Eurogam project **EDOC123**

Developpement graphique

USING THE EXTERNAL CALLBACK PACKAGE (V1-00)

Edition 1.0
June 1991

L. MICHEL

Centre de Recherches Nucléaires de Strasbourg (SATD)
CNRS-IN2P3 France

Contents

Chapter 1

USER'S GUIDE

1.1 What does the ECP do ?

The user of a graphic interface program very often needs to add specific functions to it, keeping the features of the interface. The problem is that such programs are very large and elaborate, and a modifiaction needs then a good knowledge of its structure and of the graphical langage. They are certainly interesting things but which doesn't correspond necessarily to the preoccupation of any user.

A solution is to give to the graphical application the ability to call external programs. The user will have then just to write his program (or take an existing one), the appplication managing to call it using its graphical features. That's done by the ECP (External Callback Package)

The External Callback Package is a library, based on the Xview toolkit, useful to integrate external routines in a graphic application without compilation.

The external routine is a short program, callable from the shell, getting its arguments directly from the command line (without dialog) and returning its results on *stdout* .

The ECP uses a resource file *(filename.cback)* describing the extern routines to create a the menu bar and its dialog boxes. When a menu item is selected, the corresponding dialog box is opened to set the parameters of that external routine (see Fig 1). It can then be call, by clicking on the *run* button.

The user has just to know the syntax of the resource file (less than 30 words) to integrate his routines. The strings returned by the external routine can be interpreted graphically by a specific function of the graphic application, indicated by the user in the resource file.

1.2 Writting a resource file.

1.2.1 Syntax principles

When starting, the application reads a resource file giving it the useful informations on the external callback to integrate. This file must have the suffix *.cback* . Its prefix must be put (V1-00) at the first position in the arg-list. If the graphic application haven't got resource file, a fatal error is generated.

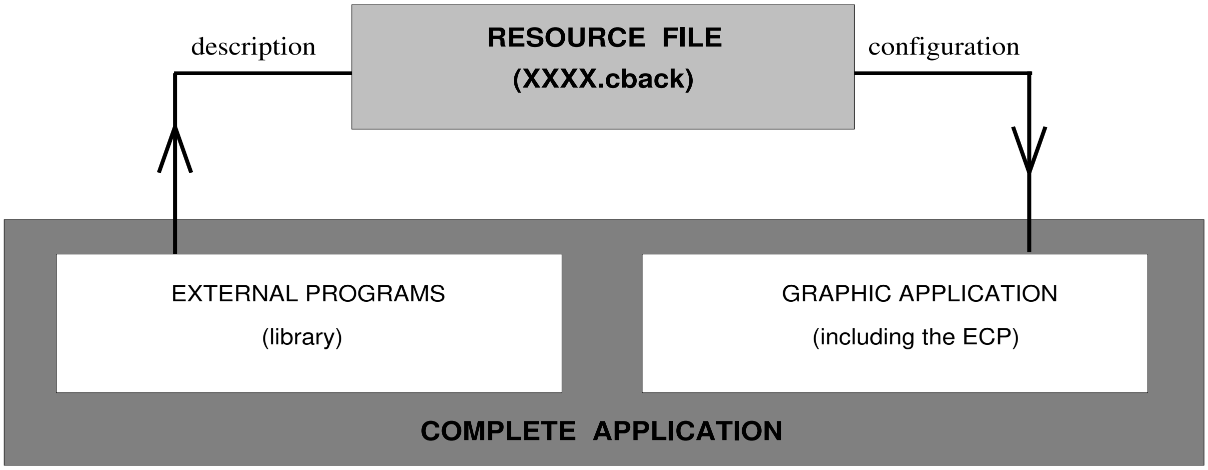


Fig 1: Building a graphic application including external programs.

The syntax of the resource file is very simple. The most useful options are taken by default and the logical links between the objects are implicit. Just the description of each object has to be written in the resource file.

RESOURCE LINE

A resource line can be more long than an file line. The fields can be separated with tabulations spaces or new line characters. The resource line ever has the same structure for all the objects:
<OBJECT IDENTIFIER> <LABEL> <OPTIONS LIST> <LIST>

- *<OBJECT IDENTIFIER>:* Keyword (upper case letters) identifying the type of object. This word marks the begining of a resource line.
- *<LABEL>:* The label of the object must be written between "", otherwise, it will be interperted like an option.
- *<OPTIONS LIST>:* List of keywords (see glossary) defining the options of the object.
- *<LIST>:* List of words between "" or keywords associated with the object.

EXAMPLE: *OBJECT_LIST "spectra" ReadOnly "spectr1" "spectr2" "spectr3"* is a resource line defining a scrolling list, named *spectra* being read only and having 3 items: *spectr1* , *spectr2* and *spectr3* .

COMMENTS

A comment line begins with a #. It's impossible to insert comments into a resource line.

EXAMPLE:
 # this is a comment will be read like a comment.
OBJECT_LIST # this is a comment > will generate an error.

FILE STRUCTURE

The resource fle contains the definitions of the menu bar and of each dialog box.
The menu bar must be defined before the dialog boxes. The order of dialog boxes descriptions is free.

