IRAM Memo 2013-2

CLASSIC Data Container

S. Bardeau[1], V. Piétu[1], J. Pety[1,2]

1. IRAM (Grenoble)
2. LERMA, Observatoire de Paris

October, 3^{rd} 2013
Version 1.0

Abstract

The **CLASS/CLIC** Data Format are digital formats used to describe single-dish/interferometric radio-astronomy data. They can be described in two layers: 1) a **CLASSIC** Data Container, which is generic enough to store many kind of data, typically several *observations* which gather observational parameters with actual data, and 2) the **CLASS/CLIC** Data Format itself, which make a particular use of the **CLASSIC** Data Container.

The size of the datasets produced by the IRAM instruments experience a tremendeous increase (because of multi-beam receivers, wide bandwidth receivers, spectrometers with thousands of channels, and/or new observing mode like the interferometric on-the-fly). This implied that the **CLASS/CLIC** Data Format were reaching common limits in the size of data which could be stored. To solve these issues, the **CLASSIC** Data Container standard was revised. This documents aims to describe the new standard. A companion document describes the **GILDAS** library which implements this standard and which is now used by **CLASS** and **CLIC**.

Related documents: The **CLASSIC** Library, IRAM memo 2013-3

Contents

We describe here the **CLASSIC** Data Container standard Version 2. Its historical version (Version 1) and the difference between the two versions are detailed in the section A. Only the **CLASSIC** Data Container standard is detailed here; the **GILDAS** implementation of this standard (e.g. reading and writing) is described in an other memo about the *CLASSIC Library*. We will name *application* the **GILDAS** programs (e.g., **CLASS**, **CLIC**, ...) that use the **CLASSIC** Data Container standard.

1 Overview

Figure 1 tries to reproduce at best the structure of a **CLASSIC** Data Container file. This can be summarized as follows:
1 File = 1 File Descriptor + 0 or more Extensions,

- File descriptor: contains file information (system type, record length, addresses of extension among others).
- Extension = 1 Extension Index + 1 or more File Entries,
  - Extension Index: for each entry of the extension, it contains a few criteria characterizing an entry (to allow quick entry search) and the address of the entry
  - Entry = 1 Entry Descriptor + 1 Observation,
    - Entry Descriptor: contains parameters describing the entry (e.g. number of sections and their addresses, address and length of data).
    - Observation = 0 or 1 Observation Header + 0 or 1 Data array
      - Observation Header = 0 or more Sections
      - Data array: the data themselves

The extension index serves to index some important information about the file entries. For convenience, we also define the concept of *File Index*, which is the (virtual) collection of all the indexes of all the extensions in the file. The **CLASSIC** Data Container enforces the following rules:

1. the File Descriptor is always found in the first record in the file.
2. the various elements involved in the data container are found thanks to the addresses provided by their parent elements. Any other mean or assumption is not guaranteed to work.
3. the File Entries have an entry number which is their rank in the File Index.

2 Detailed description

A **CLASSIC** file is a Fortran, Direct-Access, Unformatted (binary) file satisfying the so-called **CLASSIC** Data Container. Such files are usually read and written by a Fortran library and accessed *record* by *record* of equal length. A **CLASSIC** file contains only (in the following order):

1. a *File Descriptor*,
2. zero or more *Extensions*.

2.1 The File Descriptor

Table 1 summarises the contents of the File Descriptor. It always spreads (entirely and only) over the first record of the file. It describes how the file contents are stored. It is composed of 3 groups of parameters:

- The first group describes the parameters tunable by the application, and which will affect the whole file.

