INTER

Manuel de maintenance

Luc Weber
Observatoire de Genève

19 mars 1998

Table des matières

Chapitre 1

Introduction

Ce manuel est la suite logique du manuel de l'administrateur, l'installation des répertoires de base et des variables d'environnement doit être conforme à ce qu'il y est décrit.

La liste des manuels à disposition est la suivante:

manuel de l'administrateur décrit comment installer l'environnement pour le dévellopeur et les utilisateurs d'une application basée sur Inter ainsi que les divers utilitaires et librairies que cette application utilise.

manuel de maintenance destiné aux dévellopeurs d'application, ce manuel décrit comment créer une application.

manuel de l'utilisateur décrit les fonctionalités d'inter, le langage de commande, les procédures, l'utilisation des serveurs, etc...

manuel de référence décrit toutes les commandes et les fonctions. On y trouve aussi divers annexes.

quickref des fonctions petite références des fonctions.

Chapitre 2

Installation de l'*inter* de base

C'est le cas le plus simple ou aucun code n'est à écrire. Tout les fichier sont présent et sont utilisés.

A ce stade, il est toutefois possible de modifier:

- le nombre de matrices et le nombre de couches. Voir sous "matrice.par"
- la tailles des tableaux pour le stockage des variables locales. Voir sous "buffer.par"

Ces modification doivent être faites avant de lancer la construction d'Inter, qui sont:

```
> cd $INTERHOME
> setenv APP inter
> rm *.o *.a   (optionel)
> make
```

Chapitre 3

Installation d'un *inter* réduit

Un *inter* réduit est un *inter* composé des commandes interne et une partie seulement des commandes d'*inter*.

Comme ce n'est pas un *inter* standard, pour des raison de distribution et de mise à jour, il faut fabriquer une nouvelle application.

Dans le cas d'un *inter* réduit, fabriquer une nouvelle application est tres simple, car il n'y a pas de code à écrire. En plus des opérations citées dans le chapître précédente, il faut: (l'explication détaillée de l'emploi des fichiers et de la variable `$APP` est décrite dans le chapître suivant: Construction d'une application")

1. trouver le nom de l'application. On l'appelera ici: *Application*.

2. initialiser la variable d'envirronement `APP`.

3. créer et mettre à jour le fichier *Application*`.req` à partir de `inter.req`

4. créer un fichier vide nommé *Application*`.cmd.def`

5. créer un fichier vide nommé *Application*`.blk.def`

6. créer et mettre à jour le fichier *Application*`.blk.sup`

Le fichiers *Application*`.cmd.def` et *Application*`.blk.def` sont vides car ils décrivent les commandes ne faisant pas partie de la distribution. Ils sont utilisé lors de la contruction d'un *inter* avec commande supplémentaires. Voir sous "Construction d'une application".

Le fichier *Application*`.blk.sup` contient les variables globales que l'on désire voir apparaître dans le block de donnée. Il peut être vide.
Les opérations successives sont par exemple:

```
> cd $INTERHOME
> setenv APP "<application> inter"
> cp inter.req <application>.req
> <edit> <application>.req
> touch <application>.cmd.def
> touch <application>.blk.def
> <edit> <application>.blk.sup
```

Pour fixer les idées, si l'on désire faire une application nommée "gaff" qui utilise uniquement les commandes: aff, gsc, align et client, la variable `APP` s'initialise:

```
setenv APP "gaff inter"
```

le fichier `gaff.req` contient:

```
> cd $INTERHOME
> more gaff.req
AFF
GSC
ALIGN
CLIENT
```

Si cette application a besoin des variables (par exemple) USER (type caractère) et SIZMAX (type numérique), le fichier `gaff.blk.sup` contient:

```
> cd $INTERHOME
> more gaff.blk.sup
"USER__001:000     ; Nom de l'utilisateur
@SIZMAX001:        0.          ; nb de fichier traité
```

Le fichier "`Makefile`" est à modifier ainsi que les fichiers "Rules_<`OS`>" avant de construire l'application avec:

```
> cd $INTERHOME
> setenv APP "<application> inter"
> rm *.o *.a   (optionel)
> make
```

Chapitre 4

Construction d'une application.

Construire une nouvelle *Application* consiste à rassembler des éléments, provenant du noyau d'Inter, des commandes d'applications existantes si besoin et des commandes de sa propre application, dans le but de fabriquer une nouvelle *Application*.

Par exemple l'application Inter–CCD qui permet le contrôle du télescope Genevois de La Silla au Chili, possède l'interfaçage aux modules suivants:

- au télescope
- au Guide Star Catalogue (GSC)
- à un programme d'affichage d'image (AFF)
- à une connection au logiciel graphique interactif SuperMongo (SM)
- à l'ensemble des programme de réduction stellaire
- aux images et table MIDAS

Le noyau ainsi qu'une partie des commandes d'Inter–CCD peuvent être utilisés pour construire une nouvelle *Application*. Lors de la construction d'une application, on défini une variable d'environnement nommée **APP** contenant la liste des applications incluses en totalité ou en partie dans la nouvelle application.

pour construire l'application Tacos (exemple d'une autre application) on défini:

```
> setenv APP "tacos inter"
```

et pour construire une nouvelle application nommée zorglub qui utiliserai des commande de tacos et des commandes de zorglub on définirai:

```
> setenv APP "zorglub inter tacos"
```

On remarque que Inter doit toujours être nommé car il est toujours utilisé pour les commandes interne et que le nom de l'application doit être mis en première position.

Il faut noter que les 3 premières lettres de l'application serviront de préfixe aux fichier liés à l'application.

En cas de synonymes dans le nom des commandes des divers applications, l'ordre des applications dans `$APP` défini la priorité de sélection.

4.1 Fichiers de définition

La construction d'une application se fait à l'aide de scripts et de fichier (ASCII) de définition.
On reconnaît les fichiers:

Application`.req`	pour la liste de toutes les commandes locales d'une *Application*.
Application`.cmd.def`	pour la définition de toutes les commandes propres à une *Application*.
Application`.blk.def`	pour la définition de toutes les variables utilisées par les commandes propres à une *Application*.
Application`.blk.sup`	pour la définition des variables supplémentaires utilisées pour l'*Application* mais pas utilisées par les commandes.