EXAMPLE: structure of a resource file:

```
##############################
# structure of a resource file
##############################

# defining the menus
MENU .. 1st menu definition ....
.....................
.....................
MENU .. last menu definition ....
# defining the dialog boxes
DIALOGUE
.. 1st dialog box definition ....
END_DIALOGUE
.....................
.....................
DIALOGUE
.. last dialog box definition ....
END_DIALOGUE
##############################
# end of the resource file
##############################
```

The ECP will first create the menu bar, and after it'll look for the corresponding dialog box in the resource file. If it finds a menu item without dialog box, or a dialog box without associated menu item, it generates an error.

1.2.2 CREATING THE MENU BAR

The ECP can create a menu bar up to 10 buttons wich will be laid out in the control area (be careful with the size of the control area). Each menu is associated with a button (ECP V1-00 don't work with popup).

The label of an object button will be at once its name on the screen and its ECP identifier.

A menu object can just have one option, *pinable* , which permits to keep the menu displayed on the screen (see the OpenLook GUI).
CAUTION: don't use this option with the ECP V1-00, the selection of an item in a pined menu will cause a fatal error.

The menu definition must begin with the keyword MENU The list associated with a menu represents its items. They must be written between "". The maximum length of that list is unfixed.

The ECP can not associate a menu with a menu item. The menu tree has got just one level.

EXAMPLE: Definition of the menu bar in a *.cback* file. That menu bar has got two buttons, *UNIX* and *OTHER* (All the following examples will show how to realize the UNIX menu).

```
##############################
```

```
# Definition of a menu of command UNIX
##############################
MENU "UNIX"
    "file list" "file copy" "file read" "file archive"
##############################
# Definition of an other menu
##############################
MENU "OTHER"
    "other1" "other2" "other3"
.....................
.....................
##############################
# end of the resource file
##############################
```

1.2.3 Creating a dialog box

The ECP draws the dialog box from the informations of the resource file. An active object corresponds to each argument. Each kind of active object has a specific control area. The dialog box will ever have at least one control area where are the command buttons and the linedit of the comand line (see Fig2).

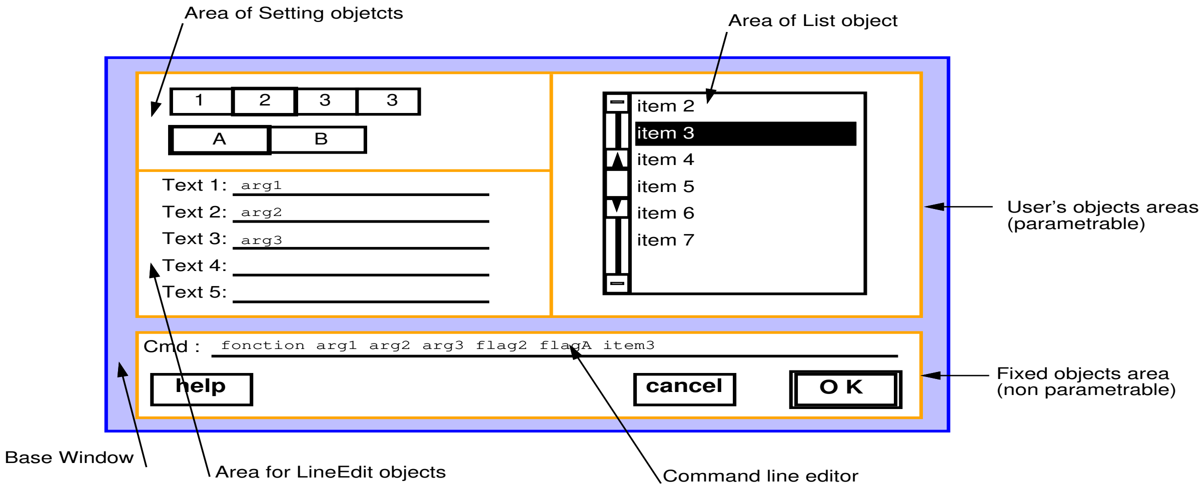


Fig 2: General structure of a dialog boxe.

There are 3 kinds of active objects:

- LINEDITS: useful to edit a string.
- SETTING: useful to select an option.
- LIST: useful to select a string from a list.

Let's build a dialog box corresponding to the *file archive* item (UNIX command "tar") in the menu *UNIX* of the previous example.

The *tar* command needs the name of the device, the name of the files to archive and the options:

- The name of the device will be selected from a list.

- The name of the file will be set from a linedit.
- The options will be selected from a setting.

CREATING THE BLOCK OF A DIALOG BOX DESCRIPTION

The description of the dialog box must begin with *DIALOGUE* followed with *"label"* and terminate with *END_DIALOGUE* .

The label must have this form: *"Menu:Label"*

- The "" means that it's not a keyword but a label.
- The : character separates the 2 fields.
- *Menu* is the label of the menu.
- *Label* is the label of the menu item.

