Multi-Beam FITS (MBFITS) format data storage module in ESCS/Nuraghe

P. Libardi, S. Righini, A. Orlati

IRA 461/12

1. From the origins to a standard format

MBFITS is a raw data format for multi-beam multi-receiver/backend single dish telescopes.
It was originally created to be used at the IRAM 30m, APEX 12m and Effelsberg 100m mm/submm telescopes, but this format is suitable for single-dish bolometers and heterodyne observations.

The MBFITS format was structurally derived from the ALMA Test Interferometer FITS (ALMA-TI FITS) raw data format. MBFITS is based on the scan-subscan-integration scheme used by ALMA-TI FITS and retains many of its keywords. It uses the FITS standard key-value format and the World Coordinate System representation. While keeping most of the original design concept from ALMA TI-FITS, new structures and keywords have been added to accommodate multiple beam observations and multiple frontend and backend combinations. These changes were required by telescopes that decided to adopt MBFITS in a single-dish configuration. The MBFITS format can now be considered to be an independent format.

Since July 2007 the MBFITS format description is an officially registered FITS convention.
The Registry of FITS Conventions (http://fits.gsfc.nasa.gov/fits_registry.html) provides a central and authoritative repository for documenting conventions that have been developed by the FITS user community for storing and transmitting various types of information in FITS format data files.

The structure of the MBFITS format has been described in many articles, and new version are currently under development as new requirements emerge from observations.

List of the most relevant documents to be used as reference for the new MBFITS format:

- Muders, D., Polehampton, E. & Hatchell, J., 2007, *"Multi-beam FITS Raw Data Format"*, APEX Report APEX-MPI-ICD-0002, Rev. 1.63
- Wells, D.C., Greisen, E.W. & Harten, R.H., 1981, *"FITS - a Flexible Image Transport System"*, A&AS, 44, 363-+
- Lucas, R. and Glendenning, B., 2001, *"ALMA Test Interferometer Raw Data Format"*, ALMA Report ALMA-SW-0015
- Greisen, E.W. & Calabretta, M.R., 2002, *"Representations of world coordinates in FITS"*, A&A, 395, 1061-1075

2. Data structure and time frames

The MBFITS format is structured as a set of extension table, as show in Figure 1. Data are stored as key-value pairs or as information in binary table sections.

From the MBFITS reference manual we get a description of the most relevant time frames, from shortest to longest:

- **dump:** the smallest interval of time for which a set of correlated data can be accumulated and output from the backend;
- **integration:** a set of dumps, all identical in configuration (except for the antenna motion and some others), that is accumulated and forms the basic recorded unit;
- **subscan:** a set of integrations achieved while the antennas complete an elemental pattern across the source, possibly while performing frequency switching, nutator switching, etc. (previous to v.1.54: observation);
- **scan:** a set of subscans with a common goal, e.g.: a pointing scan, a focus scan, an atmospheric amplitude calibration scan, a correlation scan on a continuum source or a spectral line source.

This subdivision reflects in the structure of the MBFITS tables as shown on the left side of Figure 1.

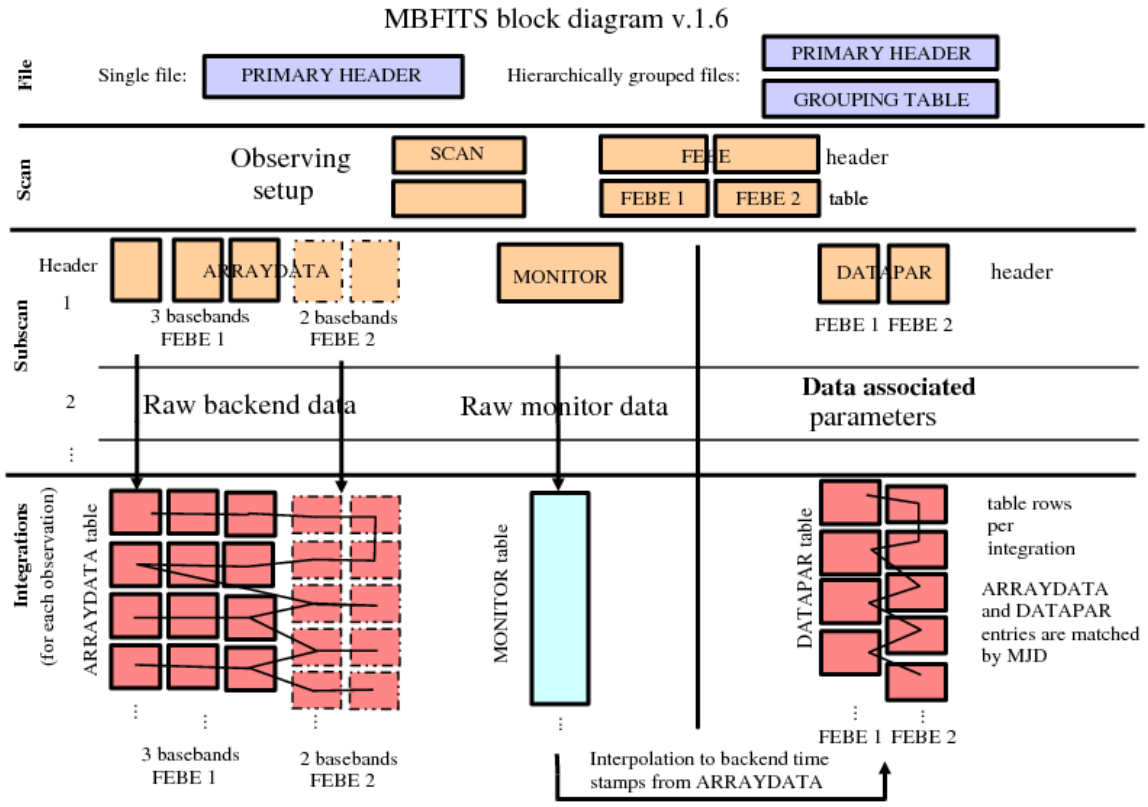

Figure 1 – MBFITS structure

3. Software for data analysis

As a consequence of the adoption of this new format in some telescopes, astronomers and software developers had to create new tools or to adapt data reduction and analysis software.
The following list is not exhaustive as development is always ongoing:

- **Toolbox** – MPIfR, Effelsberg
  The MBFITS data can be inspected and edited with any program that understands FITS Format (e.g. fv - FITS viewer). However, most users might prefer a kind of pre-reduced view where the amplitude of the scan is calibrated and with actual arcseconds for the scanning axis. This is provided by the "Toolbox" program.

- **MIRA** (**M**ultichannel **I**maging and Calibration Software for **R**eceiver **A**rrays) – IRAM
  MIRA is the new software developed at IRAM to replace the previous tool OTFCAL for the calibration of the new IMBFITS raw data at the 30m telescope.

- **BoA** (**BO**lometer **A**rray Analysis Software) – MPIfR, AIfA, APEX et al.
  BoA is a newly designed software package for the reading, handling, and analysis of bolometer array data. The primary goal of BoA is to handle data from LABOCA, the Large APEX Bolometer Camera, both for online visualization and offline processing. BoA can also be used to process data acquired with other instruments such as ASZCa (the APEX SZ Camera) or MAMBO at the IRAM 30-meter telescope. BoA includes most of the relevant functionalities of the current reduction packages (MOPSIC, NIC, SURF). The major difference to them is that BoA is written in a programming environment that is (hopefully) easier to modify, maintain, and re-use.
  Moreover, BoA naturally interfaces with APECS and the MBFITS format.
  The design of BoA has been done with two major goals in mind: to provide a comprehensive tool for the visualisation, reduction and analysis of data from the new generation of bolometer arrays, and to facilitate the extension and modification of the software by users with no strong programming background.

