Abstract

CLASS is a state-of-the-art GILDAS\(^1\) software for reduction and analysis of (sub)–millimeter spectroscopic data. Up to now, the CLASS data format could only describe a predetermined number of parameters in the observation headers, these parameters being grouped in 17 sections (e.g. the general, spectroscopic or calibration sections). However, the possibility to add telescope-specific information to the CLASS data format was requested several time. Instead of introducing a per-telescope specific and fixed section, we thus decided to introduce a new flexible section, named User Section. This document describes the functionalities available to the end-users as well as the implementation steps which must be developed by the section owner.

---

\(^1\)http://www.iram.fr/GILDAS
Contents

1 Basic description and functionalities 3

2 Adding a user subsection to an observation 3
   2.1 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   2.2 Detailed API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3 Reading a user section from an observation 6

4 Adding hooks to the Class commands 6
   4.1 User hook for command DUMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      4.1.1 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      4.1.2 API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   4.2 User hook for command SET VARIABLE USER . . . . . . . . . . . . . . . . . . . 9
      4.2.1 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      4.2.2 API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   4.3 API summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

5 Extending the standard CLASS capabilities 11

6 Backward compatibility 13

7 Portability 13

8 Conclusion 13

A Implementation details for CLASS developers 15
1 Basic description and functionalities

A user section is now part of the CLASS data format. It can be added to the observations by any external program which is linked with the CLASS library and which uses the CLASS API.

The user section is composed of a basic owner and data descriptor, and of a data block. The content and length of the data is controlled by the section owner, but for a few restrictions described in this document. The data block is untyped, which means that the standard CLASS program is not able to understand it natively. Only its owner is able to do so with the appropriate decoding routines. On the other hand, the standard CLASS program is able to detect, load and propagate the User Section when possible. CLASS can also find all the observations containing a user section. However, users and programmers should be aware that there are CLASS operations under which the user section loses its meaning (for example AVERAGE'ing). These operations thus do not propagate the user function.

More precisely, the User Section can store several user subsections. Each of these is identified by a unique owner and title pair of values. The CLASS library will properly read and write in the correct subsection as long as the program has declared properly which one it owns. Each subsection has its own version number to enable the evolution of the data format (Note that abuse of the versioning mechanism may slow down operations on this section).

We also provide code hooks which enable external programs to work on the user sections when a few usual CLASS commands are called. These commands are WRITE, GET, DUMP, SET VARIABLE USER.

2 Adding a user subsection to an observation

In order to add a new user subsection to an observation, the external program has first to define a Fortran derived type containing the data. Scalar or 1D-array values of types INTEGER(4), REAL(4) and REAL(8) are supported. Scalar CHARACTER(LEN=*) strings are also supported, but their length must be a multiple of 4. The component names, their number, and their order is defined by the programmer. The derived type used by the external program does not need to have a Fortran sequence statement: the way its elements are ordered in memory is also free.

The external program can add its user section to an observation in 3 steps:

1. tell CLASS which subsection it owns thanks to the subroutine class_user_owner,
2. declare with class_user_toclass the subroutine which knows how to order and transfer to CLASS the user data,
3. finally send the data to CLASS with the subroutine class_user_add (for a new user subsection) or class_user_update (if the subsection already exists).

The 2 first steps can be done only once, while the last has to be repeated each time a new user section has to be added. The transfer subroutine declared at step 2 is mandatory to indicate to CLASS which elements are present in the data and in which order they have to be written.

2.1 Example

The following program adds a new User Section to a CLASS observation. A more realistic program would open an output file and write the observation into it. The data here contains dummy values and names for the example.

```fortran
module mytypes
  type :: owner_title_version
    integer(kind=4) :: datai4
    real(kind=4)  :: datar4
end type
```

