85
PS/CO/Note 93-[redacted] (Spec.)
12 Octobre 93

Applications Motif

Manuel de Programmation

Ce manuel décrit l'environement de production des programmes d'application, sur station de travail, à partir d'un squelette baptisé : "Frame".

M. Arruat, F. Di Maio

CERN - PS

Contents

1 Introduction

2 Description du Frame

3 Structure de l'Application

4 Contrôle des Aquisitions

5 Edition de l'Interface Utilisateur

6 Integration de l'Interface

7 Fonctions Diverses

Figures

1

Introduction

But

Ce document décrit la procédure à suivre pour réaliser des programmes d'application sur les stations de travail en utilisant l'interface utilisateur OSF-Motif.

Les objectifs de cette procédure sont:

1 de simplifier la réalisation des programmes,

2 de restreindre le volume de code nécessaire pour chaque application spécifique,

3 d'implémenter les standards locaux concernant la présentation et le comportement des programmes d'application.

Cette procédure utilise un cadre type d'application ("frame") implémentant de façon homogène les composants standards de l'interface utilisateur.

Références

Ce manuel est disponible en ligne en version PostScript, dans le directory "/usr/local/doc/ws":

```
# lpr /usr/local/doc/ws_dev/MotifApplic.ps
```

Les manuels suivants sont complémentaires à ce document pour la réalisation des applications:

- DEC-VUIT User's Guide - Manuel d'utilisation de l'éditeur DEC-VUIT.
- OSF/Motif Programmer's Guide - Information détaillée sur le "toolkit" Motif et le language UIL.
- OSF/Motif Style Guide - Spécifications de la présentation et du comportement "standard" des applications Motif.
- OSF/Motif Programmer's référence - Description détaillée de toutes les classes de widgets.
- Widgets PS
- Equipment Interface
- Gestion des erreurs

Ces manuels sont accessibles en ligne au moyen de l'application "BookReader" (normalement disponible dans le menu "Applications" du "Session Manager"). Commande:

```
# dxbook
```

Démarrage

Choix d'un symbole pour l'application

Pour chaque application devant être, à terme, installée dans l'environnement opérationnel, un nouveau symbole doit être choisi. Ce symbole doit être utilisé pour les directory spécifiques à l'application.

La convention est la suivante:

- caractères minuscule uniquement: [a-z]
- 16 caractères maximum

Examples: "alarm", "codd", "eqpinfo", etc.

La database des programmes contient tous les symboles déja utilisés. Dans ce document, le symbole "myapp" est utilisé au lieu du symbole réel de l'application.

Création de la directory

Le symbole de l'application doit être utilisé comme identificateur de la directory:

```
# mkdir myapp
# cd myapp
```

Pour une nouvelle directory, les fichiers et liens symboliques nécessaires doivent être créés par les commandes:

```
# frame_initdir
# mv frame.uil myapp.uil
```

L'état initial de la directory est le suivant:

```
# ls
Framenews@                      SCCS/
vuit_include_template_c@        vuit_main_template_c@
vuit_makefile_template_c@       vuit_stubs_template_c@
user.c                          myapp.uil
user.h
```

Démarrage de VUIT

L'éditeur DEC-VUIT n'est disponible que sur les machines "svps14" et "svps13".

Pour démarrer VUIT:

- ouvrir une nouvelle fenêtre DECterm
- se logger sur "svps14" ou sur "svps13"
- s'assurer que votre machine autorise "svps14" à établir une connection X. Si tel n'est pas le cas agir sur "Security..." situé dans le menu "Customize" du "Session Manager" et ajouter dans la cartouche Host Name "svps14". Vérifier également que la variable DISPLAY est toujours définie après la procédure de "remote login". Si elle n'est plus définie c'est que votre fichier ".login" n'est pas complet. (voir celui du copain chez qui ça marche ...)
- aller dans la directory de l'application
- démarrer VUIT par la commande **"psvuit myapp".**

Exemple:

```
# rlogin svps14
# cd myapp
# psvuit myapp.uil
```

Remarque: seul l'editeur doit tourner sur "svps14", les autres opérations: édition des fichiers C, compilations... peuvent être exécutées sur la machine locale.

Initialisation et Tests

La version initiale des fichiers "makefile", "myapp.c" et "myapp.h" doit être générée depuis VUIT par la commande "Generate Application File":

Activities -> Generate Application File -> In C

Une application vide peut être compilée et essayée à ce stade:

```
# make
# myapp
```

Description des Fichiers

myapp.uil

Ce fichier contient la description de l'interface Motif de l'application. Il peut être édité avec VUIT ou avec un éditeur de texte si nécessaire. Suivant la complexité de l'interface, le code "UIL" peut être séparé sur plusieurs fichiers en éditant le fichier "makefile".

user.c

Ce fichier contient le code de l'application. Si nécessaire, le code peut être séparé sur plusieurs fichiers en éditant le fichier "*makefile*".

user.h

Ce fichier est le "*header file*" de *user.c*.Il doit contenir, entre autre, la déclaration des callbacks qui sont définis dans *user.c*.

makefile

Ce fichier contient les procédures de génération et d'installation de l'application. Par défaut, ce fichier est sur-écrit par la command "Generate Application File". Si ce fichier doit être édité, prendre l'une des précautions suivantes:

- Invalider l'option de sur-écriture *("Generate Build procédure")* au moment de la confirmation de la commande *"Generate Application File"*.
- Supprimer les droits d'acces en ecriture sur le fichier après chaque modification:

```
# chmod +w makefile
# e makefile
# chmod -w makefile
```

myapp.c - myapp.h

Ces fichiers doivent être génères par VUIT pendant la construction du programme et ne doivent donc pas être édités.

Le fichier "myapp.c" contient l'initialisation du programme et la boucle de traitement des évènements. Des fonctions de l'application peuvent être connectées à ce niveau (voir chapitre 3: *Structure de l'Application).*

fichiers vuit_*_template_c

Ces fichiers sont référencés par des liens symboliques. Ils ne doivent pas être modifiés et servent à identifier la version du frame utilisée.

Framenews

Ce fichier est référencé par un lien symbolique. Il a pour but de signaler les dernières modifications apportées au frame. Ces informations resteront dans ce fichier tant que la documentation n'est pas à jour.

Conventions

Afin de rendre le plus lisible possible le code des différentes applications par chacun d'entre nous, il est souhaitable de respecter certaines conventions. Ces conventions sont celles utilisées par "Motif".

Constantes

Utiliser des lettres majuscules et "_" comme séparateurs. Exemple : ***OUF_QUEL_BOULOT***.

Variables

- Si elles sont *privées* (internes au fichier, déclarées en "static") : utiliser des lettres minuscules et des "_" comme séparateurs. Exemple : ***ouf_quel_boulot***.
- Si elle sont *publiques* (visibles par d'autres fichiers, déclarées en "extern") : utiliser des lettres minuscules et des lettres majuscules comme séparateur en commençant par une minuscule. Exemple: ***oufQuelBoulot***.

Fonctions

- Si elles sont *privées* (internes au fichier, déclarées en "static") : utiliser des lettres minuscules et des "_" comme séparateurs. Exemple : ***ouf_quel_boulot()***;
- Si elles sont *publiques* (visibles par d'autres fichiers, déclarées en "extern") : utiliser des lettres minuscules et des lettres majuscules comme séparateur, en commençant par une majuscule. Exemple: ***OufQuelBoulot()***;

Callbacks

Utiliser le suffixe "Proc" pour différencier les callbacks des autres procédures. Exemple: ***OufQuelBoulotProc()***.

