall contain a floating point number giving the partial derivative of the coordinate specified by the CTYPEi keywords with respect to the pixel index, evaluated at the reference point CRPIXi, in units of the coordinate specified by the CTYPEi keyword.	True	False	False			

3.4 Customization and Global Configuration

3.4.1 The `configrc` file

This package uses a `configrc` configuration file to customize certain properties. You can control a number of key features of such as where your data will download to. HERMES packages look for this configuration file in a platform specific directory, which you can see the path for by running:

```
>>> import hermes_core
>>> hermes_core.print_config()
```

3.4.2 Using your own `configrc` file

To maintain your own customizations, you must place your customized `configrc` inside the appropriate configuration folder (which is based off the operating system you are working on). The AppDirs module provided by the sunpy package is used to figure out where to look for your configuration file.

Warning: Do not edit the configrc file directly in the Python package as it will get overwritten every time you re-install or update the package.

You can copy the file below, customize it, and then place your customized `configrc` file inside your config folder.

If you work in our developer environment you can place your configuration file in this directory:

```
/home/vscode/.config/hermes_core/
```

If you do not use our developer environment, you can run the following code to see where to place it on your specific machine as well:

```
>>> from hermes_core import util
>>> print(util.config._get_user_configdir())
/home/vscode/.config/hermes_core
```

Note: For more information on where to place your configuration file depending on your operating system, you can refer to the AppDirs module docstrings.

To learn more about how to set-up your development environment see *Developer Environment*.

See below (*A sample configrc file*) for an example configuration file.

3.4.3 Dynamic settings

You can also dynamically change most of the default settings. One setting that cannot be changed is the location of the log file which is set on import. All settings are stored in a Python ConfigParser instance called `hermes_core.config`, which is global to the package. Settings can be modified directly, for example:

```
import hermes_core
hermes_core.config.set('downloads', 'download_dir', '/home/user/Downloads')
```

A sample configrc file

```
;
; Configuration
;
; This is the default configuration file

;;;;;;;;;;;;;;;;;;;
; General Options ;
;;;;;;;;;;;;;;;;;;;
[general]

; Time Format to be used for displaying time in output (e.g. graphs)
; The default time format is based on ISO8601 (replacing the T with space)
; note that the extra '%'s are escape characters
time_format = %Y-%m-%d %H:%M:%S

;;;;;;;;;;;;;
; Downloads ;
;;;;;;;;;;;;;
[downloads]

; Location to save download data to. Path should be specified relative to the
; HERMES working directory.
; Default value: data/
download_dir = data
```

(continues on next page)

(continued from previous page)

```
;;;;;;;;;;;;
; Logger   ;
;;;;;;;;;;;;
[logger]

# Threshold for the logging messages. Logging messages that are less severe
# than this level will be ignored. The levels are 'DEBUG', 'INFO', 'WARNING',
# 'ERROR'
log_level = INFO

# Whether to use color for the level names
use_color = True

# Whether to log warnings.warn calls
log_warnings = True

# Whether to log exceptions before raising them
log_exceptions = True

# Whether to always log messages to a log file
log_to_file = True

# The file to log messages to
log_file_path = hermes.log

# Threshold for logging messages to log_file_path
log_file_level = INFO

# Format for log file entries
log_file_format = %(asctime)s, %(origin)s, %(levelname)s, %(message)s
```

3.5 Logging system

3.5.1 Overview

The logging system is an adapted version of `AstropyLogger`. Its purpose is to provide users the ability to decide which log and warning messages to show, to capture them, and to send them to a file.

All messages provided by HERMES use this logging facility which is based on the Python `logging` module rather than print statements.

Messages can have one of several levels, in increasing order of importance:

- DEBUG: Detailed information, typically of interest only when diagnosing problems.
- INFO: A message conveying information about the current task, and confirming that things are working as expected
- WARNING: An indication that something unexpected happened, and that user action may be required.
- ERROR: indicates a more serious issue where something failed but the task is continuing
- CRITICAL: A serious error, indicating that the program itself may be unable to continue running.

By default, all messages except for DEBUG messages are displayed.

3.5.2 Configuring the logging system

The default configuration for the logger is determined by the default configuration file. To make permanent changes to the logger configuration see the `[logger]` section of the configuration file (*config*).

If you'd like to control the logger configuration for your current session first import the logger:

```
>>> from hermes_core import log
```

or also by:

```
>>> import logging
>>> log = logging.getLogger('hermes_core')
```

The threshold level for messages can be set with:

```
>>> log.setLevel('DEBUG')
```

This will display DEBUG and all messages with that level and above. If you'd like to see the fewest relevant messages you'd set the logging level to WARNING or above.

For other options such as whether to log to a file or what level of messages the log file should contain, see the the HERMES configuration file (*config*).

3.5.3 Context managers

If you'd like to capture messages as they are generated you can do that with a context manager:

```
>>> from hermes_core import log
>>> with log.log_to_list() as log_list:
...    # your code here
```

Once your code is executed, `log_list` will be a Python list containing all of the messages during execution. This does not divert the messages from going to a file or to the screen. It is also possible to send the messages to a custom file with:

```
>>> from hermes_core import log
>>> with log.log_to_file('myfile.log'):
...     # your code here
```

which will save the messages to a local file called `myfile.log`.

CHAPTER

FOUR

DEVELOPER'S GUIDE

This article describes the guidelines to be followed by developers working on this repository. If you are planning on contributing to this repository please read the following carefully. This guide borrows heavily from those developed by the SunPy Project and is generally consistent with this community-developed approach.

4.1 Developer Environment

This Python package is used in the pipeline processing of scientific data from HERMES. Special consideration is therefore required to ensure that development is compatible with the pipeline environment. It is also important to ensure that this package is compatible with a user's systems such as a mac and windows.

4.1.1 Visual Studio Code

Though not required, this packate designed for development in Visual Studio Code inside of a container managed by Docker. This is the same environment that is used by the data processing pipeline. All of the configuration required by VS Code are maintained in the `devcontainer` folder including the Dockerfile. For more information see `Developing inside a Container`.

Setup

Follow these steps to set up VS Code.