3
CLASS User Section 2. Adding a user subsection to an observation

real(kind=8) :: datar8
character(len=4) :: datac4
end type owner_title_version
end module mytypes

program myprog
use class_types
use class_user_interfaces
use mytypes
external :: toclass
type(observation) :: obs
type(owner_title_version) :: mydata
integer(kind=4) :: version
logical :: error
!
! 1) Fill the data
mydata%datai4 = 111
mydata%datar4 = 222.
mydata%datar8 = 333.
mydata%datac4 = 'ABCD'
!
! 2) Tell Class who I am
call class_user_owner('OWNER','TITLE')
!
! 3) Declare my transfer subroutine
call class_user_toclass(toclass)
!
! 4) Fill the User Section in the observation
version = 1
error = .false.
call class_user_add(obs,version,mydata,error)
if (error) stop
!
end program myprog

subroutine toclass(mydata,version,error)
use class_user_interfaces
use mytypes
!---------------------------------------------------------------------
! Transfer and order the input 'mydata' object to the internal Class
! data buffer.
!---------------------------------------------------------------------
type(owner_title_version), intent(in) :: mydata
integer(kind=4), intent(in) :: version ! The version of the data
logical, intent(inout) :: error ! Logical error flag
!
if (version.ne.1) then
  print *, 'TOCLASS: Unsupported data version ',version
  error = .true.
  return
endif
!
call class_user_datatoclass(mydata%datai4)
call class_user_datatoclass(mydata%datar4)
call class_user_datatoclass(mydata%datar8)
call class_user_datatoclass(mydata%datac4)
!
end subroutine toclass

2.2 Detailed API

The following subroutines are part of the Class User Section API. The interface module named class_user_interfaces MUST be used in the calling program or subroutines:

- Declare the owner and title of the User Subsection:

  subroutine class_user_owner(sowner,stitle)
  character(len=*), intent(in) :: sowner ! Section owner
  character(len=*), intent(in) :: stitle ! Section title
  
- Declare the user’s transfer routine

  subroutine class_user_toclass(usertoclass)
  external :: usertoclass ! User’s 'toclass' subroutine
  
- Send a variable to the data buffer in CLASS:

  subroutine class_user_datatoclass(var)
  integer(kind=4), intent(in) :: var
  OR real(kind=4), intent(in) :: var
  OR real(kind=8), intent(in) :: var
  OR character(len=*), intent(in) :: var
  
- Add the User Subsection to an observation:

  subroutine class_user_add(obs,sversion,sdata,error)
  type(observation), intent(inout) :: obs ! Observation
  integer(kind=4), intent(in) :: sversion ! Version number
  type( ), intent(in) :: sdata ! The user data
  logical, intent(inout) :: error ! Logical error flag
  
- Update the User Subsection in the input observation:

  subroutine class_user_update(obs,sversion,sdata,error)
  type(observation), intent(inout) :: obs ! Observation
  integer(kind=4), intent(in) :: sversion ! Version number
  type( ), intent(in) :: sdata ! The user data
  logical, intent(inout) :: error ! Logical error flag
  
Finally, the calling sequence of the user’s transfer routine must be of the following form:

subroutine toclass(data,version,error)
  type( ), intent(in) :: data ! The data to be transferred to Class
  integer(kind=4), intent(in) :: version ! The version of the data
  logical, intent(inout) :: error ! Logical error flag
The name of this subroutine is to be defined by the user. The data type can be of any kind provided by the user. The purpose of this routine is to send the data elements to CLASS in a given order. The advantage of this mechanism is to leave CLASS compute the whole data size and store it as its will. The counterpart is that the order must be then exactly respected in the other parts of the user code, e.g. in the reading subroutine fromclass defined hereafter. Furthermore, the CLASS API does not allow to read only one element in the data.

3 Reading a user section from an observation

When the observation which is read from a file contains a user section, CLASS will load it entirely (i.e. all the subsections it may contain). The data block of each subsection is loaded “as is” in memory. Nothing more is done at read time.

4 Adding hooks to the Class commands

The key point is now: “How users can access the user subsection in the context of CLASS commands?”. Let’s take the command DUMP as first example.

4.1 User hook for command DUMP

The command DUMP displays the content of one or all sections of the R buffer. When CLASS finds a user section to dump, it can display the information it knows about each subsection. What it can not do is guessing what the data block contains.

If the section owner wants to tell CLASS how to dump its subsection, it has to:

1. tell CLASS which subsection it owns with class_user_owner. This is optional if it has been done elsewhere.

