Outil de mise au point et de surveillance d'applications, activable sous un système transactionnel

Jacques Coche

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE AGREE
DE GRENOBLE (C.U.E.F.A)

MEMOIRE

présenté en vue d'obtenir

le diplôme d'Ingénieur
du Conservatoire National des Arts et Métiers

en

Informatique

par

Jacques COCHE

OUTIL DE MISE AU POINT ET DE SURVEILLANCE D'APPLICATIONS, ACTIVABLE SOUS UN SYSTEME TRANSACTIONNEL

SOUTENU LE : le 3 Décembre 1979

JURY

Président : L. BOLLIET

Membres : F. BERGER
C. EUZET
C. CAMOZZI
C. HERNANDEZ
M. SCHLUMBERGER

Le travail présenté dans ce mémoire a été entrepris à la SEMS (Société Européenne de Mini-Informatique et Systèmes), dans le cadre de la formation permanente.

J'adresse mes remerciements :

A Monsieur C. KAISER, Maître de conférence au Conservatoire National des Arts et Métiers (CNAM), qui m'a fait l'honneur d'accepter le dossier de Thèse,

Au Président du jury, Monsieur le Professeur L. BOLLIET, Directeur du département informatique à l'Institut Universitaire de Technologie de Grenoble, pour ses conseils lors de la préparation de ce mémoire,

A Monsieur D. MANUELLO, Chef de Centre à la SEMS, région grenobloise, qui, grâce à l'intérêt qu'il porte à la formation permanente, m'a donné toutes les facilités pour réaliser ce travail,

Aux membres du jury :

Messieurs F. BERGER, Professeur au CNAM et C. EUZET, Enseignant au Centre Universitaire d'Education et de Formation des Adultes (CUEFA) pour avoir accepté de juger ce travail,

Messieurs C. CAMOZZI, Directeur des études logiciel et C. HERNANDEZ, Chef de service à la SEMS, pour les suggestions qu'ils m'ont formulées,

Monsieur SCHLUMBERGER, Ingénieur principal à CAP-SOGETI-GESTION à Grenoble, de l'aide qu'il a fournie à la rédaction de ce mémoire.

Je tiens à remercier aussi :

Mes camarades de l'équipe de développement de systèmes transactionnels, qui ont aidé à la réalisation pratique de cette étude, et plus particulièrement Monsieur R. SACCO, Chef de projet,

Tous ceux qui ont participé amicalement à ce travail et notamment Mesdames A. RUBIN et N. BOURDIGNON ainsi que Messieurs G. ENTRESSANGLE, J.P. GAYET, P. MARTIN, P. MORAND, N. PEYSSON et P. VALENTIN.

Je ne saurais oublier les encouragements de ma femme durant les moments difficiles traversés au cours de ces années de formation permanente.

SOMMAIRE

INTRODUCTION PAGES

CHAPITRE I : LES OUTILS D'AIDE A LA MISE AU POINT DE PROGRAMMES 1

I.1. - LES CARACTERISTIQUES DES OUTILS D'AIDE A LA MISE AU POINT 2

I.1.1. - TECHNIQUES DE MISE AU POINT 2

I.1.2. - METHODES DE MISE AU POINT 5

I.1.3. - FONCTIONS GENERALES 7

I.1.4. - CONCLUSION 11

I.2. - LES LANGAGES DE PROGRAMMATION 12

I.2.1. - MISE AU POINT DE PROGRAMMES ECRITS EN LANGAGE ASSEMBLEUR 12

I.2.2. - MISE AU POINT DE PROGRAMMES ECRITS EN LANGAGE EVOLUE 13

I.2.3. - CONCLUSION 14

I.3. - LES OUTILS D'AIDE A LA MISE AU POINT DANS LEUR CONTEXTE D'EXECUTION 15

I.3.1. - CONTEXTE MONOPROGRAMMATION 15

I.3.2. - CONTEXTE MULTIPROGRAMMATION 15

I.3.3. - CONCLUSION 17

CHAPITRE II : PRESENTATION DE L'OUTIL DE MISE AU POINT REALISE 18

II.1. - OBJECTIF 18

II.2. - CONTEXTE D'EXECUTION 19

II.2.1. - PRESENTATION DU SYSTEME D'EXPLOITATION MUTEX 19

II.2.2. - LIEU D'IMPLANTATION DES PROGRAMMES A TESTER 22

II.2.3. - GESTION DES PAGES SOUS MUTEX 22

II.2.4. - GRAPHE D'ETAT D'UN PROCESSUS 26

PAGES

II.3. - CARACTERISTIQUES DE L'OUTIL DE MISE AU POINT 28

II.3.1. - FONCTIONS REALISEES 29

II.3.2. - EXTENSIONS POSSIBLES 35

II.3.3. - MODE D'EXECUTION DU MODULE EN MISE AU POINT 36

CHAPITRE III : REALISATION SUR SOLAR 37

III.1. - GENERALITES SUR LES COMMANDES 37

III.2. - DESCRIPTION DES COMMANDES 41

III.3. - LES TABLES 57

III.3.1. - LES TABLES DE SYMBOLES 57

III.3.2. - LA ZDCPARAM 63

CHAPITRE IV : PRINCIPE DE FONCTIONNEMENT DE DEBUG 69

IV.1. - FONCTIONNEMENT DES DIFFERENTS MODULES 70

IV.2. - INTERFACES AVEC LE SYSTEME MUTEX 94

CONCLUSION 101

GLOSSAIRE G.1

ANNEXES A.1

BIBLIOGRAPHIE B.1

NOTATIONS UTILISEES

- Les nombres précédés du caractère "'" ou "&" sont des nombres exprimés en hexadécimal.

- Dans l'écriture d'une commande, seuls les trois premiers caractères sont obligatoires :

  Exemple : SUPPRIMER '026A est équivalent à :

  SUP '026A.

- Le caractère "> " signifie que la commande émise sera interprétée par le système MUTEX.

- Les caractères " >> " signifient que la commande émise sera interprétée par le système DEBUG.

[ ] : indique une option dans la syntaxe d'une commande.

{ } : indique un choix obligatoire de l'un des éléments contenus dans l'accolade.

( ) : un nombre entre parenthèses signifie une référence à la bibliographie.

<< : zone commentaire.

INTRODUCTION

La fiabilité d'une application se mesure par une probabilité de défaillance sur une période de temps définie.

Une assez bonne fiabilité peut être obtenue en utilisant des méthodes de programmation telles que la programmation modulaire, la programmation structurée. L'utilisation de ces méthodes facilite les modifications, permet une localisation plus aisée des erreurs et en diminue les risques sans toutefois les annuler.

Notre objectif a été la réalisation d'un outil facilitant la détection d'erreurs au niveau de la programmation d'un algorithme.

Nous l'appelons "DEBUG". Il est conçu pour des applications s'exécutant sous le système d'exploitation MUTEX, écrites en langage FORTRAN, ASSEMBLEUR ou PL16 (01) - (02) - (04).

Il s'est voulu simple (il a été consacré un an de travail à une personne pour le développer).

Il s'agit d'un outil interactif de haut niveau permettant le contrôle continu de plusieurs modules à la fois, l'ensemble de ces modules formant l'application à mettre au point.

Sa mise en oeuvre est immédiate.

Son fonctionnement ne nécessite aucune modification des programmes sources (pas de recompilation due à l'utilisation de DEBUG).

Son activation et le dialogue homme-machine par commandes interprétées sur n'importe quel périphérique de la configuration, nous laissent espérer qu'il sera souvent utilisé.

Nous donnons dans le premier chapitre la raison du choix de cet outil. Ses fonctionnalités ainsi que son contexte d'exécution sont définis dans le chapitre suivant. Ce qui suit est réservé à l'analyse et à la programmation de DEBUG.

CHAPITRE I

LES OUTILS D'AIDE A LA MISE AU POINT DE PROGRAMMES

L'expérience industrielle montre qu'une application en fin de développement comporte encore des erreurs qui apparaîtront en cours d'exploitation.

En conséquence, la mise au point est un facteur essentiel à prendre en compte dans la conduite d'un projet.

Le programmeur doit choisir ces tests le plus judicieusement possible. La mise en place de ceux-ci sera facilitée bien sûr, par l'utilisation d'un outil "efficace" et "souple".

Nous constatons que l'écriture des programmes d'une application suit une logique liée au système d'exploitation.

Ce dernier est construit en fonction des possibilités de la machine sur laquelle il s'exécute. Ceci explique que les outils de mise au point développés aujourd'hui sont dépendants des systèmes d'exploitation et des calculateurs sur lesquels ils s'exécutent. En voici quelques exemples :

- PROGRAID sur T2000 (13) - DRIP16 sur SOLAR (15)
- AID sur T1600 et SOLAR (14) - DDT sur PDP-6 (18)
- ODT sur PDP-11 (16) - DEBS2 sur MITRA 125 (09)
- DEBU2 sur MITRA 125 (10) -DEBUGGING SYSTEM DE CP/CMS (23)

Nous allons voir que même si ces logiciels sont difficilement portables, ils peuvent être construits sur un certain nombre de concepts généraux, de techniques déjà expérimentées.

I.1. - LES CARACTERISTIQUES DES OUTILS D'AIDE A LA MISE AU POINT

I.1.1. - TECHNIQUES DE MISE AU POINT

Il existe aujourd'hui trois techniques de mise au point selon le type du système d'exploitation sur lequel se déroule l'application.

- La mise au point aux clés, convient pour des programmeurs connaissant parfaitement la machine, le contexte d'environnement de l'application (le système d'exploitation). Cette technique est parfois utilisée par les concepteurs de systèmes sur mini-ordinateurs, ou pour des programmes fonctionnant en autonome.

- La mise au point en mode d'enchaînement de travaux appelé aussi "batch" impose au programmeur de spécifier tous les tests à accomplir avant l'exécution des programmes et souvent même avant la compilation (Ex: DEBUG FORTRAN et PL/1 sur IBM, DRIP16 sur SOLAR).

- La mise au point en mode conversationnel appelé aussi "interactif" donne la possibilité au programmeur par l'intermédiaire d'un terminal, d'observer l'exécution de son programme et d'intervenir au moment où il le désire de manière à contrôler l'exécution.

Ces deux dernières techniques se trouvent souvent en opposition quand il s'agit de définir un logiciel de mise au point. La technique interactive représente des avantages importants dans certains types d'application (saisie de données par exemple) : mise au point rapide, souplesse d'utilisation.

Cependant elle favorise un manque de méthodes dans la définition des tests. De plus elle risque d'amener d'autres erreurs dans la mise au point d'applications fonctionnant en temps-réel (problème de sychronisation).

Une comparaison entre ces deux techniques a été faite au niveau des coûts par M. SACKMAN - W.J. ERICKSON - E.E. GRANT (figure 1). Il en ressort qu'un outil interactif nécessite moins de "temps homme" qu'un outil batch , mais accroît le temps machine.

MISE AU POINT EN	PREMIER PROGRAMME		DEUXIEME PROGRAMME	
	Conversationnel	Batch	Conversationnel	Batch
Nombre d'heures /homme	34,5	50,2	4,0	12,3
Temps calcul/s	1266	907	229	197

Figure 1 : Comparaison du nombre d'heures passées en mise au point sur deux programmes de taille différente.

Les mesures ont été réalisées sur un ensemble de douze programmeurs expérimentés (19) - (21).

Ajoutons que le temps de mise au point est un facteur important dans l'écriture d'une application, comme l'indique la figure 2.

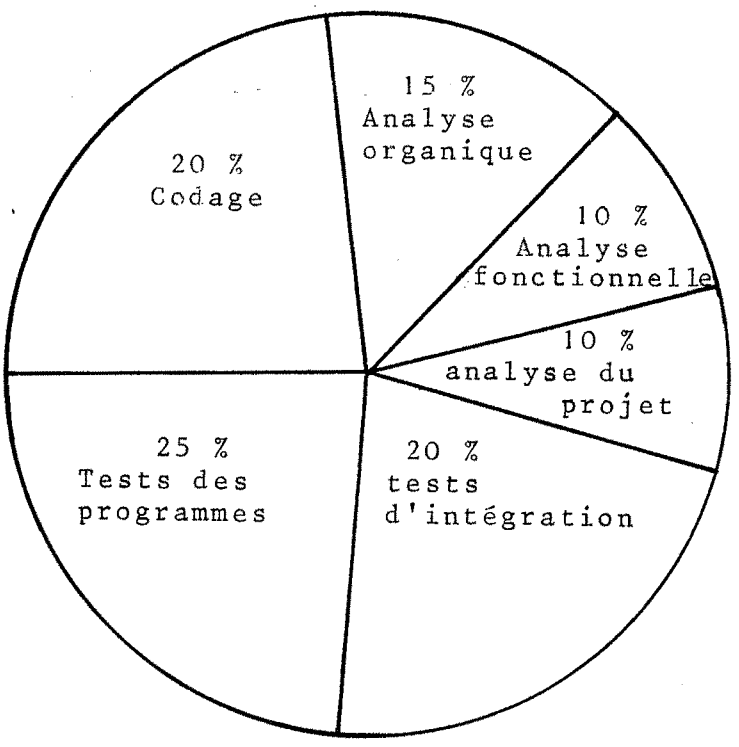


Figure 2 : Temps passé sur un projet (24).

De ce tableau, il ressort que le temps passé en tests est le facteur prépondérant dans la durée de livraison du projet. L'utilisation d'un outil de mise au point de programme bien adapté aux besoins du concepteur de projet, permettra de diminuer le temps passé en tests.

Le choix s'avère souvent limité. Pour les langages de haut niveau, nous disposons généralement de logiciels intégrés au compilateur, la technique utilisée est donc le batch. Par contre nous rencontrons des logiciels batch ou interactifs pour la recherche d'erreurs dans les programmes écrits en langage de bas niveau.

Le "Debugger system" DEBS2 (09) et le "Debugger utilisateur" DEBU2 (10) sur matériel MITRA représentent un compromis entre les deux techniques (batch et interactive). Les commandes peuvent être émises à partir de l'exécutif batch (par cartes perforées) ou à partir de l'exécutif interactif (par commande opérateur). DEBS2 permet la mise au point de systèmes d'exploitation. Celle-ci se fait par rapport à la mémoire et non par rapport à un programme du système ce qui est un handicap sérieux pour l'utilisateur ne connaissant pas l'implantation physique du programme en mémoire. DEBS2 ne peut être utilisé que par une équipe système.

DEBU2 par contre permet la mise au point de programmes d'application temps réel. Son originalité réside dans le fait de pouvoir tester plusieurs tâches à la fois. Il reste tout de même un outil difficile à manipuler par son langage de commandes de bas niveau. Il ne peut être utilisé que pour des programmes écrits en ASSEMBLEUR.

I.1.2. - METHODES DE MISE AU POINT

Il s'agit de donner ici un rapide aperçu des méthodes que l'on peut utiliser pour construire un outil de mise au point.

- L'insertion de code dans le texte source consiste à générer lors de la compilation une séquence de déroutement vers un processeur d'exécution (15). Cette méthode accroît la taille du code exécutable ainsi que le temps d'exécution. Ces deux facteurs sont importants notamment sur les mini-ordinateurs où l'adjonction d'un processeur généralement volumineux pose des problèmes d'encombrement mémoire ; De plus l'insertion de code dans des programmes s'exécutant en temps-réel (Ex: contrôle de processus) modifie la dynamique d'exécution des programmes.

  L'autre inconvénient est l'obligation de recompiler le programme une fois mis au point en supprimant les routines de tests.

  Pour résoudre ce problème, certains compilateurs possèdent des options de compilation conditionnelle (01) - (02) - (03) qui permettent de ne pas compiler certaines instructions du programme.

- Substitution dans le code exécutable. Il s'agit de remplacer dans le code compilé du programme à tester une instruction par une séquence de "déroutement" vers l'outil d'aide à la mise au point. L'instruction substituée est ensuite simulée ou exécutée hors de son contexte initial - L'instruction ACTD sur SOLAR (07) - L'instruction inexistante BRK sur MITRA (11).

  Cette implémentation dépend du type de calculateur sur lequel le programme s'exécute.

- La création d'une interruption machine permet un déroutement vers l'outil de mise au point. Dans ce cas le programme est totalement exécuté dans son contexte. Sur le SOLAR le positionnement d'un point d'arrêt se fait par l'instruction spécialisée "SBP" dont le rôle est de créer une imparité mémoire à l'adresse spécifiée dans l'instruction (07). Cette méthode évite une modification du code du programme par l'outil de mise au point, ce qui permet de ne pas alourdir les temps d'exécution. L'avantage est de pouvoir positionner des points d'arrêt aussi bien sur des variables que sur des instructions du programme. Elle reste fortement liée au calculateur.

- Simulation d'un pupitre hardware sur une console opérateur (11). Il y a basculement du mode d'utilisation de la console durant le temps de la mise au point des programmes. Pour ce faire, il y a définition d'un logiciel particulier permettant d'utiliser la console comme un outil de mise au point interactif. Cette méthode est beaucoup plus souple que la mise au point au pupitre (association d'un langage de commande) et de plus elle peut laisser une trace à l'utilisateur.

  Il ressort de ce bref aperçu, que le choix d'une méthode apparait difficile compte tenu des problèmes posés par la réalisation d'un outil de mise au point.

  - Augmentation des temps d'exécution
  - Augmentation du code généré
  - Portabilité difficilement envisageable.

I.1.3. - FONCTIONS GENERALES

Pour localiser une erreur, il faut au programmeur un certain nombre de renseignements sur l'exécution de son programme avant et au moment où se produit l'erreur. Les aides à la mise au point doivent permettre à un utilisateur d'acquérir immédiatement des informations sur l'exécution de son programme. Ensuite, la comparaison entre les résultats attendus et ceux obtenus lui permet de localiser les erreurs.

1 - Mécanisme d'arrêt :

D'une manière générale, l'arrêt de l'exécution du programme en mise au point avec retour au dialogue se fait en un point précis.

- Il est possible de demander qu'il n'ait lieu que lorsque l'instruction aura été exécutée un certain nombre de fois.

- Certains outils ne rendent le contrôle à l'utilisateur qu'en fonction du mode d'exécution demandé (mode PAS-A-PAS en général). La rencontre d'un point d'arrêt dans le programme en mise au point provoque simplement un affichage sur un périphérique de sortie si l'exécution choisie est le mode TRACE ou le mode CONTINU.

- L'arrêt peut être provoqué par un "appel opérateur" lorsque celui-ci détecte une anomalie dans sa mise au point (Ex: le programme ne passe jamais sur les points d'arrêt définis - le programme "boucle" en mémoire centrale).

2 - Reprise de l'exécution :

La reprise de l'exécution après un arrêt peut se faire séquentiellement (22). Elle peut aussi se faire en un autre point du programme par une requête "GOTO".

Toutefois cette requête n'est envisageable que si l'outil de mise au point dispose d'informations lui permettant ce branchement : informations sur la taille du programme, sur son lieu d'implantation en mémoire (pagination). Si les vérifications ne sont pas suffisantes à ce niveau, il peut s'en suivre des résultats incohérents. Aucune vérification n'est faite généralement sur les outils de bas niveau (11) - (13) - (14).

3 - Modification, affichage :

Ces fonctions représentent un aspect fondamental. Elles vérifient la cohérence de l'utilisation d'une variable par rapport à :

- sa déclaration dans le texte source du programme
- la configuration mémoire de la machine utilisée.

Des tests de cohérence sont faits sur modification de variables dans l'outil de mise au point écrit en GSL développé par HENRY SAVARY (22) ; une vérification systématique de la cohérence de la commande vis-à-vis des déclarations des éléments qui y apparaissent est faite par l'outil. Cette vérification a été possible dans la mesure où l'accès se fait symboliquement.

Certaines commandes peuvent permettre des modifications ou affichages mutiples. Le risque encouru est de définir des commandes "lourdes" d'emploi. Cependant cette forme de commande peut être efficace si la possibilité est donnée d'annuler certains caractères de la commande sans être obligé de réécrire toute la commande à cause d'un caractère mal frappé (langage de commandes des systèmes SOLAR).

L'affichage peut prendre une forme différente suivant le langage utilisé pour l'écriture du programme à mettre au point. L'affichage des registres paraît indispensable pour des langages tels que L'ASSEMBLEUR ou d'autres langages système tels que PL360, PL16. Par contre, pour des langages de haut niveau tel que FORTRAN, il sera plus important d'avoir accès à des paramètres d'appel et variables globales.

4 - Adjonction d'instruction :

Il arrive souvent qu'en phase de mise au point, l'utilisateur ait besoin d'insérer une ou plusieurs instructions dans son programme. Cette fonction, si elle existe, permet de résoudre rapidement des erreurs d'algorithme.

Un mini-langage est intégré dans le système de mise au point DEBUGS réalisé par René ODDOUX (21) ; Il permet de rajouter du code dans le programme objet. Son originalité réside aussi dans la possibilité de définir de nouvelles variables dans le programme à mettre au point. Cependant l'insertion d'un mini-langage double presque le volume du système.

Dans le système PROGRAID (13), la possibilité est offerte à l'utilisateur de simuler une instruction. Bien que cet outil soit adapté à une mise au point symbolique, la simulation d'une instruction se fait en code machine.

Un moyen simple de rajouter du code et de définir à l'avance des zones de "patches" dans les programmes. Un branchement sur ces zones permet d'effectuer des "verrues". Cette technique peut être une solution très efficace à des problèmes ponctuels. Cependant, elle impose une méthode rigoureuse de travail et une norme d'écriture des programmes ; les verrues doivent être supprimées après la période de mise au point.

Une approche originale et semble-t-il peu coûteuse a été faite (17). Au lieu de mettre en place des verrues, la partie du programme affectée par les modifications est translatée en mémoire d'un nombre de mots équivalent aux instructions à insérer. Cette méthode impose la mise à jour de toutes les références mémoire pour la partie de programme déplacée.

Toutes ces possibilités d'insertion de code évitent des pertes de temps importantes dues à des recompilations multiples de programmes.

Elles sont parfois difficiles à mettre en oeuvre (la définition d'espaces mémoires ne correspondant pas à la longueur réelle des programmes est parfois un handicap.

5 - Impression de zones mémoires :

Dans ce domaine beaucoup de choses ont été faites. Le point que l'on peut discuter est la manière de présenter les résultats à l'utilisateur ainsi que l'instant où l'impression est faite (nécessité d'une unité d'impression). Deux fonctions bien connues peuvent être proposées :

- DUMP MEMOIRE :

Cette fonction provoque l'impression du contenu d'une ou plusieurs zones mémoires. Son utilité est souvent réduite à cause de son implémentation. En effet, l'impression se fait souvent en code machine, et la sortie est souvent volumineuse. L'analyse de "DUMP" devient donc "pénible" surtout pour un programmeur qui travaille en langage de haut niveau. Des efforts ont été faits quant à la présentation.