1. Download and install VS Code.
2. Download and install Docker. The easiest way to do that is to install Docker Desktop.
3. **Open VS Code and add the following 2 extensions by navigating to View->Extensions.**
    1. Docker
    2. Remote-Container
4. Ensure that Docker is running by opening Docker Desktop. It will be required to build the container.
5. Restart VS Code and open this repository using File->Open Folder. It might recognize that a container is defined and prompt you to Reopen in Container. Do so.
6. If not, open the VS Code Command Palette by View->Command Palette (or Ctrl+Shift+P) and select: "Remote-Containers:Rebuild and Reopen in Container"
7. VS Code should build and open the container (takes as much as 10-20 minutes the first time). You will see "Starting Dev Container (show log): Building image" in the bottom right corner. Click on "show log" to see details of the build. This requires Docker to be running.

8. Once the build has finished, you will see information about the Dev Container in the bottom left.
9. Exiting VS Code will close the docker container.
10. The next time you open this folder with VS Code it should open in the built container. It should not have to rebuild the container unless the Dockerfile file has changed.

4.2 Coding Standards

The purpose of the page is to describe the standards that are expected of all the code in this repository. All developers should read and abide by the following standards. Code which does not follow these standards closely will generally not be accepted.

We try to closely follow the coding style and conventions proposed by Astropy.

4.2.1 Language Standard

- All code must be compatible with Python 3.7 and later.
- The new Python 3 formatting style should be used (i.e. `f"{spam:s}"` instead of `"%s" % "spam"`).

4.2.2 Coding Style/Conventions

- The code will follow the standard PEP8 Style Guide for Python Code. In particular, this includes using only 4 spaces for indentation, and never tabs.
- **Follow the existing coding style** within a file and avoid making changes that are purely stylistic. Please try to maintain the style when adding or modifying code.
- Following PEP8's recommendation, absolute imports are to be used in general. We allow relative imports within a module to avoid circular import chains.
- The `import numpy as np`, `import matplotlib as mpl`, and `import matplotlib.pyplot as plt` naming conventions should be used wherever relevant. `from packagename import *` should never be used (except in `__init__.py`)
- Classes should either use direct variable access, or Python's property mechanism for setting object instance variables.
- Classes should use the builtin `super` function when making calls to methods in their super-class(es) unless there are specific reasons not to. `super` should be used consistently in all subclasses since it does not work otherwise.
- Multiple inheritance should be avoided in general without good reason.
- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout.

4.2.3 Private code

It is often useful to designate code as private, which means it is not part of the user facing API, only used internally by HERMES, and can be modified without a deprecation period. Any classes, functions, or variables that are private should either:

- Have an underscore as the first character of their name, e.g., `_my_private_function`.
- If you want to do that to entire set of functions in a file, name the file with a underscore as the first character, e.g., `_my_private_file.py`.

4.2.4 Utilities

Within this reposiotory, it might be useful to have a set of utility classes or functions that are used by internally to help with certain tasks or to provide a certain level of abstraction. These should be placed either:

- `.{subpackage}.utils.py`, if it is only used within that sub-package.
- `.util` if it is used across multiple sub-packages.

These can be private (see section above) or public. The decision is up to the developer, but if these might be useful for other modules, they should be made public. These utils may be taken up by the core repository if they are generally useful for other instrument teams.

4.2.5 Formatting

We enforce a minimum level of code style with our continuous intergration. This runs a tool called pre-commit.

The settings and tools we use for the pre-commit can be found in the file `.pre-commit-config.yaml` at the root of the HERMES git repository. Some of the checks are: * Checks (but doesn't fix) various PEP8 issues with flake8. * Sort all imports in any Python files with isort. * Remove any unused variables or imports with autoflake.

We suggest you use "tox" (which is used to run the HERMES test suite) to run these tools without having to setup anything within your own Python virtual environment:

```
$ tox -e codestyle
```

This will inform you of what checks failed and why, and what changes (if any) the command has made to your code.

If you want to setup the pre-commit locally, you can do the following:

```
$ pip install pre-commit
```

Now you can do:

```
$ pre-commit run --all-files
```

which will run the tools on all files in the HERMES git repository. The pre-commit tools can change some of the files, but in other cases it will report problems that require manual correction. If the pre-commit tool changes any files, they will show up as new changes that will need to be committed.

Automate

Instead of running the pre-commit command each time you can install the git hook:

```
$ pre-commit install
```

which installs a command to `.git/hooks/pre-commit` which will run these tools at the time you do `git commit` and means you don't have to run the first command each time. We only suggest doing the install step if you are comfortable with git and the pre-commit tool. If you are running inside of a Docker container but are managing git outside of it you will have to do this outside of the Docker. This also means that you will have to install all of the dependencies on your local system.

By Hand

Sometimes it is easier to run things by hand. First, let's talk about Black. If you are using the docker container and VS Code it format be formatting your code automatically. If you want to check if all of your files are compatible with Black run the following

> $ black –check folder_name

If you want it to go ahead and format the files remote `--check`.

4.2.6 Documentation and Testing

- American English is the default language for all documentation strings and inline commands. Variables names should also be based on English words.
- Documentation strings must be present for all public classes/methods/functions, and must follow the form outlined in the *Documentation Rules* page. Additionally, examples or tutorials in the package documentation are strongly recommended.
- Write usage examples in the docstrings of all classes and functions whenever possible. These examples should be short and simple to reproduce–users should be able to copy them verbatim and run them. These examples should, whenever possible, be in the *doctests* format and will be executed as part of the test suite.
- Unit tests should be provided for as many public methods and functions as possible, and should adhere to the standards set in the *Testing Guidelines* document.

4.2.7 Data and Configuration

- We store test data in `./data/test` as long as it is less than about 100 kB.
- All persistent configuration should use the *Global Settings* mechanism. Such configuration items should be placed at the top of the module or package that makes use of them, and supply a description sufficient for users to understand what the setting changes.

4.2.8 Standard output, warnings, and errors

The built-in `print(...)` function should only be used for output that is explicitly requested by the user, for example `print_header(...)` or `list_catalogs(...)`. Any other standard output, warnings, and errors should follow these rules:

- For errors/exceptions, one should always use `raise` with one of the built-in exception classes, or a custom exception class (e.g. ValueError, TypeError). The nondescript `Exception` class should be avoided as much as possible, in favor of more specific exceptions (`IOError`, `ValueError`, etc.).
- For warnings, use the appropriate custom warning classes (e.g. *`hermes_core.util.exceptions.HERMESWarning`*, *`hermes_core.util.exceptions.HERMESUserWarning`*) to enable them to be captured by the logging system.
- For debug messages, use the logging system `log.debug()` with a descriptive message. Remember that users may access those messages as well.