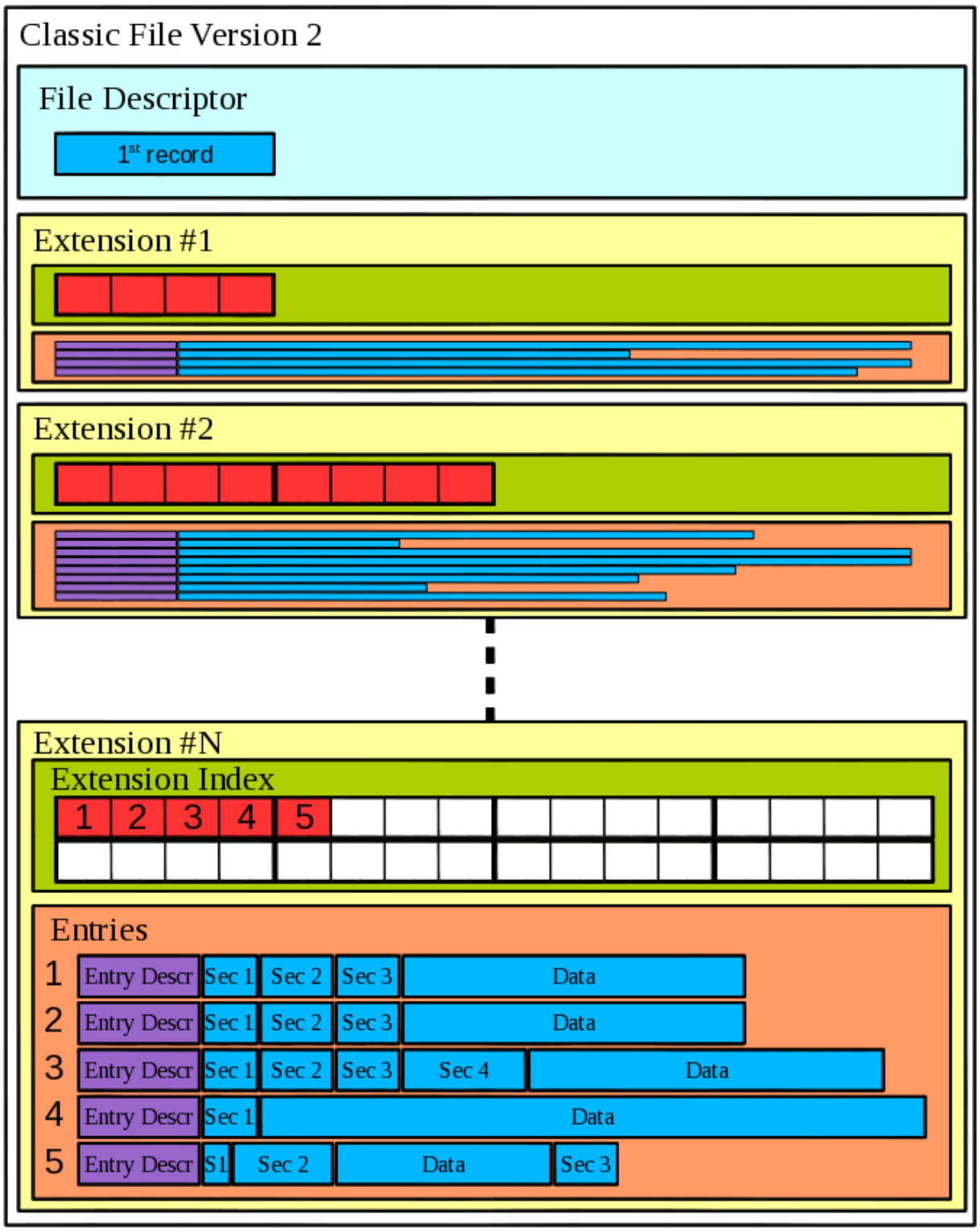

Figure 1: **CLASSIC** Data Container diagram. In this example, the file has N extensions. Extensions use an exponential growth mode (1, 2, ... 8 records devoted to the extension indexes). The last extension is incomplete and contains only 5 file entries, while there is room for 32 in the extension index. The file entries have different number of sections, and sections and data have variable lengths.