Exemples

Une "directory" contenant des exemples a été crée, afin d'illustrer concrètement la pluspart des fonctionalitées décrites dans ce document. Toutes vos propositions sont les bienvenues afin d'étoffer cette "directory" qui est localisée sous:

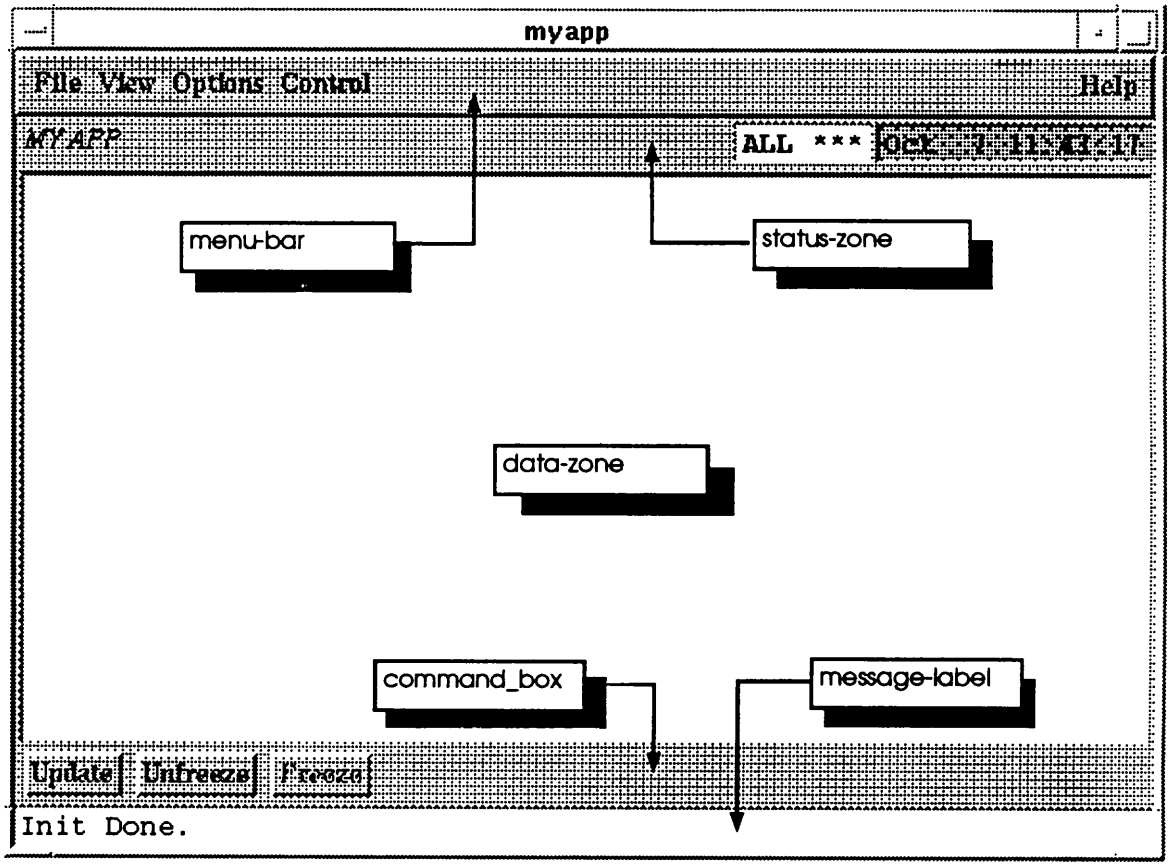
```
/src/frame/examples.
```

2

Description du Frame

Fenêtre Principale

Figure 2-1 Exemple de Fenêtre d'application



menu-bar

La barre des menus comporte les menus standards: "File", "View", "Options" et "Control" ainsi que les menus spécifiques de l'application.

status-zone

La zone d'état (state-bar) comporte les indications suivantes:

- état de l'application (frozen, polling, mdr),
- ligne PLS et numéro de cycle

- date et heure des dernières aquisitions.

data-zone

La zone des données est spécifique à chaque application et comporte, pour une application de mesure: l'état des réglages et la présentation des données.

Nota : Il est important de respecter la cohérence des données affichées dans cette zone. Par exemple, pour une application de mesure qui nécessite un réglage des instruments, les réglages affichés dans la *"data-zone"* doivent correspondre aux mesures affichées dans cette même zone et être rafraichis lors de la mesure. Si une préparation des réglages est nécessaire, il faut afficher les nouveaux réglages dans une fenêtre séparée.

command-box

La boite des commandes comporte les boutons principaux: "Update" / "One-Shot", "Freeze" et "Unfreeze" ainsi que des boutons spécifiques de l'application.

message-label

Les messages de l'application sont affichés dans la partie inférieure de la fenêtre.

Menus

File

Contient:

- Les commandes d'entrée/sortie sur fichier (voir chapitre 3: *Structure de l'Application*)
- Les commandes relatives aux références (voir chapitre 3)
- Les commandes d'impression (voir chapitre 3)
- La commande **"Exit"**

View

Contient les commandes relatives à la présentation des données.

Options

Contient les options du programmes. Les options faisant partie du frame sont:

- **"MDR"**: active ou désactive le mode MDR (voir chapitre 4: *Contrôle des Aquisitions*)
- "Select Occurence...": sélection d'une occurrence particulière du user dans le super-cycle (voir chapitre 4)
- "Specialist Mode..": lorsque ce mode n'est pas demandé, toutes les commandes dites "de spécialistes" ne sont pas présentes au niveau de l'interface (voir chapitre 3)

Control

Le frame propose un menu vide. Par convention, l'application regroupera dans ce menu toutes les actions qui agissent sur un élément "physique":

- Les contrôles du setting de l'instrument. Suivant la complexité de l'instrument, ces contrôles peuvent êtrent regroupés dans une boite de contrôle.
- Activation de knobs pour les paramètres associés (timing, power-supply etc...).
- etc ...

Help

Permet d'obtenir un "help on_line", en visualisant la version post_script de la documentation du programme. Pour ce faire, il suffit d'initialiser le "litteral" "DOC_NAME" avec le nom du document (voir chapitre 5). Le frame propose deux services standards associés aux boutons suivants:

- "Display Doc ..." : visualise le document à l'écran.
- "Print Doc ..." : imprime le document sur l'imprimante post_script la plus proche.

Remarque : si le "literal" "DOC_NAME" n'est pas initialisé par l'application, les deux boutons précédents du menu "Help" restent inactifs.