- **Fitsplode** - Steve Torchinsky @ Nançay Radio Astronomy Facility
  It is a data extractor for spectral line data in FITS files which use binary tables. In particular, it works for raw data from Effelsberg (in MBFITS format) and raw data from Arecibo (in CIMAFITS format). It explodes the file into individual FITS files each with one spectrum. These files can then be read into your favourite data processing program, such as for example *xs* by Per Bergman, and others, which do not read binary tables in FITS files, but which do read simple FITS files.

4. MBFITS implementation: hierarchical structure

The original MBFITS format was intended to be written as one file in order to facilitate transporting the data from the observatory to the users. However, the FITS standard requires the binary tables to be written sequentially in the file. When writing an MBFITS file during the actual observations, the underlying library (e.g. CFITSIO in case of APEX) thus always needs to rearrange the tables to make room for the new incoming data. To avoid this complication, starting from version 1.6 of the MBFITS reference document, the FITS hierarchical grouping standard (see Jennings et al. 1997) is employed.

The MBFITS hierarchical grouping directory structure is defined as follows (see Fig. 1):

- Main directory name according to the value of data relevant to the observation

- Inside this main directory, there are the files for the scan-level tables:
  - The grouping table file: **GROUPING**.fits
  - The scan table file: **SCAN**.fits
  - The FEBEPAR table files for each FEBE combination: **<FEBE name>-FEBEPAR**.fits

- The actual data is stored in subdirectories, one for each subscan, named according to the subscan number.
  Each subdirectory contains the following types of member files:
  - The MONITOR table file: **MONITOR**.fits
  - One ARRAYDATA table file for each FEBE combination and baseband: **<FEBE name>-ARRAYDATA-<Baseband number>**.fits
  - One DATAPAR table file for each FEBE combination: **<FEBE name>-DATAPAR**.fits

4.1. GROUPING Table

This table exists only in the hierarchical implementation of the MBFITS format and it is created once for each scan.
It is used to store the locations of the member files, plus other details which can be exploited to speed up searching when reading the files.

4.2. SCAN Table

It is stored for every scan. It contains parameters which do not change among the subscans, including:

- telescope and observatory parameters
- time system
- coordinate system
- velocity system
- project ID
- target of the scan and its coordinates
- observing mode
- pointing model coefficients

4.3. FEBEPAR Table

The FEBEPAR table is stored per FEBE (FrontEnd – BackEnd) combination for each scan and contains the frontend-backend setup. Parameters common to all FEBEs are written in the SCAN table.
It includes:

- FEBE setup: number of pixels, polarisations and basebands
- pointing model coefficients specific to this FE
- calibration parameters specific to this FEBE

4.4. ARRAYDATA Table

A new ARRAYDATA table is created for each subscan, for each FEBE and for each baseband. It stores the data description (header) and the data (table).
It includes:

- frequency band setup: frequency, bins (freq. channels), polarisations, line ID
- data axes description

If some parameters change for the individual subscan with respect to the general value stored in the SCAN table, data analysis applications should get these values from the ARRAYDATA table rather than from the SCAN one.

4.5. DATAPAR Table

A new DATAPAR table is created for each subscan and for each FEBE.
Parameters common to all the subscans are written in the SCAN table, while the FEBE setup is recorded in the FEBEPAR table (also assumed to be constant for all subscans).
The DATAPAR table contains those data-associated parameters which change with the integration, but not the data themselves – as they are stored in the ARRAYDATA table.
The table includes:

- time and coordinates information, specific to this subscan and integration
- interpolated data from the MONITOR table, resampled to the timestamps of the midpoints of the integrations, as given by the MJD timestamp.

4.6. MONITOR Table

This table stores raw monitoring data (real-time updates other than the backend data) at their natural rate, i.e. not synchronised to backend dump times.
The monitor data are stored as time-keyword-units-values.
The update intervals for any monitor stream are thus fully flexible.
It is recommended that the telescope control system should call for updates on monitor points at least at the beginning and end of the scans. As many of these as possible should be measured at these times. For points where a new measurement is not possible the last measurement should be saved again in the MONITOR table with its original timestamp. In this way, interpolation between points to fill in the DATAPAR table will be possible even without access to previous/later scan data.
MONITOR table updates:

- at the beginning/end of scans: calibration data, pointing data, radiometer data, weather station data
- at the beginning of integrations: frequencies, current real antenna positions
- at the end of observations: current real antenna positions

5. ESCS (Enhanced Single-dish Control System) and Nuraghe

At the Medicina telescope the observers operate with ESCS (*Enhanced Single-dish Control System*). A different "flavour" of the system, called *Nuraghe*, manages the Sardinia Radio Telescope antenna and several of its devices.
ESCS/Nuraghe was developed in the Alma Common Software (ACS) framework, which is optimised for the production of control software to handle complex instruments. From *IRA-TechReport 423-08* (Orlati et al.) we can extract the definitions to identify some elements that constitute the control system to setup and to drive the antenna, and to store observing data into different formats:

- **Subsystem**
  Components and table artefacts are grouped into subsystems that correspond to a single functionality unit. A subsystem is a container of components and artefacts that belong to the same sub-part of the whole system.
- **Component**
  The component corresponds to the ACS/CORBA component/remote object. Each component can implement a specific functionality that is required to realize the system. Components have relationships between each other and/or they can implement or realize interfaces.
- **Interface**
  An interface defines a common way for components to access other components or for clients to call components through common patterns. Interfaces allow to group components that show the same services but differ in the way the services are implemented or executed.
- **Artefacts**
  These objects can be files, libraries or database tables that contain data or algorithms to be used by different components as external resources.

The interactions between various components to accomplish the observations are driven by the execution of the commanded schedule. This is realized through the code implemented in the component *ScheduleExecutor*, which has been developed as a finite state machine.
The *ScheduleExecutor* makes no assumptions about the implementation of the backend and the format to store information and data: it manages a backend component and a writer component.

6. Data sender and data receiver

Figure 2 shows the hierarchy of interfaces for the sender components (left group) and for the receiver elements (right group) of the ESCS/Nuraghe system. In the lowest level we drew only a couple of the interfaces currently implemented in the system.

ScheduleExecutor creates instances of the backend and of the writer implementation, on the basis of string names that are provided by the user in the schedule files and that identify different implementations of these components. *ScheduleExecutor* then initializes these instances to setup the configuration as required by the observer in the schedule files.

On the last step of the scan start stage the *ScheduleExecutor* invokes the "*sendData*" method for the instance of the backend, to effectively start the data transmission along the stream that has been previously created between the instance of the backend and the implementation of the writer.