Dans certains DUMP (12) on trouve à la fois une interprétation du code machine sous forme de caractères ainsi qu'une définition assez claire des zones du DUMP. Nous ne disposons généralement que de DUMP "a postériori" souvent insuffisants pour trouver les causes d'un incident ou d'une dégradation de l'application. Sur le système SIRIS-8 (23) nous disposons de deux commandes PMD et PMDI qui provoquent respectivement l'impression d'un contenu mémoire après l'achèvement anormal du programme et l'autre simplement à la fin d'exécution du programme.

Si des efforts sont faits sur la présentation des DUMP (sorties symboliques, pourquoi pas, quand l'outil dispose de la table des symboles du programme) nous constatons aujourd'hui encore que le DUMP est utile surtout pour l'homme système et peu pour le programmeur d'application.

- SNAP :

Cette fonction est similaire au DUMP. La différence est que les sorties se font en cours d'exécution du programme après chaque passage sur le SNAP. Exécutée trop souvent dans un programme, elle aboutit à une sortie volumineuse, qui devient comme le DUMP difficilement exploitable.

Il est possible d'améliorer les sorties en utilisant :

- des sorties sur fichiers dont on extrait sur imprimante seulement une partie de l'information
- une sélection des impressions (notion de "degré" sur DRIP16 (15))
- des paramètres indiquant le nombre de passages souhaités avant la sortie d'informations
- des paramètres indiquant un nombre maximum de sorties sur une requête. Ces SNAP sont utilisés s'ils s'intègrent bien dans le langage et forment des instructions facilement compréhensibles par l'utilisateur.

I.1.4. - CONCLUSION

Les gains apportés en programmation par l'utilisation de logiciels d'aide à la mise au point de programmes sont indéniables.

Il nous semble que le développement de l'un deux suivant une technique interactive ou batch devrait être lié à une méthodologie de travail et à une recherche d'efficacité de la mise au point, plus qu'au langage qui lui est associé.

Le nombre de fonctions que l'on veut y intégrer dépend souvent du temps que l'on voudra passer à sa réalisation.

S'il existe plusieurs méthodes de mise au point il apparaît que le choix entre l'une d'entre elles est conditionné par le calculateur sur lequel s'exécutent les applications.

I.2. - LES LANGAGES DE PROGRAMMATION

Selon le type de système utilisé, l'application que l'on développe, nous distinguons deux niveaux de langages :

- Les langages de bas niveau :

  Ils sont dépendants de la machine. L'utilisateur est capable généralement de savoir ce que génère en instructions machine, une instruction de programme. Le plus connu de ces langages est l'assembleur.

- Les langages de haut niveau :

  Ils se veulent indépendants de la machine. Chaque instruction écrite génère plusieurs instructions machine et aucune information n'est en principe donnée à l'utilisateur sur les instructions générées (adresses des variables ou contenu des registres). Ils sont nombreux, préférés aux langages de bas niveau pour leur efficacité (21). Le FORTRAN est un exemple parmi d'autres de langage de haut niveau.

I.2.1. - MISE AU POINT DES PROGRAMMES ECRITS EN LANGAGE ASSEMBLEUR

La visualisation ou la modification de registres semble être indispensable pour la mise au point. L'accès aux registres se fait en général par nom. L'accès à la mémoire est fonction de l'outil utilisé. Cela peut se faire de différentes manières:

- Par l'adresse du mot mémoire (code machine). C'est la méthode la plus simple à mettre en oeuvre

- Par association d'un nom symbolique à une adresse physique, le mot mémoire étant référencé par la suite sous ce nom (09)

- Par la possibilité d'utiliser les noms symboliques définis dans le programme source. Il suffit pour cela de conserver après l'assemblage du programme, la table des symboles et qu'elle soit accessible par l'outil de mise au point (13)

Une fois la variable ou l'instruction référencée, il est possible de visualiser sous différents formats :

- Hexadécimal
- Décimal
- Caractères
- Format instruction (13).

I.2.2. - MISE AU POINT DES PROGRAMMES ECRITS EN LANGAGE EVOLUE

Nous distinguons deux types de compilateurs utilisés dans l'industrie :

- les compilateurs générant du code interprétable,
- les compilateurs générant du code exécutable.

Quelque soit le type de compilateur utilisé, il est difficile d'accéder à l'adresse physique d'un mot mémoire, contrairement au langage assembleur, ainsi qu'aux registres de la machine qui n'ont plus aucune signification pour le programmeur. La possibilité d'utiliser des noms symboliques devient indispendable. Selon le langage utilisé, l'arrêt peut se faire sur un numéro de ligne instruction (le système HELPER (25) réalisé sur CDC 6600) si le compilateur donne les informations nécessaires ; nous pouvons aussi imaginer des arrêts sur des étiquettes ou variables du programme, ce qui semble plus simple à mettre en oeuvre.

Le logiciel proposé à l'utilisateur doit permettre dans la mesure du possible une souplesse se rapprochant de celle du compilateur.

Pour les outils interactifs, le concepteur devra s'attacher à définir un langage de commande de haut niveau avec une facilité de mise en oeuvre.

I.2.3. - CONCLUSION

Nous sommes parfois amenés à utiliser plusieurs langages pour une même application (Ex: le "corps" de l'application en langage évolué et certains interfaces en assembleur ou langage système tel que PL16). Nous devons donc admettre que le développement d'un outil de mise au point indépendant du langage est souhaitable.

Si l'efficacité de l'outil n'est pas toujours liée au choix d'une technique(système interactif ou batch) par contre elle dépend souvent de sa facilité d'utilisation (langage de commande de haut niveau - facilité de mise en oeuvre).

I.3. - LES OUTILS D'AIDE A LA MISE AU POINT DANS LEUR CONTEXTE D'EXECUTION

La plupart des outils de mise au point interactifs réalisés aujourd'hui, sont des "Memory Debugger" et non des "Task Debugger".

En effet, le positionnement de points d'arrêt se fait généralement sur une adresse mémoire et non sur une adresse de programme.

En plus de l'adressage, se pose le problème de la dimension temps. Le déroulement d'une application en test ne doit pas avoir pour conséquence la "dégradation des temps d'exécution" d'une autre application s'exécutant parallèlement en temps-réel.

La taille de l'outil est aussi un facteur important. Il arrive fréquemment sur les mini-ordinateurs de ne pas pouvoir l'utiliser par manque de place mémoire.

Les solutions à ces problèmes, sont souvent liées au contexte d'exécution des applications.

I.3.1. - CONTEXTE MONOPROGRAMMATION

Ici il s'agit de mettre au point un seul programme.

- S'il est entièrement résident en mémoire il est possible d'utiliser des "Memory Debugger", l'utilisateur pouvant connaître l'emplacement physique du programme en mémoire. Dans ce contexte particulier, nous pouvons envisager des outils autonomes.

- S'il fonctionne en "overlay" ou suivant un dispositif de pagination , il est indispensable d'avoir des "Task Debugger", l'utilisateur ne sachant pas quelle partie du programme va s'exécuter à un instant donné.

  Leur réalisation ne peut être envisagée sans une liaison étroite avec le système d'exploitation.

I.3.2. - CONTEXTE MULTIPROGRAMMATION

Nous regroupons volontairement sous ce terme la notion multi-tâches, multi-consoles, time-sharing (ou temps partagé).

Dans ce contexte, seuls les "Task Debugger" peuvent être envisagés si l'on donne une priorité à l'efficacité. La possibilité de mise au point simultanée de plusieurs modules s'exécutant en mémoire devient indispensable pour ce type d'application.

Le logiciel que nous avons développé utilise ce principe de fonctionnement : L'utilisateur spécifie les modules qu'il veut mettre au point à un instant donné et demande le positionnement de points d'arrêt ainsi que certaines actions à réaliser sur ces points d'arrêt (visualisation, modification, etc).

Les modifications des systèmes d'exploitations pour intégrer des outils de mise au point entraînent souvent des dégradations de performance notamment pour les applications fonctionnant en temps-réel. Il faudra donc s'attacher à minimiser au maximum l'importance de ces modifications.

Sur les systèmes multi-consoles un des problèmes rencontré est la liaison entre l'outil de mise au point et la console à laquelle il est "attaché". Une indépendance de l'un par rapport à l'autre est un facteur d'efficacité et de souplesse d'utilisation dont il faut tenir compte au moment de la réalisation.

Dans un contexte multiprogrammation de type temps-réel, la difficulté de la mise au point d'application réside dans la dimension temps.

L'insertion de points d'arrêt par des logiciels interactifs introduit souvent une dilatation des temps de traitement et peut donc cacher des problèmes de synchronisation.

Nous avons alors recours à des logiciels "batch". L'originalité du système DRIP16 (15) réside dans le fait de pouvoir contrôler et surveiller une application dans des situations réelles de fonctionnement (contrôle de paramètres aléatoires par exemple) sans trop augmenter la dilatation du temps.

I.3.3. - CONCLUSION

Il ressort de cette description très succinte que le contexte d'exécution d'une application est un paramètre important dans la conception d'un logiciel interactif.

Si l'indépendance de l'outil par rapport au langage de programmation peut facilement être envisagée, elle peut difficilement l'être par rapport au système d'exploitation.

Nous constatons aussi que la possibilité de mise au point simultanée de plusieurs programmes est nécessaire pour des applications s'exécutant dans un contexte multiprogrammation.

Nous dirons qu'il n'y a pas de solution peu coûteuse au problème de dégradation des temps d'exécutions des applications en test.

Si l'on veut réaliser des outils interactifs simples d'utilisation il faut qu'ils soient avant toute chose des "Task Debugger".

CHAPITRE II

PRESENTATION DE L'OUTIL DE MISE AU POINT REALISE

II.1. - OBJECTIF

Nous avons voulu réaliser un logiciel adapté à la mise au point et à la surveillance d'application fonctionnant dans un contexte multiprogrammation, sous le système d'exploitation MUTEX.

Notre choix s'est porté sur un outil interactif de haut niveau. Les langages de programmation utilisés pour l'écriture des applications sont l'Assembleur, le PL16 et le Fortran. Nous avons recherché une indépendance vis-à-vis de ces langages.

Pour la programmation système tels que l'Assembleur et le PL16, nous avons estimé dans un premier temps que les points d'arrêt seraient définis en hexadécimal. Les listings de programmation sont suffisamment clairs pour permettre à l'utilisateur de connaître l'adresse physique d'une instruction ou d'une variable. En langage Fortran les points d'arrêt sont définis en symbolique, l'utilisateur n'étant pas supposé connaître la correspondance instruction symbolique - instructions machine générées - L'outil réalisé est capable de faire la distinction entre ces deux types d'adresse (symbolique ou hexadécimale).

Notre objectif a été aussi la recherche d'une mise au point simultanée de plusieurs modules, ce qui nous a amené à réaliser un "débugger" d'applications et non de programmes.

Le positionnement de points d'arrêt se fait sur des adresses de modules ce qui en fait un "task debugger".

La recherche d'une souplesse d'utilisation a été aussi une de nos préoccupations. Ceci s'est traduit par un fonctionnement ne nécessitant aucune modification des programmes sources ; une mise en oeuvre immédiate ; une activation par commandes opérateur et sur n'importe quel poste de la configuration ; le choix des périphériques de sorties pouvant varier d'une session à une autre.

Il englobe aujourd'hui un certain nombre de fonctions que nous décrivons dans le paragraphe 3. Son nom est "DEBUG".

II.2. - CONTEXTE D'EXECUTION

DEBUG s'exécute sur le mini-calculateur SOLAR. Comme tous les logiciels interactifs, il dépend de la machine. Bien que son langage de commandes en fasse un outil de haut niveau, il reste très "attaché" au système d'exploitation MUTEX.

II.2.1. - PRESENTATION DU SYSTEME D'EXPLOITATION MUTEX

MUTEX (Multi User Transaction Executive) est un système d'exploitation conçu pour les applications de type conversationnel multi-utilisateurs et tout particulièrement les applications de saisie de données, gestion de transactions, gestion de production.

Ce système d'exploitation permet, par sa structure en pages et par son fonctionnement même, une programmation modulaire. Son langage de commandes est interprété par l'interpréteur-générateur INTGEN (05).

Cet interpréteur-générateur a été utilisé pour l'écriture des commandes de DEBUG.

MUTEX a pour rôle de dérouler simultanément différentes applications.

Une application est l'ensemble des programmes (ou modules) qui s'exécutent sous le contrôle d'un processus,

Un processus est l'entité fonctionnelle de la partie conversationnelle : il exécute pour le compte du poste de travail les divers modules de l'application déroulée sur ce poste, il réalise les échanges sur les périphériques du poste (Ex: imprimante, bande magnétique), et sur les fichiers disque. Le processus est le seul en relation directe avec le système d'exploitation.

Sous le système MUTEX, nous ne parlons pas d'activation d'un module en mémoire, mais d'activation d'un processus contrôlant une application (figure 3).

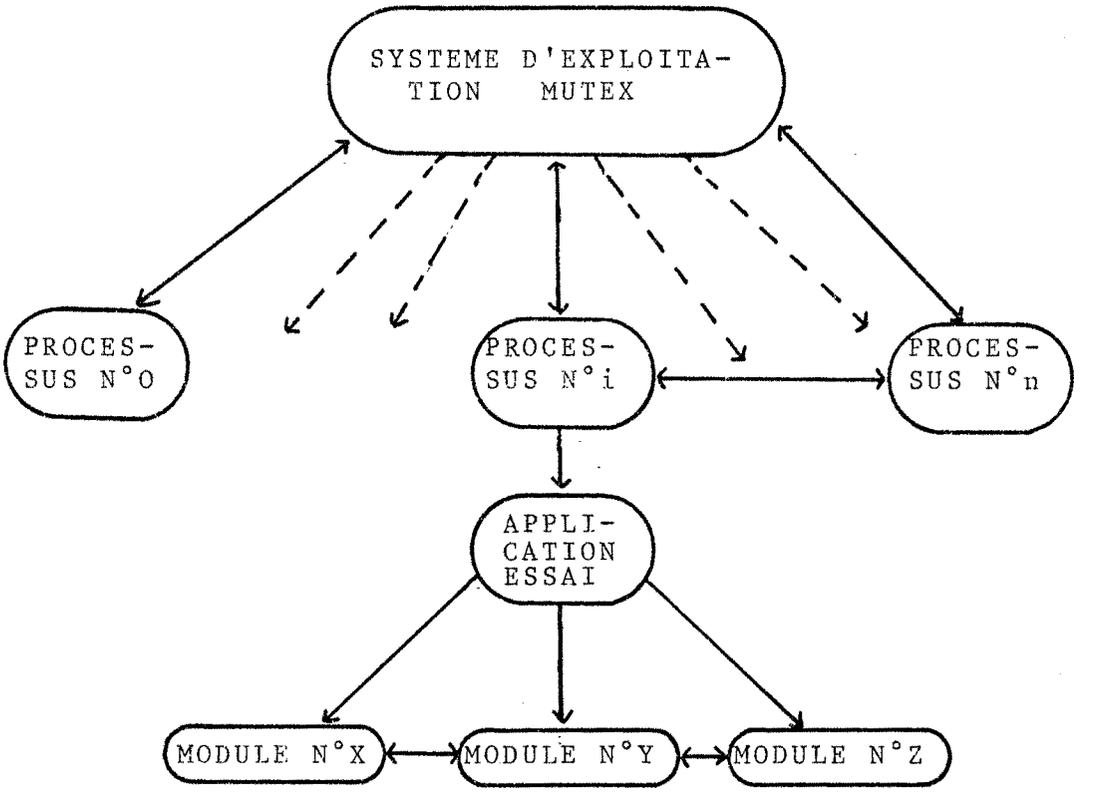


Figure 3 : Le processus est le seul en relation directe avec le système d'exploitation. Les modules d'une même application peuvent communiquer entre eux, mais pas avec le processus.

Il existe trois applications de type interpréteur pré-enregistrées dans le système d'exploitation MUTEX :

- L'application SYSCTL : c'est l'ensemble des modules permettant le langage de contrôle
- L'application SYSGEN : c'est l'ensemble des modules permettant la génération de langage. C'est elle qui active les modules du générateur d'applications de type interpréteur.
- L'application SYSDBG : c'est l'ensemble des modules que nous avons réalisé et composant l'outil de mise au point "DEBUG".

A ce niveau de définition, nous pouvons déjà faire une constatation. DEBUG a la même structure que toute autre application. Il est découpé en modules, il s'exécute pour le compte d'un processus. La technique d'enchaînement de modules ainsi que le passage d'informations entre deux modules suit la même logique que pour les applications développées par les utilisateurs. Nous verrons tout de même plus loin qu'il a besoin d'une "aide" du système d'exploitation pour résoudre un certain nombre de problèmes.

II.2.2. - LIEU D'IMPLANTATION DES PROGRAMMES A TESTER

Nous avons vu qu'une application était constituée d'un ensemble de modules. Ces derniers après compilation résident sur disque dans un fichier (bibliothécaire) repéré par un nom symbolique. Ils sont identifiés dans le fichier par un numéro (1 à 999). Ce numéro est défini après la phase de compilation, au moment du chargement dans le bibliothécaire.

La gestion d'un bibliothécaire (initialisation - sauvegarde - restitution - retassage) est faite par le processeur système GESBIB.

Le chargement, la suppression d'un module dans un bibliothécaire sont faits par le processeur système GESMOD.

Tous les programmes du système, y compris ceux de DEBUG sont stockés dans un bibliothécaire particulier nommé "BIBCOM". La figure 4 montre le chargement d'un module dans une page de la mémoire.

La communication des données entre les modules d'un même processus se fait par une zone dite "zone des données de communication" (ZDC) , elle-même stockée sur le disque.

II.2.3. - GESTION DES PAGES SOUS MUTEX

Le système d'exploitation MUTEX est capable de gérer autant de pages que le permet la mémoire centrale. Elles sont toutes de taille égale et entièrement banalisées vis-à-vis des processus. La taille d'une page est donnée lors de la phase de configuration du système ; elle varie de 4,5 Kmots (un mot égale 16 bits) à 32 Kmots. Une fois attribuée au processus, la page est chargée avec le module, la zone des données communes (ZDC), ainsi que le contexte non résident du processus (table contenant les informations nécessaires au processus pour le déroulement de l'application (05)).

Selon le cas, le module chargé dans la page peut être :

- Un module précédemment mis en attente dans le fichier système sur le disque (le nom de ce fichier est FUSYS) et ceci pour des raisons d'entrée/sortie ou encore dans le cas où un point d'arrêt dans le module a mis le processus dans l'état en attente.

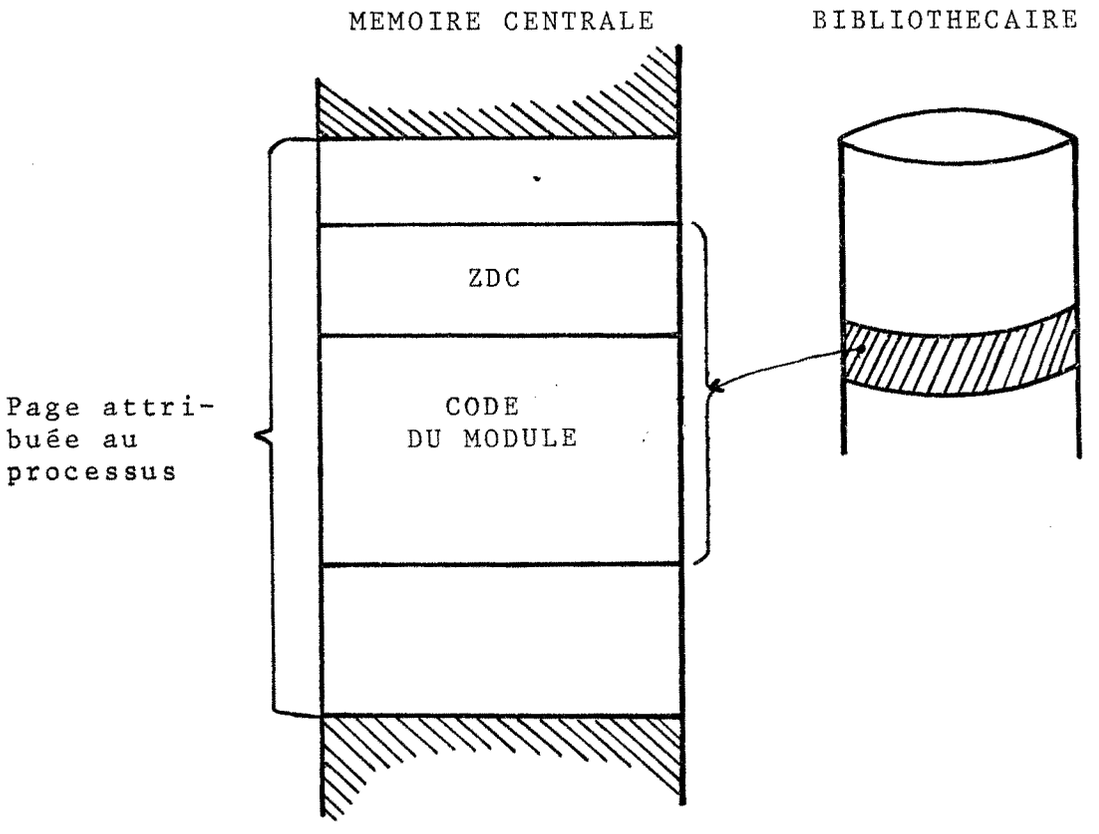


Figure 4 : Exemple de chargement du code d'un module.

- Un module en provenance d'un bibliothécaire et dont le chargement a été demandé par le processus qui le contrôle. Les modules provenant d'un bibliothécaire sont ceux activés sur primitive d'enchaînement (figure 5).

Le système effectue des SWAPP vers le fichier FUSYS :

- Sur entrée/sortie lente (Ex: imprimante ou console de visualisation).
- Sur enchaînement de module (passage d'un module à un autre.
- Sur rencontre de point d'arrêt dans le module.