The following resource lines show the syntax of the description block of a dialog box:

```
###############################
# Begining of dialog box description
###############################
DIALOGUE "UNIX:file archive"
.. dialog box definition (see later) ...
END_DIALOGUE
###############################
# end of dialog box description
###############################
```

CREATING THE SETTING OBJECT

The setting object will be used to set the option of the *tar* command (read, write or extract).

Its definition begins with the keyword *OBJECT_SETTING* , followed with its label *"option"* (an object label has up to 8 characters). There are no option in this example.

This setting will have three items: *read* , *write* and *extr.* (an item label has up to 5 characters). Each item ever has the same structure as the label of the dialog box (see previous section). The first fields is the button label and the second fields is the associated argument of the command line: *"button label:arg"* .

EXAMPLE: The argument associated with the *read* option is *-tvf* (see the UNIX command syntax). The item will be then written *"read:-tvf"* . In this way, the argument *-tvf* will be add to the command line when the button *read* of the setting will be selected.

```
###############################
# Begining of the setting definition
###############################
OBJECT_SETTING "option"
    "write:-cvf" "read:-tvf" "extr.:-xvf"
###############################
# end of the setting definition
###############################
```

CREATING THE LINEDIT OBJECT

The linedit object will be used to set the name of the files to archive. It won't have a default value.
Its definition begins with the keyword *OBJECT_LINEDIT* , followed with its label *"file"* . There are no option in this example.

EXAMPLE: The string which will be set in the linedit will be add to the command line.

```
###############################
# Begining of the linedit definition
###############################
OBJECT_LINEDIT "file"
###############################
# end of the linedit definition
###############################
```

CREATING THE LIST OBJECT

The list object will be used to set the name of the archive file. Its definition begins with the keyword *OBJECT_LIST* , followed of its label *"tarfile"* .
On our example, we suppose that the list of tar files is given by the system, so the user can not add or delete a new one. So we will put to the list the *ReadOnly* option. The list of tar files will be the list associated to the list od this object.

EXAMPLE: The string which will be selected in the list will be add to the command line.

```
###############################
# Begining of the list definition
###############################
OBJECT_LIST "tarfile"
    "/dev/fd0" "/dev/rst0" "/dev/rst1"
###############################
# end of the list definition
###############################
```

SETTING THE COMMAND LINE

The definition of the command begins with thew keyword *COMMAND_LINE* followed with a list having the same structure as the command line.
The *tar* command line has this form: *tar option tarfile filename* .

- *tar* is the command name. It never will be edited, so we put it in a constant string: *"tar "* ¿ Notice that the constant string must be terminated with a space to be separated of the next argument.

- *option* is got from the first setting (first definition). It will be symbolized in the command line by this 2 fields keyword: *Setting:1* where the 1st field (*Setting*) indicates the kind of the active object giving this argument, and the 2nd field indicating its definition position.

- *tarfile* is got from the first list. It will be symbolized in the list by this item: *Liste:1* (*Liste* is a keyword)

- *filename* is got from the first linedit. It will be symbolized in the linedit by this item: *Linedit:1* (*Linedit* is a key word)

The command line is made with constant strings and references to defined active objects. The constant strings will ever be written in the command line, the other arguments taking the same value as their associated active objects.

EXAMPLE:Command line of the *tar* command:

```
###############################
# Begining of the command line definition
###############################
COMMAND_LINE "tar " Setting:1 Liste:1 Linedit:1
###############################
# end of the command line definition
###############################
```

SETTING THE GRAPHIC INTERPRETATION ROUTINE

All the graphic applications using the ECP must have a list of routines interpreting the strings returned from the external callbacks. These routines are specific to the application and we won't discuss here about they (see further PROGRAMMER'S GUIDE). This list is given with the application documentation.

Each dialog box must yet be associate with a such routine. This declaration must begin with the keyword *GRAPH_INTER_ROUTINE* followed with the name of this routine, held keyword.

EXAMPLE: We suppose that the application has a graphic routine named *display_string* which displays in a window the strings read from the standard output (callback output).

```
###############################
# Begining of the graphic interpretation routine definition
###############################
GRAPH_INTER_ROUTINE display_string
###############################
# end of the command line definition
###############################
```