Hierarchie des Widgets

La hiérarchie des widgets est la suivante: les widgets dont les noms sont en caractères gras et italique ne peuvent pas être modifiés par l'utilisateur. Ils peuvent néanmoins, être supprimés de l'interface utilisateur par l'attribut "managed" ou rendus inactifs par l'attribut "sensitive" (voir chapitre 5). Tous les autres widgets (container, menu, ..) peuvent être modifiés pour integrer les spécificités de l'application :

```
XmMainWindow : main_window
  XmForm : top_zone
    XmMenuBar : menu_bar
      XmCascadeButton : file_cascade_button
      * XmPulldownMenu : file_menu
      XmCascadeButton : view_cascade_button
        XmPulldownMenu : view_menu
      XmCascadeButton : options_cascade_button
      * XmPulldownMenu : options_menu
      XmCascadeButton : help_cascade_button
        XmPulldownMenu : help_menu
      XmCascadeButton : control_cascade_button
        XmPulldownMenu : control_menu
    XmRowColumn : date_zone
      XmFrame : pls_frame
        XmDigit : pls_digit
      XmFrame : date_frame
        XmDigit : date_digit
    XmRowColumn : status_zone
      XmLabel
  XmFrame : message_frame
    XmLabel : message_label
  XmForm : content_manager
   /* Espace utilisateur */
    XmFrame : data_zone
* XmRowColumn : command_box

      * XmPulldownMenu : file_menu
          XmPushButton : save_ref_measure_btn
          XmPushButton : export_measure_btn
          XmSeparator : pulldown_menu_sep
          XmPushButton : load_ref_setting_btn
          XmPushButton : save_ref_setting_btn
          XmSeparator : pulldown_menu_sep
          XmPushButton : exit_button

      * XmPulldownMenu : options_menu
          XmToggleButtonGadget : mdr_button
          XmPushButton : occurrence_btn
          XmSeparator : pulldown_menu_sep
          XmPushButton : specialist_btn
          XmSeparator : pulldown_menu_sep
```

```
        * XmPulldownMenu : help_menu
            XmPushButton : display_doc_button
            XmPushButton : print_doc_button
            XmSeparator : pulldown_menu_sep

    * XmRowColumn : command_box
        XmPushButton : update_button
        XmPushButton : unfreeze_button
        XmPushButton : freeze_button
```

3

Structure de l'Application

Etats de l'Application

Rafraichissements

Deux indicateurs contrôlent le rafraichissement:

- ***frozen:*** si cet indicateur est présent, les aquisitions sont suspendues.
- ***MDR:*** si cet indicateur est présent, les aquisitions se font par le méchanisme "MDR". Sinon, elles se font par "polling".

Présence à l'Ecran

- ***mapped:*** si cet indicateur est présent, la fenêtre est présente à l'écran. Sinon, l'application est iconifiée et frozen.

Variables Globales

Les variables suivantes sont définies dans le fichier "myapp.c" et dans les bibliothèques. Leur déclaration se trouve dans le fichier:

```
#include <psm.h>
```

Variables d'état

- **frameFrozenFlag** contient l'état de l'indicateur *frozen.*
  Syntaxe :

  ```
  extern Boolean frameFrozenFlag;
  ```

- **frameMdrFlag** contient l'état de l'indicateur MDR.
  Syntaxe :

  ```
  extern Boolean frameMdrFlag;
  ```

- **frameMappedFlag** contient l'état de l'indicateur *mapped.*
  Syntaxe :

  ```
  extern Boolean frameMappedFlag;
  ```

Variables Motif

Les variables **display, appContext, topLevelWidget** et **s_MrmHierarchy** sont définies dans le module myapp.c.
Syntaxe :

```
extern Display *display;   /* Display variable */

extern XtAppContext appContext; /* application context */
```

```
extern Widget topLevelWidget;/* Root widget ID of application */
extern MrmHierarchy s_MrmHierarchy;/*MRM database hierarchy ID*/
```

Les variables qui suivent sont définies et initialisées par la bibliothèque. Si l'interface qui leur est associé n'est pas "managée" elles sont initialisées à "NULL", sinon elles contiennent l'identificateur de l'interface.

Les variables **frameUpdateBtn, frameFreezeBtn, frameUnfreezeBtn, frameMdrBtn.**
Syntaxe :

```
extern Widget frameUpdateBtn;/*Update Button identifier*/
extern Widget frameFreezeBtn;/*Freeze Button identifier*/
extern Widget frameUnfreezeBtn;/*Unfreeze Button identifier*/
extern Widget frameMdrBtn; /*Mdr Button identifier*/
```

NB: l'état de ces boutons et leurs fonctionnalités sont gérés par le frame. Toutefois l'application peut agir, dans une certaine mesure, sur ces objets, notamment sur-écrire les valeurs de défaut de certains attributs. (Exemple : les labels par défaut "Freeze" et "Unfreeze" peuvent, pour une application particuliére devenir "New Measure" et "Stop".)

Les variables **framePlsDigit frameDateDigit** et **framêmessageLabel.**
Syntaxe :

```
extern Widget framePlsDigit;/*Pls Digit identifier*/
extern Widget frameDateDigit;/*Date Digit identifier*/
extern Widget framêmessageLabel;/*Message Zone identifier*/
```

Les variables PLS

Les variables plsOption et **plsLineName** sont définies en bibiothèque. Elles contiennent respectivement la ligne PLS courante et le label correspondant à cette ligne (ex: 33 et "SFT").
Syntaxe :

```
extern int plsOption;
extern char * plsLineName;
```

Les variables MDR

Les variables suivantes sont définies et pilotées par la bibliothèque. Elles fournissent des informations liées au *MDR*

- ***frameMdrCycle*** contient le numéro de cycle retourné par le *MDR*.
  Syntaxe :

  ```
  extern int frameMdrCycle;
  ```

- ***frameMdrPlsLine*** contient la ligne PLS retournée par le *MDR*.
  Syntaxe:

  ```
  extern int frameMdrPlsLine;
  ```

variable Polling

La variable **frameUpdatePeriod** est définie en bibliothèque. Cette variable fixe la fréquence de rafraichissement des données en millisecondes, lorsque l'application travaille en mode "polling". Sa valeur, (4000) soit 4 secondes par défaut, peut-être modifiée par l'application.
Syntaxe :

```
extern int frameUpdatePeriod; /* donner la valeur en ms */
```

Fonctions Principales

Les fonctions suivantes doivent être implémentées par l'utlisateur dans le fichier "user.c" en respectant la syntaxe décrite.

FrameInit

Le code de cette fonction doit implémenter l'initialisation de l'application. La fonction est exécutée après que l'interface ait été "réalisée" (`XtRealize`) et avant d'entrer dans la boucle de distribution des évènements (`XtAppMainLoop`). à ce niveau , tous les widgets et "window" ont été crées mais rien n'a encore été "mappé" à l'écran.

```
int FrameInit(unsigned int argc,  char *argv[])
    {}
```

Une seconde fonction **FrameInitBeforeRealize** peut être implémentée pour les actions devant être effectuées avant XtRealize (ex: manipulation de la ressource XmNgeometry).

```
int FrameInitBeforeRealize (unsigned int argc,  char *argv[])
    {}
```

Ces fonctions doivent renvoyer une indication d'erreur (0 si OK).

FrameUpdate

Le code de cette fonction doit implémenter l'aquisition et l'affichage des données. Si l'indicateur *frozen* est présent, cette fonction est appelée chaque fois que le bouton "Update" est activé, sinon son appel est automatique et dépend du mode d'aquisition (Pooling ou MDR). (voir chapitre 4)

```
int FrameUpdate()
    {}
```

Cette fonction doit renvoyer une indication d'erreur (0 si OK).