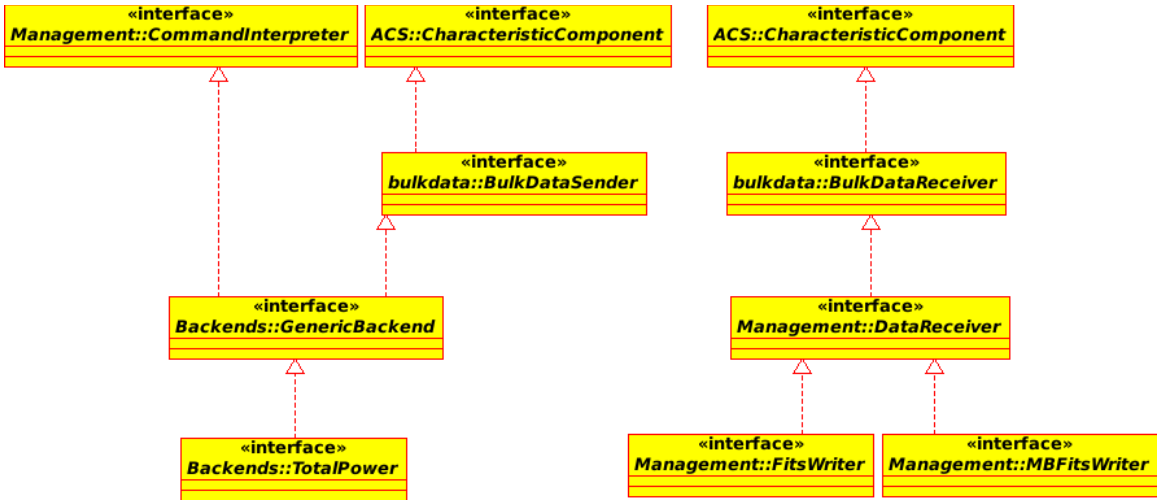

Figure 2 – Hierarchy for the sender components' interfaces (left) and for the receiver elements' interfaces (right)

To store information and data in the MBFITS format we developed a component that implements the interface required by the *ScheduleExecutor* for the writer component. We realized this interface in the new class *MBFitsWriterImpl*, which is an element of the model component-container *MBFitsWriter*. Other elements of this model implementation are shown in Figure 3 and will be described later: they implement the intermediate and the low level methods between the ESCS *ScheduleExecutor* and the actual storing operation to MBFITS files.
We also defined some CDB (Configuration DataBase) schemas and the corresponding tables to store parameters relative to the different setup configurations, together with the values describing the models used by the antenna control system.

7. MBFITS implementation of the component-container model

Figure 3 shows the UML diagram of the classes that constitute the component-container model implemented in ESCS/Nuraghe in charge of the registration of the device configuration and the observational data in the MBFITS format, according to the structure described in the reference manual.
The base element activated by the *ScheduleExecutor* component is an instance of the class *MBFitsWriterImpl* in which we implemented the methods required by the interface of the ACS Bulk Data Transfer. This instance of the class *MBFitsWriterImpl* then manages the receiving of data, their handling and the storage to MBFITS format structures, by means of an instance for each of the following classes:

- CConfiguration
- CDataCollection
- CSecureArea
- EngineThread
- CCollectorThread

7.1. CConfiguration

It is the delegate class in reading information stored in the CDB for the instance of the component. This includes the values of the timeouts for the execution of threads to acquire the configuration and the data from other system components, the paths and the filenames for the definitions of the components used by the writing component-container model.

7.2. CDataCollection

Information related to the acquisition/recording status of the observational data, in addition to the configuration parameters of the observation and to the data themselves, is stored in this class.

Only one instance of this class can exist for all the elements of the component-container model implementation of the MBFITS writer. It is created in the *MBFitsWriterImpl* instance and then it is assigned to the different classes to temporary store the information provided by the devices, before it is stored to the MBFITS tables.

It is important to note that this class is not implemented safely for multiple threads.

In order to permit proper operations with the various threads used in the component, we instantiate a realization of the template class *CSecureArea* setting its parameter to the class *CDataCollection*.

7.3. CSecureArea

This is a template class that implements a mechanism to protect a resource, being the instance of the class that the template parameter is valued to. This protection enables to manage sequential access requests to the resource. Such requests can come asynchronously from various instances of different classes, possibly even belonging to different threads.

The resource has to be accessed through a method implemented in this class, allowing in this way to allocate the resource to a single applicant at a time. The allocation of the resource by means of this method also allows to avoid the explicit release of the resource: when the variable to which the resource is assigned goes out of scope, the resource is released automatically and transparently to the user.

7.4. EngineThread

It is a class derived from ACS::Thread. It is used to manage the information that may be available from the backend and other components of the ESCS/Nuraghe system to coordinate data acquisition and its subsequent registration of the MBFITS files.

The instance of this class uses the state variables stored in the instance of the class *CDataCollection* to determine the available information and to invoke methods to store data in the MBFITS files. These methods, closely related to the structure of the MBFITS format, are implemented in the classes *MBFitsManager*, *CMBFitsWriterTable*, *CMBFitsWriter* as shown in the UML diagram of the classes (Figure 3).

The process of data storage in the MBFITS tables is commanded by a sequence of interactions between the instance of this class and the instances of other classes as shown in Figures 4 to 10.

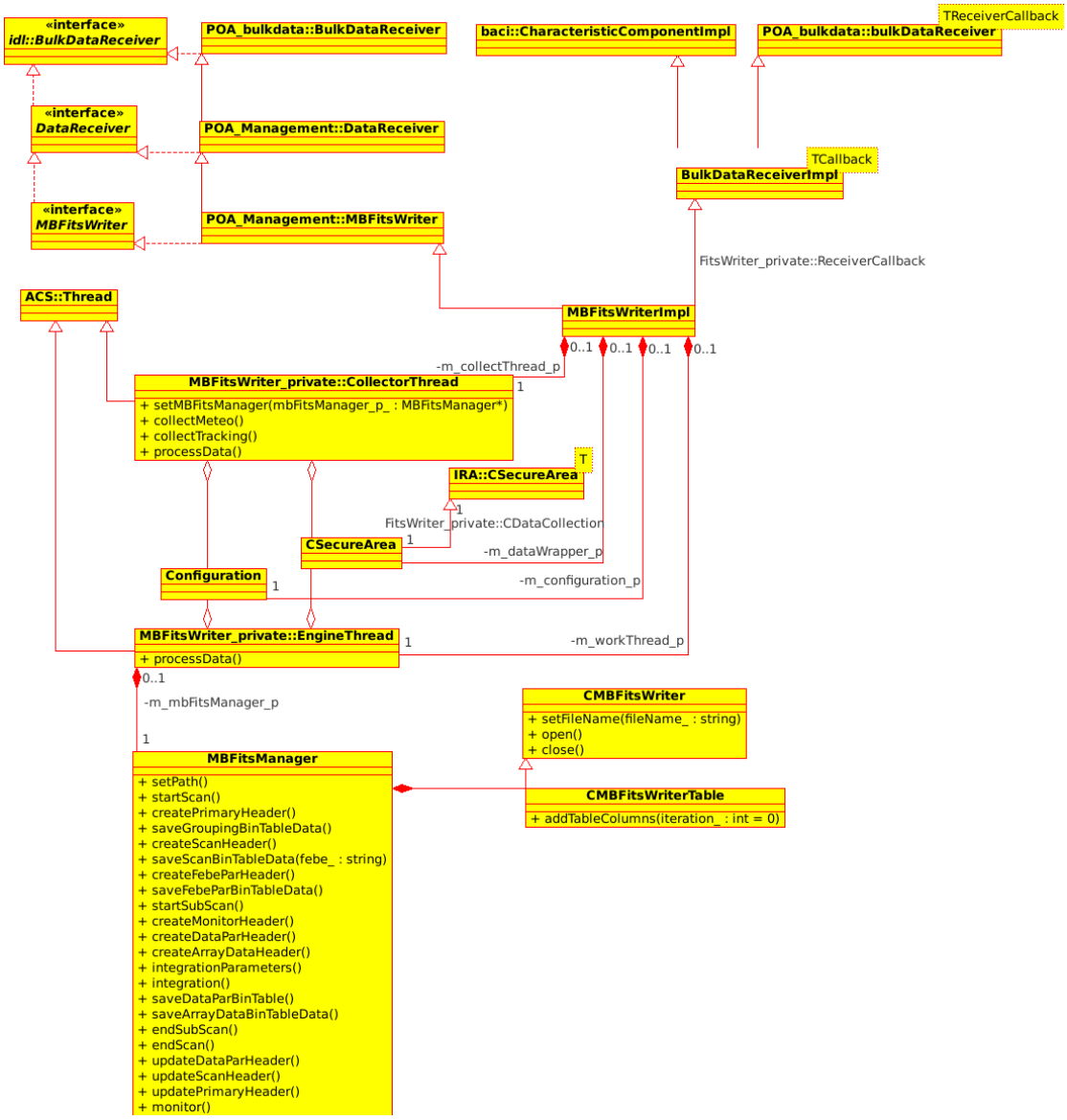

Figure 3 - UML diagram of the classes that constitute the component-container model.