Table 1: File Descriptor Version 2

Position	Parameter	Fortran Kind	Purpose	Unit
1	`code`	Character*4	File code	-
2	`reclen`	Integer*4	Record length	words
3	`kind`	Integer*4	File kind	-
4	`vind`	Integer*4	Index version	-
5	`lind`	Integer*4	Index length	words
6	`flags`	Integer*4	Bit flags. #1: single or multiple, #2-32: provision (0-filled)	-
7:8	`xnext`	Integer*8	Next available entry number	-
9:10	`nextrec`	Integer*8	Next record which contains free space	record
11	`nextword`	Integer*4	Next free word in this record	word
12	`lex1`	Integer*4	Length of first extension index	entries
13	`nex`	Integer*4	Number of extensions	-
14	`gex`	Integer*4	Extension growth rule	-
15:`reclen`	`aex(:)`	Integer*8	Array of extension addresses	record

Table 2: **CLASSIC** Data Container encoding of the system and kind. Most of the recent computers are little endian (IEEE) machines. The (very) old PDP-11 files (with Integer*2 words) are recognized but not supported anymore (XX means 2 non-blank characters). Note that in V2 files, the single/multiple attribute is not encoded anymore in this code.

Kind	IEEE	EEEI	VAX	PDP-11
V1 multiple	'1A␣␣'	'1B␣␣'	'1␣␣␣'	'1␣XX'
V1 single	'9A␣␣'	'9B␣␣'	'9␣␣␣'	N/A
V2	'2A␣␣'	'2B␣␣'	'2␣␣␣'	N/A

1. the first 4 bytes of the file are a `code` identifying the files. They also provide the bit encoding system, as detailed in Table 2.
2. the next word (`reclen`) is the record length of this Fortran direct-access file. It is stored in 4-bytes word units[1]. There is no restrictions for `reclen` other than the Fortran ones, except that at least one extension address should fit, *i.e.* `reclen` must be at least 16 words. It can be a non-power of 2, it can be odd, but programmer should keep in mind file access efficiency.
3. the `kind` of the file is stored next (e.g. **CLASS**, **CLIC**, or any other application). This helps the reading application to detect whether it opens a file which it is able to read. In order to avoid conflicts, the possible values are NOT let free to the applications (see Table 3).

[1] A *4-bytes word* is defined here as a block of 4 consecutive bytes. It is the basic unit for all the kind of variables involved here, *e.g.* a Fortran Integer*4 uses 1 word and a Fortran Character*12 uses 3 words.

Table 3: **CLASSIC** Data Container encoding of the owner (`kind` parameter in the File Descriptor). These values are NOT let free to the calling application to avoid conflicts.

Owner	Identifier
CLASS	1
CLIC	2
MRTCAL	3

4. the next 2 words are the Index Version and Length (see section 2.2.1). The Index is a program-specific object, and it is likely to evolve through time. Note that there is no restriction on the Index Length (it can be a non-power of 2, or odd).
5. the next word is used as 32 bit-flags, describing some boolean attributes of the file. Currently only the first[2] bit is used to indicate if the observation numbering is single or multiple. The other 31 bits are provisioned and set to 0, indicating the historical status of some future attributes.

- The 2nd group indicates where the next free memory positions are available.

6. `xnext` provides the next free entry number. File entries are numbered continuously from 1 to `xnext-1`.
7. `nextrec` identifies the record at the end of the file where space is still available to append new data.
8. `nextword` indicates where this space starts in the `nextrec` record.

- The last group describes how the size of extensions increases and the extension addresses.

9. `lex1` is the length of the first extension, in number of entries. This length can vary from one file to another, and can be considered as a scaling factor for the extensions.
10. `nex` is the number of extensions currently in use in the file. It can increase or not when an entry is appended to the file.
11. `gex` describes the extension growth (see below).
12. finally `aex(:)` is an array providing the starting record number for each extension index. Since they are stored as Fortran Integer*8 values, 2 words are needed for each address. Given the fact that the `aex(:)` array fills the 1st record up to its end, it is limited to $nex = (reclen - 14)/2$ elements, which is the maximum number of extensions (`mex`) in the file.

The extension can either all have the same size or exponentially increase their size. Both cases can be summarized as

$$lex_E(iex) = lex1 \times m^{iex-1} \tag{1}$$

where $lex_E(iex)$ is the size of the `iex`-th extension index in number of file entries. The total number of file entries which can be stored in a file with `nex` extensions is then:

$$N_{entry} = lex1 \times (m^0 + m^1 + m^2 + ... + m^{nex-1}) \tag{2}$$

If the mantissa m is 1, all the extensions have the same size ($N_{entry} = lex1 \times nex$). If $m > 1$, the growth is exponential ($N_{entry} = lex1 \times (2^{nex} - 1)$ if $m = 2$). Only equal size extensions were used in the version 1 of the **CLASSIC** Data Container. The new exponential growth solves two problems: 1) the file size will remain small when it contains only a few file entries (no pre-allocation of a large index), and 2) there is virtually no more limit on the number of entries the file can hold[3].