4.2.9 Including C Code

- C extensions are only allowed when they provide a significant performance enhancement over pure Python, or a robust C library already exists to provided the needed functionality.
- The use of Cython is strongly recommended for C extensions.
- If a C extension has a dependency on an external C library, the source code for the library should be bundled with the HERMES repository, provided the license for the C library is compatible with the HERMES license. Additionally, the package must be compatible with using a system-installed library in place of the library included in HERMES.
- In cases where C extensions are needed but Cython cannot be used, the PEP 7 Style Guide for C Code is recommended.
- C extensions (Cython or otherwise) should provide the necessary information for building the extension.

4.3 Testing Guidelines

This section describes the testing framework and format standards for tests. Here we have heavily adapted the Astropy version, and **it is worth reading that link.**

The testing framework used by HERMES is the pytest framework, accessed through the `pytest` command.

Note: The `pytest` project was formerly called `py.test`, and you may see the two spellings used interchangeably.

4.3.1 Writing tests

`pytest` has the following test discovery rules:

```
* ``test_*.py`` or ``*_test.py`` files
* ``Test`` prefixed classes (without an ``__init__`` method)
* ``test_`` prefixed functions and methods
```

We use the first one for our test files, `test_*.py` and we suggest that developers follow this.

A rule of thumb for unit testing is to have at least one unit test per public function.

Where to put tests

Each package should include a suite of unit tests, covering as many of the public methods/functions as possible. These tests should be included inside each package, e.g:

```
hermes_core/util/tests/
```

"tests" directories should contain an `__init__.py` file so that the tests can be imported.

doctests

Code examples in the documentation will also be run as tests and this helps to validate that the documentation is accurate and up to date. We use the same system as Astropy, so for information on writing doctests see the astropy documentation.

You do not have to do anything extra in order to run any documentation tests. Within our `setup.cfg` file we have set default options for `pytest`, such that you only need to run:

```
$ pytest <rst to test>
```

to run any documentation test.

Bugs Testing

In addition to writing unit tests new functionality, it is also a good practice to write a unit test each time a bug is found, and submit the unit test along with the fix for the problem. This way we can ensure that the bug does not re-emerge at a later time.

4.4 Documentation Rules

4.4.1 Overview

All code must be documented and we follow these style conventions described here:

- numpydoc

We recommend familiarizing yourself with this style.

Referring to other code

To link to other methods, classes, or modules in your repo you have to use backticks, for example:

```
`hermes_core.io.read_file`
```

generates a link like this: `hermes_core.io.read_file`.

Other packages can also be linked via intersphinx:

```
`numpy.mean`
```

will return this link: `numpy.mean`. This works for Python, Numpy and Astropy (full list is in `docs/conf.py`).

With Sphinx, if you use `:func:` or `:meth:`, it will add closing brackets to the link. If you get the wrong pre-qualifier, it will break the link, so we suggest that you double check if what you are linking is a method or a function.

```
:class:`numpy.mean()`
:meth:`numpy.mean()`
:func:`numpy.mean()`
```

will return two broken links ("class" and "meth") but "func" will work.

Project-specific Rules

- For **all** RST files, we enforce a one sentence per line rule and ignore the line length.

4.4.2 Sphinx

All of the documentation (like this page) is built by Sphinx, which is a tool especially well-suited for documenting Python projects. Sphinx works by parsing files written using a a Mediawiki-like syntax called reStructuredText. It can also parse markdown files. In addition to parsing static files of reStructuredText, Sphinx can also be told to parse code comments. In fact, in addition to what you are reading right now, the Python documentation was also created using Sphinx.

Usage and Building the documentation

All of the documentation is contained in the "docs" folder and code documentation strings. Sphinx builds documentation iteratively, only adding things that have changed. For more information on how to use Sphinx, consult the Sphinx documentation.

HTML

To build the html documentation locally use the follownig command, in the root directory run:

```
$ sphinx-build docs docs/_build/html -W -b html
```

This will generate HTML documentation in the "docs/_build/html" directory. You can open the "index.html" file to browse the final product.

PDF

To build the pdf documentation locally use the follownig command, in the root directory run:

```
$ sphinx-build docs docs/_build/pdf -W -b pdf
```

This will generate HTML documentation in the "docs/_build/html" directory. You can open the "index.html" file to browse the final product.

4.5 Workflow for Maintainers

This page is for maintainers who can merge our own or other peoples' changes into the upstream repository.

Seeing as how you're a maintainer, you should be completely on top of the basic git workflow in Developer's Guide and Astropy's git workflow.

4.5.1 Integrating changes via the web interface (recommended)

Whenever possible, merge pull requests automatically via the pull request manager on GitHub. Merging should only be done manually if there is a really good reason to do this!

Make sure that pull requests do not contain a messy history with merges, etc. If this is the case, then follow the manual instructions, and make sure the fork is rebased to tidy the history before committing.

To check out a particular pull request to test out locally:

```
$ git checkout pr/999
Branch pr/999 set up to track remote branch pr/999 from upstream.
Switched to a new branch 'pr/999'
```

When to remove or combine/squash commits

In all cases, be mindful of maintaining a welcoming environment and be helpful with advice, especially for new contributors. It is expected that a maintainer would offer to help a contributor who is a novice git user do any squashing that that maintainer asks for, or do the squash themselves by directly pushing to the PR branch.

Pull requests **must** be rebased and at least partially squashed (but not necessarily squashed to a single commit) if large (approximately >10KB) non-source code files (e.g. images, data files, etc.) are added and then removed or modified in the PR commit history (The squashing should remove all but the last addition of the file to not use extra space in the repository).

Combining/squashing commits is **encouraged** when the number of commits is excessive for the changes made. The definition of "excessive" is subjective, but in general one should attempt to have individual commits be units of change, and not include reversions. As a concrete example, for a change affecting < 50 lines of source code and including a changelog entry, more than a two commits would be excessive. For a larger pull request adding significant functionality, however, more commits may well be appropriate.

As another guideline, squashing should remove extraneous information but should not be used to remove useful information for how a PR was developed. For example, 4 commits that are testing changes and have a commit message of just "debug" should be squashed. But a series of commit messages that are "Implemented feature X", "added test for feature X", "fixed bugs revealed by tests for feature X" are useful information and should not be squashed away without reason.