7.5. CCollectorThread

It is a class derived from ACS::Thread. It is used to manage the information that may be collected from the ESCS/Nuraghe components that are not directly involved in the acquisition of observational data. This material, such as the weather station data and information related to the status of the tracking system, will be useful in the subsequent analysis phases.

The MONITOR table, documented in the MBFITS file structure, is designed to record information that is available asynchronously with respect to the backend data stream. The implementation of this class, as a thread being independent from the management of the observational data, matches the behaviour expected in MBFITS.

The component in charge of writing data in the MBFITS files exploits some additional classes, closely related to the MBFITS format:

- MBFitsManager
- MBFitsWriter
- MBFitsWriterTable

7.6. MBFitsManager

In the design phase we decided to implement the hierarchical version of the MBFITS format, as described in the reference manual from version 1.6. With this implementation choice, each table is stored in a separate file and we need to write an additional table, named GROUPING, that lists the references to, and other information on, the various files related to a single scan.
Some of the methods implemented in the *MBFitsManager* class handle the process of creating, writing and closing these files. Other methods have been created to realize an interface to the *EngineThread* class to handle the start and the conclusion of a scan, the beginning and the end of each subscan and the recording of observational data and of the parameters associated to them.
Another method of this class is invoked by the *CCollectorThread* class to record the monitor data.

7.7. MBFitsWriter

The low-level operations for recording information in the MBFITS files and tables are managed through the library *CCFits*. It permits to create FITS files, to add tables to them, to manage columns and to assign the values to keywords and to binary data.
In the *MBFitsWriter* class we implemented an interface to the *CCFits* library in order to simplify the operations of reading and writing the values. These methods have been developed to be abstract with respect to the data type to be read and to be stored, so that they can be handled in a homogeneous manner. For instance, a certain method can lead to the storage of single or multiple values, according to the system setup, without changing the parameters list.
With reference to this last aspect, the implementation of this class greatly simplifies the management of the different configurations that are possible for a scan observation, since it is not necessary to implement different blocks of code, in the methods of classes *MBFitsManager* and *EngineThread*, to handle different setup configurations.

7.8. MBFitsWriterTable

The abstraction realized by the *MBFitsWriter* class with respect to the CCFits library has been developed focusing on the library methods that handle data reading and storing operations, both on single and multiple values.
Each instance of the *MBFitsWriterTable* class represents a FITS table, created and managed by the instance of the *MBFitsManager* class, so that the code realizes an abstraction of tables as objects of the code. Most of the operations activated by the *EngineThread* to store setup information or observation data require to operate on almost all the headers or the binary sections of FITS tables. As a consequence, creating an abstraction of these tables as instances in the code immediately translates into a simpler way to handle these objects.

8. Sequence of interactions in data acquisition and storage

Figures from 4 to 10 illustrate the sequence of the most important interactions that occur between instances of the classes that are involved in the process of recording observational data, and the related information, in MBFITS format.
We have previously described that the *EngineThread* class manages the activation of the *MBFitsManager* class when configuration information or observational data are available, delegating their management to this class for storing them in the MBFITS files. These interactions also determine the creation and saving of FITS tables that constitute the file structure of MBFITS, on the key points in time defined in the reference manual MBFITS:

- **Scan**
  A scan is the lowest level object normally used by an observer. It is a sequence of one or more subscans that share a single goal: for instance pointing and focus scans, cross scans or map scans involve a pattern of subscans. Whether OTF (On-The-Fly) maps mosaicing observations are considered a single scan or a series of scans is rather a matter of how the user would like to define it. In our implementation each map is considered a scan.

- **Subscan**
  A subscan is the minimal amount of data taking that can be commanded at the script language level, the script language being a format and a set of telescope instructions to be interpreted by the Telescope Control Software.
  It is highly desirable for the subscan to be a simple enough element, so that the script language may be used to define (at the staff-member/expert level) the content of scans as a means to develop and debug new observing modes.

These terms define frames which reflect into the organization of tables in the MBFITS format: SCAN and FEBE tables store information about the scan as defined by the observer, DATAPAR and ARRAYDATA are created and store data for each subscan.
In the previous version of ESCS the schedules could specify only parameters at the subscan level: for each subscan all data were stored in a single FITS file and the information to connect different subscans with a common goal was left to the observer initiative.
With the MBFITS format this information is required to be stored in the SCAN and FEBE tables. The list of all the tables composing a single scan is stored, for the hierarchical implementation, in the GROUPING table.

9. New format for the schedules

The new data/info arrangement required to implement an updated format for the schedules and to manage some more information in the *ScheduleExecutor*. We defined a new format for the schedule: after an introductory header, the first schedule line is used to specify information common to the whole scan, while the following lines are dedicated to the individual subscans.
In the development of this new format for the schedules, we preserved the possibility for the user to decide, scan by scan, which format (FITS or MBFITS) should be used as data output.

The information required by the MBFITS format to be stored in the SCAN table is relevant only to MBFITS, or to be more precise, to the data reduction software that will be applied to the MBFITS files. We decided to modify the schedule format by adding an attachment file that is not parsed by the *ScheduleExecutor* and will be used only by the writer component. This decision allowed us to implement a very flexible solution, that will be adapted in the future if new formats are required for additional writer components, while keeping the schedule format as simple as possible.

The result we obtained was twofold:

- the files composing the schedule are essentially identical to those used in previous versions of ESCS for all the specified information, almost exclusively the ones functional to the observations. As a consequence, the implementation of the *ScheduleExecutor* and all the related code required only minor updates
- values that do not modify the antenna setup or the pointing in the course of the observation and are required by a new storage format can be specified in the attached file
- the same need may arise in the course of the observation by components with particular implementations. The attached file will be therefore useful not only for the MBFITS format, but also for other formats, or to integrate new components in ESCS/Nuraghe that might need additional information.

9.1. Attachment data

Here we list the keywords to be included in the attached file, when the MBFITS format is chosen for data storage. They are divided into groups according to their position which, in turn, depends on the keyword level (scan or scubscan).

Groups of keywords that should appear in the main section of the attached file:

- Description of the scan
  The data reduction software should not calculate positions based on this description; it must instead rely on the actually observed positions provided in the DATAPAR table
- Description of the basis and the native reference systems and of the relative coordinates
- Description of the projections

Groups of keywords that could appear in the subsections of the attached file:

- Description of the native reference system for the observation of bodies in motion: if the value of MOVEFRAM in group 1 is "true", these values have to be set for each subscan and the corresponding values of the first group must not to be considered.

Here follows the list of the keywords, grouped as indicated above.