Par exemple: chaque commande utilisée par Inter–CCD est décrite dans le fichier `inter.cmd.def` toutes les variables utilisée par ces commandes et seulement celle-là sont décrites dans le fichier `inter.blk.def` et les variables supplémentaires sont décrites dans le fichier `inter.blk.sup`.
Chaque application doit posséder ces trois fichiers ainsi que les fichiers *Application*`.*.def` des applications sitées dans `$APP`. Pour une application nommée "`tacos` ", on a `tacos.cmd.def, tacos.blk.def` et `tacos.blk.sup` en plus des deux fichier `inter.cmd.def` et `inter.blk.def` . On remarque que le fichier `inter.blk.sup` n'est pas utilisé car il est utilisé uniquement par l'application `inter` et non `tacos`.

Le but de ces divers fichiers est de générer les fichiers:

`block.ref`	qui contient la liste exacte des commandes demandées par l'application.
`command.ref`	qui contient la liste exacte des variables demandées par les commandes plus les variables liées à cette application.
`userd.par`	qui contient le code pour l'appel des commandes locales utilisant des variables du bloc.
`usere.par`	qui contient le code pour l'appel des commandes locales n'utilisant pas des variables du bloc.
`for/makefile`	qui est le makefile complet pour la compilation des commandes locales.

La génération de ces fichiers est faite par le **Makefile**. Qui regarde la liste de commande que désire l'*Application* dans le fichier *application*.req. Par exemple **inter.req** pour inter ou **tacos.req** pour tacos. Il contient, en colonne, la liste des commande locales, les commandes internes étant prises d'office.

Par exemple, le fichier nommé `tacos.req` contient:

```
> cd $INTERHOME
> more tacos.req
AELMS2D
AMPFIT
BACKGROUND
BERVDET
CONVOL
...
AFF
CLIENT
DESCR
EXTRACT
...
```

permettra la génération des fichiers **block.ref, command.ref, usere.par, userd.par** et **for/makefile** pour *tacos*.

4.1.1 Utilitaires de génération d'applications

Ces utilitaires sont des *shell scripts* lancés dans le **Makefile**:

`inicommandref.sh`	pour créer command.ref en fonction de *Application*`.req`
`iniblockref.sh`	pour créer `block.ref` en fonction de `command.ref`
`iniuserpar.sh`	pour créer `userd.par` et `usere.par` en fonction de `command.ref`
`iniformakefile.sh`	pour créer `for/makefile` en fonction de `command.ref`

Il faut remarquer que ces scripts peuvent s'utiliser indépendament les uns les autres.

Utilitaires annexes

Ces scripts sont utilisés par les scripts ci–dessus et sont listés ici juste pour information.

testenvapp.sh	test si l'environnement est prêt
collect.sh	extrait les rubriques compactées des fichiers *application*.cmd.def
collectblk.sh	extrait les variables dont ont besoin les commandes décrites dans command.ref
internelist.sh	sort les commande interne de inter (utilisé pour construire command.ref)
localelist.sh	sort les commande locale de inter
selectcmd.sh	sélectionne une commande et ses rubrique. Utilise TMP.command.ref crée par inicommandref
unused.sh	sort la liste de toutes les variables décrites dans *application*.blk.def mais pas utilisées dans command.ref
used.sh	sort la liste de toutes les variables décrites dans *application*.blk.def et utilisées dans command.ref

4.2 Les fichiers paramètres

Les fichiers paramètres (extension **.par**) décrivent l'environnement local d'une version d'INTER. Certains doivent être édités lors de la première installation de l'interpréteur pour contrôler si leur contenu correspond à la configuration des directories et au besoin qu'on attend du logiciel. Cinq d'entre eux (**usera.par, userb.par, userc.par, userd.par** et **usere.par**) sont des fichiers de type include qui vont s'insérer à des endroits précis de l'interpréteur pour permettre une modification locale de ses fonctions. Même si ces derniers fichiers ne sont pas utilisés, il faut tout de même qu'ils existent.

Les voici tous décrits.

4.2.1 matrice.par

C'est là que sont définis le nombre de matrices de travail et le nombre de couche. Ce fichier est lu par **inicode** lors de la génération de **common.cod**. **Une modification dans ce fichier nécessite une compilation complète de l'interpréteur et de toutes les commandes.**

4.2.2 buffer.par

On y définit le nombre de variables (taille des stacks) globales et locales que l'on désire utiliser en cours de session.

4.2.3 common.par

C'est un fichier décrivant les **common** internes à INTER. Il n'est pas à modifier.

4.2.4 usera.par

Il s'insert dans la source d'INTER au moment de la déclaration des variables.

4.2.5 userb.par

Il s'insert dans la source d'INTER au moment de l'initialisation des variables (DATA).

4.2.6 userc.par

Il s'insert dans la source d'INTER au début des instructions exécutables.

4.2.7 userd.par

Fabriqué automatiquement par le script **iniuserpar.sh** donc de devant pas être modifié manuellement, il s'insert dans la source d'INTER au moment de l'exécution des commandes locales n'utilisant pas le transfert automatique de variables du bloc.

4.2.8 usere.par

Fabriqué automatiquement par le script **iniuserpar.sh**, donc de devant pas être modifié manuellement, il s'insert dans la source d'INTER au moment de l'exécution des commandes locales utilisant le transfert automatique de variables du bloc.

4.2.9 usersave.par

Il s'insert dans la source d'INTER au moment de l'exécution du SAVE du bloc courant.

4.2.10 userblockrd.par

Il s'insert dans la source de subblock.f (subroutine: read_blk) apres la lecture d'un bloc standard.

4.2.11 userexit.par

Il s'insert dans la source d'INTER au moment de l'exécution du EXIT.

4.2.12 userquit.par

Il s'insert dans la source d'INTER au moment de l'exécution du QUIT.

4.2.13 handlerdec.par

Contient la déclaration des variables utilisées par les handlers de signaux.

4.2.14 handlerini.par

Contient l'initialisation des handlers de signaux.

4.2.15 socket.par

Contient la déclaration des variables utilisées pour les transferts interprocess.

4.2.16 pblock.par

Il définit la structure dans les blocs de données. C'est à dire la taille des variables, le nombre de chiffres pour les indices, la longueur des contenus des variables de type caractère et la position de ces divers éléments. **Une modification de ce fichier (pas recommandé) demande la restructuration manuelle de tout les blocs existants afin de les rendre compréhensible par INTER. De plus l'utilitaire inicode doit être recompilé et exécuté et INTER doit être recompilé**.

4.3 Génération complète d'une application

A ce stade l'application ne peut pas être compilée car le dévellopeur doit mettre à jour les fichiers de définitions et écrire le code des commande.

La création des commandes est décrites dans les chapîtres suivants.