When squashing, extra care should be taken to keep authorship credit to all individuals who provided substantial contribution to the given PR, e.g. only squash commits made by the same author.

When to rebase

Pull requests **must** be rebased (but not necessarily squashed to a single commit) if:

- There are commit messages include offensive language or violate the code of conduct (in this case the rebase must also edit the commit messages)

Pull requests **may** be rebased (either manually or with the rebase and merge button) if:

- There are conflicts with main
- There are merge commits from upstream/main in the PR commit history (merge commits from PRs to the user's fork are fine)

Asking contributors who are new to the project or inexperienced with using git is **discouraged**, as is maintainers rebasing these PRs before merge time, as this requires resetting of local git checkouts.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
$ git fetch upstream-rw

# Rebase
$ git rebase upstream-rw/main
```

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
$ git fetch upstream-rw
$ git merge --no-ff upstream-rw/main
```

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
$ git log --oneline --graph
$ git log -p upstream-rw/main..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/main`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to open pull request

Now you need to push the changes you have made to the code to the open pull request:

```
$ git push git@github.com:<username>/hermes_core.git HEAD:<name of branch>
```

You might have to add `--force` if you rebased instead of adding new commits.

4.5.2 IOssue Milestones and Labels

Current milestone guidelines:

- Only confirmed issues or pull requests that are release critical or for some other reason should be addressed before a release, should have a milestone. When in doubt about which milestone to use for an issue, do not use a milestone and ask other the maintainers.

Current labelling guidelines:

- Issues that require fixing in main, but that also are confirmed to apply to supported stable version lines should be marked with a "Affects Release" label.
- All open issues should have a "Priority <level>", "Effort <level>" and "Package <level>", if you are unsure at what level, pick higher ones just to be safe. If an issue is more of a question or discussion, you can omit these labels.
- If an issue looks to be straightforward, you should add the "Good first issue" and "Hacktoberfest" label.
- For other labels, you should add them if they fit, like if an issue affects the net submodule, add the "net" label or if it is a feature request etc.

4.5.3 Updating and Maintaining the Changelog

The changelog will be read by users, so this description should be aimed at HERMES users instead of describing internal changes which are only relevant to the developers.

The current changelog is kept in the file "CHANGELOG.rst" at the root of the repository.

4.5.4 Releases

We have a step by step checklist on the Wiki on how to make a release.

4.6 Global Settings

This package makes use of a settings file (`configrc`). This file contains a number of global settings such as where files should be downloaded by default or the default format for displaying times. When developing new functionality check this file and make use of the default values if appropriate or, if needed, define a new value. More information can be found in *Customization and Global Configuration*.

CHAPTER

FIVE

API REFERENCE

5.1 hermes_core Package

5.1.1 Functions

print_config()	Print current configuration options.

print_config

`hermes_core.`**`print_config`**`()`

Print current configuration options.

5.2 hermes_core.timedata Module

Container class for Measurement Data.

5.2.1 Classes

HermesData(timeseries[, support, spectra, meta])	A generic object for loading, storing, and manipulating HERMES time series data.

HermesData

`class` `hermes_core.timedata.`**`HermesData`**(*timeseries: TimeSeries*, *support: dict[Quantity* | *NDData]* | *None* = *None*, *spectra: NDCollection* | *None* = *None*, *meta: dict* | *None* = *None*)

Bases: `object`

A generic object for loading, storing, and manipulating HERMES time series data.

Parameters

- **`timeseries`** (`astropy.timeseries.TimeSeries`) – The time series of data. Columns must be `Quantity` arrays.

- **support** (Optional[dict[Union[astropy.units.Quantity, astropy.nddata.NDData]]]) – Support data arrays which do not vary with time (i.e. Non-Record-Varying data).
- **spectra** (Optional[ndcube.NDCollection]) – One or more ndcube.NDCube objects containing spectral or higher-dimensional timeseries data.
- **meta** (Optional[dict]) – The metadata describing the time series in an ISTP-compliant format.

Examples

```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> from ndcube import NDCube, NDCollection
>>> from astropy.wcs import WCS
>>> from astropy.nddata import NDData
>>> from hermes_core.timedata import HermesData
>>> # Create a TimeSeries structure
>>> data = u.Quantity([1, 2, 3, 4], "gauss", dtype=np.uint16)
>>> ts = TimeSeries(time_start="2016-03-22T12:30:31", time_delta=3 * u.s, data={"Bx
↪": data})
>>> # Create a Spectra structure
>>> spectra = NDCollection(
...     [
...         (
...             "test_spectra",
...             NDCube(
...                 data=np.random.random(size=(4, 10)),
...                 wcs=WCS(naxis=2),
...                 meta={"CATDESC": "Test Spectra Variable"},
...                 unit="eV",
...             ),
...         )
...     ]
... )
>>> # Create a Support Structure
>>> support_data = {
...     "data_mask": NDData(data=np.eye(100, 100, dtype=np.uint16))
... }
>>> # Create Global Metadata Attributes
>>> input_attrs = HermesData.global_attribute_template("eea", "l1", "1.0.0")
>>> # Create HermesData Object
>>> hermes_data = HermesData(timeseries=ts, support=support_data, spectra=spectra,␣
↪meta=input_attrs)
```

Raises

- **ValueError** – If the number of columns is less than 2 or the required 'time' column is missing.:
- **TypeError** – If any column, excluding 'time', is not an astropy.units.Quantity object with units.:

- **`ValueError`** – If the elements of a `TimeSeries` column are multidimensional:
- **`TypeError`** – If any `supoport` data elements are not type `astropy.nddata.NDData`:
- **`TypeError`** – If *`spectra`* is not an `NDCollection` object.:

References

- Astropy TimeSeries
- Astropy Quantity and Units
- Astropy Time
- Astropy NDData
- Sunpy NDCube and NDCollection
- Space Physics Guidelines for CDF (ISTP)

Attributes Summary

`data`	(`dict`) A `dict` containing each of *`timeseries`* and *`support`*.
`meta`	(`collections.OrderedDict`) Global metadata associated with the measurement data.
`spectra`	(`ndcube.NDCollection]`) A `NDCollection` object containing high-dimensional spectra data.
`support`	(`dict[Union[astropy.units.Quantity, astropy.nddata.NDData]]`) A `dict` containing one or more non-time-varying support variables.
`time`	(`astropy.time.Time`) The times of the measurements.
`time_range`	(`tuple`) The start and end times of the times.
`timeseries`	(`astropy.timeseries.TimeSeries`) A `TimeSeries` representing one or more measurements as a function of time.