1. Description of the scan:

- **SCANTYPE**
  Scan astronomical type
- **SCANMODE**
  Mapping mode
- **SCANGEOM**
  Scan geometry
- **SCANDIR**
  Scan direction
- **SCANLINE**
  Number of lines in a scan
- **SCANRPTS**
  Number of repetitions for each scan line
- **SCANLEN**
  Line length
- **SCANXVEL**
  Tracking rate along line
- **SCANTIME**
  Time for one line
- **SCANXSPC**
  Step along scanning line between samples
- **SCANYSPC**
  Step between scan/raster lines
- **SCANSKEW**
  Offset in scan direction between lines
- **SCANPAR1**
  Spare scan parameter
- **SCANPAR2**
  Spare scan parameter
- **CROCYCLE**
  CAL / REF / ON loop string
- **ZIGZAG**
  Is the scan performed in zigzag mode?
- **MOVEFRAM**
  True if tracking a moving frame
- **SWTCHMOD**
  Type of switched observation

2. Description of the basis and the native reference systems and of the relative coordinates:

- **CTYPE**
  Basis system type : LONGITUDE / LATITUDE
  Accepted values: RA/DEC, GLON/GLAT, AZ/EL
- **CRVAL1**
  Native frame zero in basis system (longitude)
- **CRVAL2**
  Native frame zero in basis system (latitude)
- **CTYPEN**
  Native system type: LONGITUDE / LATITUDE
  Accepted values: RA/DEC, GLON/GLAT, AZ/EL
- **SCANROT**
  Rotation of the user frame meridian w.r.t. the basis frame meridian, measured positive E of N
- **WCSNAME**
  Human-readable description of the basis and user native coordinate systems. For rotated (descriptive) user frames WCSNAME is 'descrip' followed by the basis frame description. Where the user frame is not rotated then use 'absolut' followed by the basis frame description. For moving body observations use 'Moving body coordinates'.

3. Description of the projections:

- **BASISPROJECTION**
  Projection for basis frame: LONGITUDE / LATITUDE
  Accepted values: SFL/SFL
- **NATIVEPROJECTION**
  Projection for native frame: LONGITUDE / LATITUDE
  Accepted values: SFL/SFL

4. Description of the native reference system for the observation of bodies in motion:

- **CTYPEN**
  Native system type : LONGITUDE / LATITUDE
- **MCRVAL1**
  (Moving frame) Native frame zero in basis system (longitude)
- **MCRVAL2**
  (Moving frame) Native frame zero in basis system (longitude)
- **MSCANROT**
  (Moving frame) Rotation of the user frame meridian w.r.t. the basis frame meridian, measured positive E-N

10.Data storage process – execution flow

The storage operations of data and information during each observation are started and stopped through the interactions between the *EngineThread* instance and the *MBFitsManager* object.

The first step in this process is the definition of the path for the directory that will contain all the files of the scan, as shown in Figure 4.
The beginning and the end of a scan, respectively, determine the invocation of methods "*startScan*" and "*endScan*", the start and end of each subscan are handled through the methods "*startSubScan*" and "*endSubScan*".

The transmission of the observational data is managed by the method "*processData*" of the class *EngineThread* which, in turn, invokes the methods "*integrationParameters*" and "integration" on the instance of the class *MBFitsManager*.

The monitor data are collected through methods of the instance of *CCollectorThread* and managed by the method "*processData*" of the same class, which sends data to the *MBFitsManager* instance through the "*monitor*" method.

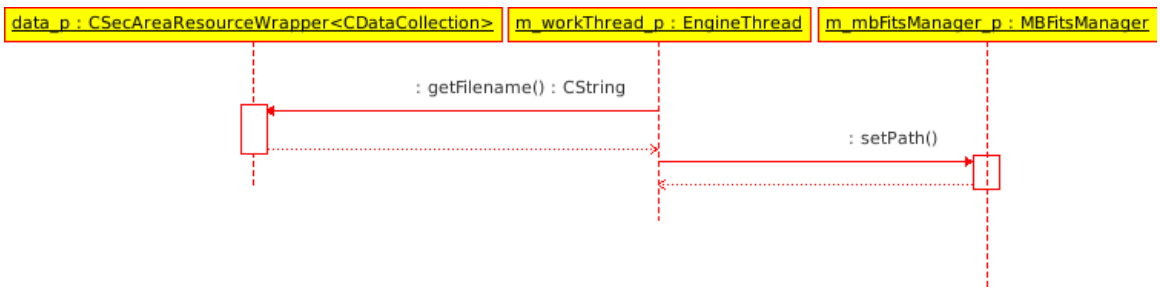

Figure 4 - definition of the path for the directory that will contain all the files of the scan.

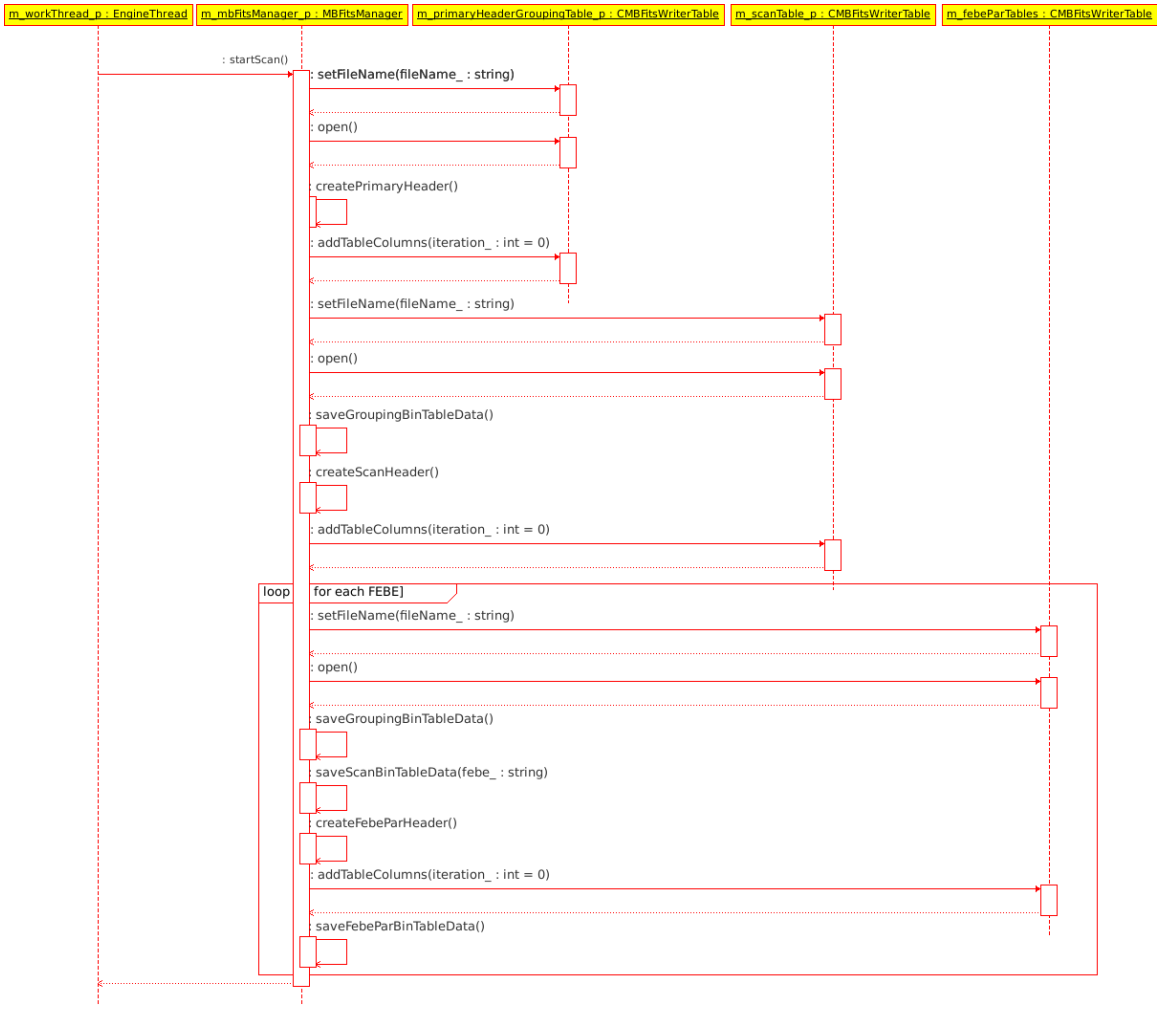

Figure 5 - interactions between the EngineThread and the MBFitsManager instances

Figure 5 shows the interactions between the *EngineThread* and the *MBFitsManager* instances and the sequence of methods that are executed in the starting phase of each scan.