If m is non-integer, floating point computations should be involved. In order to leave no room for round-off problems or ambiguity, the following rules shall be enforced:

1. $m \geq 1$.
2. m is limited to 1 significant digit after the coma.
3. the number of records devoted to each extension index must be rounded to the lowest upper-or-equal integer value large enough to store the $lex_E(iex)$ entry indexes:

$$lex_R(iex) = \text{ceiling}\left(\frac{lex_E(iex) \times lind}{reclen}\right) \tag{3}$$

[2] *first* in the sense of the Fortran bit model, *i.e.* processor-independent.

[3] In practice, the number of file entries will be limited to $2^{63} - 1$, the largest integer number coded as a IEEE Integer*8, *e.g.* $mex = 63$ if $m = 2$

Given these rules, the extensions growth can easily be implemented with integer multiplications and divisions. To simplify the application of rule 2, m is stored as an integer value in the `gex` element in the File Descriptor, multiplied by a factor 10:

$$gex = \text{nint}(10 \times m) \tag{4}$$

This means e.g. that equal-size extensions use $gex = 10$ and a 2-based exponential growth uses $gex = 20$.

2.2 The Extensions

Each extension `iex` contains a set of entries, up to $lex_E(iex) = lex1 \times \left(\frac{gex}{10}\right)^{iex-1}$ entries, according to the equations 1 and 4. It is divided in 2 main parts:

1. the *Extension Index*,
2. the *File Entries*.

When an extension is full (*i.e.* the number of entries has reached its limit), new file entries are added to a new extension.

2.2.1 The Extension Index

Each entry is basically described by a few elements in its associated *Entry Index*. Its length is given by the *Index Length* element in the File Descriptor (`lind`). The interpretation of the bytes of the *Entry Index* is delegated to the application, with one restriction: the application must provide a way to find the entry in the file. This is achieved thanks to the 2 first elements in the entry index, which must give the entry address in the file, *i.e.* its starting record (absolute position in the file, coded as a Fortran Integer*8) and word (relative from the beginning of this record, coded as a Fortran Integer*4). Since the interpretation of most of the *Entry Index* is delegated to the application, the **CLASSIC** Data Container provides an Index Version parameter (`vind`). The application can use its own versioning with the following restriction: the Index Version must be greater than 1 in File Version 2[4]. The couple `vind + lind` can be used as desired by the application to update and ensure backward support of all the revisions of the Entry Index.

The Extension Index is the collection of the entry indexes of the file entries gathered in the extension. It always starts at the beginning of a record. The `lind` words are repeated contiguously, in the limit of $lex_E(iex)$ entry indexes. If a non-integer number of indexes fit in one record ($reclen/lind \notin \mathbb{N}$), entry indexes can overlap two records. The last bytes of the last record in the extension index may be left empty if the last entry index does not reach the end of the record. This mathematically translates into a `ceiling` statement in Eq. 3. Space is always provisioned (the $lex_R(iex)$ records) for the $lex_E(iex)$ entries description, even though the associated file entries are not yet present.

2.2.2 The File Entries

The data written in each entry is very variable, *e.g.* spectra may have very different number of channels. As a consequence, the starting address of an entry cannot be computed from the file entry number and is stored in the Entry Index. An entry does not necessarily start and end on record boundaries. It is divided into

1. the *Entry Descriptor*,
2. the *Observation* itself.