FrameExit

Le code de cette fonction doit implémenter les actions nécessaires avant la sortie du programme.

```
int FrameExit()
    {}
```

Cette fonction doit renvoyer une indication d'erreur **si et seulement si** l'erreur détectée est importante. En effet un retour non NULL n'autorise pas la fin du programme.

Fonctions de Changement d'Etat

Les fonctions suivantes peuvent être implémentées pour gérer les changements d'états de l'application.

FrameFreezeChanged

Cette fonction est activée quand l'état de l'indicateur *frozen à* changé. La variable `frameFrozenFlag` doit être lue pour connaitre le nouvel état. Syntaxe :

```
int FrameFreezeChanged()
    {}
```

Cette fonction doit renvoyer un code d'erreur (0 si OK). Si un code d'erreur est retourné, l'action demandée est annulée : en particulier, dans le cas d'une demande de changement d'état de Unfreeze à Freeze ou, un code d'erreur annule la demande et force l'application à repasser en mode Unfreeze. (Exception: si l'application est iconifiée, même dans cette transition, ce code d'erreur est ignoré).

FrameMdrChanged

Cette fonction est activée quand l'état de l'indicateur MDR à changé. La variable `mdrFlag` doit être lue pour connaitre le nouvel état. Syntaxe :

```
int FrameMdrChanged()
    {}
```

Le code d'erreur retourné par cette fonction n'est pas pris en compte dans cette version.

Le code de cette fonction doit contenir en particulier l'appel aux fonctions ***FrameMdrSubscribe*** et ***FrameMdrUnsubscribe*** (voir chapitre 4).

FrameMapChanged

Cette fonction est activée quand l'état de l'indicateur *map* à changé. La variable mapFlag doit être lue pour connaître le nouvel état. Syntaxe :

```
int FrameMapChanged()
    {}
```

Cette fonction doit renvoyer une indication d'erreur (0 si OK). Si le code retourné est non NULL , l'action est annulée : en particulier dans la transition Map -> Unmap ou l'envoi d'un code d'erreur non NULL force le retour à l'état désiconifié. Si le code d'erreur est NULL la transition Map -> Unmap est effectuée et l'application est mise en mode "frozen"

FrameReqNewState

Cette fonction permet de définir un nouvel état pour l'application. Syntaxe :

```
void FrameReqNewState( unsigned char new_state)
```

```
new_state : paramètre définissant le nouvel état de
    l'application. Les valeurs possibles sont :
    FrameFREEZE_STATE,      FrameUNFREEZE_STATE,
    FrameMDR_STATE,         FramePOLLING_STATE.
```

Fonction de Sélection du mode Spécialiste

FrameSpecialistMode

Cette fonction est appelée lorsque l'utilisateur active le bouton "Specialist mode" situé dans le menu "Options". Le mode "spécialist" doit donner accés à toutes les commandes qui ne sont pas autorisées dans le mode restrictif. L'application démarre toujours dans le mode restrictif. Syntaxe:

```
int FrameSpecialistMode(Boolean flag)
```

```
flag : spécifie le mode demandé par l'utilisateur.
```

- `"True" : le mode spécialiste est demandé`
- `"False" : le mode restrictif est demandé`

Cette fonction doit renvoyer un code d'erreur (0 si OK). Si un code d'erreur est retourné la demande est annulée. Exemple : si l'application juge qu'elle ne peut pas autoriser le mode spécialiste, car certaines conditions extérieures ne sont pas remplies, elle doit retourner un code d'erreur qui permet de remettre l'interface en accord avec la situation.

Fonctions PLS

FramePlsChanged

Cette fonction doit être implémentée si l'application veut être informée d'un changement de la ligne Pls. Pour plus d'information sur le changement de la ligne Pls, (voir chapitre 4).

Fonctions Standards

Les fonctions suivantes sont justes des points d'entrées, (le frame n'implémente pas pour l'instant de code particulier), auxquels l'application peut se connecter. L'intérêt d'imposer ces noms de fonctions, réside en deux points :

- le frame conserve la possibilité d'implémenter, au niveau bibliothèque, du code qui se répèterait fréquemment dans les applications.
- toutes les applications implémenteront le code nécessaire aux services standards ("print", "save as reference", ...) dans des fonctions standards. D'ou, une meilleure lisibilité du code.

FrameExportAsText

La fonction ***FrameExportAsText*** est appelée lorsque le bouton "*Export As Text*" situé dans le "File Menu" est activé. Son rôle est de sauver les données sur un fichier dans un format compatible "spread-sheet" en s'appuyant sur des fonctions de bibliothéque prévues à cet effet (cf EqpFile note).

```
extern void FrameExportAsText();
```

FrameSaveAsRef

La fonction ***FrameSaveAsRef*** est appelée lorsque le bouton "*Save as référence*" situé dans le "File Menu" est activé. Son rôle est de sauver les données en tant que valeurs de référence. Attention : il faut vérifier que le "setting" actuel de l'instrument corresponde au "setting" de référence.

```
extern void FrameSaveAsRef();
```

FrameSaveSetting

La fonction ***FrameSaveSetting*** est appelée lorsque le bouton "*Save Settings as référence*" situé dans le "File Menu" est activé. Son rôle est de sauver le "setting" actuel de l'instrument en tant que "setting" de référence.

```
extern void FrameSaveSetting();
```

FrameLoadRefSetting

La fonction ***FrameRefSetting*** est appelée lorsque le bouton "*Load référence Setting*" situé dans le "File Menu" est activé. Son rôle est d'envoyer le "setting" de référence vers l'instrument.

```
extern void FrameLoadRefSetting();
```

FramePrint

La fonction ***FramePrint*** est appelée lorsque le bouton "*Print*" situé dans le "File Menu" est activé. Son rôle est d'imprimer en format A4 sur l'imprimante la plus proche toutes les informations utiles : mesures, résultats de calcul, "setting" de l'instrument, date des aquisitions, conditions PLS, etc ...

```
extern void FramePrint();
```

Note : le nom de l'imprimante DEC le plus proche (à utiliser pour du texte ou pour un "hardcopy") est "**1**". Le nom de l'imprimante postcript la plus proche est "**0**".

Widgets Principaux

bouton "Exit"

L'action de ce bouton est d'activer la fonction *FrameExitProc* et de sortir de l'application.

bouton "Update" / "One Shot"

Ce bouton n'est "sensitive" que si l'indicateur frozen est présent. Il revêt deux fonctionnalitées différentes suivant le mode d'aquisition.

- Mode *MDR* : le label de ce bouton est "One Shot" et l'application peut fonctionner dans un mode "single shot". L'action du bouton activera la fonction *FrameUpdate* une seule fois sur le "PLS USER" défini.
- Mode *Polling*: le label est "Update". L'action du bouton active immédiatement et une seule fois la fonction *FrameUpdate*.

boutons "Freeze" et "Unfreeze"

Ces boutons contrôlent l'état de l'indicateur *frozen* et, par voie de conséquence, activent/désactivent les appels automatiques à la fonction *FrameUpdate*.

toggle MDR

Ce "toggle" du menu "Options" contrôle l'état de l'indicateur *MDR* (voir chapitre 4)

4

Contrôle des Aquisitions

Sélection de la Condition PLS

Variables d'environement

La variable d'environement **PLS_TELEGRAM** doit être définie. Elle spécifie le télégramme PLS qu'utilise l'application.