Voici toutefois un exemple de marche à suivre pour génerer une l'application inter–tacos:

```
> cd $INTERHOME
> setenv APP "tacos inter"
> <edit> tacos.req
> <edit> tacos.cmd.def
> <edit> tacos.blk.def
> make
```

Chapitre 5

Notions de base concernant inter

Ce chapître décrit le fonctionnement de l'interpreteur dans les grandes lignes et devrait permettre au devellopeur d'application de comprendre la phylosophie Inter concernant la création des commandes. Le chapitres suivant explique comment on crée les commandes locales pour une *Application*.

5.1 Fonctionnement de l'interpréteur

Le travail d'INTER se résume en deux actions: l'interprétation puis l'exécution d'une commande.
Lors de l'interprétation, INTER détecte tout de suite les commandes spéciales ($,@, ...), les choix (CASE...ENDCASE), les labels (GOTO) et entreprend les actions adéquates. Toutes les autres commandes sont analysées selon un même algorithme. Cet algorithme décompose la ligne de commande en trois groupes: la commande, les paramètres avec leurs arguments et les qualificateurs avec leurs arguments. Ces groupes sont examinés et leurs contenus cont évalués pour finalement les laisser sous la forme d'un élément simple, c'est à dire un nombre, un vecteur ou une chaîne de caractères. (Remarque: l'analyse de la dernière commande peut être visualisée en tapant `???`. Chaque élément est stocké sous une forme codée dans une chaîne de caractères (nommée `sortie` de longueur `sortie_len`) commune à INTER et aux commandes d'applications. L'*Application*, grâce à un ensemble de subroutines peut retrouver facilement tout les éléments de la ligne de commande.
Une autre zone commune est dédiée aux variables transitant entre l'*Application* et les commandes. Ces variables ont été choisies au moment de la définition de la syntaxe de la commande (voir plus loin les fichiers `<applic>.cmd.def et command.ref`). Elles sont transférées de la zone de stockage interne du bloc de données à la zone commune automatiquement à l'appel de chaque commande. La commande les accède de façon naturelle (ce sont à ce moment là des variables FORTRAN vis à vis de la commande) et peut les retourner à INTER (mis à jour, résultat, ...). Si la commande renvoie un status indiquant que les variables de retour sont valables alors INTER les copie de la zone commune à la zone de stockage interne du bloc de données. Sinon cette zone n'est pas modifiée et reste dans l'état dans laquelle elle était avant la commande.
Toutes les zones communes décrites ci-dessus sont définies dans un `common` nommé `transfert` qui est définit dans le fichier include `common.cod`.

5.2 Inicode, les fichiers includes

A aucun moment de l'interprétation ou de l'exécution, INTER n'accède de fichier. Toutes les références dont il a besoin pour l'analyse des commandes ou pour l'accès au bloc de données sont stockées de

manière interne. INTER a été compilé avec la description des commandes et les pointeurs sur le bloc. Ces divers éléments sont contenus dans des fichier de type include qui sont insérés dans la source d'INTER au moment de la compilation.
De même, les commandes d'INTER reconnaissent certaines variables du bloc comme des variables internes. Cela est rendu possible grâce à un jeu d'équivalence entre les variables de la commande et la zone de transfert commune dans laquelle INTER a déposé une copie des variables à la fin de l'interprétation. Ces equivalences sont décrites dans les fichiers `<commande>.key`.
Les fichiers de type include sont crées automatiquement par l'utilitaire **inicode**. Celui-ci accède le fichier de description de commande (**command.ref**), le bloc de données de référence (**block.ref**) et les fichiers définissant l'environnement (d'extension **.par**). Il crée les fichiers **icomm.cod, dcomm.cod, iblock.cod** et **dblock.cod**, utilisés par l'interpréteur, le fichier **common.cod**, utilisé par l'interpréteur et les commandes, et les fichiers **<commande>.key** utilisés par les commandes. Attention les fichiers **command.ref** et **block.ref** sont des fichiers créés par les scripts `inicommandref.sh` et `iniblockref.sh` à partir des fichier `*.cmd.def` et `*.blk.def` et `<application>.blk.sup`, il ne faut donc pas les modifier (voir chapître "construction d'une application").

5.3 Le bloc de données

inicode ne reconnaît qu'un seul bloc: **block.ref** (généré par `iniblockref.sh`). C'est le bloc de référence. A l'intérieur apparaissent toutes les variables reconnues par INTER pour la version en cours. C'est lui qui sera utilisé pour la génération des fichiers de type include. On utilise des copies du blocs pour l'utilisation de /inter. Leur extension standard est `.blk`.

Le bloc peut comporter trois type de lignes:

1. les lignes de commentaires commençant par un point d'exclamation
2. les lignes vides
3. les lignes de déclaration de variables

Ces dernières comportent dans l'ordre: le nom de la variable, son indice, sa valeur puis des commentaires facultatifs. La position de ces éléments est fixe, elle est définie dans le fichier **pblock.par**. **En cas de modification du fichier pblock.par (pas recommandé), il est nécessaire de recompiler "inicode" puis de mettre à jour INTER:**

```
> make
```

5.3.1 Mise à jour du bloc du données

Le bloc de données peut être édité. **Mais il est absolument interdit d'ajouter ou de supprimer une variable.**

5.3.2 Blocs de données périmés

Les blocs de données périmés sont reconnus par INTER qui lit la variable caractère formé que de "A" (ex AAAAAAAAAA(1) pour des variables de 10 caractères) et qui contient la date de son dernier passage dans **inicode** (sous entendu la version d'INTER correspondante). Si cette date est différente, le fichier est comparé à **block.ref**. Les nouvelles variables sont alors rajoutés dans le fichier avec la valeur qu'elles ont dans le bloc de référence, celles qui n'ont plus cours sont éliminées et celles commununes aux deux fichiers sont conservées sans modification de leur valeur. Ce transfert automatique n'est pas possible pour les blocs qui ont changés de structure (ex: rallongement du nom des variables, ...), il faut alors le faire manuellement.

5.4 Description des commandes

Toute les commandes que reconnaît INTER ont leur syntaxe décrite dans un seul fichier: **command.ref** (créé par inicommandref.sh). A l'intérieur y sont décrites les commandes (voir chapître suivant: "création de commande")

5.5 Compilation de INTER, utilisation de make

La mis à jour d'INTER est faite par **make** et le fichier **makefile**.

Une mise à jour d'INTER se résume à:

1. génération de **block.ref** (iniblockref.sh)
2. génération de **command.ref** (iniblockref.sh)
3. génération de **userd.par** et **usere.par** (iniuserpar.sh)
4. génération de **for/makefile** (iniformakefile.sh)
5. compilation et link d'**inicode**
6. génération des fichiers ***.cod** (inicode)
7. compilation et link **Inter**

5.6 Les fichier spéciaux

Voici les fichiers qui peuvent être utilisés lors d'une session INTER.