Methods Summary

add_measurement(measure_name, data[, meta])	Add a new time-varying scalar measurement (column).
add_spectra(name, data[, meta])	Add a new time-varying vector measurement.
add_support(name, data[, meta])	Add a new non-time-varying data array.
append(timeseries)	Add additional measurements to an existing column.
global_attribute_template([instr_name, ...])	Function to generate a template of the required ISTP-compliant global attributes.
load(file_path)	Load data from a file.
measurement_attribute_template()	Function to generate a template of the required measurement attributes.
plot([axes, columns, subplots])	Plot the measurement data.
remove(measure_name)	Remove an existing measurement or support data array.
save([output_path, overwrite])	Save the data to a HERMES CDF file.

Attributes Documentation

data

(`dict`) A `dict` containing each of *timeseries* and *support*.

meta

(`collections.OrderedDict`) Global metadata associated with the measurement data.

spectra

(`ndcube.NDCollection]`) A `NDCollection` object containing high-dimensional spectra data.

support

(`dict[Union[astropy.units.Quantity, astropy.nddata.NDData]]`) A `dict` containing one or more non-time-varying support variables.

time

(`astropy.time.Time`) The times of the measurements.

time_range

(`tuple`) The start and end times of the times.

timeseries

(`astropy.timeseries.TimeSeries`) A `TimeSeries` representing one or more measurements as a function of time.

Methods Documentation

add_measurement(*measure_name: str*, *data: Quantity*, *meta: dict* | *None* = *None*)

Add a new time-varying scalar measurement (column).

Parameters

- **measure_name** (`str`) – Name of the measurement to add.
- **data** (`astropy.units.Quantity`) – The data to add. Must have the same time stamps as the existing data.

- **meta** (dict, optional) – The metadata associated with the measurement.

Raises

- **TypeError** – If var_data is not of type Quantity.:
- **ValueError** – If data has more than one dimension:

add_spectra(*name: str*, *data: NDCube*, *meta: dict | None = None*)

Add a new time-varying vector measurement. This include higher-dimensional time-varying data.

Parameters

- **name** (str) – Name of the measurement to add.
- **data** (ndcube.NDCube) – The data to add. Must have the same time stamps as the existing data.
- **meta** (dict, optional) – The metadata associated with the measurement.

Raises

TypeError – If var_data is not of type NDCube.:

add_support(*name: str*, *data: Quantity | NDData*, *meta: dict | None = None*)

Add a new non-time-varying data array.

Parameters

- **name** (str) – Name of the data array to add.
- **data** (`Union[astropy.units.Quantity, astropy.nddata.NDData]`,) – The data to add.
- **meta** (`Optional[dict]`, optional) – The metadata associated for the data array.

Raises

TypeError – If var_data is not of type NDData.:

append(*timeseries: TimeSeries*)

Add additional measurements to an existing column.

Parameters

timeseries (astropy.timeseries.TimeSeries) – The data to be appended (rows) as a `TimeSeries` object.

static global_attribute_template(*instr_name: str = ''*, *data_level: str = ''*, *version: str = ''*) → OrderedDict

Function to generate a template of the required ISTP-compliant global attributes.

Parameters

- **instr_name** (str) – The instrument name. Must be “eea”, “nemisis”, “merit” or “spani”.
- **data_level** (str) – The data level of the data. Must be “l0”, “l1”, “ql”, “l2”, “l3”, “l4”
- **version** (str) – Must be of the form X.Y.Z.

Returns

template (collections.OrderedDict) – A template for required global attributes.

classmethod load(*file_path: str*)

Load data from a file.

Parameters

file_path (str) – A fully specificed file path.

Returns
: **data** (`HermesData`) – A `HermesData` object containing the loaded data.

Raises
: `ValueError` – If the file type is not recognized as a file type that can be loaded.:

`static measurement_attribute_template()` → OrderedDict

Function to generate a template of the required measurement attributes.

Returns
: **template** (`collections.OrderedDict`) – A template for required variable attributes that must be provided.

`plot`(*axes=None*, *columns=None*, *subplots=True*, ***plot_args*)

Plot the measurement data.

Parameters

- **axes** (`Axes`, optional) – If provided the image will be plotted on the given axes. Defaults to `None` and creates a new axis.
- **columns** (`list[str]`, optional) – If provided, only plot the specified measurements otherwise try to plot them all.
- **subplots** (`bool`) – If set, all columns are plotted in their own plot panel.
- ****plot_args** (`dict`, optional) – Additional plot keyword arguments that are handed to `Axes`.

Returns
: `Axes` – The plot axes.

`remove`(*measure_name: str*)

Remove an existing measurement or support data array.

Parameters
: **measure_name** (`str`) – Name of the variable to remove.

`save`(*output_path: str* | *None* = *None*, *overwrite: bool* = *False*)

Save the data to a HERMES CDF file.

Parameters

- **output_path** (`str`, optional) – A string path to the directory where file is to be saved. If not provided, saves to the current directory.
- **overwrite** (`bool`) – If set, overwrites existing file of the same name.

Returns
: **path** (`str`) – A path to the saved file.

5.3 hermes_core.util Package

5.3.1 Functions

create_science_filename(instrument, time, ...)	Return a compliant filename.
parse_science_filename(filepath)	Parses a science filename into its consitutient properties (instrument, mode, test, time, level, version, descriptor).
warn_deprecated(msg[, stacklevel])	Raise a *HERMESDeprecationWarning*.
warn_user(msg[, stacklevel])	Raise a *HERMESUserWarning*.

create_science_filename

hermes_core.util.**create_science_filename**(*instrument: str*, *time: str*, *level: str*, *version: str*, *mode: str = ''*, *descriptor: str = ''*, *test: bool = False*)

Return a compliant filename. The format is defined as

hermes_{inst}_{mode}_{level}{test}_{descriptor}_{time}_v{version}.cdf

This format is only appropriate for data level >= 1.

Parameters

- **instrument** (str) – The instrument name. Must be one of the following "eea", "nemesis", "merit", "spani"
- **time** (str (in isot format) or ~astropy.time) – The time
- **level** (str) – The data level. Must be one of the following "l0", "l1", "l2", "l3", "l4", "ql"
- **version** (str) – The file version which must be given as X.Y.Z
- **descriptor** (str) – An optional file descriptor.
- **mode** (str) – An optional instrument mode.
- **test** (*bool*) – Selects whether the file is a test file.