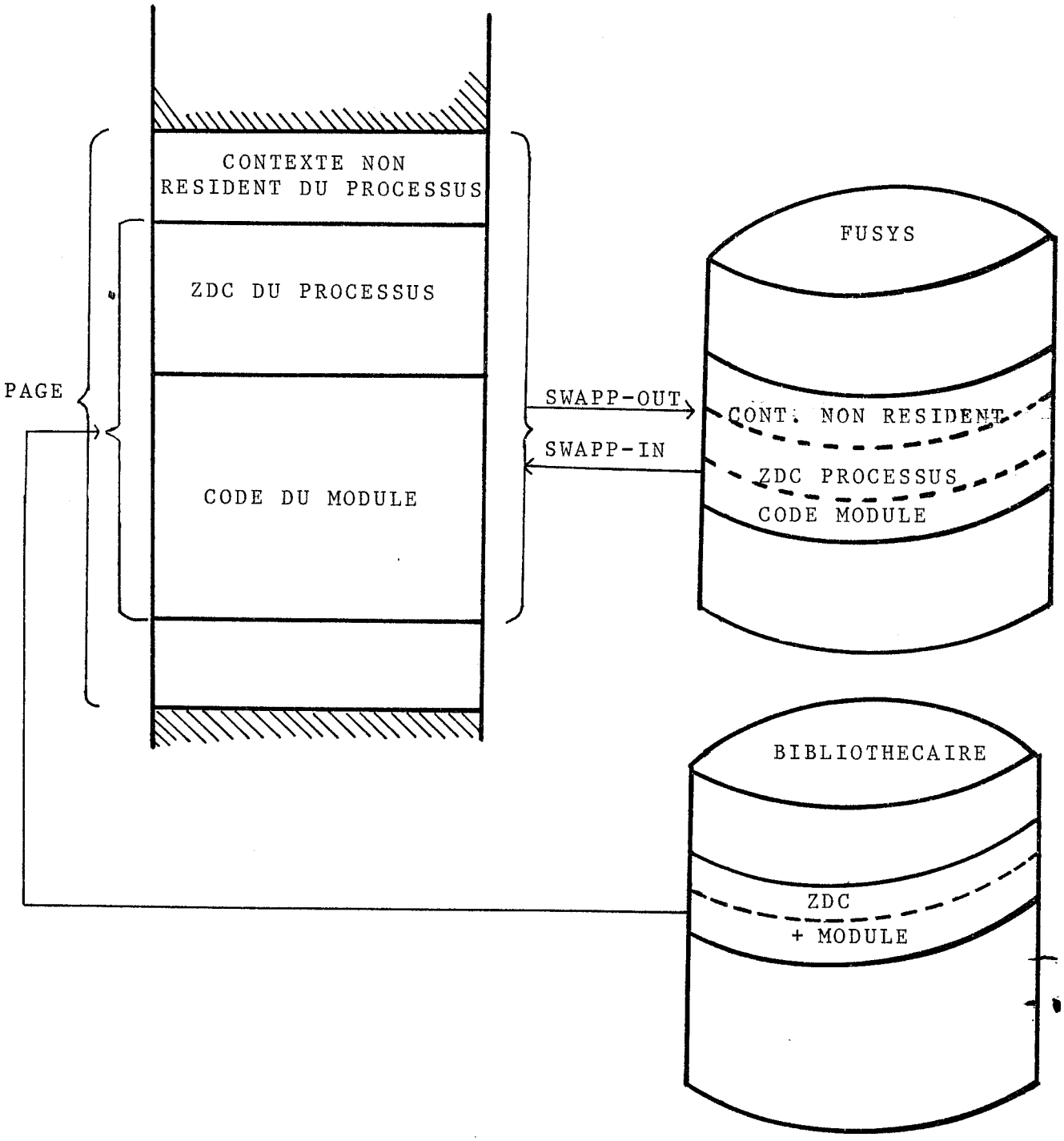


Figure 5 : Chargement du code du module et de sa ZDC depuis un bibliothécaire.
SWAPP du code du module, du contexte non résident et de la ZDC d'un processus.

II.2.4. - GRAPHE D'ETAT D'UN PROCESSUS

Pour être actif, un processus passe par divers états dont les principaux apparaissent sur la figure 6. La description de ces différents états est donnée à la suite de la figure.

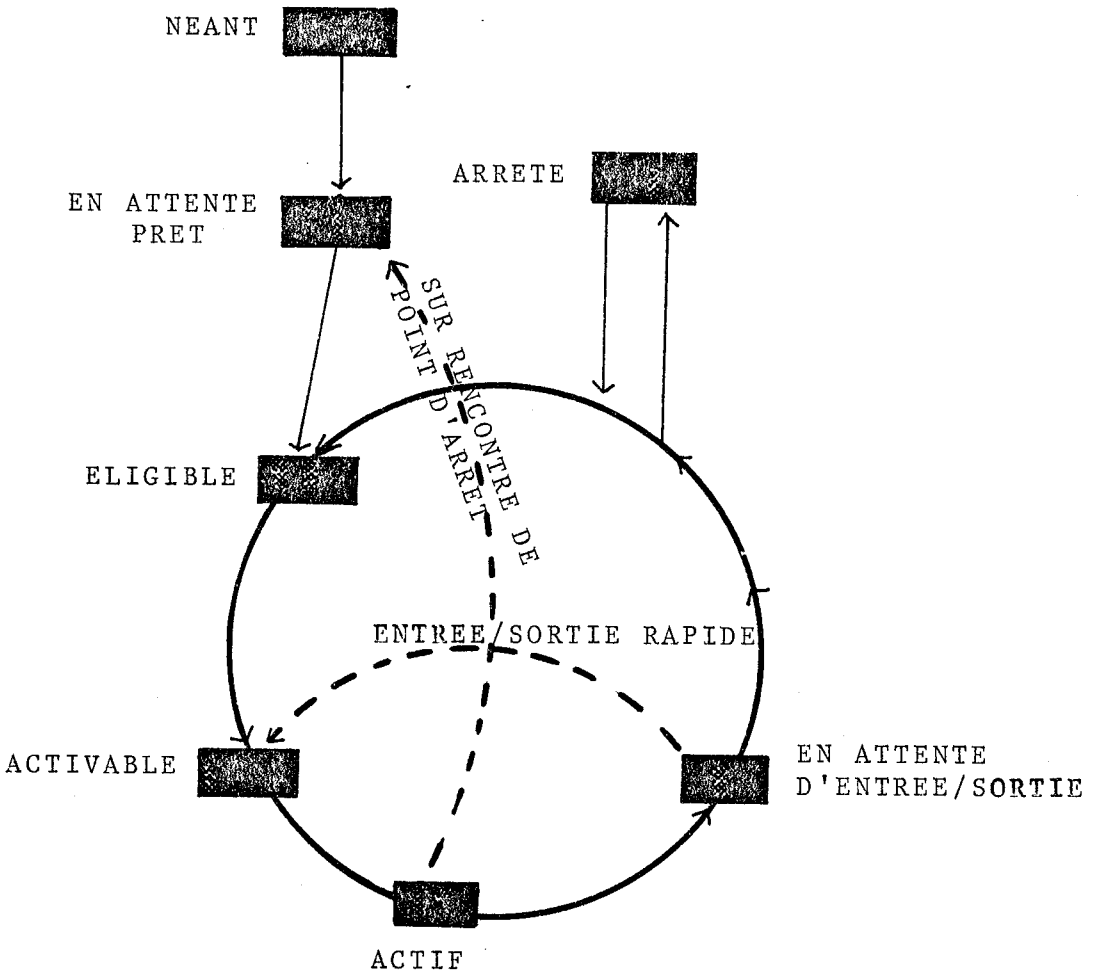


Figure 6 : Graphe d'un processus.

Description des différents états :

Actif : le module du processus est en mémoire centrale où il s'exécute sous une priorité commune à tous les processus. Le module "rend la main" lors de certaines primitives au système ou sur rencontre de point d'arrêt si une mise au point est en cours.

Eligible : On trouve dans cet état les processus en attente de page disponible. Les processus éligibles sont chaînés dans la file d'attente dite des "éligibles".

Activable : Un processus passe de l'état éligible à l'état activable, lorsqu'une page lui a été attribuée et qu'elle a été chargée. Le module du processus est en attente d'exécution dans la file d'attente dite des "activables".

Arrêté : Tout passage par l'état "arrêt" du processus (cause d'alarme ou d'une fin de transaction) déclenche la fermeture des fichiers ouverts par le processus et la récupération des périphériques par le système

Néant : Le processus n'est pas encore connu du système d'exploitation.

En attente-prêt : Le processus est connu du système mais l'activation n'a pas encore été demandée. En phase de mise au point, DEBUG se trouvera dans cet état sur rencontre de point d'arrêt dans le module qu'il déroule.

En attente d'entrée/sortie : Le processus est en attente dans le système d'entrée/sortie soit parce qu'il n'a pas encore eu l'accès au système d'entrée/sortie, soit qu'il lui manque momentanément l'accès à une ressource. Il passe de cet état à l'état activable sur une entrée/sortie rapide (entrée/sortie sur un fichier disque). Il passera à l'état éligible sur une entrée/sortie lente (entrée/sortie sur un périphérique lent). De toute façon le processus est chaîné dans la file d'attente du processus en attente d'entrée/sortie.

II.3. - CARACTERISTIQUES DE L'OUTIL DE MISE AU POINT

DEBUG est une application contrôlée par un processus particulier.

Ce processus est pré-configuré et n'est lié à aucun poste. C'est l'utilisateur lui-même qui l'associera à un poste lorsqu'il débutera une session de mise au point. Cette originalité permet à un utilisateur de faire de la mise au point d'application sur n'importe quel poste de la configuration. D'où une grande facilité de mise en oeuvre, ainsi qu'une grande souplesse dans l'utilisation des périphériques.

DEBUG permet :

- de faciliter la détection d'erreurs dans un module et la localisation de leurs sources

- de faciliter la correction de ces dernières.

Une des caractéristiques de cet outil est qu'il permet de positionner plusieurs points d'arrêt à la fois. Les points d'arrêt pouvant être positionnés aussi bien sur une instruction du programme que sur une donnée.

Il n'est actif en mémoire que pour le positionnement d'une série de points d'arrêt et la définition d'actions sur les différents modules à mettre au point. Le reste du temps, il est inactif.

En conséquence, le module en test s'exécute dans son contexte. C'est le micro-programme du calculateur qui détecte les points d'arrêt.

II.3.1. - FONCTIONS REALISEES

La mise en attente d'un module se fait en des points précis définis à l'avance par l'utilisateur à l'aide du langage de commande.

1 - Deux types de commandes peuvent être émises par l'utilisateur (figure 7) :

- commandes pour la réalisation d'actions immédiates.

- commandes pour la réalisation d'actions différées.

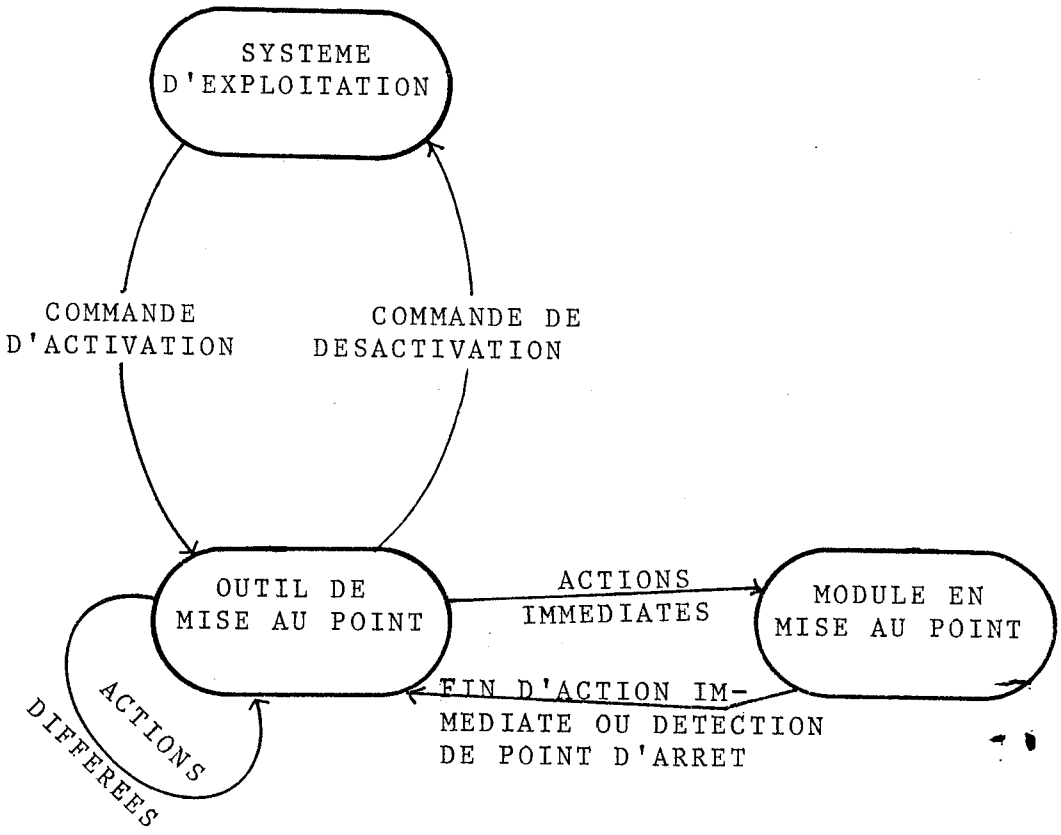


Figure 7 : Liens entre le système d'exploitation, l'outil de mise au point et le module en mise au point. L'activation et la désactivation sont expliquées dans le chapitre III paragraphe 2.

- Les actions immédiates :

Elles sont exécutées par DEBUG dès que la commande est émise. Elles permettent entre autre d'effectuer des opérations de visualisation de variables ou de registres, de supprimer des points d'arrêt ou bien encore de les afficher sur la console opérateur.

- Les actions différées :

Elles sont exécutées par DEBUG sur rencontre d'un point d'arrêt. Si l'action correspond au point d'arrêt, alors elle est exécutée ; sinon elle reste en attente du point d'arrêt correspondant. Ces actions permettent en particulier d'effectuer des traces en certains points de l'application. Il est à noter que le nombre d'actions différées n'est limité que par la taille de la table où elles sont en attente de réalisation.

2 - Mécanisme d'arrêt :

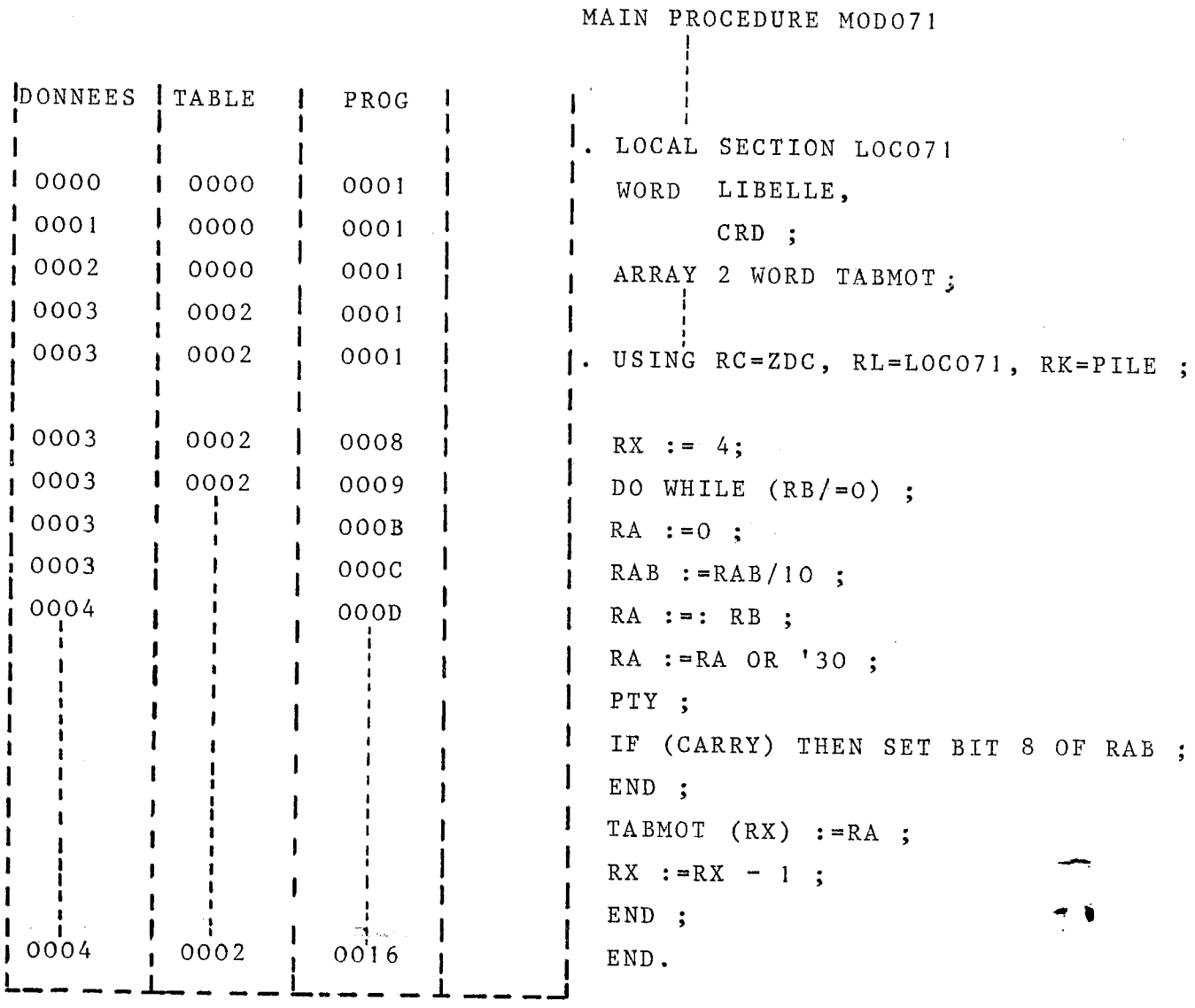
Le programme source est référencé de deux manières différentes selon que le programme est écrit en PL16, ASSEMBLEUR, ou FORTRAN.

- Cas de l'assembleur et du PL16 :

L'utilisateur choisit un ou plusieurs endroits dans le programme pour implémenter une série d'actions.

Il calcule l'adresse physique par rapport au début du programme. Ce calcul est facilité par la structure des listings de compilation et de chargement. Cette adresse calculée est celle qu'il écrira dans la commande de positionnement de point d'arrêt.

Soit la séquence suivante en PL16 :



Supposons :

- que le programme débute à l'adresse hexadécimale : '025E
- que le local débute à l'adresse hexadécimale : '0253.

A noter que ces deux informations sont données au chargement du programme .

- Calcul de l'adresse du mot LIBELLE : '0253 +'0000 = '0253
- Calcul de l'adresse de l'instruction DO WHILE (RB/=0) : '025E +'0009 = '0267
- L'adresse du tableau TABMOT est contenue dans le mot d'adresse : '0253 +'0002 = '0255

- Cas du FORTRAN :

En FORTRAN, l'utilisateur choisit aussi plusieurs endroits dans le programme pour implémenter une série d'actions. Il n'a pas à faire de calcul d'adresse. Il inscrit dans la commande un nom de variable ou bien encore un numéro d'étiquette correspondant à une instruction du programme source.

Soit la séquence suivante en FORTRAN :

```
00001                COMMON ZDC (256)
00002                INTEGER N
  ¦                    ¦
  ¦                    ¦
00005              10 DO 100 I = 1,N
00006                 TAB (I) = I * 2
00007             100 CONTINUE
00008                 J = J + 1
  ¦                    ¦
  ¦                    ¦
```

L'utilisateur peut demander un arrêt sur les variables N, I, J, ou sur le tableau TAB ou bien encore sur les étiquettes 10 ou 100.

Il doit écrire des étiquettes supplémentaires bien placées dans son programme source s'il veut des possibilités de mise au point efficaces.

3 - Reprise de l'exécution :

La reprise de l'exécution se fait soit en séquence, soit en un autre point du programme si l'utilisateur a demandé une action de branchement. C'est l'outil de mise au point qui vérifie la validité de l'action de branchement.

4 - Modification, affichage :

Il est possible sur des actions différées ou immédiates de demander d'afficher ou de modifier :

- Certains registres de la machine (si PL16 ou Assembleur)

RC, RL, RW ⟶ les bases
RX ⟶ le registre index
RK ⟶ le pointeur de pile
RS ⟶ le registre d'état
RA, RB ⟶ les accumulateurs
RY ⟶ le registre de travail

La modification des registres RSLO, RLSE (registres de protection mémoire) est impossible. Ces registres sont manipulés uniquement par le système, leur visualisation est par contre possible. Elle permet de connaître l'implantation et la longueur de la page où s'exécute le programme. Le registre RP (compteur ordinal) peut lui aussi être visualisé. Sa modification se fait par une action de branchement (GOTO).

- Toutes les variables du programme :

Le format de visualisation ou de modification est l'hexadécimal, le décimal, le caractère, le binaire.

- Les modules en attente d'activation :

Il est possible pour une même application de visualiser la pile des modules et même d'en modifier le contenu. Aucune vérification n'est faite sur les modifications de la pile des modules.

- Les entrées/sorties :

L'utilisateur peut obtenir le compte-rendu du dernier échange effectué. Il lui suffit de demander la visualisation du dernier échange.

- Les points d'arrêt :

Il est aussi possible de visualiser ou de supprimer des demandes de point d'arrêt dans un module.

5 - Dump mémoire :

Il est possible avant, pendant ou après la détection d'une erreur, de demander l'édition de toute la zone mémoire relative au module.

Ce dump est en hexadécimal mais structuré de telle manière que son exploitation soit "aisée" pour une utilisateur PL16 ou ASSEMBLEUR.

L'activation de ce dump peut être différée ou immédiate.

6 - Enchaînement d'un module :

Il est possible sur une action immédiate ou différée d'enchaîner un module. Ce dernier s'exécute sous le contrôle de l'outil de mise au point. Après son exécution, l'utilisateur peut demander la reprise de l'exécution du module en mise au point. Ceci permet à l'utilisateur de rajouter lui-même de nouvelles fonctions de mise au point (Ex. lecture d'un fichier disque).

7 - Appel opérateur :

Cette fonction permet de redonner le contrôle au dialogue DEBUG. Pour l'activer, il suffit à l'utilisateur d'appuyer sur la touche "BREAK" de la console utilisée pour le passage des commandes.

Cette fonction est essentielle. Elle permet de palier à un certain nombre d'erreurs de manipulations dans les demandes de positionnement de point d'arrêt (Ex: le programme boucle sur une zone où aucun point d'arrêt n'a été défini).