2. declare to CLASS with class_user_dump a subroutine which is able to display what is in the data block.

If CLASS tries now to DUMP the user section, it will loop over all the subsections it contains. If one matches the owner+title couple, it will put the data block in a specific place, and then call user’s dump subroutine. This subroutine will then just have to translate back the data block to its own data type.

This last point must be fulfilled by a subroutine exactly symmetric to the transfer routine used at write time, i.e. elements should be reread in the same order they have been written.

4.1.1 Example

If no user hooks have been declared, the output of the command DUMP /SECTION USER will be (with the example shown in the previous section):

USER ------------------------------------------------------
Number of subsections: 1
User Section # 1
   Owner: OWNER
   Title: TITLE
   Version: 1
   Data length: 5
   Data: (can not dump)
**CLASS User Section**

**CLASS** claims it is not able to understand the content of the data buffer in the User Subsection.

The following subroutines show basically what should be done by the section owner to allow **CLASS** to **DUMP** a specific user section:

```fortran
subroutine mydump_init
  !----------------------------------------------------------------------
  ! Preliminary declarations
  !----------------------------------------------------------------------
  ! 1) Tell Class who I am
  call class_user_owner('OWNER','TITLE')
  ! 2) Declare my dumping routine
  call class_user_dump(mydump)
  ! Nothing more: return and wait for Class to ask for my User
  ! Subsection dump.
end subroutine mydump_init

subroutine mydump(version,error)
  use mytypes
  !---------------------------------------------------------------------
  ! Dump to screen my User Subsection
  !---------------------------------------------------------------------
  integer(kind=4), intent(in) :: version ! The version of the data
  logical, intent(inout) :: error ! Logical error flag
  ! Local
  type(owner_title_version) :: mydata
  !
  call fromclass(mydata,version,error) ! Read the Class buffer and fill mydata
  if (error) return

  ! Display to screen
  print *," datai4 = ",mydata%datai4
  print *," datar4 = ",mydata%datar4
  print *," datar8 = ",mydata%datar8
  print *," datac4 = ",mydata%datac4

end subroutine mydump

subroutine fromclass(mydata,version,error)
  use mytypes
  use class_user_interfaces
  !---------------------------------------------------------------------
  ! Transfer the data values from the Class data buffer to the 'mydata'
  ! instance.
  !---------------------------------------------------------------------
  type(owner_title_version), intent(out) :: mydata !
  integer(kind=4), intent(in) :: version ! The version of the data
  logical, intent(inout) :: error ! Logical error flag
```

7
if (version.ne.1) then
   print *, 'FROMCLASS: Unsupported data version ',version
   error = .true.
   return
endif
!
call class_user_classtodata(mydata%datai4)
call class_user_classtodata(mydata%datar4)
call class_user_classtodata(mydata%datar8)
call class_user_classtodata(mydata%datac4)
!
end subroutine fromclass

With these subroutines, CLASS is now able to ask for a translation of the User Subsection. The output of the command DUMP /SECTION USER will be:

```
USER ------------------------------------------------------
Number of subsections: 1
User Section #  1
 Owner: OWNER
 Title: TITLE
 Version: 1
Data length: 5
Data:
datai4 = 111
datar4 = 222.00000
datar8 = 333.000000000000000
datac4 = ABCD
```

4.1.2 API

The following subroutines have to be used to declare user’s dump subroutine:

- Declare the user’s dump routine:

  ```fortran
  subroutine class_user_dump(userdump)
     external :: userdump ! User’s ‘dump’ subroutine
  end subroutine class_user_dump
  ```

- Get a variable from the data buffer in CLASS:

  ```fortran
  subroutine class_user_classtodata(var)
     integer(kind=4), intent(out) :: var
     OR real(kind=4), intent(out) :: var
     OR real(kind=8), intent(out) :: var
     OR character(len=*), intent(out) :: var
  end subroutine class_user_classtodata
  ```

  The calling sequence of the user’s dump routine must be of the following form:

  ```fortran
  subroutine mydump(version,error)
     integer(kind=4), intent(in) :: version ! The version of the data
     logical, intent(inout) :: error  ! Logical error flag
  end subroutine mydump
  ```