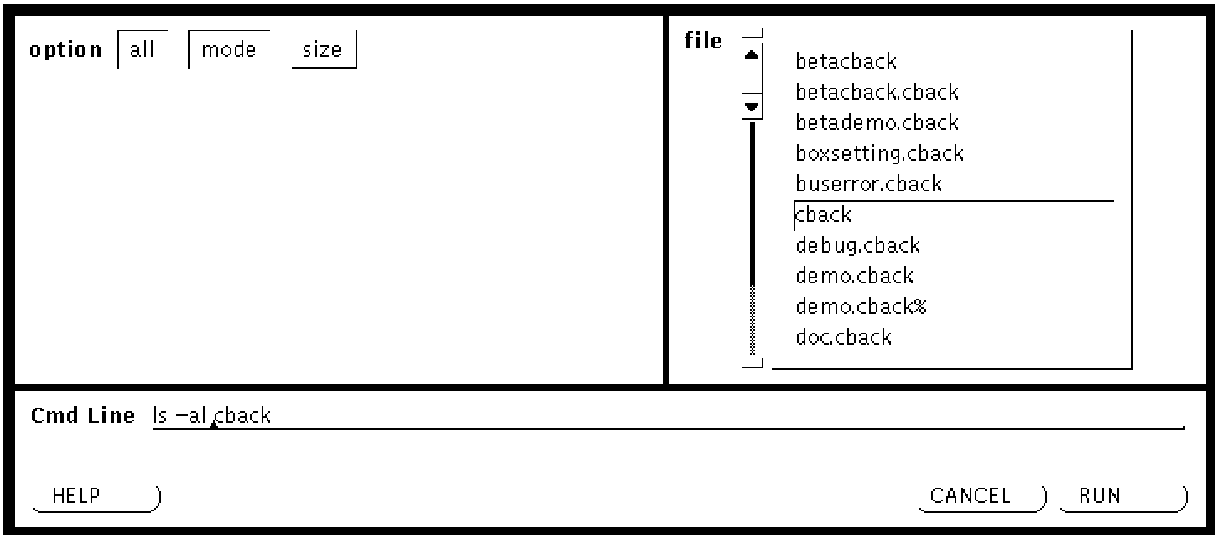
1.2.4 Using the dialog box

THE COMPLETE DEFINITION OF THE DIALOG BOX

The following extract of the resource file shows the complete defintion of the dialog box setting the *tar* command.

```
###############################
# Begining of dialog box description
###############################
DIALOGUE "UNIX:file archive"
    OBJECT_LINEDIT "file"
    OBJECT_SETTING "option"
        "write:-cvf" "read:-tvf" "extr.:-xvf"
    OBJECT_LIST "tarfile"
        "/dev/fd0" "/dev/rst0" "/dev/rst1"
    GRAPH_INTER_ROUTINE display_string
    COMMAND_LINE "tar " Setting:1 Liste:1 Linedit:1
END_DIALOGUE
###############################
# end of dialog box description
###############################
```

When the item *file archive* will be selected in the menu *UNIX* , this dialog box (see fig3) appears. Notice that just one dialog can be open at once.



All the modification in an active object is transmitted to the command line, but the opposite isn't true.

The *tar* will be executed clicking on the *run* button. The dialog box will be closed after the command is terminate.

The *canel* button closes the box without execution of the callback.

The *help* button doesn't work in the version V1-00.

1.3 Digest of the resource file syntax

1.3.1 objects identifier

The following figure shows all the ECP identifier:

objects	ECP identifier	commandline identifier
menus	MENU	(no menu in cmdline)
dialog	DIALOGUE (END_DIALOGUE)	(no dialog in cmdline)
setting	SETTING_OBJECT	Setting
linedit	LINEDIT_OBJECT	Linedit
list	LIST_OBJECT	Liste

Tab 1: ECP identifiers.

The *menu* and *dialog* objects can not be referenced in the command line. The *dialog* object has 2 ECP identifier (begining and end of definition).

1.3.2 options availables

The following figure shows all the ECP options available for each type of object:

	MENU	DIALOGUE	SETTING	LINEDIT	LIST
NonResizable		NI			
NonMovable		NI			
Pinable	NI				
Focused		NI			
NonExclusiveChoice			OK		OK
Default			NI	NI	NI
NumericOnly				OK	
XChanelClick				OK	
YChanelClick				OK	
ZChanelClick				OK	
WChanelClick				OK	
ReadOnly				OK	OK
SystemDefaultList					OK
NonEditableCommandline		NI			
NoSeparator			OK	OK	OK

NI: option no implemented OK: option available for this object

Tab 2: Available options.

OPTION *NonExclusivChoice*

This option permits to select several items in a *setting* or in a *list* . In this way, the argument associated with that object becomes a list instead a single string.

OPTION *NumericOnly*

The *NumericOnly linedit* accepts only characters with numerical value. Its value can be increased or decreased with small buttons. This option is useful to ensure to have numeric argument.

OPTION *XChanelClick*

With these options X(Y,Z or W)ChanelClick permits to set a value in an *linedit* object without editing. A dialog box can have up to 4 (x, y, z and w) ring buffers of linedits identifiers, having each one a function setting the current object with a value, and setting the current position to the next object. These function must be call from the graphic application. So, this option can not be used with an application providing not it (see the documentation).

This option is very usefull to get value directly from another window (such as a chanel number). The value sent to the *linedit* is set by the application. The ECP just receives it, and puts it in the current object of the ring buffer.

OPTION *ReadOnly*

The *readonly* objects (*linedit* or *list*) vorbiden to editing its values. It's very useful for a *list* set by a system call (see the *SystemDefaulList* option) or for a linedit using a *ChanelClick* option.

OPTION *SystemDefaulList*

This option permits to set a *list* with the results of a UNIX command (or any program). That command must be written as first item of the associated list, the other item being ignored. The UNIX command is executed once when the application is starting and the *list* kepp these value while all its execution.