II.3.2. - EXTENSIONS POSSIBLES :

Compte tenu des délais de réalisation, un certain nombre de fonctions ne sont pas développées dans cet outil.

On ne peut demander des actions différées que lorsque les points d'arrêt sont définis sur des instructions de programme. Il serait souhaitable de pouvoir demander aussi ce type d'actions lorsque les points d'arrêt portent sur des variables du programme.

Les arrêts sur modification de variables seraient une extension à envisager dans la suite à donner à ce projet.

En fin de session de mise au point, toutes les actions différées sont détruites. Il faudrait pouvoir les sauvegarder sur un fichier disque particulier. Cette nouvelle fonctionnalité permettrait à un utilisateur au moment de l'activation de DEBUG de récupérer ou non les demandes d'actions différées de la session précédente.

Sur les appels de procédures aucune trace n'est donnée sur les paramètres d'appel. Cette extension serait intéressante à réaliser dans la mesure où les modules d'une application fonctionnant sous MUTEX contiennent beaucoup d'appels par CALL. L'utilisateur peut aujourd'hui contourner cette difficulté en positionnant simultanément des points d'arrêt sur tous les appels de procédures ; ceci n'étant pas une solution en soi.

Remarque :

A notre avis certaines extensions ne sont pas envisageables.

La visualisation ou la modification du registre interruption (IM) n'est pas souhaitable. Les interruptions étant entièrement gérées par le système d'exploitation, nous avons pensé qu'il n'était pas nécessaire de permettre des actions sur les interruptions machine ou logicielles.

Pour faire des verrues dans son programme, l'utilisateur est obligé de se réserver une zone de "patches" qu'il remplit avec des instructions machine. Si cette méthode est imaginable pour un utilisateur système, par contre elle ne l'est pas du tout pour un programmeur d'application. Il faudrait pouvoir insérer des instructions ou des variables dans un langage simple. Si d'un point de vue externe cela parait envisageable, sachons tout de même qu'il faut beaucoup de temps pour réaliser cette fonctionalité.

II.3.3. - MODE D'EXECUTION DU MODULE EN MISE AU POINT :

Il est défini par l'utilisateur à l'aide d'une commande du langage. Nous avons retenu deux modes d'exécution. Nous les avons appelés:

* Mode pas-à-pas
* Mode trace.

- Mode pas-à-pas :

Le programme s'exécute jusqu'à la rencontre d'un point d'arrêt.

Entre deux conditions d'arrêt, il n'y a pas trop de dégradations des temps d'exécution du programme.

A la rencontre du point d'arrêt le contrôle est donné au DEBUG pour qu'il réalise les actions prévues lors de la définition du point d'arrêt. Enfin le contrôle est donné à l'utilisateur pour définir d'autres points d'arrêt ou pour demander la réalisation d'actions immédiates.

Si l'utilisateur a pris soin de définir de nombreux points d'arrêt, cela lui permettra de suivre de très près l'exécution de son programme.

Ce mode de fonctionnement est utile pour dérouler les programmes non opérationnels.

- Mode trace :

Comme dans le mode pas-à-pas, le programme s'exécute jusqu'à la rencontre d'un point d'arrêt. En mode trace, le contrôle n'est pas donné à l'utilisateur après la réalisation des actions prévues lors de la définition du point d'arrêt, mais au programme qui continuera de s'exécuter jusqu'à la rencontre d'un prochain point d'arrêt.

Ce mode de fonctionnement convient très bien pour faire de la surveillance d'application.

En effet un utilisateur peut définir plusieurs points d'arrêt sur chaque module de l'application ; à chaque point d'arrêt il associe une action à réaliser par le DEBUG (une visualisation par exemple) puis il demande l'activation de l'application en mode trace. Cette technique lui permet donc une surveillance en des points précis qu'il s'est définis lui-même. Il est bien évident que ce mode peut être dangereux pour des programmes non opérationnels.

Remarques : De toutes façons l'utilisateur pourra toujours reprendre le contrôle par "appel opérateur".

CHAPITRE III

REALISATION SUR SOLAR

III.1. - GENERALITES SUR LES COMMANDES :

Les commandes du langage de l'application SYSDBG (outil de mise au point) s'exécutent sous l'interpréteur INTGEN (05).

Chaque commande est analysée par l'interpréteur. Ce dernier fait réaliser au système d'exploitation MUTEX le traitement correspondant à la commande émise ; ce traitement est exécuté par une suite de modules du système.

L'interpréteur est prêt à recevoir les commandes dès qu'il a émis le caractère [≫]. La commande de lancement de l'application SYSDBG appartient au langage de commandes du système d'exploitation. Le caractère émis dans ce cas est [>].

Supposons qu'un utilisateur émette la commande "MODULE 071 MODE PP".

Cette commande permet de spécifier que l'on désire mettre au point le module 71 et que le mode d'exécution de ce module est le pas-à-pas.

La figure 8 schématise le travail fait par l'interpréteur :

1) - empilement du module exécutant la commande.
2) - sauvegarde des paramètres de la commande dans la ŻDC interpréteur.
3) - demande le chargement du module empilé dans la pile des modules.

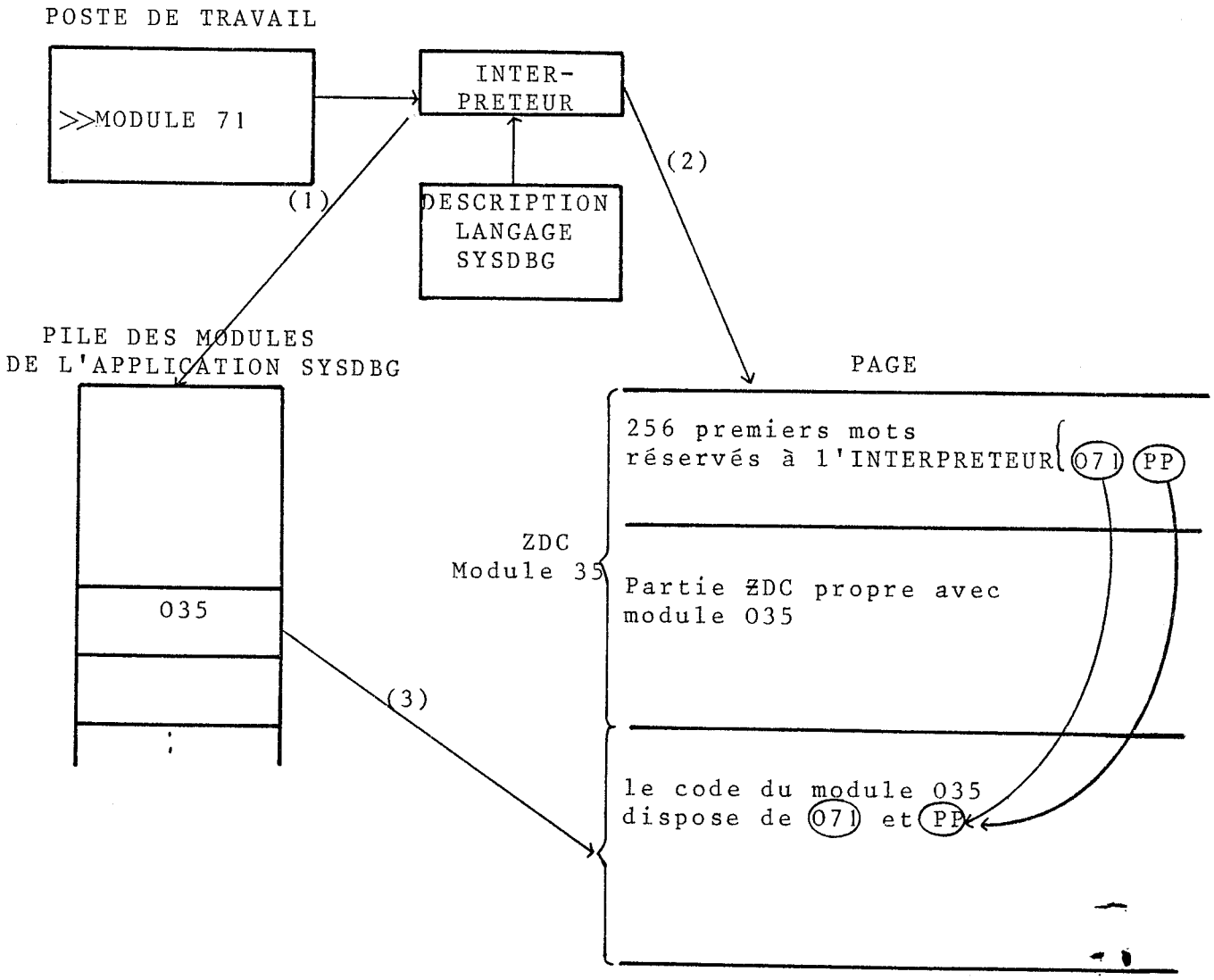


Figure 8. : Exemple du travail de l'interpréteur.
La commande "MODULE" active le module 035. Ce dernier accède aux paramètres de la commande par une simple lecture de la ZDC.

Obtention des commandes de l'application SYSDBG :

Pour décrire le langage de l'application SYSDBG, nous avons utilisé l'application SYSGEN. C'est une application livrée en STANDARD et indispensable aux applications de type INTERPRETEUR, qui permet la génération de langage. Elle nous a permis de fabriquer facilement le langage d'application SYSDBG, notre DEBUG. La description obtenue après génération est implantée par le générateur lui-même dans le bibliothécaire où a été déclarée l'application.

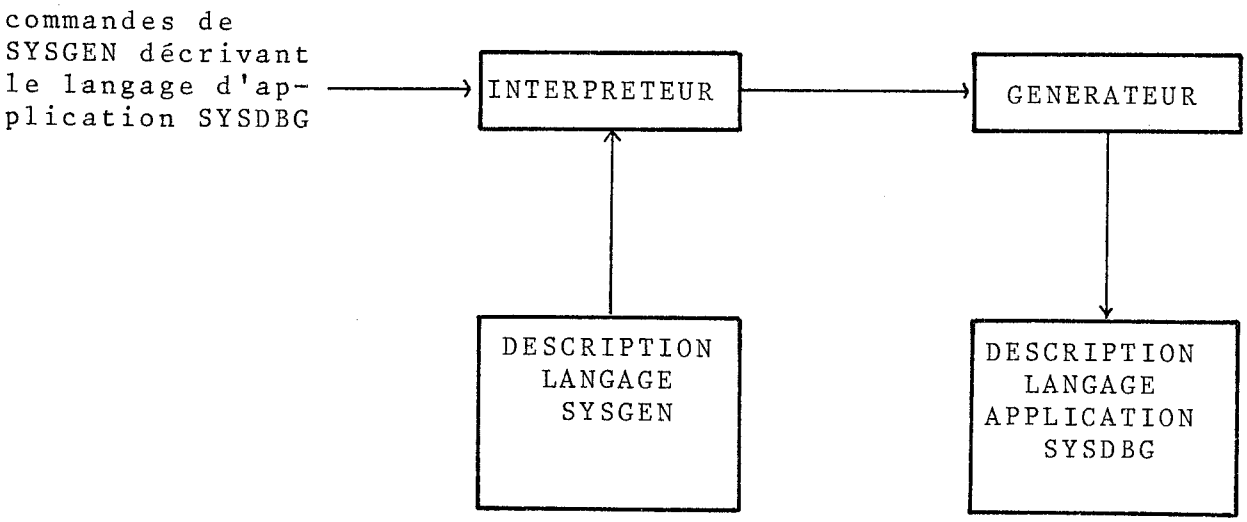


Figure 9. : Travail de l'interpréteur-générateur.

Dynamique du système :

La figure 10 représente les commandes du système DEBUG dans leur environnement. La description détaillée de toutes les commandes est faite dans le chapitre suivant (chapitre III).

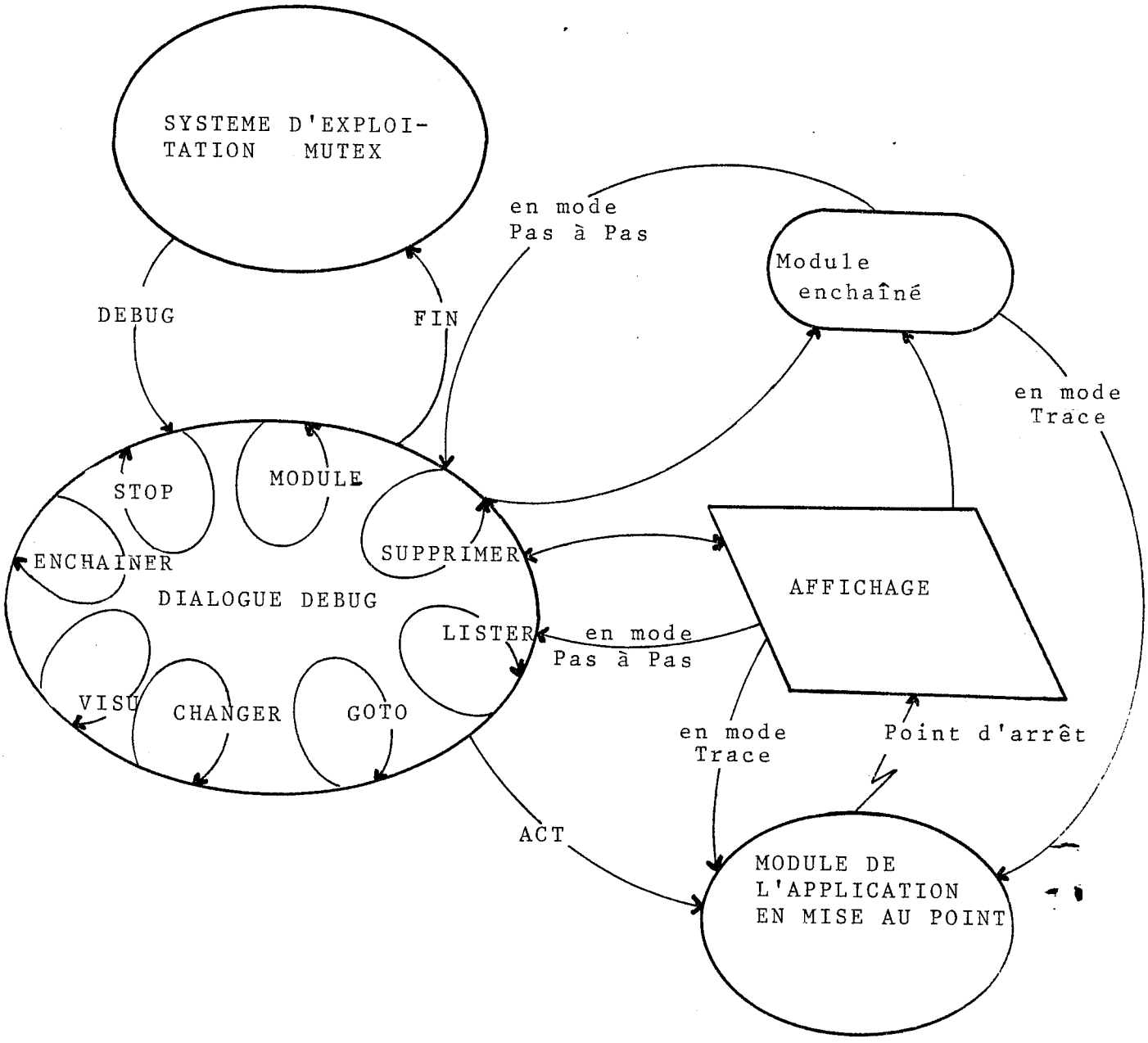


Figure 10. : Dynamique du système de mise au point.

III.2. - DESCRIPTION DES COMMANDES

Remarques préliminaires :

- Les commandes précédées du caractère [>] appartiennent au langage de contrôle SYSCTL. L'utilisateur les fait précéder du caractère [/]

- Les commandes précédées du caractère [>>] appartiennent au langage de l'application SYSDBG (outil de mise au point)

- Dans une commande seuls les trois premiers caractères du mot sont significatifs

- Le lancement de l'application SYSDBG se fera comme le lancement des autres applications, c'est-à-dire par la commande du langage contrôle SYSCTL :

```
> / LANcer  APPlication  SYSDBG  [SUR NOMPOSTE]
```

Nous rappelons que l'application SYSDBG représente l'outil de mise au point.

Signification du paramètre NOMPOSTE :

NOMPOSTE est le nom du poste sur lequel nous voulons activer l'application.

Par défaut, il s'agit du poste où est émise la commande.

III.2.1. - ACTIVATION ET DESACTIVATION DE DEBUG

A - Activation :

Pour activer l'outil de mise au point, l'utilisateur émet la commande "DEBUG" après avoir lancer l'application SYSDBG.

Syntaxe des commandes :

a - DEBug

Ne seront testés que les modules qui s'exécutent sur le processus où est émise cette commande.

b - DEBug SUR NOMPROC

Les points d'arrêt dans les modules à tester ne seront effectifs que pour le processus dont le nom est donné dans le paramètre NOMPROC.

c - DEBug SUR TOUS PROCESSUS

Dans ce cas les points d'arrêt dans les modules seront pris en compte quelque soit le processus exécutant ces modules.

Remarques :

.Après l'émission de la commande DEBUG, la phase du dialogue est signalée par les caractères " >> " imprimés en début de ligne sur le périphérique où a été émise la commande "DEBUG". Les modules à tester sont spécifiés au niveau du dialogue par les commandes "STOP" et "MODULE" (voir paragraphe suivant).

.Le processus sur lequel a été émise la commande est alors mis en attente ; cette technique permet d'activer l'outil de mise au point sur un poste déroulant une application à mettre au point (commande "DEBUG" sans paramètre).

B - Désactivation :

Pour désactiver DEBUG (pour terminer une session), l'utilisateur émet la commande "FIN".

Syntaxe de la commande :

FIN [DEBug]

Cela revient à demander un logout des processus DEBUG.

Si un processus est en attente sur point d'arrêt au moment de l'émission de la commande, alors il est réactivé.

III.2.3. - COMMANDE DU DIALOGUE

Ce dialogue a pour but de faire connaître au processus DEBUG et au système MUTEX les modules à mettre au point et donc les points d'arrêt à positionner sur ces modules.

Le rôle de DEBUG est d'enregistrer les points d'arrêt à poser dans sa ZDC (appelée ZDCPARAM) et dans la table des symboles du module. En fin de dialogue, les points d'arrêt seront effectivement positionnés par MUTEX après chargement en mémoire des modules concernés.

Le chargement d'un module en mémoire se fait avec sa table des symboles (voir paragraphe 3.).

Une des particularités de ce dialogue est que l'on peut demander la pose de plusieurs points d'arrêt sur plusieurs modules à la fois.

L'utilisateur peut aussi demander pour chaque point d'arrêt une action associée à ce point d'arrêt (visualisations, modifications, branchements inconditionnels, enchaînement d'un module).

Durant la phase du dialogue, l'utilisateur peut lister ou supprimer des points d'arrêt.

Le dialogue est composé d'un certain nombre de commandes décrites dans les pages suivantes.

Remarque :

Les points d'arrêt dans un programme ne sont effacés que sur une demande de l'utilisateur.

A - Commande MODULE :

L'utilisateur indique au niveau du dialogue le module à tester à un instant donné, ceci afin de permettre au processus DEBUG d'accéder à la table des symboles correspondant au module en question. C'est dans la table des symboles de ce module que DEBUG inscrit les différents points d'arrêt à positionner par le système MUTEX lors du chargement du module en mémoire. Cette table sert donc de moyen de communication d'informations entre le système MUTEX et le processus DEBUG.

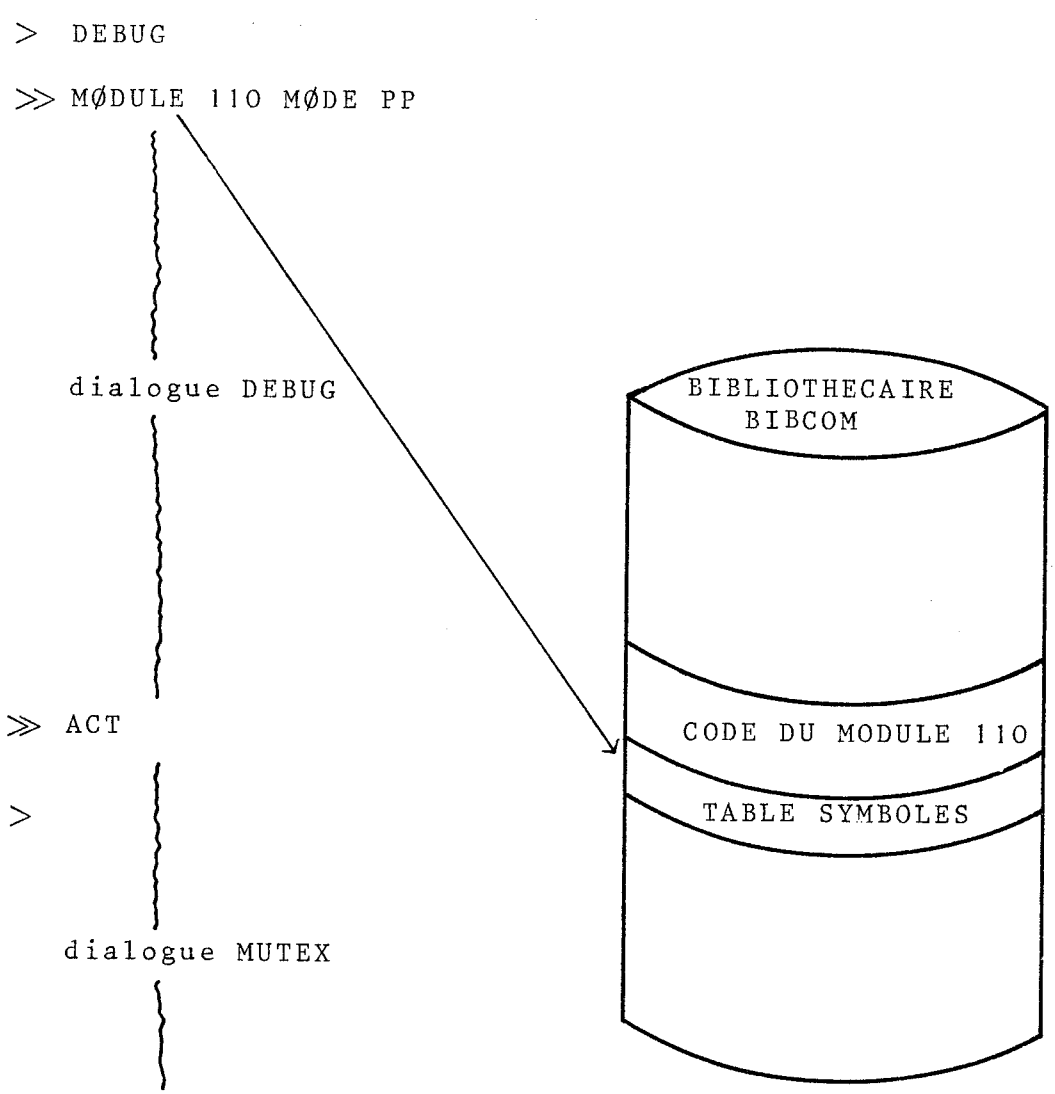
Syntaxe de la commande :

MODule NUMOD [MODE {TR / PP}] [DANS [BIBLIOTHECAIRE] NOMBIB]

Signification des paramètres :

NUMOD	Numéro de module
TR	demande d'exécution du module en mode TRACE
PP	demande d'exécution du module en mode pas-à-pas ; ce paramètre est pris par défaut
NOMBIB	nom du bibliothécaire où est stocké le module (par défaut : bibliothécaire BIBCOM).