The first step in this sequence is setting the name for the file where the GROUPING table is written. This file is then opened and the initialization of the header values and of the structure of the binary table section takes place.
Analogous operations are executed on the instance of *CMBFitsWriterTable* that will handle the SCAN table and the file to contain it. We also add some information about this file to the GROUPING table, as described in the MBFITS reference manual for the hierarchic implementation.
For each frontend-backend pair we create a new file, setting and initializing a FEBEPAR table to be stored in each of these files. We add the parameters describing this frontend-backend configuration to the binary table section of the SCAN table.
We use one instance of the *MBFitsWriterTable* class for each table we need to create for the MBFITS format, so the code of the methods is specialized on the basis of the particular table owned and managed by each instance. The methods defined in the *MBFitsManager* have similar names for analogous duties to be accomplished on different instances of the *MBFitsWriterTable* class.

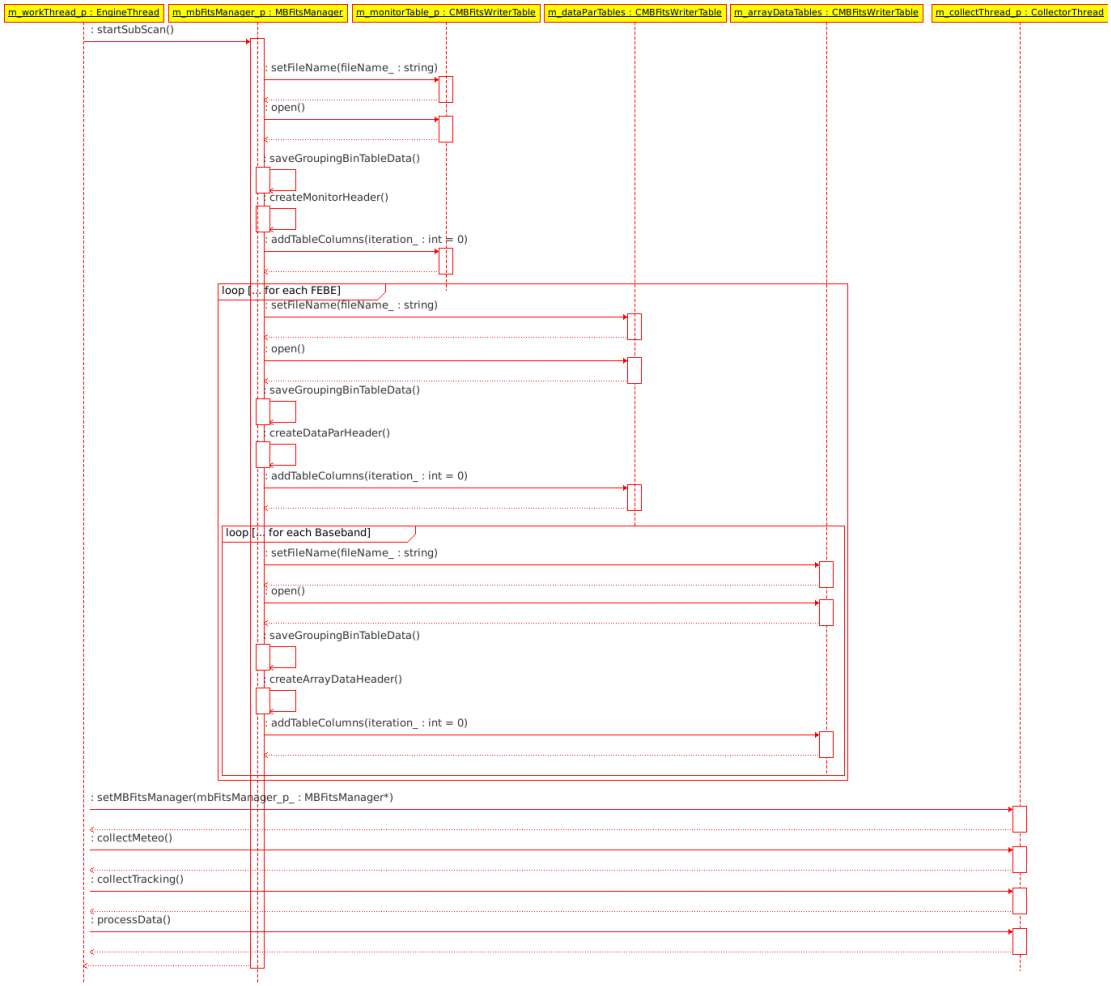

Figure 6 - A subscan start activates the execution of some methods from EngineThread to MBFitsManager

The start of each subscan activates the execution of some methods from the *EngineThread* instance to the *MBFitsManager* object, as shown in Figure 6.
These methods are required to create and initialize the tables in the MBFITS format to store all the acquired data, and the information to describe a single subscan.

As a next step we create and initialize the file to contain the MONITOR table: this file will be created in a new subdirectory where all the files relative to the same subscan will be stored.
For each frontend-backend pair we create a file to save the DATAPAR table: we set the filename, open the file, initialize the header with the required keywords and values and we configure the structure of the binary table section. We also add the information of this file to the main GROUPING table.
Inside the same loop, iterating over a nested loop for all the configured basebands, we also create and initialize new files to store the ARRAYDATA table(s), one file for each baseband.

The last step in the starting phase of each subscan is to measure and save the related environmental information. We currently collect some meteo and tracking information, through the *CollectorThread* instance, and we store them into the MONITOR table created for the subscan.

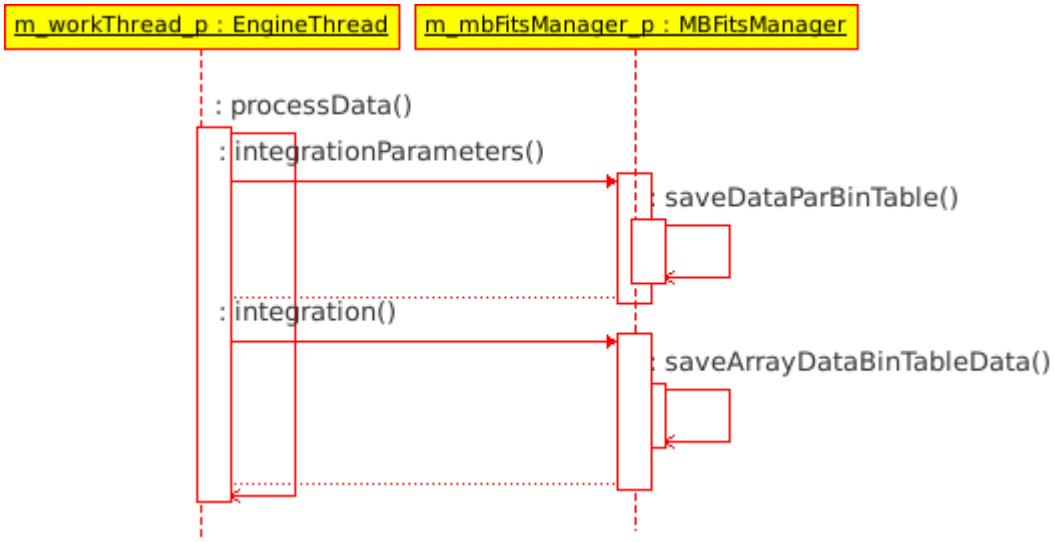

Figure 7 - high level methods invoked during the main loop.
The EngineThread instance invokes the processData method on itself.