Exemple:

```
# setenv PLS_TELEGRAM LPI
```

La variable d'environement **PLS_LINE** peut être utilisée pour spécifier la ligne PLS. Le console manager utilise ce moyen pour passer la ligne PLS aux applications. La syntaxe de cette variable comprend le télégramme, le groupe et la ligne, séparés par un point (.).

Exemple:

```
# setenv PLS_LINE LPI.USER.PPE
```

Arguments

La ligne pls peut être spécifiée dans les arguments de la commande UNIX de démarrage de l'application au moyen de l'option **"-pls_line"**. La syntaxe est la même que celle de la variable d'environement PLS_LINE.

Exemple:

```
# myapp -pls_line LPI.USER.PPP
```

Sélection dynamique

La fonction **PlsChangeOption** peut être utilisée à l'interieur du programme pour permettre à l'utilisateur de modifier de façon interactive la ligne PLS.

```
external int PlsChangeOption (int flags);
    /* flags:    control the dialog options */
```

Cette fonction construit un interface contenant au plus deux options à définir : le groupe et une ligne dans ce groupe. La valeur de l'argument passé à la fonction, construit à l'aide de masques, permet de restreindre les possibilités de ces choix aux cas désirés. Ces différents masques, dont la signification est détaillée ci-dessous, peuvent êtres cumulés au moyen de l'operateur OU (|) :

- ***PCO_EXCLUSIVE_GR***: ce masque offre la possibilité de sélectionner un groupe parmi tous les groupes PLS de type "*exclusif*".

- ***PCO_BITPATTERN_GR***: ce masque offre la possibilité de sélectionner un groupe parmi tous les groupes PLS de type "*bit-pattern*".
- ***PCO_ALL_LINES***: ce masque offre la possibilité de sélectionner une ligne parmi toutes les lignes appartenant au groupe sélectionné, indépendamment de l'état de la "user-matix". Si ce masque n'est pas utilisé seules les lignes du groupe sélectionné présentes dans la "user-matrix" seront proposées.
- ***PCO_STATIC_GR***: ce masque supprime la possibilité de définir le groupe car il est forcé à la valeur du groupe auquel la PlsLine courante appartient. Exemple : si l'application travaille avec la PlsLine "SFT", la valeur du groupe sera "USER".
- ***PCO_NEXT_GR***: ce masque offre la possibilité de sélectionner un groupe parmis les groupes "*NEXT*" du télégramme.
- ***PCO_NEG_LOGIC***: traitement de la logique négative (cf PLS decoders).

Remarque : toutes les combinaisons ne sont pas possibles, car certains masques sont indispensables. Au moins un des deux masques (PCO_EXCLUSIVE_GR, PCO_BITPATTERN_GR) doit être toujours présent car ils définissent le type de groupe : soit des groupes "exclusifs", soit des groupes "bit-pattern". Ensuite les autres options permettent de raffiner ce premier choix.

Exemple: changement de la ligne PLS, sans changer de groupe, sans se limiter aux lignes présentes dans la user-matrix et en supposant que la ligne PLS actuelle soit "SFT"):

```
result = PlsChangeOption (PCO_EXCLUSIVE_GR | PCO_STATIC_GR |
   PCO_ALL_LINES);
```

Regardons la contribution de chacun des masques : le masque `PCO_EXCLUSIVE_GR` fait un filtre sur le type de groupe en excluant tous les groupes qui ne sont pas de type exclusif. Ensuite le masque `PCO_STATIC_GR` supprime le choix parmis les groupes restants en imposant le groupe auquel appartient la PlsLine courante, en l'occurrence le groupe "USER". à ce stade, le groupe est défini. Enfin, le masque `PCO_ALL_LINES` fait que les lignes qui seront proposés ne seront pas limités aux lignes présentes dans la "user-matrix" courante.

Remarque : Attention à la combinaison entre le masque qui définit le type de groupe et le masque qui choisit le groupe en fonction de la PlsLine. En effet, si dans l'exemple précédent on avait mis : `PCO_BITPATTERN_GR | PCO_STATIC_GR`, la fonction aurait retournée une erreur car les deux masques sont imcompatibles du fait de la valeur de la PlsLine courante qui appartient à un groupe "exclusif".

La fonction renvoie un indicateur d'erreur (0 si OK). En cas de changement confirmé par une action sur le bouton "OK" ou "Apply" de la boite, les variables d'état *plsOption* et *plsLineName* sont mises à jour, le widget d'affichage de la condition PLS est rafraichi (en jaune pour indiquer que la ligne PLS de l'application n'est plus cohérente avec celle du contexte de travail) et la fonction de changement d'état *FramePlsChanged* est executée (si elle existe).

Note: Depuis le "Console Manager", il est toujours possible de démarrer dans des contextes differents la même application avec deux conditions PLS différentes. La sélection dynamique peut introduire des écrans incohérents au niveau de PLS. Elle n'est donc pas à généraliser.

contrôle du Mode d'Aquisition

Arguments

L'état intilial de l'indicateur *frozen* peut être spécifié dans les arguments de la ligne de commande de démarrage de l'application au moyen de l'option **"frozen"** précédé du signe "**+**" ou "**-**". La syntaxe est la suivante :

```
# myapp -frozen  /* l'application démarre non "frozen" */
# myapp +frozen  /* l'application démarre "frozen" */
```

Le mode d'aquisition peut être spécifié dans les arguments de la ligne de commande de démarrage de l'application au moyen de l'option "**mdr**" précédé du signe "+" ou "-". La syntaxe est la suivante:

```
# myapp -mdr  /* l'application démarre en mode polling */
# myapp +mdr  /* l'application démarre en mode MDR */
```

contrôle Dynamique

Pour les applications qui le souhaitent, il est possible de changer de mode d'aquisition à tout moment. Pour ce faire, dans le menu "Options", le toggle "**MDR**" fait passer l'application du mode Polling au mode MDR et vice-versa. Lors d'un changement du mode d'aquisition la fonction *FrameMdrChanged* est executée (si elle existe).

La fonction *FrameMdrChanged* doit contenir, outre le code spécifique nécessaire à l'application, l'appel à deux fonctions qui permettent respectivement de s'abonner et se "désabonner" aux services du MDR . Ces fonctions sont les suivantes :

- La fonction ***FrameMdrSubscribe*** permet de faire enregistrer par le MDR la ou les listes d'aquisition d'un objet équipement:

```
int FrameMdrSubscribe ( Equipment equipment,
    int * property_list,  int property_count);

arguments:

equipment : objet équipement (cf Equipment Interface)

property_list : liste de toutes les proprietés auxquelles on
    veut s'abonner pour cet équipement.

property_count : nombre de proprietés.
```

- La fonction ***FrameMdrUnsubscribe*** permet de supprimer du MDR la ou les listes d'aquisition d'un objet équipement:

```
int FrameMdrUnsubscribe ( Equipment equipment);

arguments:

equipment : objet équipement (cf Equipment Interface)
```

Note : Pour un exemple d'utilisation de ces routines dans un contexte simple voir l'exemple "trafo".(fonction FrameMdrChanged dans le fichier user.c)

Note : le passage du mode MDR au mode pooling puis au mode MDR peut etre utilisé pour ré-initialiser la connection avec le MDR

Sélection d'Occurences

Lorsque l'application utilise le mode d'aquisition "*MDR*", il est possible de sélectionner une occurrence particulière du cycle dans le super-cycle.