The Entry Descriptor The Entry Descriptor gives a technical description (version of the observation, memory layout, ...) of the associated observation in the Entry (see Table 4). Except the `code` identifier, all these elements can be different from one entry to an other:

1. the `code` is a unique identifier used to check the file consistency. It is always set to the 4 characters '2␣␣␣' (number 2 followed by 3 blanks).
2. the `version` can be used by the application to keep track of the evolutions of the content of the sections and/or data array.
3. `nsec` describes the number of sections actually written in the current descriptor.
4. `nword` is the actual number of words of the whole entry, this descriptor included. It is used to prevent overflow at reading/writing time.
5. `adata` is the data address, in words from the beginning of the descriptor.
6. `ldata` is the data length, in words (4-byte words whatever the kind and precision of data actually stored).
7. `xnum` is the entry number, available here for self-consistency of the file (*i.e.* the entry being N^{th} in the file index must have `xnum` $= N$).
8. `seciden` is an array of `nsec` elements providing the identifiers (Integer*4 codes, private to the application) of the sections present in the observation.
9. `secleng` is an array of `nsec` elements providing the length of each of these sections.
10. `secaddr` is an array of `nsec` elements providing the addresses of each of these sections, in words from the beginning of the entry.

`msec` is the maximum number of sections which can be written in a given entry (while `nsec` is the actual number of sections which are written in this entry). `msec` is let to the choice of the application. It is not stored in the descriptor, but makes sense at the time each new section is written and the descriptor is filled accordingly. If the application writes less `nsec` than `msec`, the remaining space in the descriptor is left empty. This can lead to non-negligible byte losses (see Section 3). As `nsec`, `msec` may vary from entry to entry in the same file.

The Observation The science measurements associated to each observation are shared between

- the *Observation Header*. Each Observation Header is divided in several sections, containing scalar values or arrays. They are defined and controled by the calling application. Each type of section is coded with an integer code, known only by the application. It is not required that all sections be present. The space available for the section elements is reserved at first *write* time (`secleng(i)`). One or more *update*s of the section, including on its number of elements, are possible, as long as they keep the length smaller or equal than the reserved space. Enlarging a section during an update is impossible and is then forbidden[5].
- and usually (but not necessarily) *Data*, e.g., a spectrum coded as a single floating point array of intensities.

[4] In other words: Index V1 in File V1, Index V2 or more in File V2. Index V1 can not be used in File V2 since at least the first word (Entry Address) is different.

[5] If enlarging of a section is required, the application or the user can write a new version of the observation in the same file (*multiple* files) or in a new file.

The final user will probably have access only to these information, all other information described in this document remaining hidden. Sections and Data can be mixed in any order. They are not necessarily aligned on the file records. Their address can be retrieved from the Entry Descriptor. The first element always starts after the entry descriptor, *i.e.* at the address $11 + 5 \times msec + 1$, in words from the beginning of the entry. This structure makes observations self-contained: all the associated information is recorded on a few contiguous file records.

3 Storage efficiency

The **CLASSIC** Data Container is defined such as it introduces unused bytes here and there. We paid special attention to reduce this to a negligeable fraction of the file. In particular, the Version 1 of the **CLASSIC** Data Container had an important source of unused bytes, *i.e.* the fact that each entry had to start at the beginning of a record. Depending on the record length in comparison to the entry size, the number of lost words goes from 0 to `reclen`-1. Which can lead to terrible results (e.g. with $1 + \epsilon$ records actually used for 2 allocated, the mathematical limit is 50% of the file unused). This has been addressed in the Version 2: Each entry can now start anywhere in a record, *i.e.*, right after the previous entry. The key point to keep in mind in this discussion is that all of these unused bytes will still be read (at least by the **CLASSIC** Library implementation because of the record per record access to the file). This means basically that if 10% of the file is unused, 10% of the time is spent for useless I/O.

This section is a exhaustive account of the storage efficiency, in **CLASSIC** Data Container version 2.

1. The only lost bytes in the File Descriptor are associated to the unused extension addresses. Their impact is low.

2. The File Index has several sources of bytes loss. The first is the unused bytes in the Entry Index itself, but this is under the control of the application. Note that this value is proportional to the number of entries in the file.

3. The second element in the Index is the provisioned but unused bytes in the last extension index. This point explains why a file with a few entries can be very large (depending on the `lex1` scaling factor).