  The name of this subroutine is free.
4.2 User hook for command SET VARIABLE USER

If CLASS knows about a User Subsection owner and title when SET VARIABLE USER is invoked, it can instantiate the SIC variables which describe the data of this subsection. This is done if the associated hook has been defined and declared to CLASS. The steps to do so are, as usual:

1. tell CLASS which subsection it owns with class_user_owner. This is optional if it has been done elsewhere.
2. declare to CLASS with class_user_setvar a subroutine which is able to instantiate the SIC variables associated to the User Subsection.

The subroutines class_user_def_inte, class_user_def_real and class_user_def_dble and class_user_def_char will create respectively an integer*4, real*4 and real*8 and character string in R%USER%OWNER% (where OWNER is the owner previously declared). The subroutines order must match the order the data was written in the User Section buffer.

The calling sequence of these subroutines is described in the following sections. The suffix name of the SIC variable is given as first argument, e.g. FOO for R%USER%OWNER%FOO. The arguments ’ndim’ and ’dims’ describe the variable dimensions, but only scalars and 1D-arrays are currently supported (ndim=0 or 1). For the character variables, ’strlen’ is the string length which has to be read from the buffer.

Note that once it has been instantiated, the SIC structure R%USER%OWNER% will be updated each time the data changes (e.g. after a GET NEXT).

4.2.1 Example

When CLASS does not know about the User Subsection, it will not accept to instantiate the SIC structure R%USER%:

LAS90> set variable user
E-SETVAR, No user function set for SET VAR USER
LAS90> exa r%user%
E-EXAMINE, No such variable R%USER%

The following subroutines show basically what should be done by the section owner to allow CLASS to instantiate SIC variables pointing to the User Subsection when the command SET VARIABLE USER is invoked.

subroutine mysetvar_init
  !------------------------------------------------------------------
  ! Preliminary declarations
  !------------------------------------------------------------------
  !
  ! 1) Tell Class who I am
  call class_user_owner(’OWNER’,’TITLE’)
  !
  ! 2) Declare my instantiation routine
  call class_user_setvar(mysetvar)
  !
  ! Nothing more: return and wait for Class to ask for my User
  ! Subsection instantiation.
  !
end subroutine mysetvar_init

subroutine mysetvar(version,error)
  use mytypes

4. ADDING HOOKS TO THE CLASS COMMANDS

With these routines, CLASS is now able to instantiate the SIC variables:

```
MYPROG> set variable user
MYPROG> exa R%USER%
R%USER%      ! Structure GLOBAL
R%USER%OWNER%DATAC4 = ABCD       ! Character*4 GLOBAL RO
R%USER%OWNER%DATAR8 = 333.0000000000000 ! Double GLOBAL RO
R%USER%OWNER%DATAR4 = 222.0000      ! Real GLOBAL RO
R%USER%OWNER%DATAI4 = 111          ! Integer GLOBAL RO
R%USER%OWNER%DATAR8 = 222.0000      ! Real GLOBAL RO
R%USER%OWNER%DATAI4 = 111          ! Integer GLOBAL RO
R%USER%OWNER%DATAC4 = ABCD       ! Character*4 GLOBAL RO
R%USER%OWNER%DATAR8 = 333.0000000000000 ! Double GLOBAL RO
R%USER%OWNER%DATAR4 = 222.0000      ! Real GLOBAL RO
R%USER%OWNER%DATAI4 = 111          ! Integer GLOBAL RO
```

4.2.2 API

The following subroutines have to be used in order to instantiate SIC variables in the structure R%USER%.