Figure 7 shows the high level methods invoked during the main loop of the observation.
The *EngineThread* instance invokes the "*processData*" method on itself.

If new data are available to be stored, *the EngineThread* instance invokes the "*integrationParameters*" method of the *MBFitsManager* instance, in order to save the information related to the antenna pointing into the DATAPAR table.
Then the "*integration*" method of the same object is invoked to store the observation data into the ARRAYDATA table.

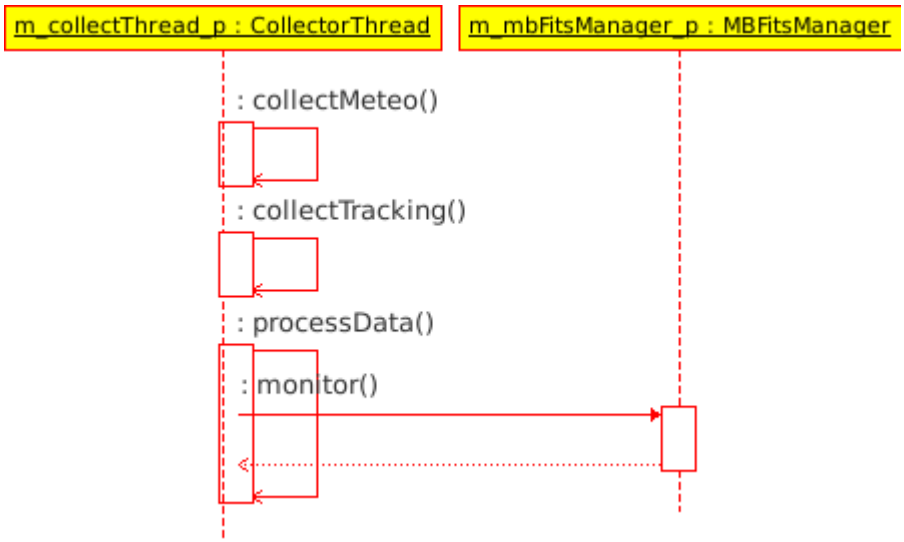

Figure 8 – collecting meteo and tracking data for the MONITOR table.

The *CollectorThread* is executed as an independent thread and the core operations of its loop are activated only by the start of a subscan. They are ended by the closing event of the same subscan.

If the subscan is in execution, every time we enter the loop we collect new meteo and tracking information and store them, through the *MBFitsManager* instance, into the MONITOR table created for this subscan, as shown in Figure 8.

Figure 9 shows the methods invoked for the "*stopSubScan*" event: we get the meteo and the tracking information and store them into the MONITOR table, then we disconnect the *CollectorThread* from the *MBFitsManager* instance to stop the recording of these data.

As a last step in the subscan storage procedure, we update the DATAPAR and the SCAN table to save the correct completeness status of the operations and we close the subscan-related files, looping over all the frontend-backend pairs and configured basebands to close all files that were created as the subscan started.

When the scan is completed we need to close the main files created for it: for each frontend-backend pair it is necessary to close the corresponding FEBEPAR table file, then the SCAN and the GROUPING table files are closed.

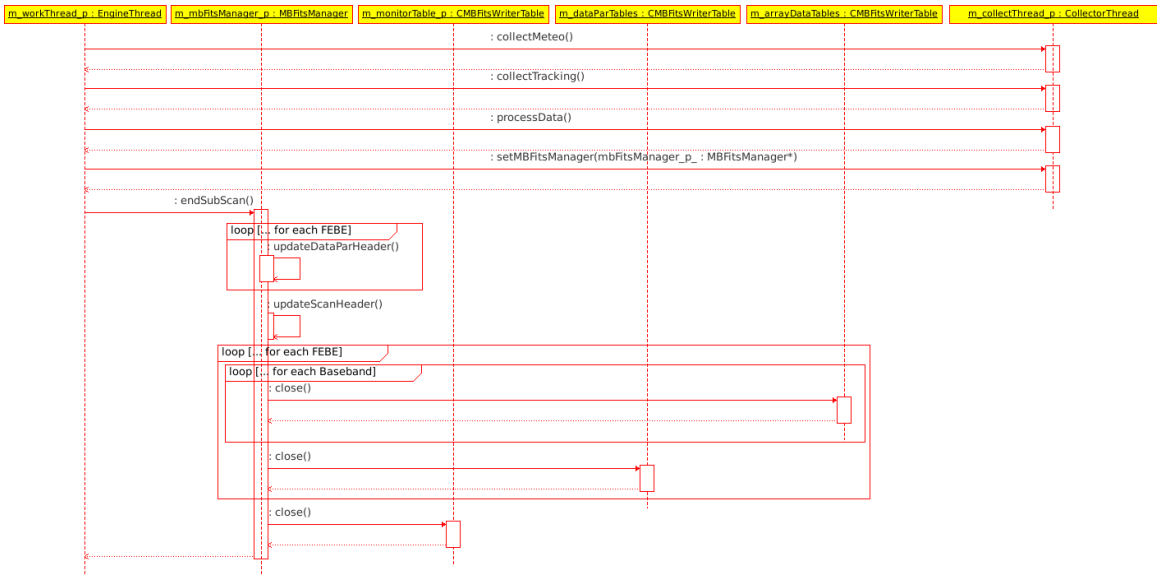

Figure 9 – methods invoked for the "stopSubScan" event.

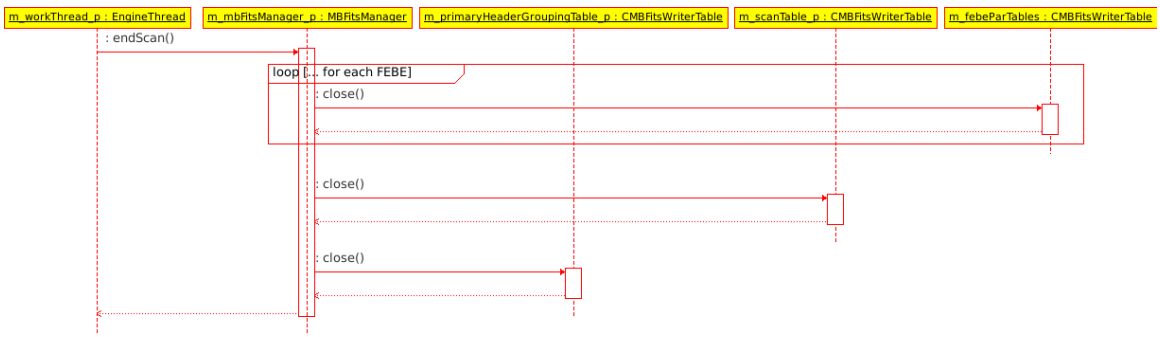

Figure 10 – closing the files.