EXEMPLE :


> DEBUG
≫ MØDULE 110 MØDE PP
dialogue DEBUG
≫ ACT
>
dialogue MUTEX
BIBLIOTHECAIRE
BIBCOM
CODE DU MODULE 110
TABLE SYMBOLES

B - Commande STOP :

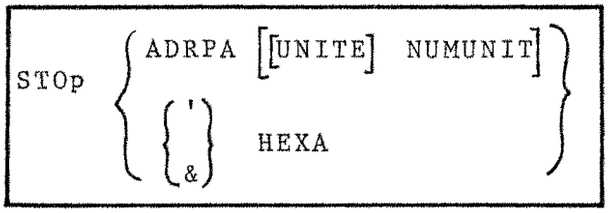
Cette commande permet de faire des demandes de points d'arrêt dans le module référencé par la commande "MODULE".

Selon le langage utilisé, ces points d'arrêt sont définis en symbolique (si langage FORTRAN) ou en hexadécimal (si langage PL16 ou Assembleur).

Nous rappelons que les points d'arrêt symboliques ne peuvent correspondre qu'à des étiquettes ou variables du programme. Ceci implique que l'utilisateur devra faire précéder les lignes FORTRAN d'un maximum d'étiquettes s'il veut une mise au point efficace.

Si le point d'arrêt est défini en hexadécimal, il est précédé obligatoirement du caractère "'" s'il s'agit d'une adresse effective (par rapport au début du programme) ou du caractère "&" s'il s'agit d'une adresse de relais.

Syntaxe de la commande :



Signification des paramètres :

ADRPA	nom symbolique correspondant à une étiquette ou variable du programme
NUMUNIT	numéro d'unité Fortran dans laquelle est défini le nom symbolique ($1 \leqslant NUMUNIT \leqslant 63$). Par défaut ce paramètre est égal à 1 ; il correspond au programme principal.
HEXA	adresse hexadécimale, du programme Assembleur ou PL16
"'"	l'adresse hexadécimale est une adresse effective par rapport au début du programme
"&"	l'adresse hexadécimale est une adresse de relais (Ex. adresse d'un tableau).

REMARQUE 1 :

A chaque arrêt, le système DEBUG sort un message indiquant l'étiquette ou la variable sur laquelle le module utilisateur est en attente. Cette méthode simplifie la mise au point.

REMARQUE 2 :

Dans le cas de positionnement de point d'arrêt sur variable du programme, il y a arrêt chaque fois qu'une instruction adresse la variable.

Exemple :

```
>  DEBUG
≫  MØDULE 31 MØDE PP
≫  STØ 100
≫  STØ INDIC1
≫  STØ 120 02
≫  ACT

   MØDULE 31 PRØCESSUS PØSTE1 STØP ADRESSE 100

≫  ACT

   MØDULE 31 PRØCESSUS PØSTE1 STØP ADRESSE INDIC1
```

C - Commande LISTER :

Syntaxe de la commande :

LISter [ARRETS]

Cette commande a pour but de lister tous les points d'arrêt d'un module. Le module concerné est celui défini par la commande "MODULE".

EXEMPLE :

```
≫ MØDULE 31 MØDE PP
≫ LIS ARR
   LISTE  PØINTS D'ARRETS  MØDULE 31
   01 - 0100      ⟹ il y a seulement 1 point d'arrêt (sur
                     l'unité FORTRAN 01 à l'étiquette 100)
≫ SUP 100
≫ ACT
```

D - Commande SUPPRIMER:

Cette commande permet de supprimer un ou plusieurs points d'arrêt dans un module.

Elle est obligatoirement précédée par la commande "MODULE".

Nous rappelons que la définition des points d'arrêt se fait par la commande "STOP".

La suppression de tous les points d'arrêt dans un module supprime aussi l'action définie par la commande "MODULE". Donc si un utilisateur veut définir de nouveaux points d'arrêt dans ce même module, il devra auparavant émettre la commande "MODULE".

Syntaxe de la commande :

```
                  ( ADRPA  [[UNITE] NUMUNIT] )
                  (                          )
SUPprimer         ( {'}                      )
                  ( {&} HEXA                 )
                  (                          )
                  ( TOUT [ARRET ]            )
```

Signification des paramètres :

ADRPA	nom symbolique correspondant à une étiquette ou variable du programme.
NUMUNIT	numéro d'unité Fortran dans laquelle est défini le nom symbolique ($1 \leqslant NUMUNIT \leqslant 63$). = 1 par défaut, correspond au programme principal.
HEXA	adresse hexadécimale du programme Assembleur ou PL16
"'"	l'adresse hexadécimale est une adresse effective par rapport au début du programme
"&"	l'adresse hexadécimale est une adresse de relais.

EXEMPLE :

```
≫ MØDULE 120
≫ STØP &300
≫ ACT

> dialogue MUTEX


MØDULE 120 PRØCESSUS PØSTO1 STØP ADRESSE 300
≫ SUP &300
≫VISU VAR '210, '211
VISU VARIABLES
'210      '211
'0FFF     '0001
≫ACT

> dialogue MUTEX
```

Remarque :

Dans cet exemple l'adresse 300 correspond au contenu du relai situé à l'adresse 300.

E - Commande VISUALISER :

Au cours du dialogue, l'utilisateur peut demander la visualisation d'un certain nombre d'informations parmi lesquelles :

- le dernier échange (entrée-sortie - demande/réponse)
- un tableau (ZDC ou autre)
- suite de variables et instructions (si assembleur ou PL16)
- la pile des modules
- les registres au moment de la rencontre du point d'arrêt.

Remarque :

Visualiser le dernier échange signifie :

* Si c'est une requête du type Demande/Réponse, (dialogue entre le périphérique et le processus (le périphérique pouvant être un périphérique logique))on visualise le compte rendu de sortie.

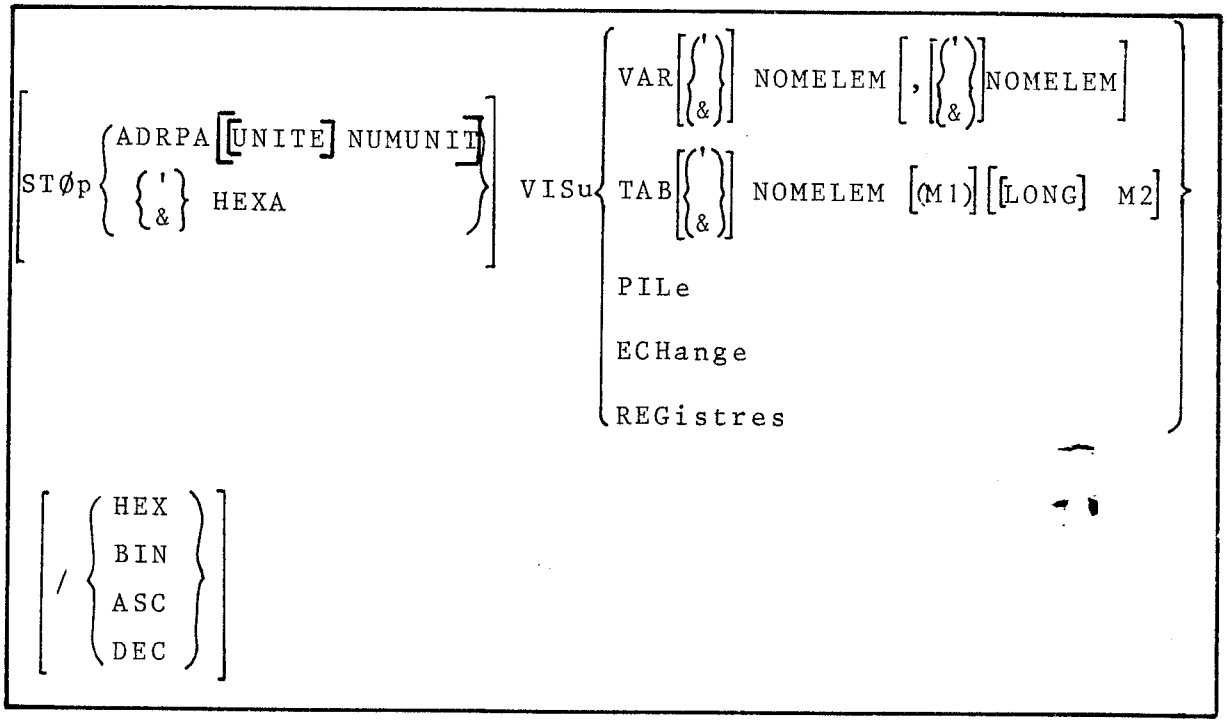
* Si c'est une sortie on visualise aussi le compte rendu de sortie.

* Si c'est une entrée, on visualise le compte rendu d'entrée ainsi que le nombre d'octets du buffer.

Si la commande est précédée du paramètre "STOP" l'action de visualisation est différée. La visualisation aura lieu sur la rencontre du point d'arrêt.

Si la commande n'est pas précédée du paramètre "STOP", alors l'action de visualisation est immédiate.

Syntaxe de la commande :



Signification des paramètres :

ADRPA — nom symbolique correspondant à une étiquette ou une variable du programme

NUMUNIT — numéro d'unité Fortran dans laquelle est défini le nom symbolique (01 ⩽ NUMNIT ⩽ 63). Par défaut ce paramètre est égal à 01 (il correspond au programme principal).

HEXA	adresse hexadécimale du programme Assembleur ou PL16
"'"	le paramètre HEXA ou NOMELEM correspond à une adresse hexadécimale par rapport au début du programme
"&"	le paramètre HEXA ou NOMELEM correspond à une adresse de relais (Ex. positionner un point d'arrêt sur une adresse de tableau en PL16)
NOMELEM	adresse symbolique ou hexadécimale (si précédé de caractère "'" ou "&") d'une variable ou d'un tableau
M1	déplacement par rapport au début du tableau, exprimé en mot et en décimal (début du tableau par défaut)
M2	longueur du tableau à visualiser, exprimée en octets (les deux premiers octets par défaut)
VAR	visualisation de variable simple
TAB	visualisation de tableau
PIL	visualisation de la pile des modules (toujours en décimal)
ECH	visualisation du dernier échange
REG	visualisation des registres au moment de l'arrêt du programme en mise au point. On visualise tous les registres (A-B-X-Y-C-L-W-K-P-S-SLO-SLE) au moment de l'arrêt. Ce paramètre a la même signification pour les programmes FORTRAN, notons qu'il est dans ce cas difficilement exploitable.
HEX-BIN ASC-DEC	Format de visualisation. Ce format n'a de signification que pour les variables simples, flottantes et tableaux. Par défaut, le format est l'hexadécimal.

Exemple :

```
>   DEBUG                          ⟶ activation de DEBUG
≫   MØD 30 MØDE TR
≫   STØ 21 VISU VAR I,J,K
≫   STØ 10 VISU TAB (10) LØNG 5 /DEC
≫   STØ 10 VISU PIL
≫   STØ 20 VISU ECH
≫   MØD 31 MØDE PP
≫   STØ 10
≫   ACT

≫   FIN DEBUG                      ⟶ fin de DEBUG.
```

Dans cet exemple les visualisations apparaissent sur le périphérique de sortie ; le retour au dialogue se fait au passage sur l'étiquette 10 du module 31. En effet le module 31 du biblio BIBCOM s'exécutera en Pas-à-Pas, c'est-à-dire avec retour au dialogue après passage sur un point d'arrêt.

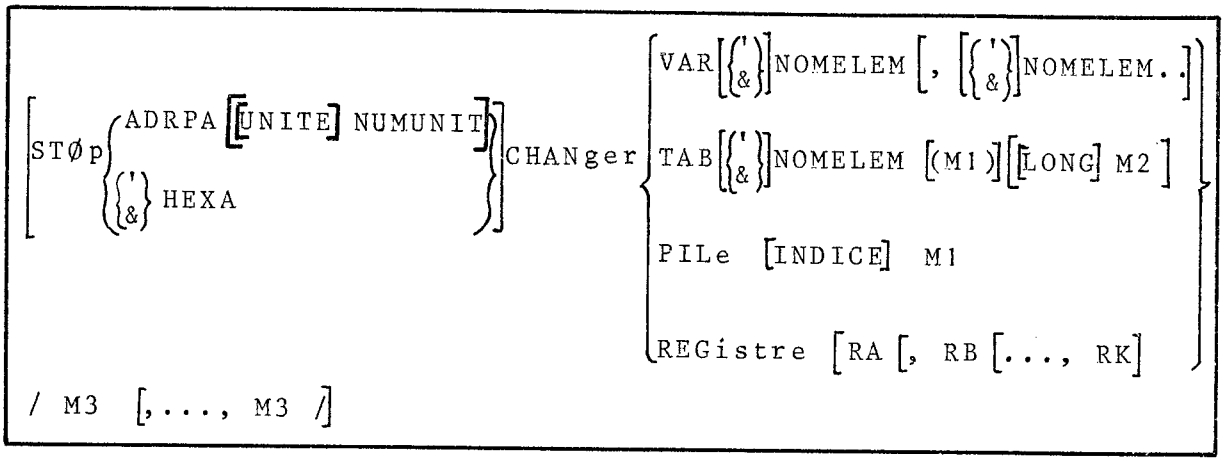
```
MØDULE 30 PRØCESSUS PØSTE1 STØP ADRESSE  0010
VISU TABLEAU
TAB (10)      + 1      + 2      + 3      + 4
  + 1         + 2      + 3      + 4      + 5
VISU PILE
  +600     +601     +602
MØDULE 30 PRØCESSUS PØSTE1 STØP ADRESSE 0020
VISU DERNIER ECHANGE
ENTREE          CPTRENDU = 0000
MØDULE 30 PRØCESSUS PØSTE1 STØP ADRESSE 21
VISU VARIABLES
I         J         K
'0000    '0000    '000F
MØDULE 31 PRØCESSUS PØSTE1 STØP ADRESSE 0010 ⟶ imprimé par DEBUG

≫ VISU VAR I, J /DEC
  VISU VARIABLES
  I         J
  +10       +3
≫ STØP 110 VISU ECH
≫ ACT
```

F - Commande CHANGER :

Comme dans la commande VISU, la commande peut être précédée ou non d'une demande de positionnement de point d'arrêt.

Syntaxe de la commande :



Signification des paramètres autres que ceux définis dans la commande VISU :

M3 : Nouvelle valeur que prendra l'élément spécifié. Ce qui précèdera M3 indiquera le format de modification.

[+] M3 = constante décimale positive

\- M3 = constante décimale négative

'M3 = constante hexadécimale

"M3"= constante ASCII

Remarque :

Nous avons utilisé ce type de représentation pour respecter la syntaxe des constantes Fortran sur le SOLAR.

Exemple :

```
> DEBUG
>>MØDULE 30 MØDE PP
>>STØ 100 VISU VAR I / ASCII
>>ACT
   MØDULE 30 PRØCESSUS PØSTE2 STØP ADRESSE 100
   VISU VARIABLES
   I
   "DE"
>>CHAN VAR I/50/
>>STØ 150
>>ACT
   {
   }
```

G - Commande GOTO :

Il peut arriver que l'utilisateur ne veuille pas exécuter une partie d'un module située entre deux étiquettes.

Pour résoudre ce problème, on utilise une commande de branchement inconditionnel sur une étiquette.

Syntaxe de la commande :

```
[STØP { ADRPA [[UNITE] NUMUNIT] }] GØTO { ADRPA }
      { 'HEXA                   }       { 'HEXA }
```

Cette commande pourra être émise derrière une demande d'activation en Pas-à-Pas ou en mode TR. Les paramètres sont ceux des commandes précédentes.

Exemple :

```
      {
>> MØD 31 MØDE PP
>> STØ 120
>> ACT
   MØDULE 31 PRØCESSUS PØSTE1 STØP ADRESSE 0120
>> VISU VAR  L,M
     L       M
  'C3C2    'C3C3
>> GØTØ 130
>> STØ  140
>> ACT
     }
```

H - Commande ENCHAINER :

DEBUG permet à l'utilisateur d'enchaîner un module particulier à un endroit quelconque du programme. Ce module doit obligatoirement être stocké dans le bibliothécaire "BIBCOM" car il s'exécute sous le contrôle de DEBUG. Un module particulier, faisant partie des modules système, peut être enchaîné, c'est le module 24. Il permet de sortir un DUMP complet du module en test.

Syntaxe de la commande :

```
[STØ p{ADRPA [[UNITE] NUMUNIT]}]   ENChaîner NUMØD
      {{'} HEXA                }
      {{&}                     }
```

Signification des paramètres :

NUMØD : Numéro de module à enchaîner.

Exemple :

```
>> MØDULE 31 MØDE PP
>> STØP '140
>> ACT
   MØDULE 31 PRØCESSUS PØSTE 1 STØP ADRESSE 0140
>> VISU VARIABLE '100, '110

   '0100  '0110
   '000F  '0000
>> ENCH 24                                  réponse de l'utilisateur ↓
   PERIPHERIQUE DU POOL POUR LA SORTIE DU DUMP: LP
>> FIN DEBUG
```

I - Commande activer :

Pour que le système Mutex puisse prendre en compte les points d'arrêt sur les modules à tester, il faut sortir du dialogue DEBUG.

C'est la commande ACT qui le permet.

Syntaxe de la commande :

ACT

Sur l'émission de cette commande, le processus DEBUG est mis en attente et donc les caractères " >> " n'apparaissent plus sur le périphérique de sortie.

Si un module est en attente sur point d'arrêt, l'émission de la commande ACT permet sa réactivation en mémoire. Le module en mise au point sera remis en attente sur rencontre d'un nouveau point d'arrêt, ce qui permettra de "redonner la main" au dialogue DEBUG (s'il s'exécute en mode Pas-à-Pas).

III.3. - LES TABLES

III.3.1. - LES TABLES DE SYMBOLES

Nous avons vu dans le chapitre précédent que l'outil de mise au point réalisé est adapté à la mise au point de programmes écrits en FORTRAN, PL16 ou ASSEMBLEUR sur le SOLAR. C'est un outil interactif de haut niveau. Il permet à l'utilisateur d'acquérir en dynamique des informations sur la conduite de son programme. Pour les programmes écrits en FORTRAN, DEBUG accepte de travailler avec des adresses symboliques. Pour les programmes écrits en PL16 et ASSEMBLEUR, DEBUG accepte des adresses hexadécimales. Cette technique de mise au point demande l'accès à deux sortes de tables des symboles :

- Une table des symboles pour les programmes FORTRAN
- Une table des symboles pour les programmes PL16 et ASSEMBLEUR.

A - Table des symboles pour programmes FORTRAN :

Elle est stockée en fin du fichier IMAGE MEMOIRE (code objet) du module dans le fichier bibliothécaire. Elle est composée d'une suite de quadruplets dont la description est donnée en annexe. Dans chacun de ces quadruplets un emplacement est réservé au positionnement d'un point d'arrêt. Les symboles contenus dans cette table ne sont pas chargés dynamiquement par l'utilisateur devant sa console de visualisation. La table des symboles est obtenue par le compilateur Fortran ainsi que l'éditeur de liens.

Le compilateur FORTRAN :

Le compilateur FORTRAN fournit à la fois la table des symboles du programme source de même que la table des sections; cette dernière est chargée en début du fichier BO (Binaire objet obtenu après compilation). La table des symboles est détruite en fin de compilation d'une unité FORTRAN (08). Une modification du compilateur FORTRAN permet de récupérer cette table, sur le fichier BO, à la suite de la table des sections. La modification est simplifiée sur le SOLAR. La table des symboles est déjà structurée pour sortir une carte mémoire en fin d'unité Fortran, avant sa destruction (à condition d'avoir précisé l'option MAP dans la commande d'activation du compilateur (02). Une nouvelle branche sera rajoutée au compilateur. Elle aura le même rôle que la branche qui sort la carte.

L'édition ne se fera pas sur le listing de compilation mais sur le fichier BO (option TSYMB). Un filtre permettra de ne garder que les symboles utiles (annexe). Un CKECKSUM sera placé entre l'image mémoire et la table des symboles. Malgré le filtre, le risque de génération de symboles inutiles est important. De plus la table peut être vite saturée. Sa longueur maximum aujourd'hui est de 256 mots (ce qui correspond à deux secteurs sur le disque). Une suppression de symboles ainsi qu'une réorganisation de la table seront permises par l'utilitaire TSFOR (fonctionne sous le système MUTEX en BACKGROUND).

L'éditeur de liens plus le chargeur :

L'éditeur de liens a pour rôle de faire le lien entre les différentes unités Fortran compilées. Une modification lui permettra de tenir compte de la table des symboles et d'en faire la gestion (associer une adresse relative à un symbole). C'est donc lui qui permettra d'obtenir la table des symboles finale. L'emplacement final de la table sera dans un article du fichier indexé obtenu avec le chargeur disque, l'article TSYMB. Enfin l'éditeur de liens imprimera sur le listing de l'utilisateur les différents numéros d'unités Fortran (ce numéro est indispensable à l'utilisateur car il peut définir deux étiquettes identiques sur deux unités de compilation différentes).