5.6.1 block.ref ou *.blk

Ce sont les blocs de données. Il faut absolument avoir accès à l'un d'eux dès l'entrée dans INTER.

5.6.2 dbmerr.dir et dbmerr.pag

Ce sont les fichiers de la base de données des messages d'erreur (format dbm). Leur nom (sans l'extension) est contenu dans la variable **ERRNAM** ou pris par défaut si ce dernier n'existe pas (défaut initialisé dans **inter.inc**). Il est fabriqué à partir du fichier **erreur.texte** par le programme **create_dbmerr**.

5.6.3 *.prc

Procédure. Le path de recherche doit être contenu dans la variable **dirprc**.

5.6.4 hlp/*.hlp

Fichiers helps standard. Un autre path peut être donné par la variable **dirhlp** indiquant un autre directory où les trouver.

Chapitre 6

Création de commandes

Ce chapitre décrit pas à pas la procédure permettant de créer une commande fonctionnant sous INTER ainsi que les méthodes de programmation à utiliser.

6.1 La commande

Il faut décrire la syntaxe de la commande, c'est à dire déterminer tout les qualificateurs qui pourront inluencer son déroulement, qu'ils aient des arguments ou non. Par exemple:

```
/LIST  /NAME="FILE"  /PARAMETRES=1,2,3  /RECORD=START:1,END:20
```

il faut aussi déterminer les paramètres qu'elle attend, qu'ils soient obligatoires ou facultatifs. Cette syntaxe est alors écrite dans le fichier **command.ref** selon le mode d'emploi qui y est décrit.

Pour fixer les idées, prenons une commande nommée "RECHERCHE". Elle permet l'entrée des qualificateurs décrit plus haut et demande un nom de fichier (obligatoire) et un numéro (facultatif).

Dans le fichier **command.ref** nous écrirons:

```
0 RECHERCHE /LIST/NAME(,#01",)/PARAMETRES(,#03@,)-
            /RECORD(,START@,END@,)/   #01#01CN
```

Attention: si cette description doit se prolonger sur plusieurs lignes, on utilise le caractère de continuation '–' en fin de chaque ligne intermédiaire. De plus les lignes doivent tapées en majuscule et ne doivent pas contenir de tabulateur.

6.2 Les variables en transit

Il faut déterminer les variables de la commande qui doivent être lus ou sauvés dans le bloc de données. On distingue alors les variables numériques réels des variables caractères et les variables d'entrée de celles de sortie.

On reconnaît plusieurs groupes de variables:

- les variables numériques en sortie: ce sont les variables de type numérique qui sont passées de INTER à la commande.
- les variables caractères en sortie: ce sont les variables de type caractère qui sont passées de INTER à la commande.
- les variables numériques en retour: ce sont les variables de type numérique qui sont passées de la commande à INTER.
- les variables caractères en retour: ce sont les variables de type caractère qui sont passées de la commande à INTER.

Tout les groupes ne sont pas obligés d'être présent.
Par exemple, la commande "RECHERCHE" utilise les matrices de travail. Elle a donc besoin des paramètres de celles-ci en entrée. De plus elle va rendre une variable nommée "NTR" et utiliser la variable caractère "NAME" dans laquelle se trouve le nom d'un fichier à prendre par défaut si le qualificateur /NAME=<fichier> n'est pas donné. S'il est donné alors "NAME" sera mis à jour.

On rajoute donc au fichier **command.ref**:

```
1           XSTART-NX-XSTEP-YSTART-NY-YSTEP
2           NAME
3           NTR
4           NAME
```

Si certains des variables demandées par la commande n'existent pas, il faut les rajouter dans **block.ref** (selon l'ordre alphabétique).

6.3 Type de commande

Le type de commande définit l'emplacement où doit être codé l'appel à la commande. On le définit en désignant l'usage de la commande par une série de mots clés sur la ligne numéro 5. On a principalement:

```
5     LOCALE INTERPRETATION EXECUTION KEYWORD
```

Pour les commandes locales (crées pour une utilisation locale d'Inter) utilisant les variables de transferts. Dans ce cas l'appel est codé dans **userd.par**.

```
5     LOCALE INTERPRETATION EXECUTION
```

Pour les commandes locales (crées pour une utilisation locale d'Inter) n'utilisant pas les variables de transferts. Dans ce cas l'appel est codé dans **usere.par**.

6.4 Code de l'appel

Pour la commande "RECHERCHE", son code d'appel (dans `userd.par` ou `usere.par`) est simplement:

```
      call recherche()
```

si l'appel doit être différent (un autre nom ou il faut des arguments), on le code sur la ligne numéro 6 (syntaxe fortran):

```
6       call recherche(recherche_status)
```

6.5 Fichiers relatif à la commande

La ligne numéro 7 est maintenant obligatoire, elle contient le nom du ou des fichier dont a besoin la commande. Il faut se souvenir que les noms des fichiers ont comme préfixe les trois premières lettres du nom de l'application suivit d'une barre de soulignement. Par exemple "`int_`" pour /inter/ et "`tac_`" pour /tacos.
Par exemple, la commande "RECHERCHE" a besoin de "`int_recherche.f`" et "`int_fonction.f`".
La ligne 7 devient:

```
7 int_recherche.f int_fonction.f
```

Le fichier **command.ref** contiendra finalement:

```
0 RECHERCHE /LIST/NAME(,#01",)/PARAMETRES(,#03@,)-
            /RECORD(,START@,END@,)/   #01#01CN
1           NX-NY-XSTART-YSTART-XSTEP-YSTEP
2           NAME
3           NTR
4           NAME
5           LOCALE INTERPRETATION EXECUTION KEYWORD
6       call recherche(recherche_status)
7 int_recherche.f int_fonction.f
```

6.6 Génération des fichiers de type include

On lance simplement **inicode**, qui va mettre à jour les fichiers ***.cod** et créer (ou mettre à jour), dans notre cas, le fichier **recherche.key**. Voici un extrait de son contenu pour exemple:

```
      integer              size_i_NX
      parameter           (size_i_NX=0022)
      real                      i_NX(0022)
      equivalence(reel_org(0006),i_NX)

      integer              size_i_NY
      parameter           (size_i_NY=0022)
      real                      i_NY(0022)
      equivalence(reel_org(0028),i_NY)

      integer              size_i_NAME
      parameter           (size_i_NAME=1)
      character*(ils)           i_NAME
      equivalence(cara_org(0003),i_NAME)

      ...

      integer              size_o_NAME
      parameter           (size_o_NAME=1)
      character*(ils)           o_NAME
      equivalence(cara_org(0003),o_NAME)
```

On reconnaît ici la déclaration des variables locales et leurs équivalences à la zone de transit commune déclarée dans le **common transfert** lui-même défini dans **common.cod**. Le paramètre **ils** définissant la taille du contenu des variables de type caractères est le même que celui définit dans **pblock.par**.