11.Implementation of the interfaces

We previously showed that the component-container model *MBFitsWriter* declares an interface derived from Management::DataReceiver, and this has been realized through the implementation of the class *MBFitsWriterImpl*.
In the ACS structure, the inheritance of a component from an interface involves the declaration of an XML schema and its definition as a table stored in the CDB. The derivation from the *DataReceiver* interface involves the inheritance of properties listed and described below:

- **fileName**
  Definition of the filename that is currently written
- **projectName**
  name of the project currently running
- **observer**
  name of the observer
- **scanIdentifier**
  identifer of the currently running scan
- **deviceID**
  identifer of the device currently used as primary
- **scanAxis**
  it stores the information about the axis along which the subscan is taking place; since one axis at a time is allowed, the antenna movement has the precedence over the ServoMinor. When the telescope is not scanning or it is simply tracking, in that case "MNG_NO_AXIS" is reported.
- **dataX**
  istantaneous X data
- **dataY**
  istantaneous Y data
- **arrayDataX**
  all X data from the beginning of the data acquisition
- **arrayDataY**
  all Y data from the beginning of the data acquisition
- **status**
  general status of the subsystem
- **recv_protocols**
  type of the protocol, address and port for receiving data

Here follow the time-related parameters for the execution of threads to acquire observational data – and other information – from the components of the ESCS/Nuraghe system:

- **WorkingThreadTime**
  sleep time of the working thread (microseconds), this is the thread that saves the data into the file
- **WorkingThreadTimeSlice**
  time slice of the thread (microseconds): the thread must complete one iteration within that time

- **CollectorThreadTime**
  sleep time of the collector thread (microseconds), this is the thread that collects the complementary information to be stored in the file
- **RepetitionCacheTime**
  cache time (microseconds) for logging repetition filter
- **RepetitionExpireTime**
  expire time (microseconds) for logging repetition filter
- **MeteoParameterDutyCycle**
  gap between two weather station parameters refreshes (microseconds)
- **TrackingFlagDutyCycle**
  gap for the tracking flag refresh (microseconds)

Paths of the interfaces for the components of the ESCS/Nuraghe system used by the component-container model *MBFitsWriter*:

- **AntennaBossInterface**
  tinterface of the component that leads the Antenna subsystem
- **ObservatoryInterface**
  tinterface of component that stores observatory information
- **ReceiversBossInterface**
  interface of component that leads the receivers subsystem
- **SchedulerInterface**
  interface of the component that leads the system and carries out the observation
- **MeteoInstance**
  instance name of the component that samples the weather station data

Some information stored in MBFITS files concern the configuration of the acquisition system of the observational data, such as the time system adopted, or the parameters that depend on the setup of the antenna chosen by the observer.
We created new schemas and added two new tables to the CDB to store these values. Here is the list and the description of these parameters:

TimeData

- **timeSys**
  time system
- **tai2utc**
  TAI - UTC time correction
- **et2utc**
  Ephemeris Time - UTC time correction
- **gps2tai**
  GPS - TAI time correction

AntennaParameters

- **band**
  identifier of the band
- **apertureEfficency**
  aperture efficiency
- **beamEfficency**
  beam efficiency
- **forwardEfficency**
  forward efficiency
- **HPBW**
  Half-Power Beam Width
  During the observations the value of HPBW is computed from other components of the ESCS/Nuraghe system. These online values are stored in the MBFITS files.
- **antennaGain**
  antenna Gain
- **calibrationTemperature_LCP**
  calibrationTemperature_RCP
  calibration Temperatures for Left/Right Circular Polarization
- **dsbImageRatio**
  double Side Band Image Ratio
- **gainPolynomParameters_a**
  gainPolynomParameters_b
  gainPolynomParameters_c
  coefficients of the gain polynomial

12.Pointing model description parameters

The MBFITS format requires to record information on the antenna pointing, as it might be needed during the data analysis phase.
The corrections to the pointing can be divided into different contributions:

- antenna
- subreflector and receiver-dependent static terms
- dynamic pointing
- focus corrections from observations of pointing sources during the observations
- focus/elevation interplay
- refraction.

For the APEX antenna the pointing corrections are dealt with in two stages: the telescope control system (TICS) handles the refraction correction, the dynamic antenna pointing correction, and the receiver terms; and the antenna pointing computer deals internally with the static pointing, the dynamic focus correction, and the focus/elevation interplay. This reflects in the MBFITS format by storing the pointing coefficients in two groups of keywords.

The static pointing coefficients for the APEX antenna, for which the MBFITS format has been created at first, follow the 7-coefficient model described by Mangum (2001), which follows the Stumpff (1972) model plus an extra flexure term, which behaves in the same way as receiver offsets at the Nasmyth focus. Following ALMA developments, this has been extended to include higher order sine and cosine terms and this is reflected in the SCAN header keywords from v.1.55.

PointingCoefficients

- **ia**
  azimuth encoder zero offset
  -P1 in the 7-coefficient model described by Mangum (2001)
- **ie**
  collimation error of the electromagnetic axis
  P7 in the 7-coefficient model described by Mangum (2001)
- **hasa**
  azimuth correction, function of sin(Az)
- **haca**
  azimuth correction, function of cos(Az)
- **hese**
  gravitational flexure parallel to optical axis plus horizontal receiver offset at Nasmyth focus
- **hece**
  gravitational flexure perpendicular to optical axis plus vertical receiver offset at Nasmyth focus
  P8 in the 7-coefficient model described by Mangum (2001)
- **hesa**
  elevation correction, function of sin(Az)
- **hasa2**
  azimuth correction, function of sin(2Az)
- **haca2**
  azimuth correction, function of cos(2Az)
- **hesa2**
  elevation correction, function of sin(2Az)
- **heca2**
  elevation correction, function of cos(2Az)
- **haca3**
  azimuth correction, function of cos(3Az)
- **heca3**
  elevation correction, function of cos(3Az)
- **hesa3**
  elevation correction, function of sin(3Az)
- **npae**
  collimation of the axes / non-perpendicularity between mount azimuth and elevation axes
  -P3 in the 7-coefficient model described by Mangum (2001)
- **ca**
  collimation error of the electromagnetic axis
  -P2 in the 7-coefficient model described by Mangum (2001)
- **an**
  azimuth axis offset / misalignment north-south / zenith shift
  -P5 in the 7-coefficient model described by Mangum (2001)

- **aw**
  azimuth offset / misalignment east-west / zenith shift
  -P4 in the 7-coefficient model described by Mangum (2001)
- **hece2**
  elevation correction, function of cos(2El)
- **hece6**
  elevation correction, function of cos(6El)
- **hesa4**
  elevation correction, function of sin(4Az)
- **hesa5**
  elevation correction, function of sin(5Az)
- **hsca**
  horizontal correction, function of cos(Az)
- **hsca2**
  horizontal correction, function of cos(2Az)
- **hssa3**
  horizontal correction, function of sin(3Az)
- **hsca5**
  horizontal correction, function of cos(5Az)
- **nrx**
  horizontal displacement of Nasmyth receiver
- **nry**
  vertical displacement of Nasmyth receiver
- **hysa**
  pointing coefficient for Azimuth Hysteresis
- **hyse**
  pointing coefficient for Elevation Hysteresis
- **setLinX**
  focus X linear zero position
- **setLinY**
  focus Y linear zero position
- **setLinZ**
  focus Z linear zero position
- **setRotX**
  focus X rotational zero position
- **setRotY**
  focus Y rotational zero position
- **setRotZ**
  focus Z rotational zero position
- **moveFoc**
  harmonic oscillation of focus?
- **focAmp**
  harmonic oscillation focus amplitude
- **focFreq**
  harmonic oscillation focus frequency
- **focPhase**
  harmonic oscillation focus phase

We list also the parameters used to store the values of the coefficients for the corrections of the pointing that depend from the receiver.
Using values from the MBFITS files, each of the following parameter must be added to the corresponding coefficient stored in the table SCAN; the correspondence is

given by the name of the keyword to which, for the coefficients due to the receiver, a "rx" suffix is added.

PointingCoefficientsReceiver

- **iarx**
- **ierx**
- **hasarx**
- **hacarx**
- **heserx**
- **hecerx**
- **hesarx**
- **hasa2rx**
- **haca2rx**
- **hesa2rx**
- **heca2rx**
- **haca3rx**
- **heca3rx**
- **hesa3rx**
- **npaerx**
- **carx**
- **anrx**
- **awrx**
- **hece2rx**
- **hece6rx**
- **hesa4rx**
- **hesa5rx**
- **hscarx**
- **hsca2rx**
- **hssa3rx**
- **hsca5rx**
- **nrxrx**
- **nryrx**

13.Summary