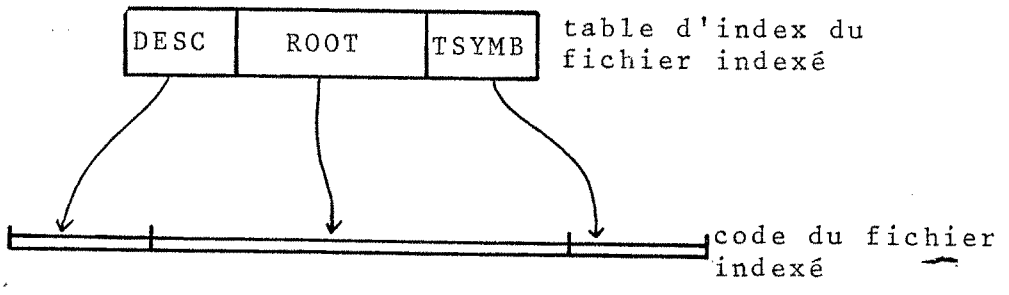


Figure 11. : Tous les fichiers obtenus par le chargeur disque sont des fichiers de type indexé.

DESC = Descripteur du fichier

ROOT = Image mémoire du programme

TSYMB = Table des symboles.

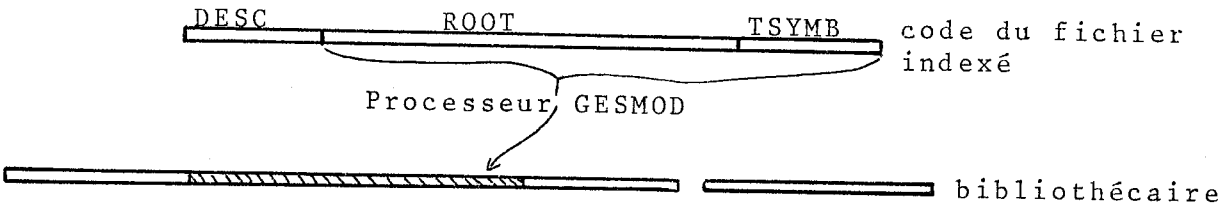


Figure 12. : Chargement du module avec sa table des symboles dans un bibliothécaire.

L'utilitaire TSFOR :

Cet utilitaire n'a de signification que pour les programmes écrits en langage FORTRAN ou tout autre langage délivrant une table de symboles ayant la même structure.

Son rôle est de supprimer des symboles dans la table de manière à obtenir une table de longueur inférieure ou égale à deux secteurs. Ce qui représente environ une cinquantaine de symboles par module. La programmation sous MUTEX est très modulaire, il s'en suit beaucoup de modules de faible longueur pour une même application.

Il est activable sous le background de MUTEX ou sous le système d'exploitation BOS-D (06)

Il est souple d'utilisation. La référence à un symbole se fait par son nom suivi du numéro d'unité FORTRAN.

En fin de suppression des symboles, il y a édition de la longueur de table.

L'activation de cet utilitaire se fait par :

CALL TSFOR

L'utilisateur doit indiquer par le biais d'un dialogue le nom du fichier contenant la table des symboles à modifier.

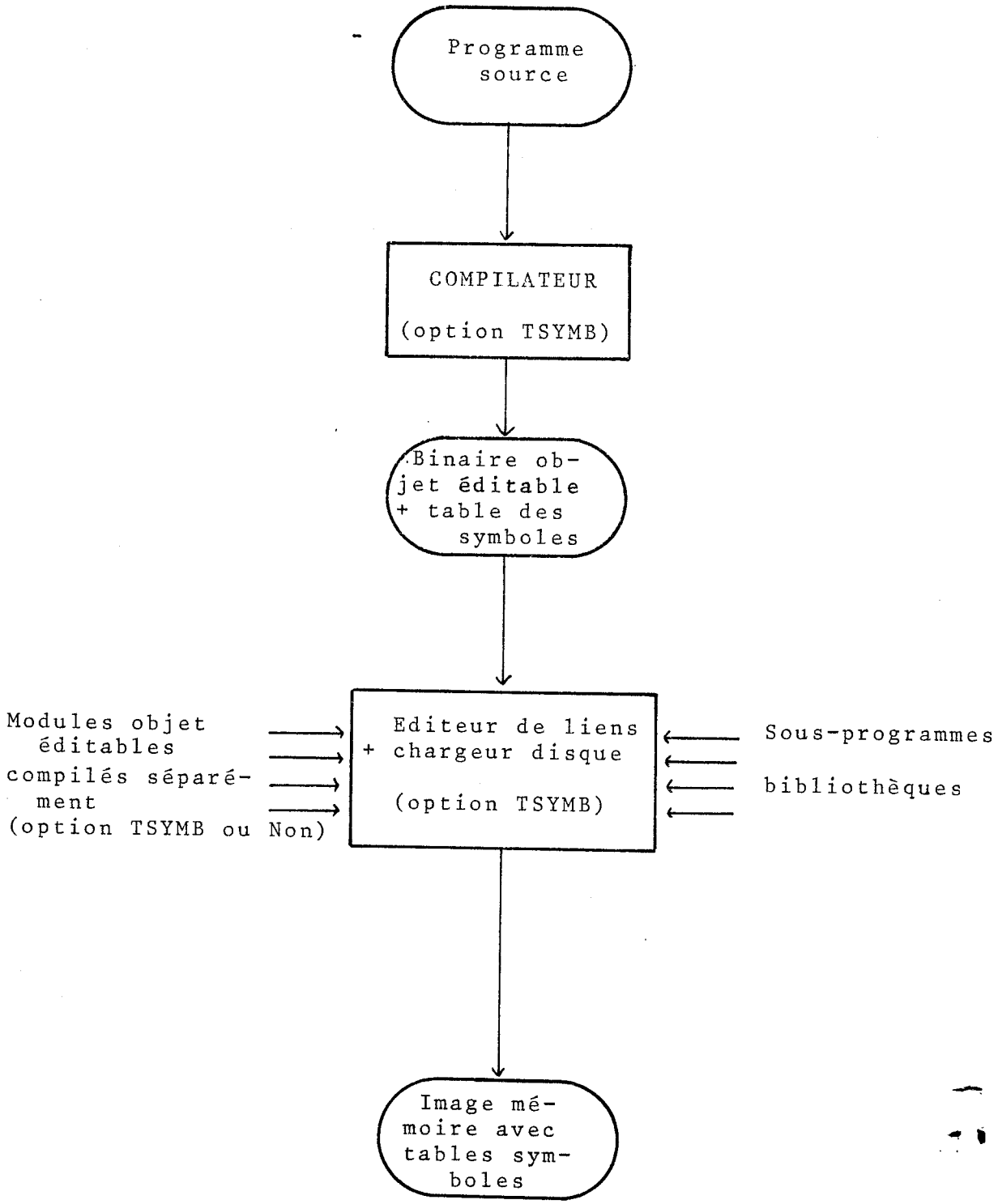


Figure 13 : Production de programme sur solar.

Remarque :

L'édition de liens + chargement de l'image mémoire sur disque dans un fichier indexé sont résolus par un seul processus appelé LKLOAD (08).

B - Table des symboles pour programmes ASSEMBLEUR et PL16

Pour ces deux langages la table des symboles est créée directement sous le dialogue de l'outil de mise au point, lors de la demande de points d'arrêt dans un module.

Les adresses sont données en hexadécimal sous forme d'adressage direct ou indirect.

Elles sont rangées successivement derrière le module dans le bibliothécaire. Une zone est réservée à cet effet au moment du chargement du module dans le bibliothécaire par le processeur GESMOD.

Cette zone a une longueur de deux secteurs ; elle est alignée sur une frontière de secteur (un secteur est égal à 128 mots).

Un symbole défini par l'utilisateur tient sur un mot, ce qui permet de pouvoir stocker plus de 250 symboles dans la table.

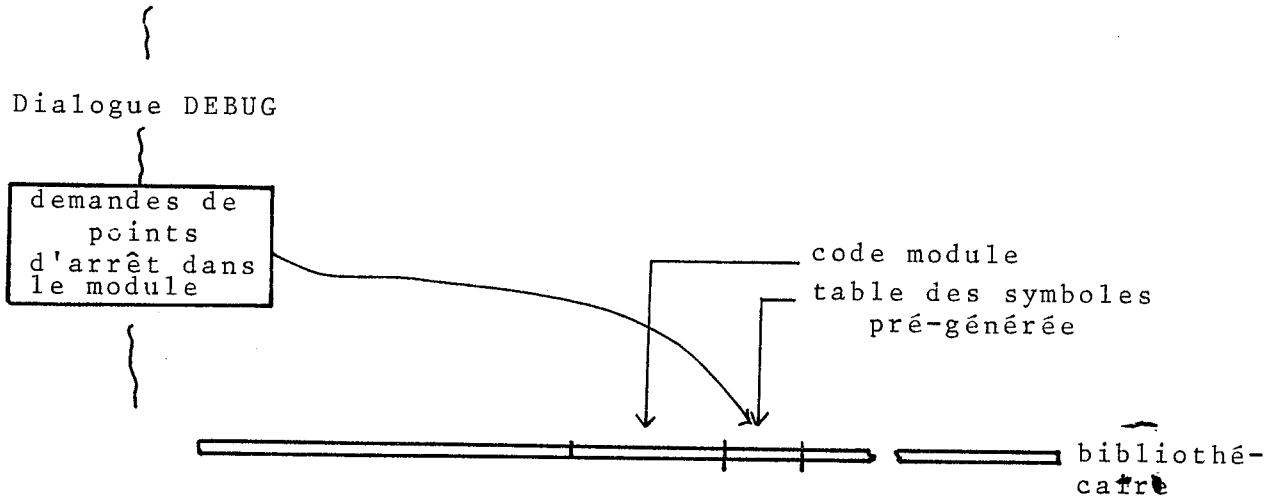


Figure 14 - définition de points d'arrêt dans la table des symboles PL16 ou ASSEMBLEUR.

C - Remarque :

L'outil de mise au point a la possibilité de reconnaître le type d'adresse qu'il manipule (adresse hexadécimale ou adresse symbolique). Pour cela il accède au premier mot de la table des symboles qui a une signification particulière.

La table des symboles est chargée en mémoire par le système d'exploitation MUTEX en même temps que le module. Cette technique évite de nombreux accès disques qui diminueraient les performances du système (figure 15).

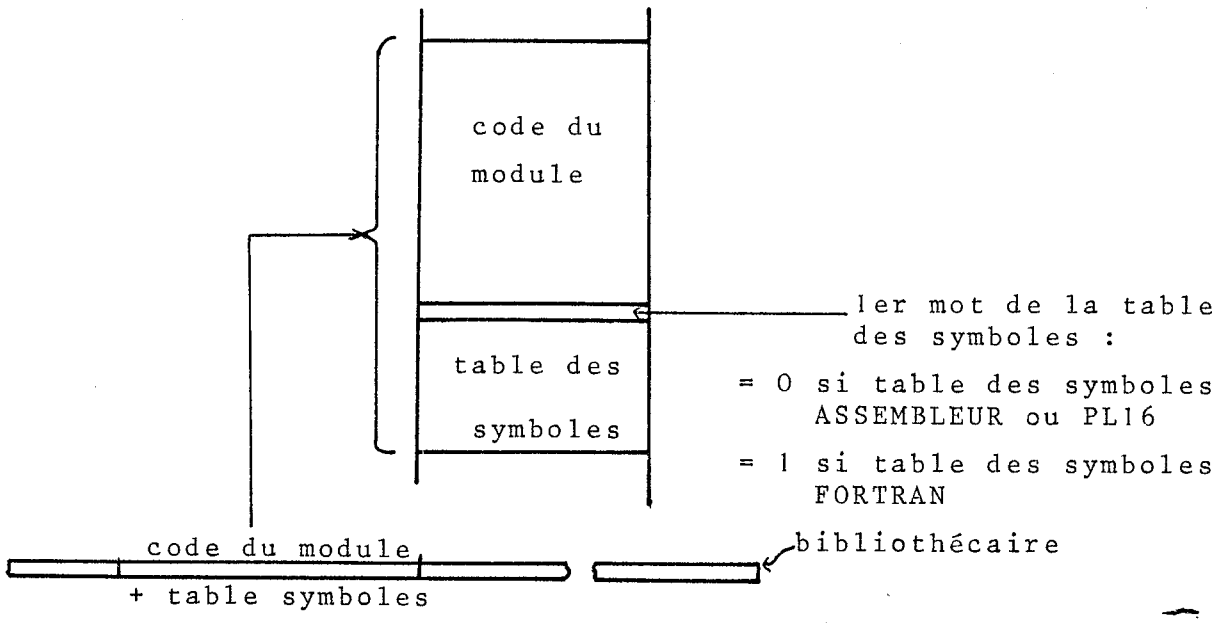


Figure 15 - chargement d'un module en mémoire.

III.3.2. - LA ZDCPARAM

Les 256 premiers mots de la ZDC sont réservés à l'interpréteur pour le stockage des paramètres des commandes. Nous avons décidé d'utiliser derrière la ZDC une table de 256 mots appelée ZDCPARAM.

Cette zone est utilisée pour enregistrer les différentes actions différées correspondant aux différents modules à mettre au point.

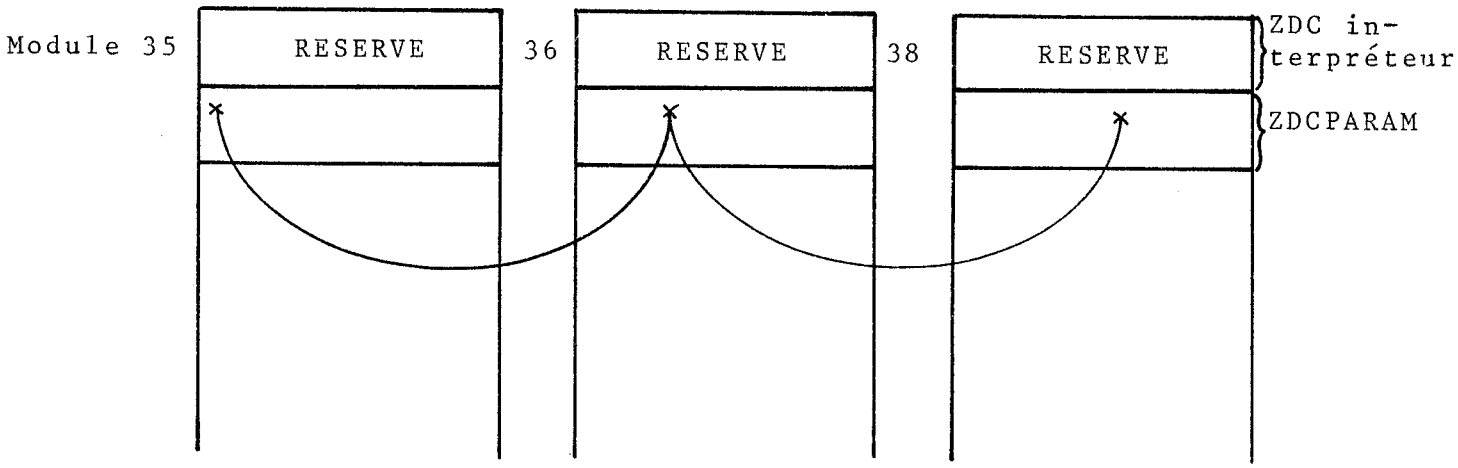


Figure 15 : Passage de paramètres par la ZDCPARAM.

Nous avons étendu la ZDC à une zone que l'on appelle ZDCPARAM.

D'un point de vue interne, la ZDC est l'ensemble ZDC interpréteur plus ZDCPARAM.

La ZDCPARAM est découpée en blocs de 10 mots initialisés à zéro au moment de l'activation de DEBUG.

Ces blocs sont chaînés entre eux. Le premier mot de chaque bloc pointe sur l'adresse du bloc suivant.

Ces blocs sont remplis au fur et à mesure que l'on définit des actions sur les modules à mettre au point.

Les six premiers mots de la ZDCPARAM ont une signification particulière :

MØTO = adresse du bloc concernant la dernière commande module émise

MØT1 = adresse du premier bloc alloué

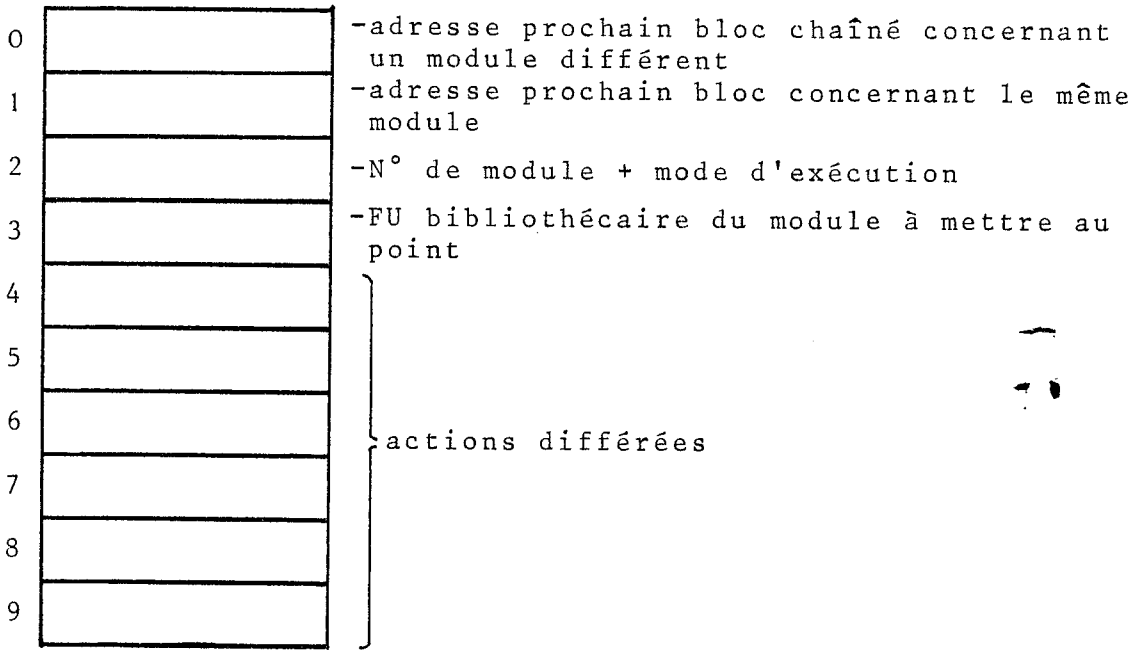
MØT2 = adresse du premier bloc libre

MØT3 = adresse du dernier bloc alloué

MØT4 = réservé

MØT5 = nombre de blocs alloués (ce qui correspond au nombre de modules en test).

La structure d'un bloc est la suivante :

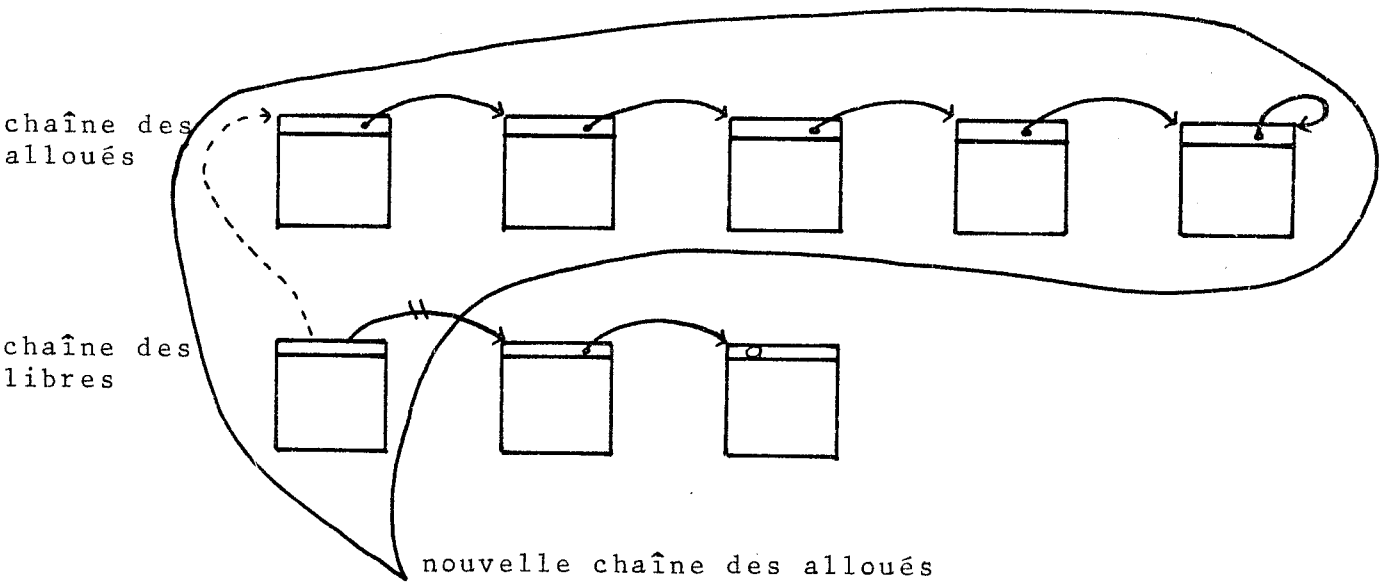


Insertion d'un bloc pour un nouveau module

Tous les blocs alloués sont chaînés entre eux, de même que les blocs libres. Nous obtenons ainsi deux chaînes (1 chaîne des libres, 1 chaîne des alloués).

Pour chaîner un bloc de la chaîne des libres vers la chaîne des alloués, on prend le premier de la chaîne des libres et on l'insère au début de la chaîne des alloués.

Le dernier de la chaîne des alloués pointe sur lui-même.



Remarque :

- au lancement du DEBUG, il n'existe qu'une chaîne des blocs libres.
  Le premier mot de chaque bloc pointe sur le bloc suivant, comme cela est indiqué par les flèches sur le schéma ci-dessus.

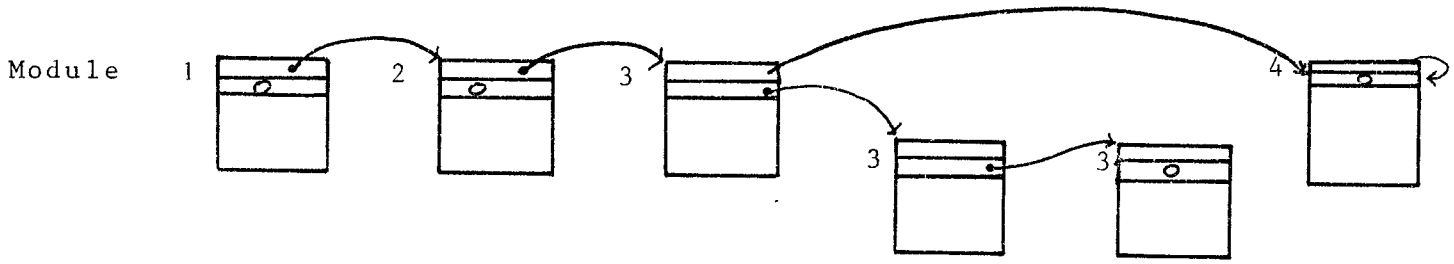
- le dernier bloc de la chaîne des blocs libres a son premier mot égal à 0.