EXAMPLE: Creating a list filled the *ls *.c* command:

```
###############################
# Begining of the list definition
###############################
OBJECT_LIST "files" SystemDefaulList
    "ls *.c"
###############################
# end of the list definition
###############################
```

The *list* af that dialog box will ever be set with the list of the C files.

OPTION *NoSeparator*

This option supresses the space betwenn 2 arguments. It's useful to merge a pathname or several options.

If The *NoSeparator* option is used with the "NonExclusiveChoice", all the elements of the list will sticked up, as well as the following argument. To avoid that, it's necessary to put a constant string " " between the both identifiers. in the command line.

EXAMPLE: The folowing example defines a dialog box interfacing the UNIX command *ls* . The *setting* sets the options which must be stick on. The available files are put in the *list* with the option *SystemDefaultList*

```
###############################
# Begining of dialog box description
###############################
DIALOGUE "UNIX:list file"
    OBJECT_LIST "file" ReadOnly SystemDefaultList
        "ls"
    OBJECT_SETTING "option" NonExclusiveChoice NoSeparator
        "a:all" "mode:l" "size:s"
    GRAPH_INTER_ROUTINE display_string
    COMMAND_LINE "ls " "-" Setting:1 " " Liste:1
END_DIALOGUE
###############################
# end of dialog box description
###############################
```

The constant string, " ", between *Setting:1* and *Liste:1* in the *command line* definition separates the option group and the name of the file to list.

1.3.3 Geometric limits of the ECP objects

The following tab shows the limits of the definition of the ECP objects. These values are verified when reading the resource file, but it's advised to respect these values writting it.

	MENU	DIALOGUE	SETTING	LINEDIT	LIST
max number	10	–	3	5	1
label length	32	8	8	8	8
item number	undef.	no item	5	no item	256
item label length	32	no item	5	no item	24

Tab 3: Geometric limits of the ECP objects.

1.4 Glossary

- **AllOption :** (option - all objects) Sets all the options (Should not use this)
- **COMMAND_LINE :** (object identifier) Identifies the structure of the command line.
- **Default :** (option - unimplemented) Sets the object's default value when opening the dialog box .
- **DIALOGUE :** (object identifier) Marks the begining of a dialog box definition.
- **END_DIALOGUE :** (object identifier) Marks the end of a dialog box definition.
- **Focused :** (option - unimplemented) Focused all the X Events to the opened dialog box.
- **GRAPH_INTER_ROUTINE :** (object identifier) Identifies the name of the graphic interpretation routine.
- **Linedit :** (argument identifier) Identifies a command line argument associated with a linedit object.
- **Liste :** (argument identifier) Identifies a command line argument associated with a list object.
- **MENU :** (object identifier) Marks the begining of a menu definition.
- **NonEditableCommandline :** (option - DIALOGUE) The command line is read-only.
- **NonExclusiveChoice :** (option - SETTING, LIST) Several items can be selected.
- **NonMovable :** (option - unimplemented) The dialog box can't be moved
- **NonResizable :** (option - unimplemented) The dialog box can't be resized.
- **NoSeparator :** (option - LINEDIT, LIST, SETTING) The argument won't be separated of the following.
- **NumericOnly :** (option - LINEDIT) The linedit will accept only the numeric values.
- **OBJECT_LINEDIT :** (object identifier) Identifies the definition of a linedit object.
- **OBJECT_LIST :** (object identifier) Identifies the definition of a list object.
- **OBJECT_SETTING :** (object identifier) Identifies the definition of a setting object.
- **Pinable :** (option - unimplemented) The menu can be pined on the screen.
- **ReadOnly :** (option - LIST, SETTING) The object is read only.

- **Setting :** (argument identifier) Identifies a command line argument associated with a setting object.
- **SystemDefaultList :** (option - LIST) The list will be initialized with a Unix command (for the version V1-00, the initialization is done once while the application starts).
- **WChanelClick :** (option - LINEDIT, LIST) The linedit can be set from an X-Event external of the dialog box, it will be stored in the W ring buffer.
- **XChanelClick :** (option - LINEDIT, LIST) The linedit can be set from an X-Event external of the dialog box, it will be stored in the X ring buffer.
- **YChanelClick :** (option - LINEDIT, LIST) The linedit can be set from an X-Event external of the dialog box, it will be stored in the Y ring buffer.
- **ZChanelClick :** (option - LINEDIT, LIST) The linedit can be set from an X-Event external of the dialog box, it will be stored in the Z ring buffer.

Chapter 2

PROGRAMER'S GUIDE

2.1 What does the ECP do?

Integrate the ECP into a Xview application give it the ability to run any external program. The ECP includes a parser which reads a resource file an create some Xview objects from it, and an event manager which is called everytime a event is detected on an ECP object. These both controls don't disturb the execution of the rest of the graphic program.

2.2 Integrating the ECP in your Xview application.

2.2.1 Integrating the ECP calls in your C code

Structure of the main loop

The ECP must be called before starting the main loop of the Xview application. It creates then the Xview objects.