Pour bénéficier de l'interface proposé par le frame il faut ajouter dans myapp.uil, soit par le biais d'un éditeur de texte, soit par VUIT (File / Include Files / Add New File) :

```
include file '/usr/local/frame/frame_occurence.uil'
```

Cette fonctionnalité est entièrement gérée par le frame et ne nécessite pas de point d'entrée dans le code de l'application. Une fois l'occurrrence choisie, l'application sera réveillée (par le biais de la fonction *FrameUpdate*) uniquement sur cette occurrence.

5

Edition de l'Interface Utilisateur

L'édition de l'interface utilisateur se fait au moyen de VUIT. La description complète de cet éditeur est disponible comme manuel DEC (disponible on-line par bookreader).

Insertions et Suppressions

Suppressions de parties du frame

Les parties non nécessaires du frame de l'application doivent être enlevées en rendant les widgets "unmanaged" (ils ne doivent pas être supprimés par la commande "*Delete*"). Cette operation doit se faire au niveau du parent du widget concerné.

La méthode est la suivante:

1 Editer le parent du widget (*"Widget Tree"* pour connaitre le parent).

2 Editer la list des "children": *"Edit"->"Children"*

3 Invalider l'état *"Managed"* du widget concerné.

Insertions de menus ou de commandes dans les menus

Le frame propose, par défaut, des menus que l'on retrouve dans la plupart des applications, ainsi que les commandes standard qui y sont rattachées. Tous les menus proposés peuvent être modifiés (par addition de boutons dans les widgets pull-down-menu) pour s'adapter au spécificités de l'application. Bien entendu de nouveaux menus peuvent être ajoutés (par addition de cascade-buttons dans le menu-bar)..

Nota : le frame possède un objet séparateur pour les menus, dont le nom est : "pulldown_menu_sep". Dans les menus, où il est nécessaire, il faut l'ajouter par *référence* (voir VUIT manuel page 2.23).

Definitions Standardisées

Tous ces "literals" sont définis dans la bibliothèque du frame. Ils sont regroupés par catégories définies par le type de données :

Couleurs

- *bg_light, bg_dark* : sont les couleurs de "background" par défaut. Par exemple, bg_light peut-être utilisé pour les données dynamiques et bg_dark pour les données statiques(titres, boutons ..).
- *on_bg, off_bg, error_bg, warning_bg, unknown_bg* : sont les couleurs de "background" pour indiquer le status. L'utilisation de ces couleurs nécessite la couleur "foreground" noire.

- *shadow_dark, shadow_light* : sont les couleurs utilisées pour définir les couleurs bottom et top de l'ombrage ("shadow") des widgets de type frame. Ces couleurs s'harmonisent avec le couleur "bg_light" en "background".

Tables de Couleurs (data type : asciz_table)

- *colors_table_bg* : c'est une table de couleurs regroupant les couleurs qui définissent le status. Elle doit être utilisé pour définir l'attribut "XmNbackgroundColors" d'un digit. Sa composition est la suivante :

```
colors_table_bg: exported asciz_table("light_bg", "on_bg",
                 "off_bg", "warning_bg", "error_bg",
                 "unknown_bg", "dark_bg");
```

Polices de caractères

- *symbol_fontl* : cette font-list combine deux polices de caractères afin de réaliser des chaînes de caratères contenant des caractères normaux et des caractères grecs. Exemple : $4\sigma^2/\beta$.
- *button_fontl* : elle doit être utilisée pour définir la "font_list" des "push_buttons":

```
-*-TIMES-BOLD-R-*--*-140-*-*-*-*-ISO8859-1
```

- *param_fontl* : elle doit être utilisée pour définir la "font_list" des "digit":

```
-*-COURIER-BOLD-R-*--*-140-*-*-*-*-ISO8859-1
```

- *title_fontl* : elle doit être utilisée pour définir la "font_list" des titres:

```
-*-TIMES-BOLD-R-*--*-140-*-*-*-*-ISO8859-1
```

- *wsw_fontl* : elle doit être utilisée pour définir la "font_list" des "WheelSwitch":

```
-*-COURIER-BOLD-R-*--*-180-*-*-*-*-ISO8859-1
```

String

- *APP_CLASS* : ce string doit être **impérativement** initialisé car il définit la classe de l'application. La convention utilisée consiste à prendre le nom de l'application avec la premiére en lettre en majuscule.
  Exemple : nom du programme : myapp, APP_CLASS : Myapp
- *DOC_NAME* : ce string doit contenir le nom de la version poscript de la documentation de l'application, si nécessaire. Ce fichier poscript doit être installé dans :

```
/usr/local/doc/ws_applic/mydoc.ps DOC_NAME : "mydoc.ps"
```

Listes d'arguments

La définition de listes d'arguments est fortement recommendée, car elle apporte plus de lisibilité dans le fichier UIL et en réduit le nombre de lignes. De plus cela permet de gérer plus facilement certaines évolutions de l'interface : supposons que vous souhaitiez changer le "FontList" de tous les "PushButton" de votre application. Si, cet attribut est défini dans une liste d'argument, le changement est localisé à un seul endroit. Voici quelques listes d'arguments utilisée par le frame que vous pouvez peut être utiliser dans votre application :

- button_args :

```
button_args: arguments {
    XmNbackground = dark_bg;
    XmNfontList = button_fontl; };
```

- digit_args :

```
digit_args: arguments {
    XXmNfontList = param_font1;
    XmNbackground = light_bg;
    XmNmarginWidth = 3; }
```

- shadow_colors :

```
shadow_colors: arguments {
    XmNbottomShadowColor = shadow_dark;
    XmNtopShadowColor = shadow_light;}
```

Recommendations

Ne pas dépendre de la taille des caracteres

Il faut éviter de positionner les widgets les uns par rapport aux autres au moyen des arguments XmNx et XmNy. Si les polices de caractères sont changés, les positions relatives doivent être conservées.

Il faut éviter de préciser la taille des widgets "container" par les resources "XmNwidth" et "XmNheight" lorsque ces widgets ne contiennent que du texte ou des objets de taille fixe.

Les containers "XmRowColumn" et "XmForm" sont à utiliser systématiquement pour résoudre ces problemes.

Utiliser les "literals" et les listes d'arguments

Laisser le Console Manager positionner la fenêtre.

6

Integration de l'Interface

Gestion des Widgets

Enregistrement des widgets

Afin d'être manipulés par l'application, les widgets doivent être enregistrés dans une table. Dans cette table, les widgets sont référencés au moyen de leur nom.

Pour qu'un widget soit enregistré, la procédure UIL ***RegisterSelf*** doit être ajoutée dans la liste de ses *callbacks*, sous la ressource "MrmNcreateCallback". En général cette ressource n'est utilisé que pour les widgets qui sont manipulés en cours d'exécution.