Insertion d'un bloc pour un même module

C'est le deuxième mot de chaque bloc qui permet de chaîner entre eux les différents blocs concernant le même module.

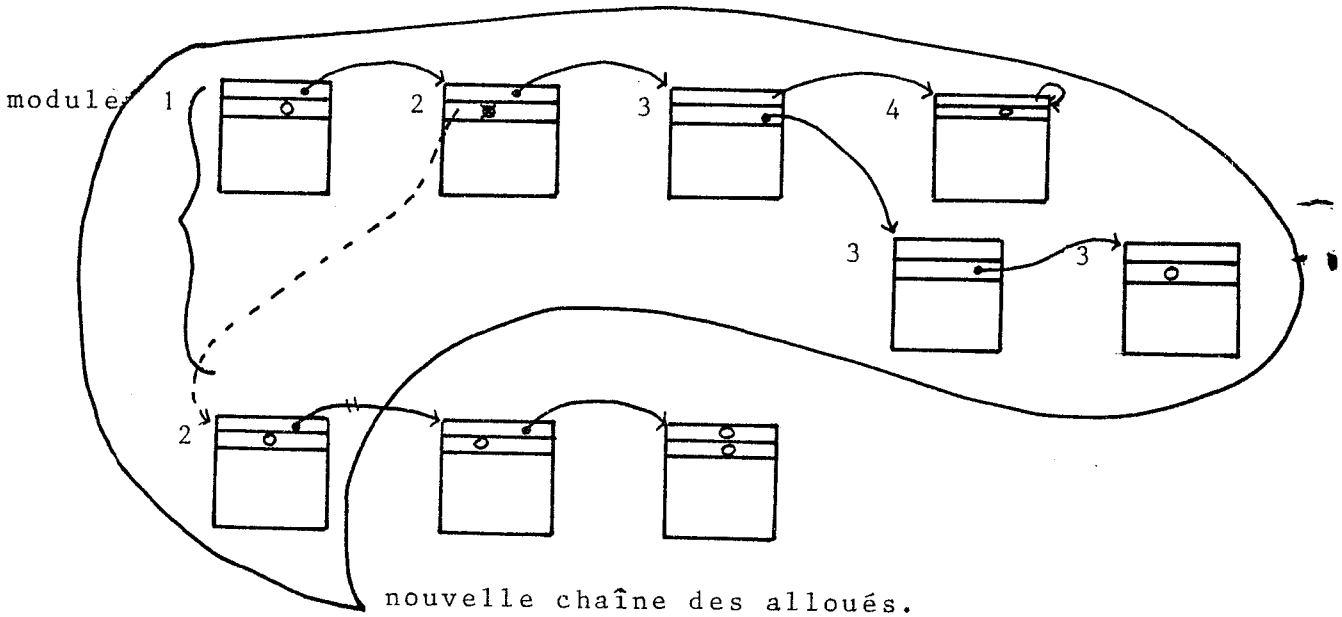
L'insertion se fait derrière le dernier bloc chaîné concernant le même module.

Exemple de chaînage :



N.B. Si le 2ème mot contient 0, alors il y a un seul bloc pour le module.
Si le 2ème mot est différent de 0, alors il y a d'autres blocs chaînés pour ce même module. Le deuxième mot contient l'adresse du bloc suivant concernant le même module.

Exemple de chaînage d'un bloc derrière le dernier bloc concernant le module 2 :



Remarque :

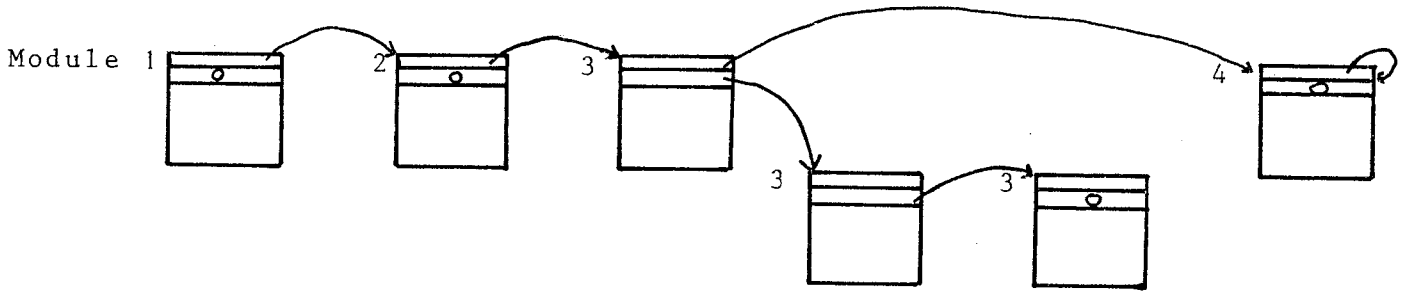
Le premier mot du bloc alloué est sans signification.

Suppression d'un bloc :

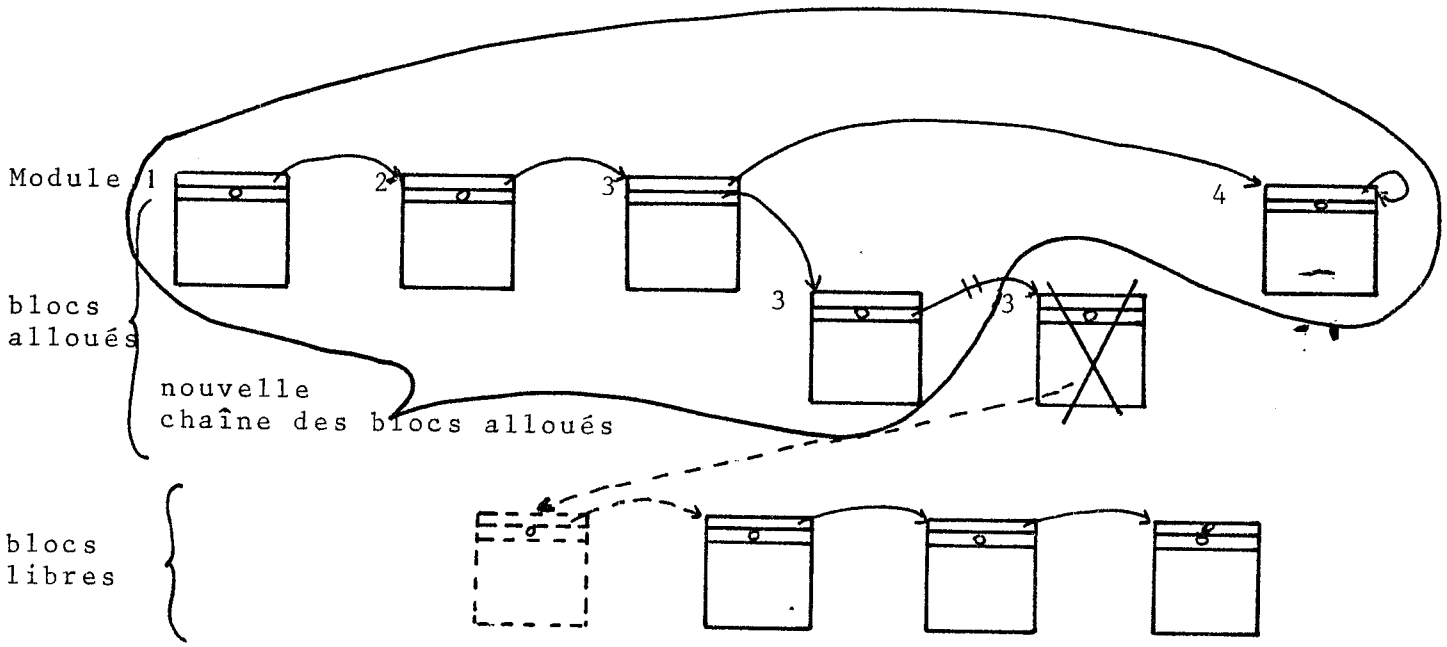
Le bloc supprimé de la chaîne des alloués vient en tête de la chaîne des blocs libres. Pour trouver le bloc qu'il faut supprimer, il suffit de parcourir la chaîne des alloués depuis le début jusqu'au bloc à supprimer.

Si l'on doit supprimer tous les blocs concernant un même module (Ex: supprimer tous les points d'arrêts d'un module), alors il faut supprimer d'abord le dernier bloc concernant ce module, puis l'avant dernier et ceci jusqu'au premier bloc qui lui aussi est à supprimer.

Ex: Nous voulons supprimer tous les blocs concernant le module 3 :

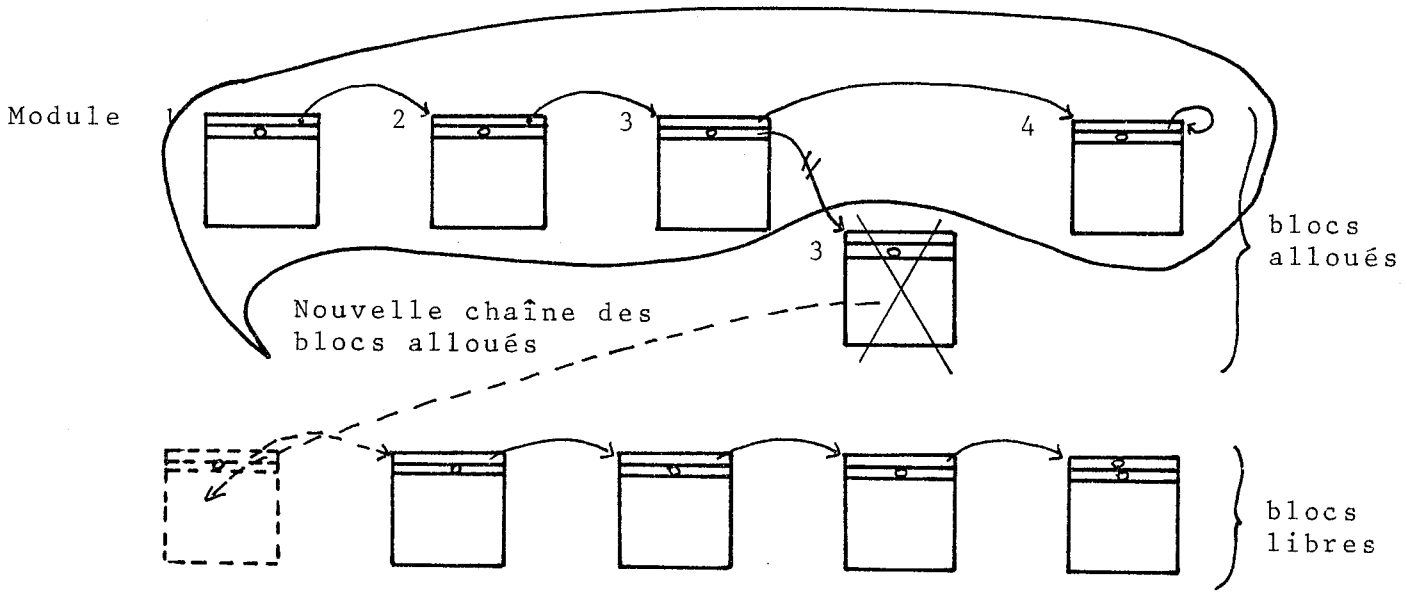


a) Suppression du troisième bloc (le dernier)



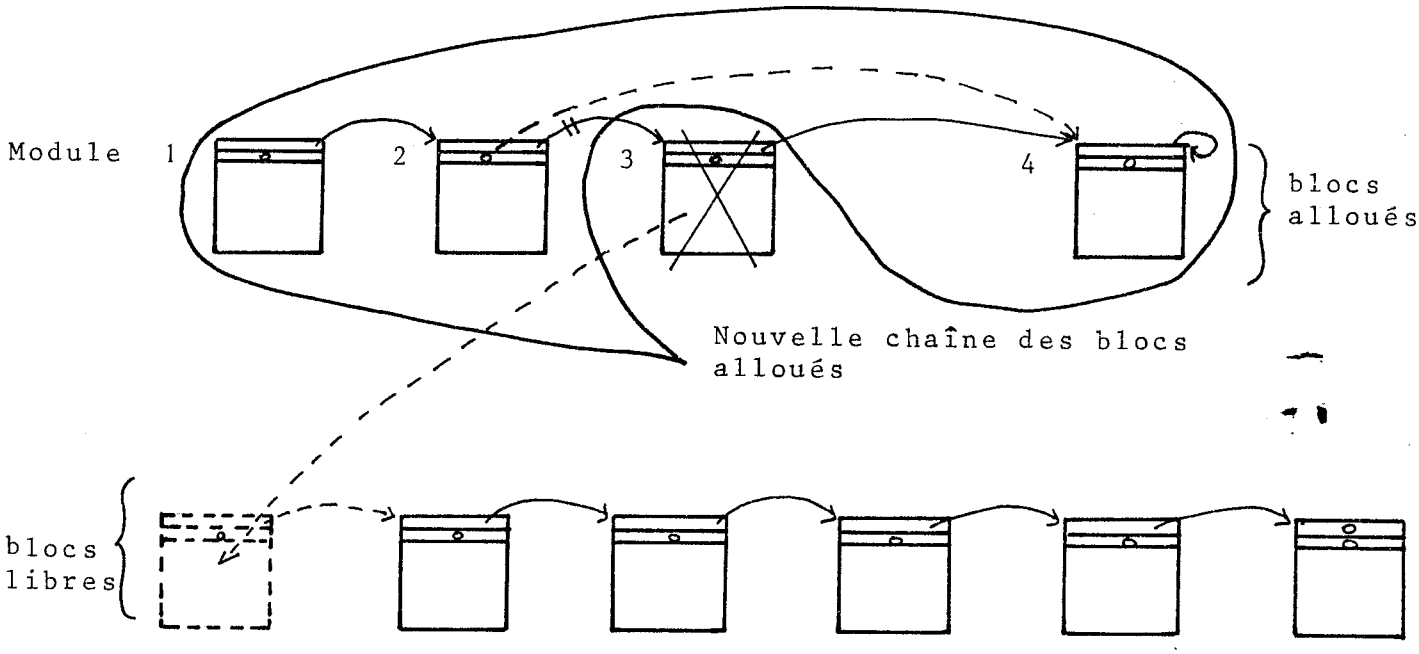
Le deuxième mot du deuxième bloc du module 3 prend la valeur zéro. Il devient le dernier chaîné. Le troisième bloc devient le premier bloc dans la chaîne des blocs libres. Son premier mot pointe sur l'adresse du prochain bloc libre.

b) Suppression du deuxième bloc (l'avant-dernier)



Le deuxième mot du seul bloc qu'il reste alloué pour le module 3 prend la valeur zéro. Le deuxième bloc qui lui était chaîné devient le premier bloc dans la chaîne des blocs libres. Son premier mot pointe sur l'adresse du prochain bloc libre.

c) Suppression du premier bloc (le seul qui reste concernant le module 3)



A ce niveau, il n'existe plus de bloc concernant le module 3. Le premier mot du bloc concernant le module 2 pointe maintenant sur le bloc concernant le module 4 . Le seul bloc qui restait concernant le module 3 devient le premier bloc parmi les blocs libres.

CHAPITRE IV

PRINCIPE DE FONCTIONNEMENT DE DEBUG

DEBUG est composé d'un ensemble de modules non résidents (ils sont chargés en mémoire centrale au moment de leur activation).

MUTEX a lui aussi une structure en modules dont certains sont en relation étroite avec ceux de DEBUG. La figure 16 montre l'interface entre les deux systèmes.

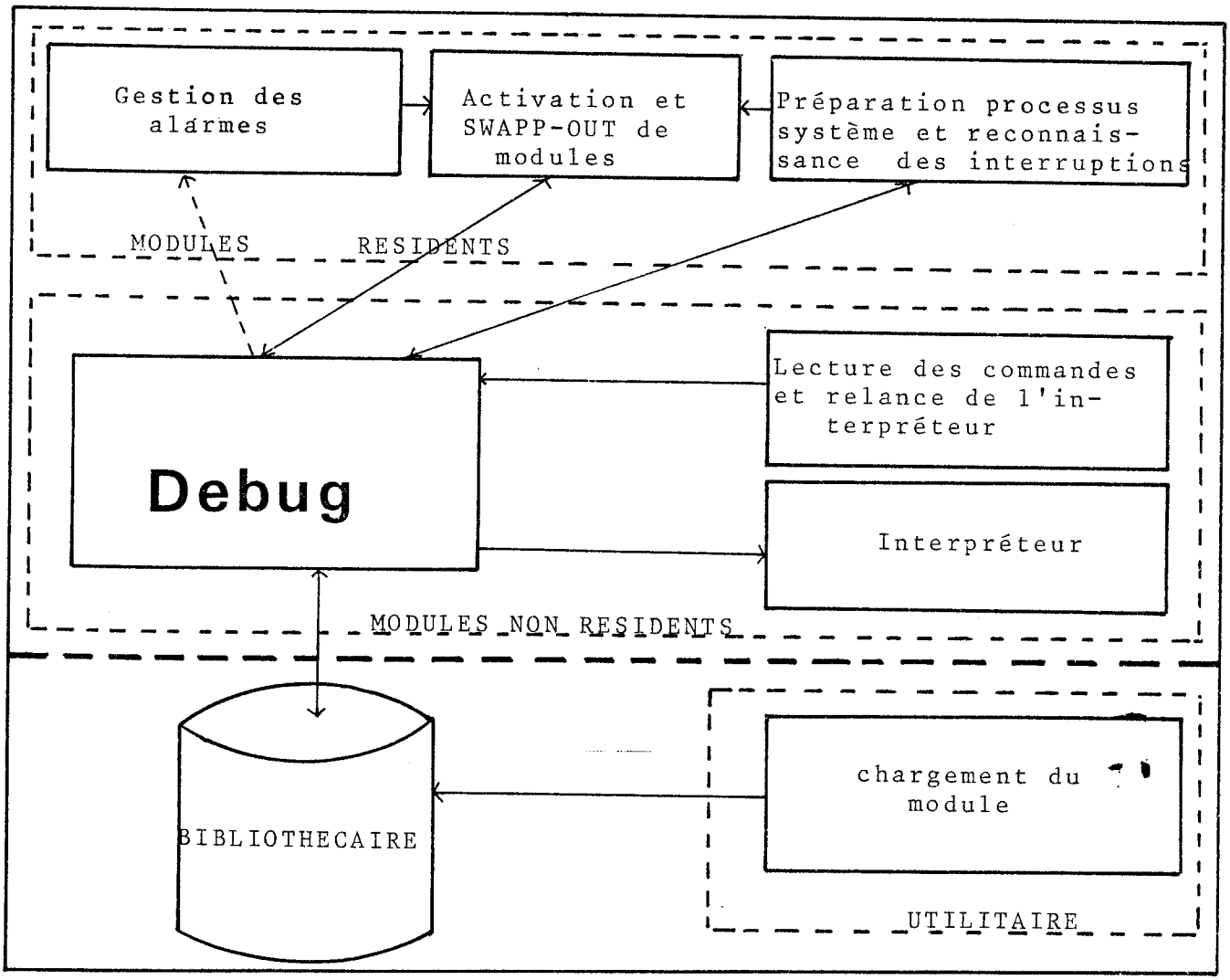


Figure 16. : Interface DEBUG-MUTEX.
Les flèches indiquent les relations directes ou indirectes entre les modules de DEBUG et ceux de MUTEX.

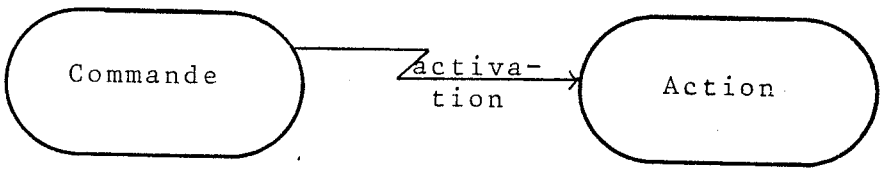
IV.1. - FONCTIONNEMENT DES DIFFERENTS MODULES DE DEBUG

DEBUG est structuré en 12 modules. Le principe de fonctionnement de chacun d'eux est donné dans le paragraphe suivant. Chaque module est lié à une action.

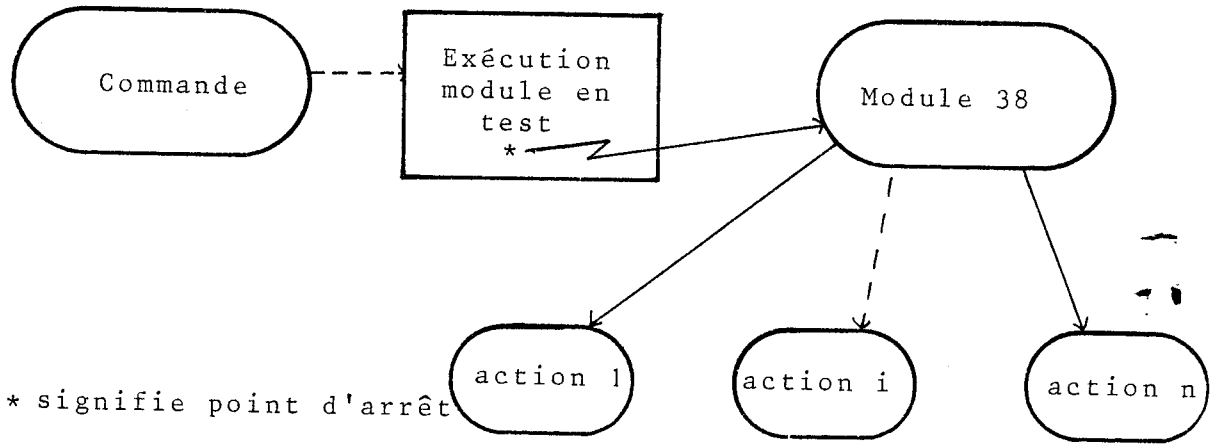
Selon l'action demandée (immédiate ou différée à la rencontre d'un point d'arrêt), l'activation d'un module peut se faire de deux manières différentes :

- directement après l'interprétation d'une commande opérateur s'il s'agit d'une demande d'action immédiate.
- après le passage sur un point d'arrêt lorsqu'il s'agit de demandes d'actions différées. Dans ce cas, c'est un module particulier (module 38) qui est toujours activé et dont l'exécution a pour but de rechercher quelles sont les actions demandées lors de l'émission des commandes.