On remarque aussi que les variables sont dimensionnées selon la dimension qu'elles ont dans le bloc.

6.7 Construction du programme

6.7.1 Variables prédéfinies

Le fichier **common.cod** définit en plus du **common transfert** les variables suivantes:

ILS	nb de caractères dans les variables de type caractère
SORTIE	tableau contenant la ligne de commande codée
SORTIE_LEN	longueur utile de cette ligne
REEL_ORG	zone de transit des variables
REEL_RET	zone de transit des variables
CARA_ORG	zone de transit des variables
BIDON_A	variable bidon
CARA_RET	zone de transit des variables
BIDON_B	variable bidon
NB_MAT_MAX	nb de matrices maximum
NB_PIX_MAX	nb de pixels actuellement alloués em memoire partagée
NB_COUCHE_MAX	nb de couches max
SHAMEM	si différent de zéro, indique que l'on utilise la mémoire partagée.
MATLOC(m,c)	adresse de chaque matrice
MATSIZ(m,c)	taille de chaque matrice
MATSHM(m,c)	flag indiquant si une matrice est en mémoire partggée
POINTER_MATRICE	pointeur de matrice à disposition
MATRICE	tableau associé à POINTER_MATRICE
TEST_PROG	vaut 1 si la commande "DEBUG /ON" a été tapée
ERREUR_PROG	doit être mis à zéro si la commande se termine bien

6.7.2 programme minimum

Voici l'allure d'un programme minimum.

On reconnaîtra l'initialisation d'un flag avec la variable TEST_PROG dont le contenu vaut "1" après la commande "DEBUG /ON" ou "0" après la commande "DEBUG /OFF". Ce flag peut servir afficher des messages en mode debug.

On remarquera également la mis à zéro de la variable ERREUR_PROG signifiant que le programme s'est bien déroulé. ERREUR_PROG vaut "1" à l'entrée de la routine et s'il n'est pas mis à zéro INTER affichera une erreur et mettra la variable de bloc ERREUR à "1". Si ERREUR_PROG est posé à "2" alors la variable du bloc EOF est également mise à "1".

```
      subroutine minimum

      include '../common.cod'
      include 'demo.key'
      logical flag,erreur

      flag = test_prog.eq.1.

      ....

      if (erreur) goto 9999

      ....

      erreur_prog = 0.
9999  return
      end
```

6.7.3 Accès à la ligne de commande

Les divers éléments décodés de la ligne de commandes peuvent être récupéré avec l'aide des routines suivantes. Chacune renvoie un flag logique nommé FLAG, pour indiquer si l'élément recherché existe et le contenu de celui-ci sous forme réel ou caractère selon les cas. Les exemples donnés ci-dessous montrent d'abord la ligne de commande tel qu'elle peut apparaître sous INTER et ensuite la manière de traiter l'élément dont il est question.

Récupération d'un qualificateur

Syntaxe: `CALL IFQUAL(qualificateur,flag)`

Ex: On veut voir si le qualificateur /LIST a été donné.

```
inter: > XXX /L

appel:   call ifqual('LIST',flag)
```

Récupération d'un argument de type numérique associé à un qualificateur

Syntaxe: `CALL IFQNNN(qualificateur,No,destination,flag)`

Ex: On veut récupérer l'argument No 2 du qualificateur /PARAMETRE et le placer dans UNITE

```
inter: > XXX /PAR=12,20,42

appel: call ifqnnn('PARAMETRE',2,unite,flag)
```

Récupération d'un argument de type caractère associé à un qualificateur

Syntaxe: `CALL IFQCNN(qualificateur,No,destination,longueur,flag)`

Ex:On veut récupérer l'argument du qualificateur /NAME et le placer dans O_NAME avec sa longueur dans ILEN

```
inter: > XXX /NAME="TEMPO.DAT"

appel: call ifqcnn('NAME',1,o_name,ilen,flag)
```

Récupération d'un argument de type inconnu associé à un qualificateur

Syntaxe: `CALL IFQNEX(qualificateur,next,dest_num,dest_car,longueur,type,flag)`

Ex: On veut récupérer successivement les arguments /COLUMNS et le placer dans O_NAME avec sa longueur dans ILEN si le type est caractère ou dans O_NO s'il est numérique. TYPE indique si le résultat est numérique (TRUE) ou s'il est de type caractère (FALSE). NEXT doit impérativement être à 1 pour la première recherche. Il représente une position de caractères pour la routine. Il ne doit pas être modifié.

```
inter: > XXX /COLUMNS="ALPHA",2,4,"DELTA"

appel: call ifqcnn('NAME',next,o_no,o_name,ilen,type,flag)
```

Récupération d'un argument de type numérique associé à un keyword et un qualificateur

Syntaxe: `CALL IFQNKW(qualificateur,keyword,destination,flag)`

Ex: On veut récupérer l'argument du keyword START du qualificateur /RECORD et le placer dans START

```
inter: > XXX /RECORD=START:12,END:20

appel: call ifqnkw('RECORD','START',start,flag)
```

Récupération d'un argument de type caractère associé à un keyword et un qualificateur

Syntaxe: `CALL IFQCNN(qualificateur,keyword,destination,longueur,flag)`

Ex: On veut récupérer l'argument du keyword UNIT du qualificateur /OUT et le placer dans UNIT_OUT avec sa longueur dans ILEN.

```
inter: > XXX /OUT=UNIT:PRINTER

appel: call ifqcnn('OUT','UNIT',unit_out,ilen,flag)
```

Récupération d'un paramètre

Syntaxe: `CALL IFPARA(No,destination,longueur,flag)`

Ex: On veut récupérer le paramètre 1, le placer dans la chaîne TEMPO et sa longueur dans ILEN. Remarque: Les paramètres numériques sont aussi passés sous formede chaîne de caractères et on les récupère avec l'instruction:

```
inter: > XXX FILE.DAT

appel: call ifpara(1,tempo,ilen,flag)
       read (tempo(1:ilen),*) param
```

6.7.4 Accès aux variables

Les variables passées à une commande sont déclarées et équivalencées à la zone de transit commune dans le fichier **<commande>.key** lui-même crée par le fichier **inicode** à la lecture de **command.ref**. Leur nom est préffixé de "**I_**" pour les variables provenant d'Inter et "**O_**" pour les variables qu'on lui retourne.
Pour la commande, les variables de sortie et de retour sont des entités différentes même s'ils référencient les même variables du bloc. **C'est au programmeur de faire transiter les valeurs de sortie sur celles d'entrée pour les variables qui ne sont pas utilisées**.