The main function in a classical Xview application has three steps:

```
/********************************************************
*         classical xview main function
*********************************************************/
main() {
     /* parsing the command line */
     xv_init( ..................);
     /* init. the xview objects */
     object_initialize(.........);
     /* Starting the main loop */
     xv_main_loop(.........);
}
```

Using the ECP adds 2 steps to that main function:

```
/********************************************************
*         Xview main function integrating the ECP
*********************************************************/
main() {
     /* parsing the command line */
     xv_init( ..................);
     /* init. the xview objects */
     object_initialize(.........);
```

```
        /* creating the list of graph. interpreation routine */
        set_routine_list();
        /* creating the dialog boxes from the .cback file */
        set_callback(filename,base_window,ctrl_area,x0,y0);
        /* Starting the main loop */
        xv_main_loop(.........);
}
```

The *set_routin_list()* function has to be written by the programer to give to the ECP a list of routines available to process the strings returned from the callbacks (see further).

The *set_callback(...)* function is a ECP callback which parses the resource file and creates the dialog boxes.

Setting the graphic interpretation routines list

The program using the ECP must have at least one graphic interpretation routine which will receive the strings returned from the external callback and will process them (see further how to write it):

```
/*******************************************************
* declaration af a graphic interpretation routine
*******************************************************/
void gph_routine(char* command_line){
...............................
...............................
...............................
}
```

The parameter is always the command line edited in the dialog box. These routines must be put in a list internal to the ECP. They are referenced by a string (their names) and a pointer in a structure defined in the *cback_anal.h* file:

```
typedef struct { char* nom; /* symbolic name of the function*/
                 void (*ptr)(char*); /* pointer to the function */
               } fct_id;
```

The string will be use to identify the function invoqued in the resource file after the *GRAPH_INTER_ROUTINE* identifier. The pointer will be use to set this routine like the notify procedure of the *run* button of a dialog box.

The ECP function *void add_routine_list(int*, int)* must be called to append a routine to the list. The first parameter is a pointer to the structure *fct_id* , the second is the length of this array (2 integers). The following C code extract shows how to create the routines list:

```
/*******************************************************
* Prototypes of the graphic interpretation routines
*******************************************************/
void gph_routine1(char*);
void gph_routine2(char*);
void gph_routine3(char*);
........................
........................
/*******************************************************
*         Xview main function integrating the ECP
*******************************************************/
main() {
        ........................
        ........................
```

```
    /* creating the list of graph. interpreation routine */
    set_routine_list();
    ........................
    xv_main_loop(.........);
    }
........................
........................
........................
void set_routine_list(){
    union { fct_id;
            int tab[2]; } id;
    /* adding the first routine */
    id.id.nom = "routine1";
    id.id.ptr = routine1;
    add_routine_list(id.tab,2);
    /* adding the second routine */
    id.id.nom = "routine2";
    id.id.ptr = routine2;
    add_routine_list(id.tab,2);
    /* adding the third routine */
    id.id.nom = "routine3";
    id.id.ptr = routine3;
    add_routine_list(id.tab,2);
    }
........................
........................
........................
```

Setting the external callbacks

The application using the ECP must be associated with a resource file (*filename.cback*) defining the callback. The file is read by the ECP library which creates the Xview objects. This is processed by the *void set_callback(char* filename,Xv_opaque base,Xv_opaque control,int x0,int y0)* function.

filename is the prefix of the resource file. The suffix, *.cback,* is added by the ECP. If *set_callback* doesn't find this file, a fatal error is generated and the program exits.

base is the Xview identifier of the base window of the application. The dialog boxes will be daughter of that window.

control is the Xview identifier of the control area where will be drawn the menu bar. It's horizontal, and can have up to 10 buttons. The ECP V1-00 accepts just one menu bar.

x0 and *y0* are the coordonates of upper left corner of the left button of the menu bar.

2.3 Writting the graphic interpretation routine

These routines (see above) must be written by the programer (in a next version, a library of standard routine will be available). They must first run the external callback, and then process the strings returned.

Running the external routine

The easiest way to run an external routine is to use the *system(char* cmdline)* system call. That is useful if the callback returns nothing.

EXAMPLE: A simple graphic interpretation routine.

```
/*********************************************************
* simple graphic interpretation routines
*********************************************************/
void gph_routine1(char* str){
     system(str);
     };
```

str is the command line edited in the dialog box. The routine returns at the end of the callback but does nothing else.

If the external callback returns strings, the *grph_routine* must read them with a pipe. That can be done with the *popen* standard call.

EXAMPLE: Piping an external callback:

```
/*********************************************************
* a graphic interpretation routines using a pipe
*********************************************************/
void gph_routine1(char* str){
          FILE* pp;
          buffer[MAX_LINE_LENGTH];
     /* creating the pipe with the external callback */
     if( (pp = popen(str,"r")) == NULL ){
          fprintf(stderr,"cannot open the pipe");
          exit(1);
          }
     /* reading the pipe */
     while( fgets(buffer,MAX_LINE_LENGTH,pp) ) {
          ................................
          /* processing the read strings */
          ................................
          }
     return;
     }
```

Note that the call of these routines are done by the ECP from its list. The programer never needs to read anything in a dialog box, or to process an event occured in it.