Returns

filename (str) – A CDF file name including the given parameters that matches the HERMES file naming conventions

Raises

- **ValueError** – If the instrument is not recognized as one of the HERMES instruments:
- **ValueError** – If the data level is not recognized as one of the HERMES valid data levels:
- **ValueError** – If the data version does not match the HERMES data version formatting conventions:
- **ValueError** – If the data product descriptor or instrument mode do not match the HERMES formatting conventions:

parse_science_filename

`hermes_core.util.`**`parse_science_filename`**(*filepath: str*) → dict

Parses a science filename into its consitutient properties (instrument, mode, test, time, level, version, descriptor).

Parameters

`filepath` (`str`) – Fully specificied filepath of an input file

Returns

result (`dict`) – A dictionary with each property.

Raises

- **`ValueError`** – If the file's mission name is not "HERMES":
- **`ValueError`** – If the file's instreument name is not one of the HERMES instruments:
- **`ValueError`** – If the data level >0 for packet files:
- **`ValueError`** – If not a CDF File:

warn_deprecated

`hermes_core.util.`**`warn_deprecated`**(*msg*, *stacklevel=1*)

Raise a *`HERMESDeprecationWarning`*.

Parameters

- **`msg`** (*`str`*) – Warning message.
- **`stacklevel`** (*`int`*) – This is interpreted relative to the call to this function, e.g. `stacklevel=1` (the default) sets the stack level in the code that calls this function.

warn_user

`hermes_core.util.`**`warn_user`**(*msg*, *stacklevel=1*)

Raise a *`HERMESUserWarning`*.

Parameters

- **`msg`** (*`str`*) – Warning message.
- **`stacklevel`** (*`int`*) – This is interpreted relative to the call to this function, e.g. `stacklevel=1` (the default) sets the stack level in the code that calls this function.

5.3.2 Classes

`HERMESDeprecationWarning`	A warning class to indicate a deprecated feature.
`HERMESPendingDeprecationWarning`	A warning class to indicate a soon-to-be deprecated feature.
`HERMESUserWarning`	The primary warning class for HERMES.
`HERMESWarning`	The base warning class from which all HERMES warnings should inherit.

HERMESDeprecationWarning

exception `hermes_core.util.`**`HERMESDeprecationWarning`**

A warning class to indicate a deprecated feature.

HERMESPendingDeprecationWarning

exception `hermes_core.util.`**`HERMESPendingDeprecationWarning`**

A warning class to indicate a soon-to-be deprecated feature.

HERMESUserWarning

exception `hermes_core.util.`**`HERMESUserWarning`**

The primary warning class for HERMES.

Use this if you do not need a specific type of warning.

HERMESWarning

exception `hermes_core.util.`**`HERMESWarning`**

The base warning class from which all HERMES warnings should inherit.

Any warning inheriting from this class is handled by the HERMES logger. This warning should not be issued in normal code. Use "HERMESUserWarning" instead or a specific sub-class.

5.4 hermes_core.util.io Module

5.4.1 Classes

CDFHandler()	A concrete implementation of HermesDataIOHandler for handling heliophysics data in CDF format.

CDFHandler

class `hermes_core.util.io.`**`CDFHandler`**

Bases: `HermesDataIOHandler`

A concrete implementation of HermesDataIOHandler for handling heliophysics data in CDF format.

This class provides methods to load and save heliophysics data from/to a CDF file.

Methods Summary

load_data(file_path)	Load heliophysics data from a CDF file.
save_data(data, file_path)	Save heliophysics data to a CDF file.

Methods Documentation

load_data(*file_path: str*) → Tuple[TimeSeries, dict]

Load heliophysics data from a CDF file.

Parameters

file_path (str) – The path to the CDF file.

Returns

- **data** (TimeSeries) – An instance of TimeSeries containing the loaded data.
- **support** (dict[astropy.nddata.NDData]) – Non-record-varying data contained in the file
- **spectra** (ndcube.NDCollection) – Spectral or High-dimensional measurements in the loaded data.

save_data(*data*, *file_path: str*)

Save heliophysics data to a CDF file.

Parameters

- **data** (*hermes_core.timedata.HermesData*) – An instance of HermesData containing the data to be saved.
- **file_path** (str) – The path to save the CDF file.

Returns

path (str) – A path to the saved file.

5.5 hermes_core.util.schema Module

This module provides schema metadata derivations.

This code is based on that provided by SpacePy see
licenses/SPACEPY.rst

5.5.1 Classes

HermesDataSchema()	Class representing the schema of a file type.

HermesDataSchema

class hermes_core.util.schema.**HermesDataSchema**

Bases: `object`

Class representing the schema of a file type.

Attributes Summary

default_global_attributes	(`dict`) Default Global Attributes applied for all HERMES Data Files
global_attribute_schema	(`dict`) Schema for variable attributes of the file.
variable_attribute_schema	(`dict`) Schema for variable attributes of the file.

Methods Summary

derive_global_attributes(data)	Function to derive global attributes for the given measurement data.
derive_measurement_attributes(data, var_name)	Function to derive metadata for the given measurement.
derive_time_attributes(data)	Function to derive metadata for the time measurement.
global_attribute_info([attribute_name])	Function to generate a `astropy.table.Table` of information about each global metadata attribute.
global_attribute_template()	Function to generate a template of required global attributes that must be set for a valid CDF.
measurement_attribute_info([attribute_name])	Function to generate a `astropy.table.Table` of information about each variable metadata attribute.
measurement_attribute_template()	Function to generate a template of required measurement attributes that must be set for a valid CDF measurement variable.

Attributes Documentation

default_global_attributes

(`dict`) Default Global Attributes applied for all HERMES Data Files

global_attribute_schema

(`dict`) Schema for variable attributes of the file.

variable_attribute_schema

(`dict`) Schema for variable attributes of the file.

Methods Documentation

derive_global_attributes(*data*) → OrderedDict

Function to derive global attributes for the given measurement data.

Parameters

data (*hermes_core.timedata.HermesData*) – An instance of `HermesData` to derive metadata from.

Returns

attributes (`OrderedDict`) – A dict containing `key: value` pairs of global metadata attributes.

derive_measurement_attributes(*data*, *var_name: str*, *guess_types: list[int] | None = None*) → OrderedDict

Function to derive metadata for the given measurement.