Par exemple, pour une variable nomme NX ,celle-ci s'appellera I_NX dans le programme (attention, les variables sont toujours des réels ou des chaînes de caractères) et si elle est modifiée, sa nouvelle valeur devra être stockée dans O_NX. Certains variables étant dimensionnées, dans le cas d'une modification que de l'une d'entre elles, les autres devront tout de même transiter sans plus de modification.

Exemple: Une commande pose le nombre de colonne de la matrice 2 à deux fois le nombre de colonnes de la matrice 1. On a donc avec **NB_MAT_MAX** (égal au nombre de matrices déclarés dans **common.cod**):

```
C
C    mise à jour des variables non-modifiees
C
     DO I=1,NB_MAT_MAX
       O_NX(I) = I_NX(I)
     ENDDO
C
C    modification desiree sur le variable NX(2)
C
     O_NX(2) = 2*I_NX(2)
```

6.7.5 Accès aux matrices

Dans la plupart des cas, le numéro de matrice est donné comme paramètre sur la ligne de commande (codé par exemple [2,4]). Dans ce cas, on le récupère avec (No étant le numéro de paramètre):

```
call read_mat_nb_param(no,flag,no_mat,no_couche,ok)
```

Cet appel retourne flag à true si le paramètre est présent, le numéro de matrice, le numéro de couche et un flag indiquant si le paramètre est bien un numéro de matrice. Si le paramètre est absent, le

Numéro de matrice est le numéro de couche vallent 1.

Une fois que le numéro de matrice et le numéro de couche sont connus, on demande un accès sur la matrice en précisant la taille désirée avec:

```
call getmat(no_mat, ncouche, ncol, nline,
            creat, save, isize, iptr, erreur)
```

Deux cas sont possibles selon que la commande à besoin d'une matrice (ex: MAT, SHOW, ...), ou que la commande crée la matrice (ex: EXTRACT, SET, ...). Cette distinction est faite avec la variable logique **creat** qui indique par .TRUE. une création.

Si une matrice existe (donc avec une taille donnée) mais que le nouveau nombre de pixels est différent, la matrice est détruite et réallouée. La variable logique **save** indique par .TRUE. que les données que la matrice contenait sont à sauvegarder.

La variable **isize** indique la taille réellement allouée pour la matrice. En effet, si la matrice est en mémoire partagée, sa taille ne peut pas être modifiée et le programmeur doit tenir compte de cette possibilité.

La variable **iptr** est un pointeur sur le premier pixel de la matrice. Il permet d'utiliser la notion de POINTER Fortran exemple:

```
      ...
      real mat(1)
      pointer  (ptr,mat)
      ...
      call getmat(...,ptr,...)
      if(erreur)...
      ...
      call sub(mat)
      ...
```

Il faut toutefois remarquer que le tableau **MATRICE** est à disposition et pointe la matrice accédée par le dernier appel à getmat. Ainsi dans la plupart des cas, où une commande n'accède qu'une matrice, la suite des opératiuons est la suivante.

Exemple avec utilisation de la matrice K utilisée par la commande EXTRACT:

```
      call read_mat_nb_param(no,flag,no_mat,no_couche,ok)
      ...
      call getmat(no_mat, no_couche, nc, nl, .true., .false.,
     .            isize, iptr, erreur)
      if(erreur)goto 9999
      if(nc*nl .gt. isize)goto 9007
c
c     appel à la routine
c
      call extr_mat (matrice,nc,nl,...)
      o_nx(k) = nc
      o_ny(k) = nl
      ...
      end

      subroutine xxxxx (mat,nx,ny,...)
      integer nx,ny
      real    mat(nx,ny)
      ...
      return
      end
```

Pour une commande utilisant plusieurs matrices (par exemple une matrice et une autre prise sur la couche suivante) le code ressemble à:

```
      real mat1(1), mat(2)
      pointer  (ptr1,mat1), (ptr2,mat2)
      ...
      call read_mat_nb_param(no,flag,no_mat,no_couche,ok)
      ...
      call getmat(no_mat, no_couche, nc, nl, .true., .false.,
     .            isize, ptr1, erreur)
      if(erreur)goto 9999
      call getmat(no_mat, no_couche+1, nc, nl, .true., .false.,
     .            isize, ptr2, erreur)
      if(erreur)goto 9999
      ...
c
c     appel à la routine
c
      call extr2_mat (mat1,mat2,nc,nl,...)
      ...
      end
```

Ce type d'appel est généralisé dans le logiciel INTER.

6.8 Messages d'erreur, warning et info

Les messages sont stockés dans une base de données au format dbm. Cela permet de dissocier les textes du code et permettre une version dans une autre langue. Cette base est fabriquée ou mise à jour par l'utilitaire **create_dbmerr** qui lit le fichier `inter.err.fr` et tout autre fichier donné sur la ligne de commande. Exemple:

```
create_dbmerr tacos.err.fr
```

met à jour `dbmerr.pag` et `dbmerr.dir` en lisant `inter.err.fr` et `tacos.err.fr`. Cette commande est lancée par **make** lors de chauqe compilation.

Ces fichiers de messages contiennent une clé suivit d'un espaceou plusieurs <TAB> puis du texte. Le texte doit être un format (syntaxe C) pour être utilisé avec les routines `int_errare`, `int_warning`, `int_info` et `get_message_db`. Exemple:

```
int_argcar    L'argument No %d doit être de type caractère
int_argkwd    Le qualificateur %s demande un argument
int_argtab    Le tableau (argument No %d) n'a pas une taille suffisante
int_attret    Le compilateur attend la déclaration RETURN
int_badunt    Unite logique interdite!  Utilisez %d, <= unit <= %d
int_colunk    La colonne "%s" n'existe pas dans le fichier "%s"
```

L'utilitaire **find_dbmerr** affiche les messages relatifs aux clés donnés sur la ligne de commande. Exemple:

```
> find_dbmerr int_argcar zorro int_colunk
int_argcar: L'argument No %d doit être de type caractère
zorro --> unknown !
int_colunk: La colonne "%s" n'existe pas dans le fichier "%s"
```