Processing the returned strings

The processing of the returned strings depends completely on the application, but we are going to see 1 example corresponding to a current case.

EXAMPLE: Writting the result into a textedit panel. (The textedit panel is an Xview object). Using the textedit panel, the programer must be careful to avoid buffer overflows (see the Xview documentation) !

```
/*********************************************************
* a graphic interpretation routines using a pipe
*********************************************************/
void gph_routine1(char* str){
          FILE* pp;
          buffe[MAX_LINE_LENGTH];
     /* creating the pipe with the external callback */
     if( (pp = popen(str,"r")) == NULL ){
          fprintf(stderr,"canot open the pipe");
          exit(1);
          }
     /* reading the pipe */
     while( fgets(buffer,MAX_LINE_LENGTH,pp) ) {
          /* inserting a new line */
          textsw_insert(texwin_id,buffer,strlen(buffer));
          }
```

```
return;
}
```

textwin_id is the Xview identifier (*Xv_opaque*) of the textedit subwindow.

2.4 Using the ring buffers

Each dialog box can have up to 4 ring buffers (X, Y, Z and W) used to set a value into an active objects directly from your application. That value must be an integer. Just the *linedit NumericOnly* objects can be set with the ring buffers in V1-00. The ring buffers are built from the objects having the option X_ChanelClick (Y_, Z_, or W_).

The ECP function *chanelclick_x(Event* event, int value) (_y, _z or _w)* sets the value *value* into the current object of the buffer, and then puts the following object like current object (see Fig 4). *event* is a pointer to the Xview event which induces that call (*event* isn't used in V1-00).

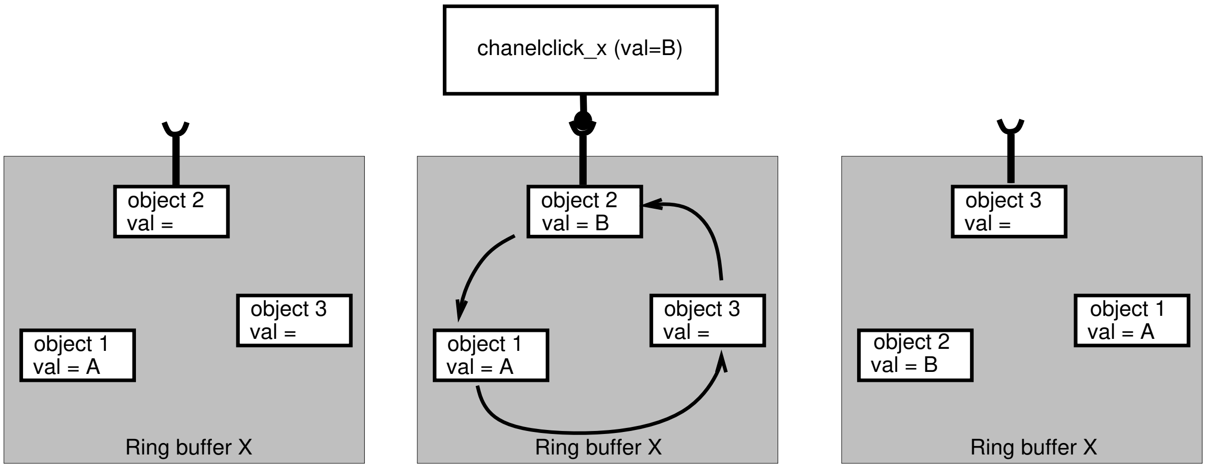


Fig 4: Effect of chanelclick_x on a rimg buffer

This feature is very useful to put a chanel number into a linedit directly clicking in the drawable window.

EXAMPLE: Notify procedure called when a mouse event is detected on a canvas. It puts into the X and Y ring buffers the coordonates of the mouse cursor when clicking.

```
/*********************************************************
* click on the canvas notify proc
**********************************************************/
void click_in_canvas_proc(Xv_window w, Event* ev, Notify_arg arg Notify_event_type type){

    /* works only if event = BUTTON_SELECT (right) release */
    if( even_is_up(ev) && (event_action(ev)= ACTION_SELECT) ){
        /* setx the X and Y ring buffers */
        chanelclick_x(event, event_x(ev) );
        chanelclick_y(event, event_x(ev) );
        }
    return notify_next_event_func(w,(Notify_event)ev,arg,type);
    }
```

If it isn't opened dialog box or if the dialog box haven't got ring buffer, the procedure does nothing.

2.5 The ECP functions

2.5.1 Procedure: *set_callback(...)*

Prototype

#include <cback_lib.h>
*void set_callback(char *filename, Xv_opaque base, Xv_opaque control , int x0, int Y0)*

- *filename* is the prefix of the resource file name. The suffix *".cback"* will be added by *set_callback.*
- *base* is the XView identifier of the application base window.
- *control* is the XView identifier of the control area where will be drawn the menu bar.
- *x0* is the abcisse of the upper left corner of the first button of the menu bar, relative to the upper left corner of the control area.
- *Y0* is the ordonate of the upper left corner of the first button of the menu bar, relative to the upper left corner of the control area.