4. The third source of loss has been added by the Version 2 format. Since a non-integer number of entry indexes can fit in one record, there can be lost words at the end of the last record devoted to each extension indexes. However this remains small (at most $lind - 1$ words per extension).

5. In the Entry Descriptor, space for up to `msec` sections has to be pre-allocated while only `nsec` will be used. The effect can become important because it is proportional to the number of entries in the file. However this particular point is under the control of the calling application. This application always can dimension the Entry Descriptor to its exact useful size.

4 Basic file reading example

Given the rules and elements described in the previous section, we list here how we can find, for example, the first section and the data array of the $ient^{th}$ entry.

1. check the first word of the file: make sure that file code is one of those described in Table 2, and if you will need to convert all the values from one bit encoding system to another. Hereafter we assume you are reading a version 2 file.

2. the second word provides the record length (in words). Many quantities (*e.g.* addresses) are provided in the **CLASSIC** Data Container in number of records of this length.

3. with the third word, the application can check that it reads a file it understands.

4. read `lind`, the entry index length, in the 5^{th} word. It will be used below.

5. then read `lex1` and `gex` in the 12^{th} and 14^{th} words. They give the number of entries in the first extension, and the rule for the number of entries in the subsequent extensions. Resolving Eq. 2 you should be able to find in which extension the $ient^{th}$ entry is (say iex^{th}), and the relative number of this entry in its extension (say $jent(iex)$).

6. find where this extension should be found. Its address is available in the two words starting at $13 + 2 \times iex$ in the file descriptor.

7. knowing the relative number $jent$ of the entry in its extension iex, you can find in which relative record (from the beginning of the extension index) the entry index is $(((jent - 1) * lind)/reclen)$, and where the starting word in this record $(\mathrm{mod}((jent - 1) * lind, reclen) + 1)$.

8. read the `lind` words of the entry index. The first 3 words provide the entry address: the record (first and second words) and starting word (third word). The others words are application specific.

9. read the given record, at the given position. The first words are the entry descriptor, as detailed in Table 4.

10. then read the number of section (`nsec`) defined in the descriptor. The length of the first section is found at the words 12-13, and its address at $12 + 2 \times nsec$ and next word. Now you can read them, taking also care of possible records overlap.

11. in order to give a signification to these bytes, get the section identifier in the word $12 + 4 \times nsec$ and refer to the documentation of the application which wrote them. Congratulations, you've just read the first section!

12. The data address (`adata`, from the beginning of the entry) and length (`ldata`), both in number of words, are encoded as two Integer*8 from words 6 to 9 in the Entry Descriptor. Just read the `ldata` words starting from `adata`, taking care that the data array can overlap several records. Now you have just read the data array.

Acknowledgments. The authors thank S. Guilloteau, R. Lucas and T. Forveille for useful comments about this document.

Table 4: Entry Descriptor Version 2

Position	Parameter	Fortran Kind	Purpose	Unit
1	`code`	Character*4	Identifier of Entry Descriptors	-
2	`version`	Integer*4	Observation version	-
3	`nsec`	Integer*4	Number of sections	-
4:5	`nword`	Integer*8	Length of this entry (descriptor included)	words
6:7	`adata`	Integer*8	Data address	word
8:9	`ldata`	Integer*8	Data length	words
10:11	`xnum`	Integer*8	Entry number	-
12:...	`seciden(1:nsec)`	Integer*4	Section numbers (identifiers)	code
...:...	`secleng(1:nsec)`	Integer*8	Section lengths	words
...:11+5*nsec	`secaddr(1:nsec)`	Integer*8	Section addresses	word
...:11+5*msec	(unused)		Empty room for up to 'msec' sections	

A The CLASSIC Data Container Version 1

This appendix briefly describes the historical version of the **CLASSIC** Data Container.