Les messages donnés sous forme de format sont affichés avec les routines décrites dans **suberr.f**. Elles sont décomposées en trois groupes:

`int_errare`(key[,params])	affiche le message en reverse vidéo et met le flag ERREUR à TRUE (voir common.cod)
`int_warning`(key[,params])	affiche le message en reverse vidéo sans modifier le flag ERREUR
`int_info`(key[,params])	affiche le message
`int_prompt`(key[,params])	affiche le message sans saut de ligne

Les messages peuvent être recherché avec routine:

`get_message_db(key, message, ilen)`	recherche du message

6.8.1 L'accès aux images MIDAS

L'accès aux images est incomplet sous MIDAS dans le sens qu'il est nécessaire à une sous-routine ouvrant une image de la refermer pour qu'une autre puisse l'accéder. Dans le cas où elle n'est pas refermée, chaque appel d'ouverture va réserver une nouvelle place mémoire et ainsi, les routines ne travailleront pas sur le même fichier. Sous INTER les images ne sont pas refermée pour des questions de rapidité. Un interface a été crée pour remédier aux problèmes décrits ci-dessus.
Il fonctionne de la sorte:

Chaque appel standard (**PUTIMAG_ST, GETIMAG_ST** et **CLOSE_ST**) à été remplacé par un appel intermédiaire (**PUTIMAG, GETIMAG** et **CLOSEIMAG** dans **for/intermidas.f**) gérant le nom et les paramètres des images que les subroutines sont en train d'accéder. Ces paramètres sont stockés dans un buffer interne de six positions où les images nouvellement ouvertes y sont notifiées. Lorsqu'une routine demande l'ouverture d'une image, le buffer est lu pour déterminer si cette image a été ouverte. Si elle l'est alors seuls ses paramètres sont retourné, si elle ne l'est pas, alors l'image est ouverte et ses paramètres sont écrit dans le buffer. Si le buffer est plein, l'image la plus ancienne est fermée et la nouvelle prend sa place.

Les arguments des trois routines de cet interface sont les mêmes que ceux des routines standards décrites dans le manuel "MIDAS Environment".

6.8.2 Utilisation de la mémoire partagée

La variable SHAMEM indique à Inter, si elle est différente de zéro, que la mémoire à allouer pour les matrices est à faire en mémoire partagée (voir "shmget", "shmat"). Si SHAMEM vaut zéro alors Inter effectue un malloc standard.

Les segment identifiers de la shared memory sont identifié par le contenu de SHAMEM et incrémenté pour chaque nouveau segment (un segment ne peut contenir que 1MB).

Si on change la taille des matrice (MAT /SET) ou si l'on modifie SHAMEM, Inter détruit les segments auxquels il était attaché et réalloue de la shared memory.

Pour effacer de la Shared Memory, il existe que deux manières: mettre SHAMEM à zéro puis effectuer un MAT /SET, ou utiliser la commande système:

```
> ipcrm -m $<$id1$>$ -m $<$id2$>$ ...
```

On peut aussi utiliser la commande maison "ipckill" qui contient:

```
#!/bin/csh
# tue tout les shared memories,semaphores et message queues
# Luc Weber 3/11/92

foreach n (`ipcs | grep ^m | awk '{print $2}'`)
echo "Killing shared memory id=" $n
ipcrm -m $n
end

foreach n (`ipcs | grep ^s | awk '{print $2}'`)
echo "Killing semaphore id=" $n
ipcrm -s $n
end

foreach n (`ipcs | grep ^q | awk '{print $2}'`)
echo "Killing message queue id=" $n
ipcrm -q $n
end
```

6.9 Documentation de la commande

La doc est écrite dans un fichier *Commande*.inc. Si la commande utilise des variables du bloc en transit il faut inclure le fichier *Commande*.var.inc. Ce fichier est fabriqué automatiquement par le script updatedoc.sh.

Chapitre 7

Fabrication de la documentation

La documentation se trouve dans le répertoire `$INTERHOME/tex`.
La mise à jour de la documentation et des utilitaires de documentations se fait avec le script `$INTERHOME/copy_documentation.sh`

Les nom fichiers dans ce directory respecte la syntaxe suivante avec par exemple pour inter:

Nom générique	Document	Exemple
Préfixe_Commande`.inc`	fichier de description de commande	int_matrix.inc
Application_Nom`.tex`	manuel	inter_reference.tex
Application_Nom`.inc`	chapître, annexe, ...	inter_introduction.inc
`util_`*Nom*`.tex`	partie de manuel	util_quickref.tex
`tmp_`*Nom*`.*`	fichier crées automatiquement	tmp_command.inc
Application`_description.var`	description des variables	inter_description.var

Le terme *Préfixe* est un mot composé de 3 lettres désignant le nom de l'*Application* de manière abrégée. Par exemple "`int`" pour Inter et "`tac`" pour Inter–Tacos.

7.1 Manuels de base

La documentation est constituées par les manuels suivant écrits en LaTex:

Application`_user.tex`	guide de l'utilisateur
Application`_reference.tex`	manuel de référence
Application`_administration.tex`	manuel de l'administrateur
Application`_maintenance.tex`	manuel de maintenance

L'usage des fichier "inclus" est géneralisé ainsi que la génération automatique de certain de ceux-ci.

Les fichiers se trouvant sous (`$INTERHOME/tex`) sont les suivants:

7.1.1 documentation liée aux commandes écrites par le dévellopeur

Préfixe_Commande`.inc`	description d'une commande (inclus dans tmp_command.inc et dans *commande*.tex)
Application`.description.var`	descriptions de toutes les variables du bloc associées à une *Application*.

7.1.2 Documentation automatique

Les fichiers generés automatiquement ne doivent naturellement pas être modifiés manuellement. Ce sont:

Préfixe_Commande`.tex`	permet de sortir la documentation d'une seule commande. Créés par inicommandtex.sh.
Préfixe_Commande`.var.inc`	tableau des variables utilisées par une commande. Créés par makevarinc.sh, modifié par ajustmakevarinc.sh.
Préfixe_Commande`.var.short.inc`	tableau des variables utilisées par une commande sous forme compressée (sans la short description). Créés par shortvarinc.sh à partir de *commande*.var.inc.
`tmp_command.inc`	description de toutes les commandes (chapitre dans manref.tex). Créé inicommandinc.sh.
`tmp_description.var`	macros de description de toutes les variables. Créé par var2cmd.sh.
`tmp_vardescription.inc`	description de toutes les variables. Créé par makevardescription.sh.
`tmp_varxref.inc`	référence croisée (variables–commandes). Créé par makevarxref.sh.
`tmp_fonctions.sorted`	liste des fonctions par ordre alphabétique (quickref). Créé par quick.sh.