Procédure (VUIT):

1 Editer le widget (command "Modify...")

2 Lui donner un nom si nécessaire

3 Activer "Add Callback..." (menu "Edit)

4 Sélectionner MrmNcreateCallback

5 Activer "Callback Editor..."

6 Sélectionner la ligne de la procédure (ce n'est pas facile).

7 Activer "Select procédure..."

8 Sélectionner RegisterSelf

Code généré (procédure éditeur de texte):

```
MrmNcreateCallback = procedures
    {
    RegisterSelf();
    };
```

Identification d'un widget

La fonction ***GetWidget*** retourne l'identificateur du widget à partir de son nom. Si ce widget n'est pas enregistré, la fonction imprime un message de "WARNING" dans "l'error viewer" et retourne la valeur NULL.

```
Widget GetWidget(char * widget_name)

arguments :

  widget_name: nom du widget(nom défini dans le fichier UIL).
```

Remarques:

— si le widget trouvé n'est pas le bon, c'est que le même widget à été référencé plusieurs fois dans l'interface.

Manipulations des Widgets

La fonction ***VUIT_Manage*** permet de faire apparaitre une nouvelle partie de l'interface (ex: fenêtre supplémentaire).

```
external void VUIT_Manage (char *widget_name);
```

Son action est la suivante:

— si le widget n'est pas déja enregistré, il est chargé depuis le fichier myapp.uid (`MrmFetchWidget`) et enregistré.

— si le widget n'est pas dans l'etat "managed", il y est mis (`XtManageChild`)

— dans les autres cas, sa fenêtre est repassée devant celles qui pouvaient l'obscurcir.

La fonction ***VUIT_Unmanage*** permet de faire disparaitre une partie de l'interface.

```
external void VUIT_Unmanage (char * widget_name);
```

La procédure UIL ***UnmanageWidgetProc*** est liée à la fonction VUIT_Unmanage et peut être utilisée dans les callbacks. :

```
procédure UnmanageWidgetProc (char * widget_name);
```

Exemple: le "activate callback list" d'un bouton "Close" d'une fenêtre, pourrait être :

```
XmNactivateCallback = procedures {
  UnmanageWidgetProc(fenêtre);/*supprime la fenêtre de l'écran*/
  ClosePopupproc();/*appel de la fonction liée au bouton close*/
};
```

La fonction ***HashRegister*** permet d'enregistrer un nom de widget et son identificateur. Elle est utilisée dans le cas ou l'on cré dynamiquement plusieurs instances d'un objet décrit dans le fichier "UIL".

```
int HashRegister(char * widget_name, Widget id)
```

Elle retourne le code 1 si l'enregistrement s'est éffectué correctement, 0 en cas d'erreur.

Les procédures UIL ***SetWidgetSensitiveProc*** et ***SetWidgetUnsensitiveProc*** peuvent être utilisées dans les callbacks pour respectivement rendre "sensitive" ou "unsensitive" un widget.

```
procédure SetWidgetsensitiveeProc(char * widget_name);
procédure SetWidgetUnsensitiveeProc(char * widget_name);
```

Addition de Callbacks

Pour ajouter des callbacks, la méthode est la suivante:

1 Implémenter le callback dans le fichier user.c (la description du widget doit être consultée pour la syntaxe exacte)

2 Ajouter la déclaration de la fonction dans le fichier user.h.

3 Ajouter la déclaration d'une procédure du même nom dans le fichier myapp.uil (avec VUIT ou l'éditeur de texte)

4 Regénérer le fichier myapp.c (avec VUIT).

Exemple:

1 Dans user.c:

```
void WswValueChanged (widget, tag, call_data)
      Widget widget;
      int * tag;          /* application specific */
      XmWheelSwitchCallbackStruct *call_data;
{ printf ("WheelSwitch new value is %f\n",
*call_data->value);}
```

2 Dans user.h

```
   extern void WswValueChanged(
   #ifdef ANSI
        integer * tag
   #endif /* ANSI */);
```

3 Dans myapp.uil `!***VUIT_no_Generate***`

```
      WswValueChanged(integer);
```

Ceci peut être fait depuis VUIT par la commande "Create"->"procédure"en entrant le nom de la procédure et le type du paramétre et en invalidant l'option "Generate in Application".

4 Depuis VUIT: "Generate Application File"->"In C"

Echanges de Données

En supplément des fonctions ***XtSetValues***, ***XtVaSetValues***, ***XtGetValues*** et ***XtVaGetValues***, des fonctions simples sont utisables avec certains widgets.

GetWidgetInt

La fonction ***GetWidgetInt*** retourne la valeur contenue dans le widget, en opérant une conversion de type de "*double*" à "*integer*". Cette fonction supporte uniquement la class "*wheelSwitchWidgetClass*".

```
int GetWidgetInt (Widget w,int * data)

arguments:

  w: spécifie le widget;

  data: adresse de la variable qui doit recevoir la valeur.
```

SetWidgetInt

La fonction ***SetWidgetInt*** écrit dans le widget la valeur de type "integer" qui lui est passée. Cette fonction supporte uniquement la class "*wheelSwitchWidgetClass*".

```
int SetWidgetInt (Widget w,int * data)

arguments:

  w: spécifie le widget;

  data: adresse de la variable contenant la valeur à écrire.
```

7

Fonctions Diverses

Colors

SetBackgroundValue

La fonction ***SetBackgroundValue*** permet de changer la couleur de fond d'un widget. Le frame défini une palette de couleur de background, qui respecte la convention des couleurs, à utiliser. Elles sont au nombre de sept :

COLOR_NONE ***COLOR_ON*** ***COLOR_OFF***
COLOR_WARNING ***COLOR_ERROR*** ***COLOR_UNKNOWN***
COLOR_DARK.

```
void SetBackgroundValue(Widget w, int color)

arguments:

  w: spécifie le widget.

  color: spécifie la couleur.
```

Affichage de Messages

DisplayMessage

La fonction ***DisplayMessage*** fait apparaitre le texte du message dans la zone de messages ("message-label").

```
void DisplayMessage(int color, char *format, [, arg ])

arguments:

  color: spécifie la couleur du background de la zone message.
    Les valeurs possibles sont les couleurs standards définies
    par le frame.

  La syntaxe des derniers arguments est la même que celle de la
    fontion printf.
```

La procédure UIL ***DisplayMessageProc*** peut être utilisée dans les callbacks pour afficher un texte dans la zone de message. La couleur de background est par défaut: COLOR_NONE.

```
procédure DisplayMessageProc(char * message);
```

TemporaryDisplayMessage

La fonction ***TemporaryDisplayMessage*** fait apparaitre le message pendant 10 secondes uniquement dans la zone de messages ("message-label").

```
void TemporaryDisplayMessage(int color, char *format, [, arg ])
```

Les arguments sont les mêmes que pour la fonction précédente.

Boites de Dialogue

MessageBoxDialog

La fonction ***MessageBoxDialog*** fait apparaitre une boite de message sur la fenêtre choisie comme parent. Elle peut contenir un message et trois boutons. Celui situé le plus à gauche est associé à l'action positive (confirmation), tandis-que le deuxième est associé à l'action négative (annulation). Le troisième, est le bouton "Help".

```
int MessageBoxDialog (Widget parent, unsigned char dialog_type,
    char *s1, char *s2, Boolean default_action, int help_id,
    char *format, [, arg ]
```

arguments :

parent: specifie l'idendificateur de la fenêtre sur laquelle la boite de message doit apparaitre.

dialog_type: specifie le type de la boite de message. Les valeurs possibles sont définies dans Xm.h (cf. XmMessageBox).

s1: spécifie le nom du premier bouton. Si la valeur est nulle, le bouton est supprimé. Ce bouton représente la confirmation de l'action.

s2: spécifie le nom du deuxième bouton. Si la valeur est nulle, le bouton est supprimé. Ce bouton représente l'annulation de l'action.

default_action: si sa valeur est "True", c'est le premier bouton qui défini l'action par défaut (s'il existe).Si la valeur est "False", c'est le deuxième bouton (s'il existe).

help_id: non implémenté pour le moment.Passer la valeur -1.