Table 5: File Descriptor Version 1

Position	Parameter	Fortran Kind	Purpose
1	`code`	Character*4	File code
2	`next`	Integer*4	Next free record
3	`lex`	Integer*4	Length of first extension (number of entries)
4	`nex`	Integer*4	Number of extensions
5	`xnext`	Integer*4	Next available entry number
6:2*`reclen`	`ex(:)`	Integer*4	Array of extension addresses

Note the following remarks about the File Descriptor in the Version 1:

1. the File Code was starting with the letter '1' or '9' (see Table 2).
2. the record length was always set to 128 words. This proved to be a limiting factor in the I/O accesses, especially for data acquisition at the telescope where the system buffering is disabled for the real-time processing purpose.
3. there was no File Kind. The reading application had to open the observations one by one and accept or reject them depending of their kind, assuming the section providing this information could be read from any program.
4. it was difficult to update the Index (no versioning). Its length was always 32 words, some of them unused.
5. the *single* or *multiple* attribute was encoded in the File Code.
6. since `next` was encoded as a Fortran Integer*4, it was impossible to write a file larger than $2^{31} - 1$ records ($\sim$ 1 TB).
7. file entries always started at the beginning of the `next` available record, possibly leaving unused space in the previous record.
8. all extensions had the same length (`lex`). This made impossible to have a file at the same time i) dimensioned to accept a large number of entries, and ii) remaining small if storing a few entries (because of the Index provision in each extension).
9. the `xnext` parameter was encoded as a Fortran Integer*4, which limited the maximum number of entries per file to $2^{31} - 1$.
10. the extension addresses were spreading over the 2 first records, which always implied at most 251 extensions.

Regarding the File Index in Version 1 Files, not much is to be said since it is not strictly under the control of the **CLASSIC** Data Container. However, here also the first word of the Index is expected to be the Entry address (in absolute records), but encoded as a Fortran Integer*4. Note also that since its length was always 32 words, 4 of them fitted exactly in one record (always 128 words).

In the Entry Descriptor Version 2, only the number of records used by the entry (`nbloc`, redundant with `nword`) and the unused words have been removed in comparison to the Version 1 (see Table 6). In version 1, the various addresses and measurements parameters were coded as Fortran Integer*4, which basically limited the Observation size to $2^{31} - 1$ words ($\sim$ 8 GB).

The Table 7 summarizes the internal limits of the **CLASSIC** Data Container Version 1. All these limits could be gathered into 3 main categories

Table 6: Entry Descriptor Version 1

Position	Parameter	Fortran Kind	Purpose
1	`code`	Character*4	Identifier of Entry Descriptors
2	`nbloc`	Integer*4	Number of records of this entry
3	`nword`	Integer*4	Number of words of this entry
4	(unused)		
5	`adata`	Integer*4	Data address (in words) from the beginning of the entry
6	`ldata`	Integer*4	Data length (in words)
7	(unused)		
8	`nsec`	Integer*4	Number of section
9	`xnum`	Integer*4	Entry number
10:...	`seciden(1:nsec)`	Integer*4	Section lengths (in words)
...:...	`secleng(1:nsec)`	Integer*4	Section addresses (in words)
...:9+3*nsec	`secaddr(1:nsec)`	Integer*4	Section numbers (identifiers)
...:9+3*msec	(unused)		Empty room for up to 'msec' sections

Table 7: **CLASSIC** Data Container Version 1 limits. Most of them are the limit an IEEE Fortran Integer*4 provides ($2^{31} - 1 = 2147483647$).

Parameter	Limit	Unit	Comment
`next`	2147483647	records per file	Maximum 1 TB for the whole file
`lex`	2147483647	entries per extension	
`nex`	251	extensions per file	
`xnext`	2147483647	entries per file	
`ex(:)`	2147483647	record	Absolute extension addresses in file
`nbloc`	2147483647	records per entry	Maximum 1 TB for one observation
`nword`	2147483647	words per entry	Maximum 8 GB for one observation
`adata`	2147483647	word	Relative data address in the entry
`ldata`	2147483647	data values	e.g. maximum number of channels
`secaddr(:)`	2147483647	word	Relative section address in the entry

- the limit on the file size. The more restrictive is the number of records per file: 2147483647 records = 1 TB.
- the limit on the number of observations per file: at most 2147483647.
- the limit on the observation size, ceiled by the physical size of the data: 2147483647 words = 8 GB.

The Version 2 of the **CLASSIC** Data Container standard have addressed all these limitations, by turning the corresponding parameters into Fortran Integer*8 values.