- Declare the user's subroutine for variables instantiation:

  ```
  subroutine class_user_setvar(usersetvar)
    external :: usersetvar
  end subroutine class_user_setvar
  ```

- Instantiate a SIC variable in R%USER%OWNER% (numeric types, respectively integer*4, real*4 and real*8). Only scalars (ndim=0) and 1D-arrays (ndim=1) are currently supported.

  ```
  subroutine class_user_def_inte(suffix,ndim,dims,error)
    character(len=*) :: suffix ! Component name
    integer(kind=4), intent(in) :: ndim ! Number of dimensions (0=scalar)
    integer(kind=4), intent(in) :: dims(4) ! Dimensions (unused if scalar)
    logical, intent(inout) :: error ! Logical error flag
  end subroutine class_user_def_inte
  ```

  ```
  subroutine class_user_def_real(suffix,ndim,dims,error)
    character(len=*) :: suffix ! Component name
    integer(kind=4), intent(in) :: ndim ! Number of dimensions (0=scalar)
    integer(kind=4), intent(in) :: dims(4) ! Dimensions (unused if scalar)
    logical, intent(inout) :: error ! Logical error flag
  end subroutine class_user_def_real
  ```
subroutine class_user_def_dble(suffix, ndim, dims, error)
  character(len=*), intent(in) :: suffix  ! Component name
  integer(kind=4), intent(in) :: ndim    ! Number of dimensions (0=scalar)
  integer(kind=4), intent(in) :: dims(4) ! Dimensions (unused if scalar)
  logical, intent(inout) :: error     ! Logical error flag

- Instantiate a SIC variable in R%USER%OWNER% (scalar character strings).

subroutine class_user_def_char(suffix, lstring, error)
  character(len=*), intent(in) :: suffix  ! Component name
  integer(kind=4), intent(in) :: lstring ! String length
  logical, intent(inout) :: error     ! Logical error flag

The calling sequence of the user’s instantiation routine must be of the following form:

subroutine mysetvar(version, error)
  integer(kind=4), intent(in) :: version ! The version of the data
  logical, intent(inout) :: error     ! Logical error flag

The name of this subroutine is free.

4.3 API summary

The table 1 describes the general hooks to be used in order to read or write the User Section. They have to be considered as prerequisites before doing anything with this section in CLASS.

The table 2 describe the hooks to be used when adding a User Section to an observation (e.g. before writing it to the output file).

Finally the table 3 summarizes the subroutines which can be used to declare the hooks to some CLASS commands, plus some subroutines which are useful for some of them.

5 Extending the standard CLASS capabilities

The previous sections describe how to write and read a User Section in/from a CLASS file. This was assumed to be done from an external program linked to be CLASS library. However, it is also possible to do this directly from the standard CLASS executable. The key point is to define all the support subroutines for the User Section in a binary, dynamically loadable library (say libowner.so), linked to the CLASS library. libowner.so must also define a specific entry point and its associated definition subroutines, as described in the document named New package initialization².

From a standard CLASS session, the command SIC\IMPORT can then be used to load the libowner.so library and import the User Section capabilities. Such a session will look like:

LAS90> ! First, Class does not know how to read the User Section:
LAS90> DUMP /SECTION USER
USER ------------------------------------------------------
Number of subsections: 1
User Section #   1
  Owner:     OWNER
  Title:     TITLE
  Version:   1
  Data length:  5
  Data:

²contact gildas@iram.fr for more information
### 5. Extending the Standard class Capabilities

#### Class User Section

**Table 1: General hooks**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class_user_owner</code></td>
<td>declare who is the owner and what is the title of the subsection the hooks can read or write</td>
</tr>
</tbody>
</table>

**Table 2: Writing hooks**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class_user_add</code></td>
<td>add a new user subsection to the input observation</td>
</tr>
<tr>
<td><code>class_user_update</code></td>
<td>update a user subsection which already exists</td>
</tr>
<tr>
<td><code>class_user_toclass</code></td>
<td>declare the subroutine which will write the data to the Class internal buffer</td>
</tr>
<tr>
<td><code>class_user_datatoclass</code></td>
<td>transfer a scalar or 1D-array value to the Class internal buffer (generic subroutine)</td>
</tr>
</tbody>
</table>