Parameters

- **data** (*hermes_core.timedata.HermesData*) – An instance of `HermesData` to derive metadata from
- **var_name** (`str`) – The name of the measurement to derive metadata for
- **guess_types** (`list[int]`, optional) – Guessed CDF Type of the variable

Returns

attributes (`OrderedDict`) – A dict containing `key: value` pairs of derived metadata attributes.

derive_time_attributes(*data*) → OrderedDict

Function to derive metadata for the time measurement.

Parameters

data (*hermes_core.timedata.HermesData*) – An instance of `HermesData` to derive metadata from.

Returns

attributes (`OrderedDict`) – A dict containing `key: value` pairs of time metadata attributes.

static **global_attribute_info**(*attribute_name: str | None = None*) → Table

Function to generate a `astropy.table.Table` of information about each global metadata attribute. The `astropy.table.Table` contains all information in the HERMES global attribute schema including:

- description: (`str`) A brief description of the attribute
- default: (`str`) The default value used if none is provided
- **derived: (`bool`) Whether the attibute can be derived by the HERMES** *HermesDataSchema* class
- required: (`bool`) Whether the attribute is required by HERMES standards
- **validate: (`bool`) Whether the attribute is included in the** *validate()* checks (Note, not all attributes that are required are validated)
- **overwrite: (`bool`) Whether the *HermesDataSchema*** attribute derivations will overwrite an existing attribute value with an updated attribute value from the derivation process.

Parameters
`attribute_name` (`str`, optional, default None) – The name of the attribute to get specific information for.

Returns
info (`astropy.table.Table`) – A table of information about global metadata.

Raises
`KeyError` – If attribute_name is not a recognized global attribute.:

`static global_attribute_template()` → OrderedDict

Function to generate a template of required global attributes that must be set for a valid CDF.

Returns
template (`OrderedDict`) – A template for required global attributes that must be provided.

`static measurement_attribute_info(`*attribute_name: str | None = None*`)` → Table

Function to generate a `astropy.table.Table` of information about each variable metadata attribute. The `astropy.table.Table` contains all information in the HERMES variable attribute schema including:

- description: (`str`) A brief description of the attribute
- **derived: (`bool`) Whether the attibute can be derived by the HERMES**
  `HermesDataSchema` class
- required: (`bool`) Whether the attribute is required by HERMES standards
- **overwrite: (`bool`) Whether the *`HermesDataSchema`***
  attribute derivations will overwrite an existing attribute value with an updated attribute value from the derivation process.
- **valid_values: (`str`) List of allowed values the attribute can take for HERMES products,**
  if applicable
- **alternate: (`str`) An additional attribute name that can be treated as an alternative**
  of the given attribute. Not all attributes have an alternative and only one of a given attribute or its alternate are required.
- **var_types: (`str`) A list of the variable types that require the given**
  attribute to be present.

Parameters
`attribute_name` (`str`, optional, default None) – The name of the attribute to get specific information for.

Returns
info (`astropy.table.Table`) – A table of information about variable metadata.

Raises
`KeyError` – If attribute_name is not a recognized global attribute.:

`static measurement_attribute_template()` → OrderedDict

Function to generate a template of required measurement attributes that must be set for a valid CDF measurement variable.

Returns
template (`OrderedDict`) – A template for required variable attributes that must be provided.

5.6 hermes_core.util.util Module

This module provides general utility functions.

5.6.1 Functions

create_science_filename(instrument, time, ...)	Return a compliant filename.
parse_science_filename(filepath)	Parses a science filename into its consitutient properties (instrument, mode, test, time, level, version, descriptor).

create_science_filename

`hermes_core.util.util.`**`create_science_filename`**(*instrument: str, time: str, level: str, version: str, mode: str = '', descriptor: str = '', test: bool = False*)

Return a compliant filename. The format is defined as

hermes_{inst}_{mode}_{level}{test}_{descriptor}_{time}_v{version}.cdf

This format is only appropriate for data level >= 1.

Parameters

- **`instrument`** (`str`) – The instrument name. Must be one of the following "eea", "nemesis", "merit", "spani"
- **`time`** (`str` (in isot format) or ~astropy.time) – The time
- **`level`** (`str`) – The data level. Must be one of the following "l0", "l1", "l2", "l3", "l4", "ql"
- **`version`** (`str`) – The file version which must be given as X.Y.Z
- **`descriptor`** (`str`) – An optional file descriptor.
- **`mode`** (`str`) – An optional instrument mode.
- **`test`** (*`bool`*) – Selects whether the file is a test file.

Returns

filename (`str`) – A CDF file name including the given parameters that matches the HERMES file naming conventions

Raises

- **`ValueError`** – If the instrument is not recognized as one of the HERMES instruments:
- **`ValueError`** – If the data level is not recognized as one of the HERMES valid data levels:
- **`ValueError`** – If the data version does not match the HERMES data version formatting conventions:
- **`ValueError`** – If the data product descriptor or instrument mode do not match the HERMES formatting conventions:

parse_science_filename

`hermes_core.util.util.`**`parse_science_filename`**(*filepath: str*) → dict

Parses a science filename into its consitutient properties (instrument, mode, test, time, level, version, descriptor).

Parameters
`filepath` (`str`) – Fully specificied filepath of an input file

Returns
result (`dict`) – A dictionary with each property.

Raises

- **`ValueError`** – If the file's mission name is not "HERMES":
- **`ValueError`** – If the file's instreument name is not one of the HERMES instruments:
- **`ValueError`** – If the data level >0 for packet files:
- **`ValueError`** – If not a CDF File:

5.7 hermes_core.util.validation Module

5.7.1 Functions

`validate`(filepath)	Validate a data file such as a CDF.

validate

`hermes_core.util.validation.`**`validate`**(*filepath: str*) → list[str]

Validate a data file such as a CDF.

Parameters
`filepath` (`str`) – A fully specificed file path.

Returns
errors (`list[str]`) – A list of validation errors returned. A valid file will result in an emppty list being returned.

5.7.2 Classes

`CDFValidator`()	Validator for CDF files.

CDFValidator

class hermes_core.util.validation.**CDFValidator**

Bases: `HermesDataValidator`

Validator for CDF files.

Methods Summary

validate(file_path)	Validate the CDF file.

Methods Documentation

validate(*file_path: str*) → list[str]

Validate the CDF file.