7.1.3 Fichier liée aux manuels:

Les fichier suivants se trouvent aussi dans le répertoire `$INTERHOME/tex`:

cmd.tex	macros latex pour les manuels
inter_fonctions.inc	descriptions de toutes les fonctions (inclus dans operations.inc)
util_quickref.tex	permet de créer juste le quickref des fonctions
util_crossref.tex	permet de créer juste la liste des référence croisée

7.2 Utilitaires pour la préparation de la documentation

Ces utilitaires permettent de créer les fichiers nécessaire à une applications lors de la création initiale de la documentation.
Ce sont:

iniinclude.sh	génère tout les fichiers *Préfixe_Commande*`.inc`, uniquement s'ils n'existent pas.
inidescriptionvar.sh	*Application*`.description.var.inc` à partir de *Application*`.blk.def`

7.3 Utilitaires de génération de documentation

Les principaux utilitaires de génération automatique sont:

make	avec le fichier makefile, génère *Application*`.reference.dvi` avec la liste des commandes, le quickref et les références croisées des variables et aussi permet de générer tout les fichiers *commandes*`.dvi`.
makehlp.csh	fabrique tout les fichiers de help
dvi2hlp.sh	fabrique un fichier de help

Les utilitaires secondaires sont:

fct2hlp	filtre utilise par dvi2hlp (fabriqué à partir du fichier "lex" fct2hlp.l)
fct3hlp.sh	fabrique la liste des fonctions en ASCII pour le help
inicommandinc.sh	fabrique tmp_command.inc
inicommandappinc.sh	fabrique tmp_command.inc avec uniquement les commandes spécifiques à l'application. (voir Makefile)
inicommandtex.sh	fabrique un ou tout les *Préfix_Commande*.tex
makedefi.sh	fabrique un ou tout les ../tmp/*Préfix_commande*.defi
makevarinc.sh	fabrique un ou tout les *Préfix_commande*.var.inc
ajustmakevarinc.sh	met à jour un ou tous les fichiers "inclus" *commande*.var.inc en enlevant les variables déjà décrites dans *commande*.inc
shortvarinc.sh	fabrique un ou tous les fichiers "inclus" *commande*.var.short.inc à partir des fichiers *commande*.var.inc en decrivant les variables sur 2 colonnes au lieu d'une et en enlevant la short description
makevarxref.sh	fabrique tmp_varxref.inc
quick.sh	fabrique tmp_fonctions.sorted
var2cmd.sh	concatene tout les fichier *.description.var en un fichier tmp_description.var où les variables deviennent des macros de description en latex. Attention les chiffres contenus dans les variables sont convertis en leur équivalent en anglais. (ex: "ZONE1" devient "ZONEone")
makevardescription.sh	fabrique le chapitre tmp_vardescription.inc qui est la description de toutes les variables du bloc
liste_include.csh	fabrique la liste des commandes décrites pour le Makefile.

7.4 Opérations à suivre pour générer la documentation

Remarque: l'application doit être à jour, et doit ce compiler sans problème. Donc `make` doit avoir été lancé sous `$INTERHOME` et le repertoire `$INTERHOME/tmp` ne doit pas avoir été purgé.

Le fichier `Makefile` doit être édité pour donner les arguments spécifiques à l'application.

7.4.1 Pour générer la doc complète

```
> cd $INTERHOME/tex
> make inter_reference.dvi
> latex inter_reference;latex inter_reference
> dvips inter_reference
> latex inter_user;latex inter_user
> dvips inter_user
> latex inter_administration;latex inter_administration
> dvips inter_administration
> latex inter_maintenance;latex inter_maintenance
> dvips inter_maintenance
```

7.4.2 Pour générer le quickref des fonctions

```
> cd $INTERHOME/tex
> quick.sh
> latex util_quickref
> dvips util_quickref
```

7.4.3 Pour générer la doc d'une commande (nouvelle ou existante)

```
> cd $INTERHOME/tex
> <edit> <prefixe>_<commande>.inc
> make cmd=<prefixe>_<commande>
> dvips <prefixe>_<commande>
```

7.5 Opérations à suivre pour générer les fichiers d'aide

7.5.1 Pour générer tout les messages de help

```
> cd $INTERHOME/tex
> makehlp.csh
> fct3hlp.sh
```

7.5.2 Pour générer un message de help

```
> cd $INTERHOME/tex
> dvi2hlp.sh <prefixe>_<commande>
```

7.6 Descriptions des fonctions

La descriptions des fonctions est écrite dans `$INTERHOME/tex/inter_fonctions.inc`. Cette description permet de mettre à jour: le manuel de référence, le quick référence des fonctions et le fichier de help des fonctions `$INTERHOME/hlp/fct.hlp`. Il y a donc un seul endroit où l'on doit apporter des modifications!

Chapitre 8

Résume des opérations de création d'une commande

Voici la liste des opérations à suivre pour ajouter une commande. L'application (Inter–CCD, Tacos, ...) est notée *Application* et la commande *Commande*.

1. Entrer le nom de la commande dans ”*Application*.def”
2. Entrer les nouvelles variables qu'utilise *Commande* dans ”*Application*.blk.def”
3. Entrer les descriptions de ces variables dans ”tex/*Application*.description.var”
4. Entrer la syntaxe et les fichiers associé à *Commande* dans ”*Application*.cmd.def”
5. Ecrire le code de la commande dans ”for/<`prefix`>*Commande*.f”
6. Lancer: ”make”
7. Ecrire la doc de la commande dans ”tex/<`prefix`>*Commande*.inc” à partir du modèle: ”skeleton.inc”
8. Mettre à jour le manuel de référence avec: ”make inter_reference”
9. Mettre à jour le fichier de help avec: ”dvi2hlp.sh *Commande*”

Chapitre 9

Modification de la longueur des variables du bloc de donnée

Le fichier à changer sont:

- pblock.par (changer IF, I1, I2, IC, TEMPLATE_KW_NUM, TEMPLATE_KW_CAR)
- inter.cmd.def (variable AAAAAAA)
- inter.blk.def (*.blk.def) changer la taille des variables
- inter.blk.sup (*.blk.def) changer la taille des variables
- iniblockref.sh (variables AAAAAAA et PROMPTER)
- unused.sh (les commande cut)
- used.sh (les commande cut)
- tex/makevardescription.sh (la commande cut)
- tex/inidescriptionvar.sh (la commande substrdans awk)
- tex/var2cmd.sh (la commande cut)