Description

Set_callback() reads the resource file *filename.cback* and then creates the menu bar and the dialog boxes.

After the compilation of each definition, the object (menu or dialog box) is created and associated with notify procedures internal to the ECP.

The list of the graphic interpretation routines must be set (see next page, the *add_-routine_list(...)* procedure) before calling *set_callback(...)*.

Messages indicating what *set_callback(...)* is doing are displayed on *stderr* while compilating. *Set_callback(...)* returns a message and exits, if an error is detected in the resource file.

2.5.2 Procedure: *add_routine_list(...)*

Prototype

#include <cback_lib.h>
void add_routine_list(int id, int taille)*

- *id* is a pointer to the routine identifier. This routine is defined in a structure *typedef {char* nom; void (*ptr)(char*)} fct_id;* where *nom* is the name of the routine and *ptr* the pointer on it (prototype: *void fct(char*)*). The array *id[taille]* includes the same data as the *fct_id* structure (use *union {fct_id f_id; int id[2]};*).
- *taille* is the size of the *id* array. taille = sizeof(*fct_id*)/sizeof(*int*) (= 2 in V1-00).

Description

Add_routine_list(...) adds the function defined *id* to the list of the available routines for the graphic interpretation of the callbacks results. This array (8 bytes) corresponds to a structure (*fct_id*) having a pointer (4 bytes) to the name of the routine and a pointer on this routine (4 bytes).

This list of the routines must be set before begining the compilation of the resource file, in way to be able to associate a routine with each dialog box.

The name of the routine, in the *fct_id* structure, must be the same as the name declared in the resource file, the parser selecting the good routine by comparison between these names.

2.5.3 Procedure: *chanelclick_x(...)*

Prototype

#include <cback_lib.h>
Xv_opaque chanelclick_x(Event event, int val)*
Xv_opaque chanelclick_y(Event event, int val)*
Xv_opaque chanelclick_z(Event event, int val)*
Xv_opaque chanelclick_w(Event event, int val)*

- *event* is a pointer on the XView event provocating this call.
- *val* is the numeric value which will be set in the Xview active object of the opened dialog box.

Description

A dialog box can have up to 4 (x, y, z, w) ring buffers of active objects (linedit, list or setting) which can be then set directly from an Xview event detected outside of it. The routine *chanelclick_x[_y _z _w](...)* must be call to set the current object of the ring buffer with the numeric value *val* .

This feature is very useful to select chanel numbers directly from a drawable window.

2.5.4 Procedure: *menu_proc(...)*

Prototype

#include <cback_lib.h>
void menu_proc(Menu_item item, Menu_generate op)

- *item* is the Xview identifier of the opened menu.
- *op* indicates the condition in which this routine is called.

Description

This routine is called when an X_Event is detected in the ECP menu bar. It opens the dialog box corresponding to the selected menu item if neither box is already opened. The dialog box is drawn in the upper left corner of the screen, and the mouse cursor is put on the pine. Its old position will be restored at the box shutting.

menu_proc(...) can just be called if an ECP menu is opened because it needs the X_view identifier of the selected menu item to determine the corresponding dialog box.

bug

menu_proc(...) generates a fatal error (segmentation fault) if invoked from a pined menu.

DON'T USE THE *PINABLE* OPTION WITH THE ECP V1-00.

2.5.5 Procedure: *help_button(...)*

Prototype

#include <cback_lib.h>
void help_button(Panel_item panel, Event event)*

- *panel* is the Xview identifier of the panel where the event were detected.
- *event* is a pointer to the Xview event provocating this call.

Description

This routine is called when an X_Event is detected in the *help button* of the opened dialog box. The both arguments are not used, *help_button(...)* can then be called explicitly uncarefully.

bug

help_button does nothing in the ECP V1-00.

2.5.6 Procedure: *run_button(...)*

Prototype

#include <cback_lib.h>
void run_button(Panel_item panel, Event event)*

- *panel* is the Xview identifier of the panel where the event were detected.
- *event* is a pointer to the Xview event provocating this call.

Description

This routine is called when an X_Event is detected in the *run button* of the opened dialog box. Its graphic interpretation routine is then called with the edited command line as argument. When this routine returns, the dialog box is then closed calling *cancel_button(...)* and the mouse cursor is put on its old position (before opening). The both arguments of *run_button(...)* are not used, *run_button(...)* can then be called explicitly if a dialog box is opened.

2.5.7 Procedure: *cancel_button(...)*

Prototype

#include <cback_lib.h>
void cancel_button(Panel_item panel, Event event)*

- *panel* is the Xview identifier of the panel where the event were detected.
- *event* is a pointer to the Xview event provocating this call.

Description

This routine is called when an X_Event is detected in the *cancel button* of the opened dialog box. It closes the opened dialog box and put the mouse cursor on its old position (before opening).

The both arguments of *cancel_button(...)* are not used, *cancel_button(...)* can then be called explicitly if a dialog box is opened.

2.5.8 Procedure: *get_version()*

Prototype

#include <cback_lib.h>
void get_version()

Description

get_version() prints on *stderr* a message indicating the ECP version number.