La syntaxe des derniers arguments est la même que celle de la fontion ***printf***.

retour : deux codes peuvent être retournés

1: si le bouton "s1" est activé.

0: si le bouton "s2" est activé.

ErrPrompt

La fonction ***ErrPrompt*** fait apparaitre une boite de message sur la fenêtre choisie comme parent. Elle contient un message et un bouton "Acknowledged".

```
void ErrPrompt(Widget parent,ErrClass severity,char *format,[,
    arg ])
```

arguments :

parent: spécifie l'idendificateur de la fenêtre sur laquelle la boite de message doit apparaitre.

severity: spécifie le type de boite de dialogue en fonction du niveau de l'erreur.

ErrNONE: XmDIALOG_INFORMATION.

ErrWARNING: XmDIALOG_WARNING.

ErrSOFTWARE ou ErrFATAL : XmDIALOG_ERROR.

La syntaxe des derniers arguments est la même que celle de la fontion printf.

WaitDefinedTime

La fonction ***WaitDefinedTime*** fait apparaitre une boite de message sur la fenêtre choisie comme parent. Elle contient seulement un message qui informe l'utilisateur que l'action en cours va durer un temps bien défini.

```
void WaitDefinedTime(Widget parent, unsigned long time_out, char
    * format, [, arg ])
```

arguments :

parent: specifie l'idendificateur de la fenêtre sur laquelle la boite de message doit apparaitre.

time_out: spécifie le temps d'apparition de cette boite de dialogue.

La syntaxe des derniers arguments est la même que celle de la fontion printf.

WaitRequestCheck

La fonction ***WaitRequestCheck*** fait apparaitre une boite de message sur la fenêtre choisie comme parent. Elle peut être utilisée lorsque le temps d'exécution d'une action n'est pas connu et que seul la lecture d'un status ou d'une condition nous permet de dire que l'action est terminée. Elle contient un message qui informe l'utilisateur sur le type de l'action en cours et du temps écoulé depuis le lancement. De plus un bouton "Abort" permet d'interrompre le processus d'attente à tout moment.

```
int WaitRequestCheck(Widget parent, int (* check_fct) (),
    unsigned long perid, char * format, [, arg ])
```

arguments :

parent: specifie l'idendificateur de la fenêtre sur laquelle la boite de message doit apparaitre.

check_fct: cette fonction, définie par l'utilisateur, détermine si l'action est terminée. Elle doit retourner les codes suivants :

1 : l'action est terminée et s'est déroulée correctement.

0 : l'action est en cours.

-1 : cas d'erreurs jugées fatales par l'application.

period: spécifie la fréquence d'appel de la fonction de "check" en ms.

La syntaxe des derniers arguments est la même que celle de la fontion printf.

retour : Les codes suivants sont retournés :

1 : L'action s'est déroulée correctement.

0 : Le processus d'attente est interrompu par l'activation du bouton "Abort"

-1 : Le processus d'attente est stoppé car la fonction de "check" a renvoyé un code d'erreur(-1).

Date

ShowDate

La fonction ***ShowDate*** permet de changer la couleur du background de la zone date.

```
void ShowDate(int color)
```

arguments:

color: spécifie la couleur du background de la zone date. Les valeurs possibles sont les couleurs standards définies par le frame.

Cursors

ShowWaitCursor et HideWaitCursor

Les fonctions ***ShowWaitCursor*** et ***HideWaitCursor*** permettent respectivement de montrer et supprimer un curseur de type "sablier", symbolisant une activité de l'application pendant laquelle l'interface ne peut pas répondre aux sollicitations de l'utilisateur

```
void ShowWaitCursor(Window w)

void HideWaitCursor(Window w)

arguments

  w: spécifie la fenêtre dans laquelle le curseur doit changer.

Note : La "Window" associée à un "Widget" s'obtient par la
    fonction XtWindow(w) avec : Widget w.

Exemple: Si w est le widget sur lequel le curseur doit changer
    il faut passer :

        ShowWaitCursor(XtWindow(w));
```

EventHandler

CloseWindowHandler

La fonction ***CloseWindowHandler*** permet de déclencher une action particuliére lors de l'activation du bouton "Close" situé dans le menu du Window Manager de la pluspart des fenêtres. Par défaut, l'action de ce bouton fait disparaître la fenêtre de l'écran sans que l'application en soit informée.

```
void CloseWindowHandler(Widget widget, void (* fct)())

arguments :

  widget: spécifie l'identificateur d'un widget appartenant à la
    hiérarchie de la fenêtre.

Exemple: Si w est un BulletinBoardDialog et que l'on désire
    attacher la fonction CloseBulBoard au bouton close du menu
    du WindowManager de cette fenêtre, il faut passer :

        CloseWindowHandler(w, CloseBullBoard);
```

Knobs

OpenKnobs

La fonction ***OpenKnobs*** permet d'ouvrir un knob sur le ou les équipements qui lui sont passés. Les knobs apparaitront dans la fenêtre

```
void OpenKnobs(char * eqp_names[], int eqp_count)

arguments:

  eqp_names: tableau des noms des équipements.

  eqp_count: nombre d'équipements
```

EventLoop

MainLoop

La fonction ***MainLoop*** gère les Evenements X. Son utilisation reste trés spécifique.

```
void MainLoop(Boolean * flag);

arguments:

  flag: conditionne l'exécution de la boucle. Tant que le
    booléen est "false", la boucle est exécutée.
```

Voici un exemple de son utilisation : Supposons que dans un "callback", l'application désire faire apparaître une fenêtre avec des sélections possibles, attendre la réponse de l'utilisateur et, exécuter un code particulier en fonction du choix fait par l'utilisateur, puis sortir du "callback". L'appel à la fonction MainLoop permet de résoudre ce problème:

```
static Boolean flag;

static int option;

/*Callback retournant la sélection de l'option*/

void SelectProc(w, tag)

Widget w;

int *tag; /*valeur de l'option*/

{

   option = *tag;

   flag = True;

}

TotoProc {

   ...

   flag = False;

   /*Manage de la fenêtre*/

   VUIT_Manage(select_parameters_id);

   MainLoop(&flag);

   switch(option) {

   ...

}
```

Fonctions d'allocation dynamique

Les macros suivantes garantissent un alignement dans la mémoire allouée. Il est recommandé de les utiliser pour les allocations dynamiques.

NEW_STRING

La macro ***NEWSTRING*** retourne un pointeur sur l'espace mémoire alloué, et l'initialise en copiant le string qui lui est passé en argument.

```
ptr = NEW_STRING("ouf quel boulot");
```

NEW_ARRAY

La macro ***NEW_ARRAY*** retourne un pointeur sur l'espace mémoire alloué, dont la taille est le produit de la taille de l'objet par n.

```
ptr = NEW_ARRAY(n, toto);
```

NEW

La macro ***NEW*** retourne un pointeur sur l'espace alloué, dont la taille dépend de celle de l'objet qui lui est passé en argument.

```
ptr = NEW(tata);
```