Parameters

file_path (`str`) – The path to the CDF file.

Returns

errors (`list[str]`) – A list of validation errors returned. A valid file will result in an emppty list being returned.

CHAPTER

SIX

EXAMPLES

6.1 Creating a CDF File

This module provides an example for creating a CDF File using the *HermesData* class. This class is an abstraction of underlying data structures to make the handling of measurement data easier when reading and writing CDF data.

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.timeseries import TimeSeries
>>> from astropy.nddata import NDData
>>> from astropy.wcs import WCS
>>> from ndcube import NDCube, NDCollection
>>> import tempfile
>>>
>>> # Import the `hermes_core` Package
>>> from hermes_core.timedata import HermesData
>>> from hermes_core.util.validation import validate
>>>
>>> # Create a np.ndarray of example measurement data
>>> bx = np.random.choice(a=[-1, 0, 1], size=1000).cumsum(0)
>>> by = np.random.choice(a=[-1, 0, 1], size=1000).cumsum(0)
>>>
>>> # Create a TimeSeries with the example measurement and a Time column
>>> ts = TimeSeries(
...     time_start="2016-03-22T12:30:31",
...     time_delta=3 * u.s,
...     data={"Bx GSE": u.Quantity(value=bx, unit="nanoTesla", dtype=np.int16)},
... )
>>>
>>> # You can also add new measurements to the TimeSeries directly
>>> ts.add_column(col=u.Quantity(value=by, unit="nanoTesla", dtype=np.int16),
...     name="By GSE"
... )
>>>
>>> # Create support data or non-time-varying (time invariant) data
>>> support_data = {
...     "data_mask": NDData(data=np.eye(100, 100, dtype=np.uint16))
... }
>>>
>>> # Create high-dimensional data leveraging the API of NDCube
```

(continues on next page)

(continued from previous page)

```
>>> spectra = NDCollection(
...     [
...         (
...             "example_spectra",
...             NDCube(
...                 data=np.random.random(size=(4, 10)),
...                 wcs=WCS(naxis=2),
...                 meta={"CATDESC": "Example Spectra Variable"},
...                 unit="eV",
...             ),
...         )
...     ]
... )
>>>
>>> # To make the creation of global metadata easier you can use the static
>>> # `HermesData.global_attribute_template()` function.
>>> global_attrs_template = HermesData.global_attribute_template()
>>>
>>> global_attrs_template["DOI"] = "https://doi.org/<PREFIX>/<SUFFIX>"
>>> global_attrs_template["Data_level"] = "L1>Level 2"
>>> global_attrs_template["Data_version"] = "0.0.1"
>>> global_attrs_template[
...     "Descriptor"
... ] = "nemisis>Noise Eliminating Magnetometer Instrument in a Small Integrated System"
>>> global_attrs_template["Instrument_mode"] = "default"
>>> global_attrs_template["Instrument_type"] = "Magnetic Fields (space)"
>>> global_attrs_template["Data_product_descriptor"] = "odpd"
>>>
>>> global_attrs_template["HTTP_LINK"] = [
...     "https://science.nasa.gov/missions/hermes",
...     "https://github.com/HERMES-SOC",
...     "https://github.com/HERMES-SOC/hermes_nemisis",
... ]
>>> global_attrs_template["LINK_TEXT"] = ["HERMES homepage",
...     "HERMES SOC Github", "NEMISIS Analysis Tools"]
>>> global_attrs_template["LINK_TITLE"] = ["HERMES homepage",
...     "HERMES SOC Github", "NEMISIS Analysis Tools"]
>>>
>>> global_attrs_template["MODS"] = ["v0.0.1 - Original version."]
>>> global_attrs_template["PI_affiliation"] = "NASA Goddard Space Flight Center"
>>> global_attrs_template["PI_name"] = "Dr. Eftyhia Zesta"
>>> global_attrs_template["TEXT"] = "Sample HERMES NEMISIS CDF File"
>>>
>>> example_data = HermesData(
...     timeseries=ts,
...     support=support_data,
...     spectra=spectra,
...     meta=global_attrs_template
... )
>>>
>>> # To make the creation of variable metadata easier you can use the static
>>> # `HermesData.measurement_attribute_template()` function.
```

(continues on next page)

(continued from previous page)

```
>>> template = HermesData.measurement_attribute_template()
>>>
>>> # Update the Metadata for each of the Measurements
>>> example_data.timeseries["Bx GSE"].meta.update(
...     OrderedDict({"CATDESC": "X component of magnetic Field GSE"}))
>>> example_data.timeseries["By GSE"].meta.update(
...     OrderedDict({"CATDESC": "Y component of magnetic Field GSE"}))
>>>
>>> # You can add new scalar time-variant measurements to the HermesData container
>>> bz = np.random.choice(a=[-1, 0, 1], size=1000).cumsum(0)
>>> example_data.add_measurement(
...     measure_name="Bz GSE",
...     data=u.Quantity(value=bz, unit="nanoTesla", dtype=np.int16),
...     meta={
...         "VAR_TYPE": "data",
...         "CATDESC": "Z component of magnetic Field GSE",
...     },
... )
>>>
>>> # You can add new time-invariant data to the HermesData container
>>> example_data.add_support(
...     name="calibration_const",
...     data=NDData(data=[1e-1]),
...     meta={
...         "CATDESC": "Calibration Factor",
...         "VAR_TYPE": "metadata"
...     },
... )
>>>
>>> # You can ass new spectral or high-dimensional data to the HermesData container
>>> data = NDCube(
...     data=np.random.random(size=(1000, 10)),
...     wcs=WCS(naxis=2),
...     meta={"CATDESC": "Example Spectra Variable"},
...     unit="eV",
... )
>>> example_data.add_spectra(
...     name="added_spectra",
...     data=data,
...     meta={"VAR_TYPE": "data"},
... )
>>>
>>> # create the CDF File
>>> DRYRUN=True
>>> if DRYRUN:
...     with tempfile.TemporaryDirectory() as tmpdirname:
...         cdf_file_path = example_data.save(output_path=tmpdirname)
... else:
...     cdf_file_path = example_data.save(output_path="./", overwrite=True)
```

The file that this code generates is made available as a sample file in this repository in `hermes_core/data/sample/hermes_nms_default_l1_20160322T123031_v0.0.1.cdf`.

PYTHON MODULE INDEX

INDEX