**Table 3: Reading hooks**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class_user_dump</code></td>
<td>declare the hook for the command DUMP</td>
</tr>
<tr>
<td><code>class_user_find</code></td>
<td>declare the hook for the command FIND (experimental)</td>
</tr>
<tr>
<td><code>class_user_setvar</code></td>
<td>declare the hook for the command SET VARIABLE USER</td>
</tr>
<tr>
<td><code>class_user_classstodata</code></td>
<td>read a scalar or 1D-array value from the Class internal buffer (generic subroutine)</td>
</tr>
<tr>
<td><code>class_user_def_inte</code></td>
<td>create an integer*4 SIC variable in the structure R%USER%OWNER%</td>
</tr>
<tr>
<td><code>class_user_def_real</code></td>
<td>create a real*4 SIC variable in the structure R%USER%OWNER%</td>
</tr>
<tr>
<td><code>class_user_def_dble</code></td>
<td>create a real*8 SIC variable in the structure R%USER%OWNER%</td>
</tr>
<tr>
<td><code>class_user_def_char</code></td>
<td>create a character string SIC variable in the structure R%USER%OWNER%</td>
</tr>
</tbody>
</table>
6. BACKWARD COMPATIBILITY

6 Backward compatibility

Old versions of CLASS (i.e. older than apr11g) will just ignore the user section, if it exists. This section will be lost in a GET-and-WRITE process.

Starting from apr11g, CLASS is able to detect, load and give basic details about this section. All the user subsections are preserved and written to the output file, under the restrictions exposed in the next section.

7 Portability

The basic behavior of CLASS is that it loads all the user subsections (if any) at read time, and transfer all of them to the output file at write time. However, the data block of bytes is untyped in the CLASS data format, so it will make sense to re-read them if and only if the reading machine has the same endianness (IEEE, EEEI, etc) than the one used at write time. As a consequence the following restrictions apply:

1. at read time, CLASS won’t read the user section if the input file type is not native (i.e. it has not been written under the same architecture we are reading it),

2. at write time, CLASS won’t write the user section if the output file type is not native (i.e. we reopened for output a file first written under another architecture).

However, these restrictions have very low chance to occur, since nowadays most (all?) of the computers have an IEEE architecture.

8 Conclusion

The CLASS developer team provides some flexibility about the CLASS data format through a customiz-able user section mechanism. This user section can be encoded and decoded only by external programs linked with the CLASS library and which follows the CLASS API. This is the responsability of the external programs to ensure the efficiency of their codes. We recommand to observatory who would like
to use this mechanism to contact us (gildas@iram.fr) in order to discuss how to maintain the long term compatibility of their user section.
A Implementation details for CLASS developers

The User Section is added to the CLASS data format. It is defined in the Fortran source as follows:

```fortran
type user  ! Type for all user subsections
  sequence
    integer(kind=4) :: n ! Number of user subsections
    type (user_sub), pointer :: sub(:) ! Array of subsections
  end type

An object of type user is added to the type observation structure in the data format. This user object can contain an arbitrary number of subsections. This allows several user subsections to exist for the same observation.

Then each of these user subsection is described by the following type:

```fortran
 type user_sub  ! Type for each user subsection
  sequence
    character(len=12) :: owner ! Owner of the subsection
    character(len=12) :: title ! Title of the subsection
    integer(kind=4) :: version ! Version of the subsection
    integer(kind=4) :: ndata ! Length of the data(:) array
    integer(kind=4), pointer :: data(:) ! Place holder for information
  end type
```

- A user subsection is identified by a owner and a title. One owner can have several subsections with each a different title. In other words the couple owner+title is considered as a unique identifier for a user subsection.

- The version value is a mean for the owner to handle several versions of the same subsection (e.g. elements have been added in this subsection between two versions).

- The data field is the data itself. It is considered as a block of bytes with no direct meaning for CLASS. Only the owner is able to decode it.

- Since the data is a dynamic object (i.e. size is not known at compilation time), the ndata value describes how many 4-bytes blocks are present in the data.

Those two types are private to CLASS. They do not have to be known by the external program.

Finally, this new User Section is different from the other ones in the CLASS data format since its size is not constant. This implies some modifications in the subroutines. In particular, a whole copy of observation to observation (obsout = obsin) is forbidden: it must be done in a specific way, taking care of allocating the pointers in the user section before the copy. For the same reason, the User Section is not buffered and must be re-read everytime needed on the disk (in other words SET VIRTUAL ON has no effect).