Action immédiate :



Action différée :



* signifie point d'arrêt

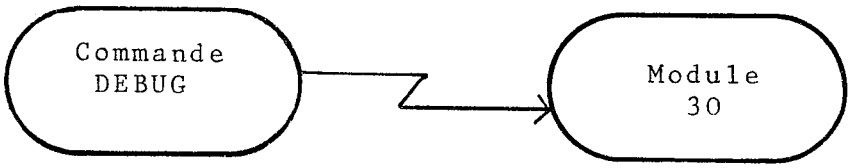
Remarque :

Dans la suite du chapitre nous appellerons :

NUMPROCTEST : le processus en mise au point

NUMPROCENCOURS : le processus mis en attente sur activation de DEBUG.

A - Module activé sur la commande "DEBUG"



La commande "DEBUG" active le module numéro 30.

Ce module rend actif le processus n° 1 (DEBUG). Ce dernier s'exécute sur le poste où est émise la commande.

a) Supposons que l'utilisateur veuille activer le DEBUG sur le poste 01 et mettre au point une application se déroulant sur le poste 02 :

   (1) Sur le poste 00 (poste système), la commande émise sera :

   / LAN APP SYSDBG SUR PØST01

   (2) Sur le poste 01, la commande émise sera :

   > DEBUG SUR PØST02

b) Supposons maintenant que l'utilisateur veuille mettre au point une application se déroulant sur le poste 02 et que l'activation du DEBUG soit aussi demandée sur le poste 02 :

   (1) Sur le poste 00, la commande émise sera :

   / LAN APP SYSDBG SUR PØST02

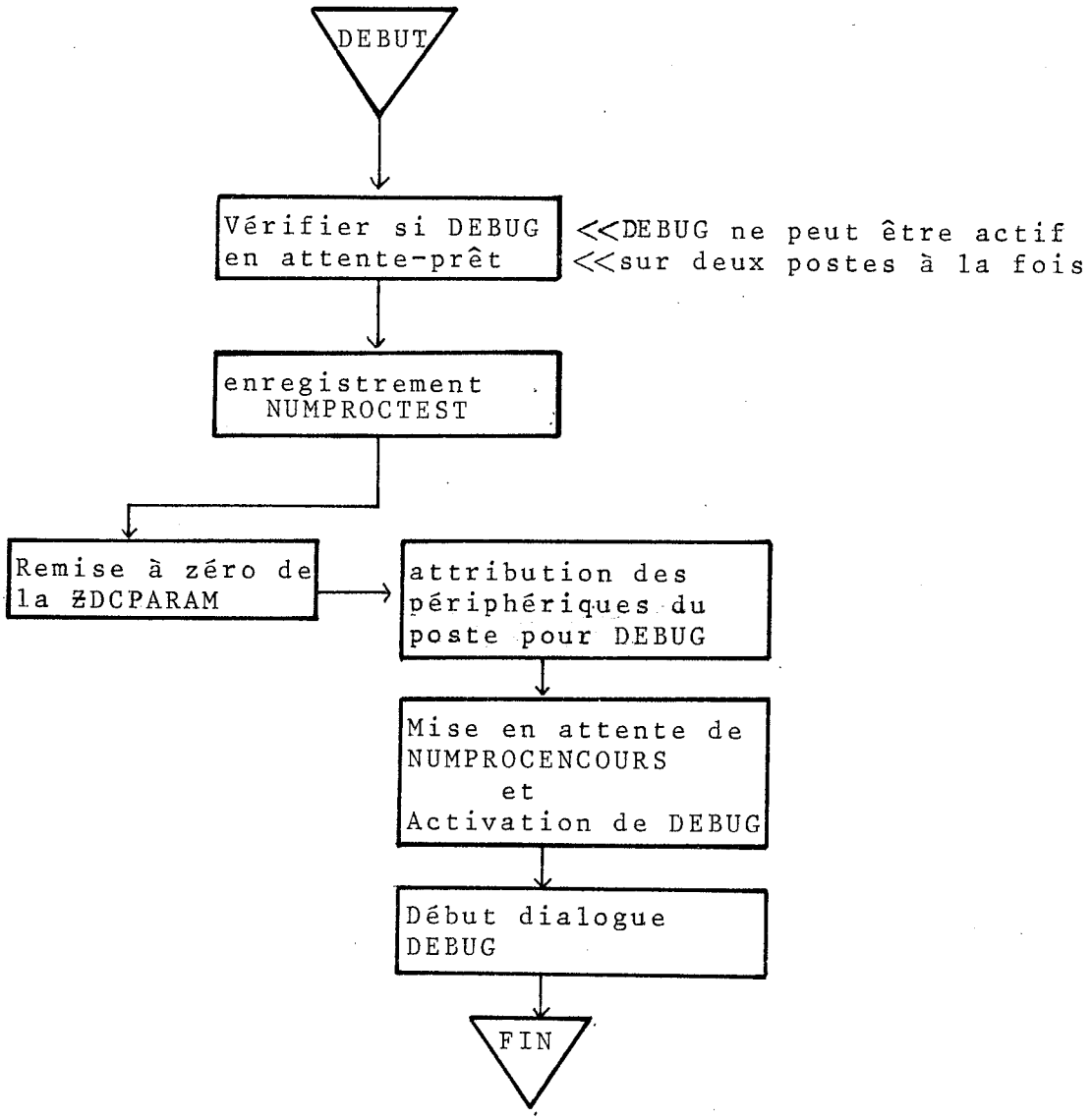
   (2) Sur le poste 02, la commande émise sera :

   > DEBUG

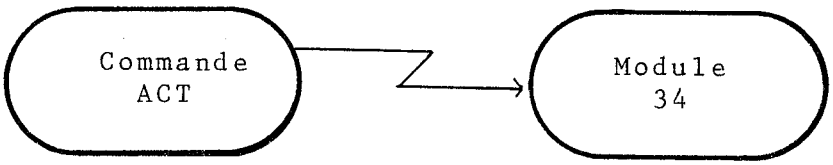
MUTEX n'autorise pas l'exécution simultanée de deux processus sur un même poste. Le module 30 doit mettre en attente le processus sur lequel vient s'exécuter le processus DEBUG. Il est appelé NUMPROCENCOURS. S'il s'agit d'un processus à mettre au point, alors il est mis en attente seulement pendant le dialogue opérateur, il est réactivé en fin de dialogue. S'il s'agit d'un processus quelconque, alors il est mis en attente durant toute la session de mise au point.

Nous avons vu dans le paragraphe précédent qu'une table appelée ZDCPARAM servait au passage de paramètres entre les différents modules du système. A chaque début de session, cette table est remise à zéro. C'est le module 30 qui en a la charge.

Organigramme de principe :



B - Module activé sur la commande "ACT"



La commande "ACT" active le module 34.

Ce module fait passer le processus 1 en attente-prêt.

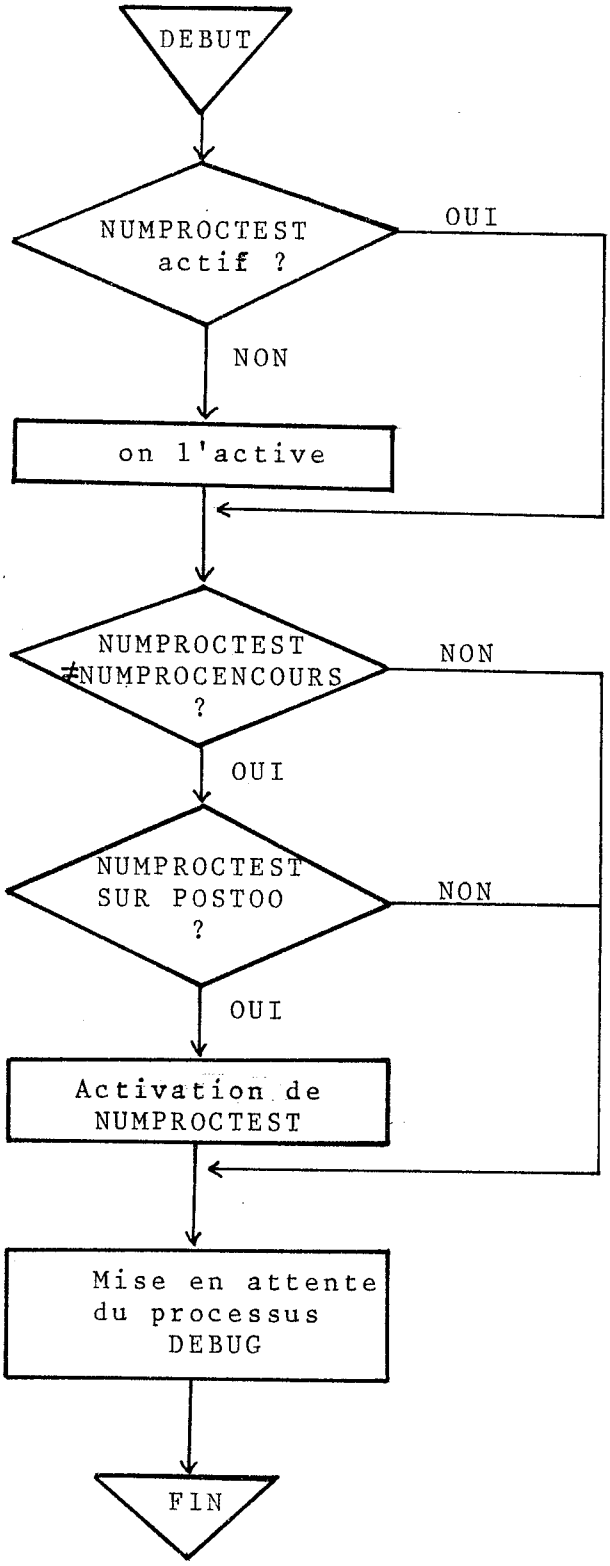
Son exécution a pour conséquence de mettre fin au dialogue DEBUG, d'activer le processus mis en attente sur activation du dialogue si celui-ci est à mettre au point ou si le dialogue DEBUG est émis depuis le poste système.

Remarque :

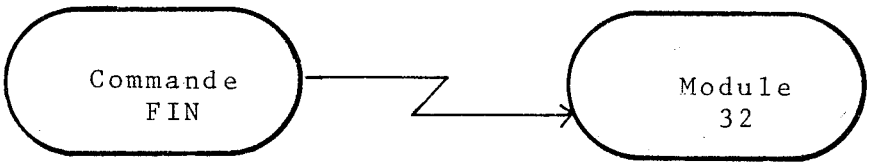
Sur activation du module 34, le processus en test (NUMPROCTEST) peut être :

- actif s'il n'a encore jamais été mis en attente sur un point d'arrêt
- en attente sur rencontre d'un point d'arrêt dans un module.

Organigramme de principe :



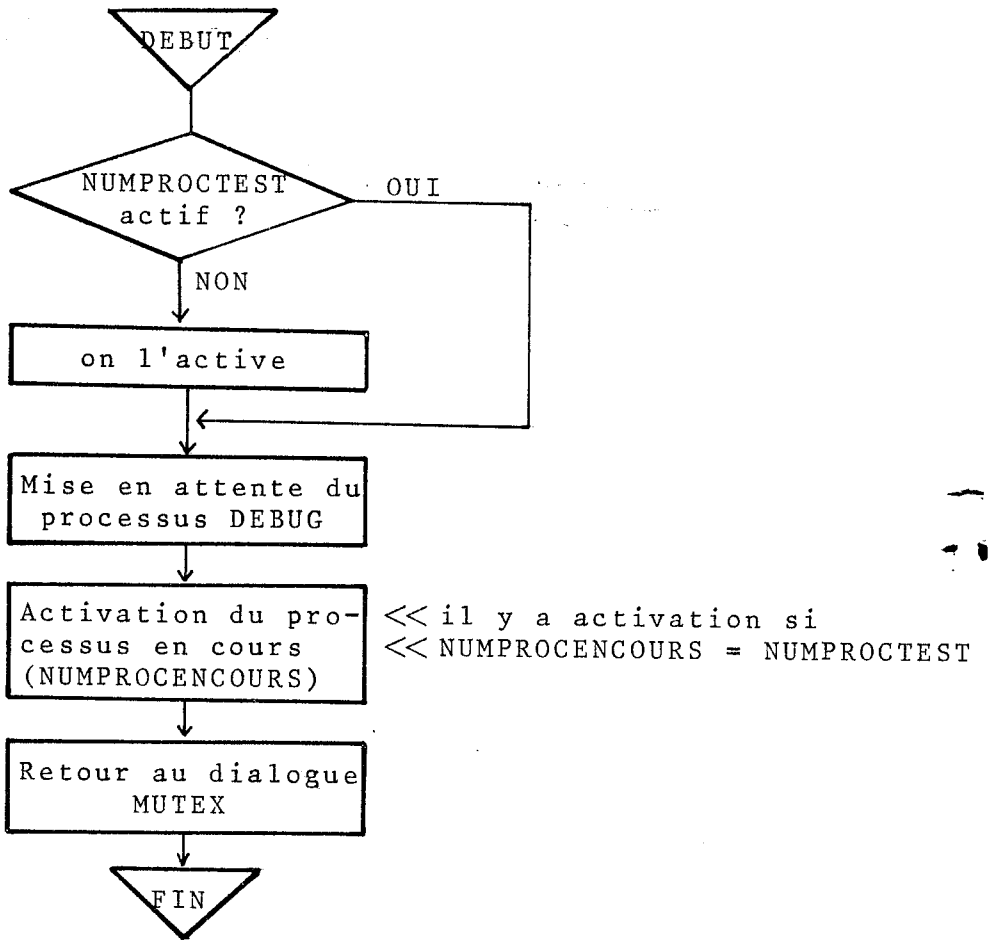
C - Module activé sur la commande "FIN"



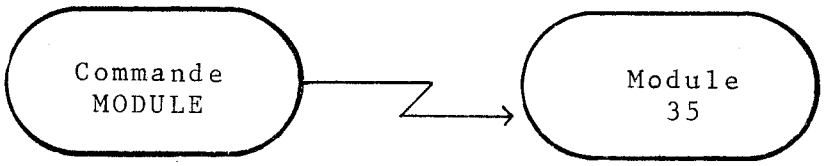
La commande "FIN" active le module 32.

Ce module détermine la fin d'une session de mise au point. Son exécution a pour conséquence de réactiver le processus en test (NUMPROCTEST) dans le cas où il aurait été mis en attente sur un point d'arrêt ; de mettre le processus DEBUG en attente d'une autre session de mise au point le cas échéant ; de réactiver le processus mis en attente par le processus DEBUG (NUMPROCENCOURS).

Organigramme de principe :



D - Module activé par la commande "MODULE"



La commande "MODULE" active le module 35.

Ce module définit un des modules du processus en test sur lequel on veut faire des demandes :

- de points d'arrêt
- d'actions immédiates ou différées.

Son exécution a pour conséquence l'allocation d'un bloc dans la table ƵDCPARAM.

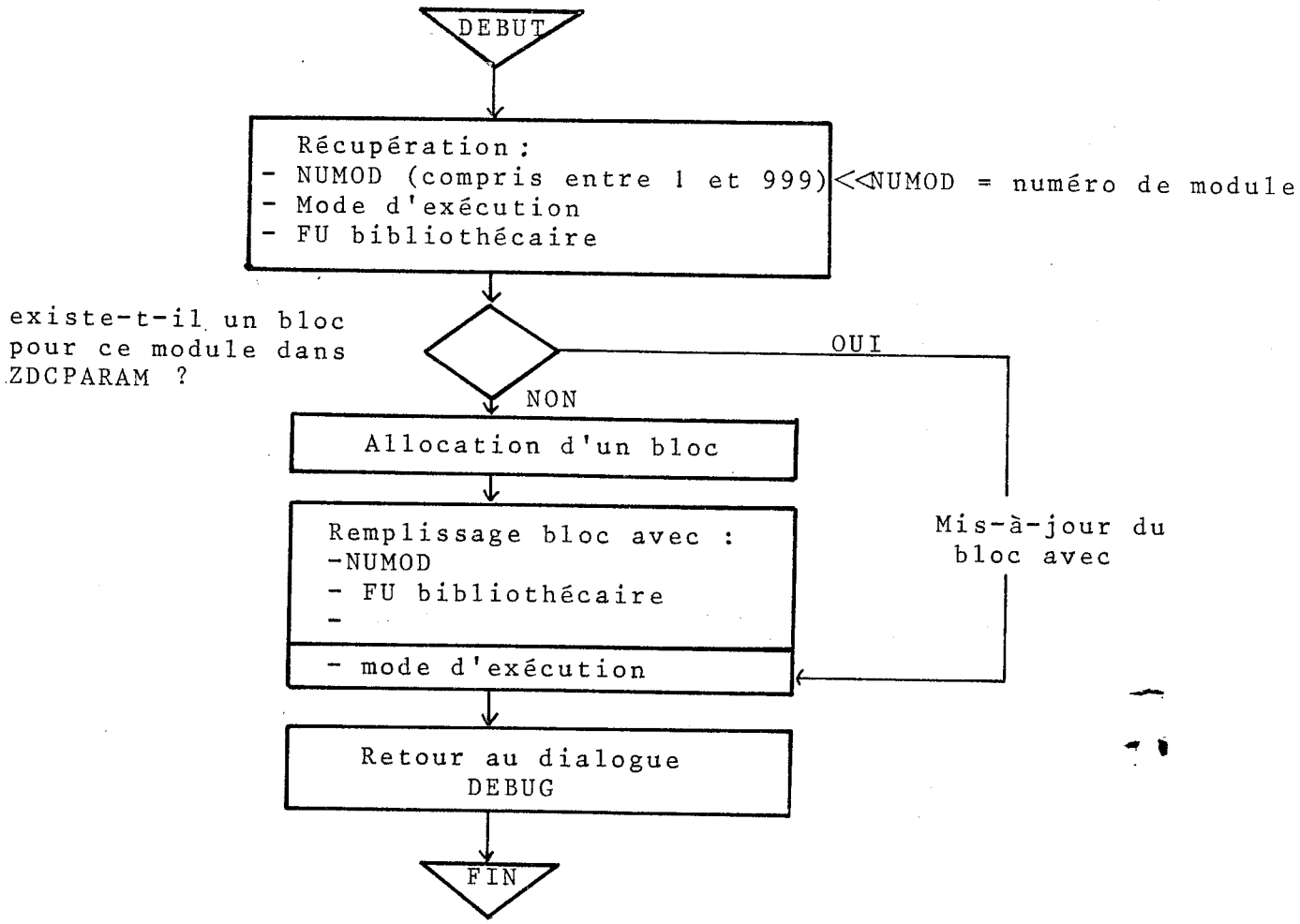
Une fois le bloc alloué, il y a stockage de trois paramètres dans le bloc :

- numéro de module
- mode d'exécution du module
- numéro de la FU bibliothécaire où est stocké le module. Nous avons vu qu'un fichier bibliothécaire était repéré par son nom. Ce fichier est stocké sur une partie physique du disque appelé "Fonctionnel unit" (FU). Le système accède à cette FU par un numéro et c'est ce numéro qui est stocké dans le bloc.

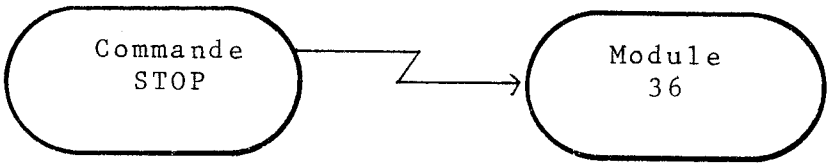
Remarque :

Si un bloc a déjà été alloué pour un même module, alors il y a simplement remise à jour du mode d'exécution du module en mise au point.

Oraganigramme de principe :



E - Module activé par la commande "STOP"



La commande "STOP" active le module 36.

Ce module permet :

- de faire une demande de point d'arrêt dans le module défini par la commande "MODULE"
- de faire une demande d'actions différées si la commande STOP est suivie des paramètres spécifiant ces actions.

Ces actions sont réalisées lors de la rencontre du point d'arrêt dans le module et avant la reprise du dialogue si fonctionnement en pas-à-pas.

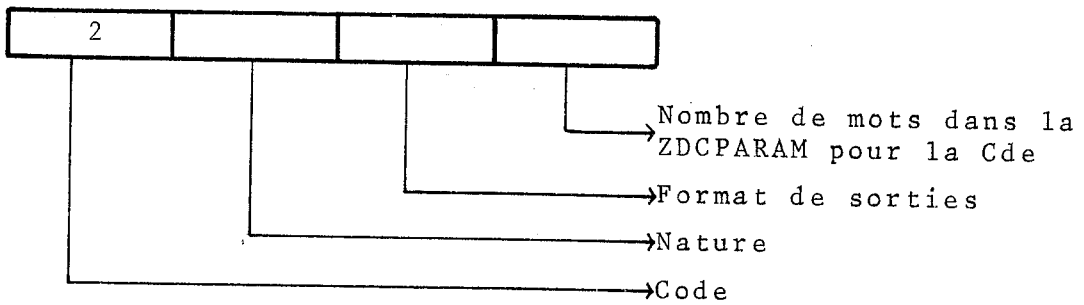
Les actions différées sont stockées dans le bloc réservé au module.

Une action est composée :

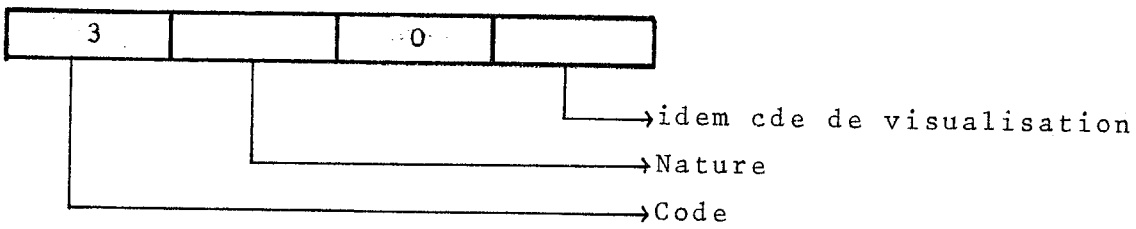
- d'un mot de commande
- de l'adresse du point d'arrêt où doit être réalisée l'action
- des informations relatives à ces actions (adresse d'une variable, nom de registre, etc).

- Représentation des mots de commande dans le bloc

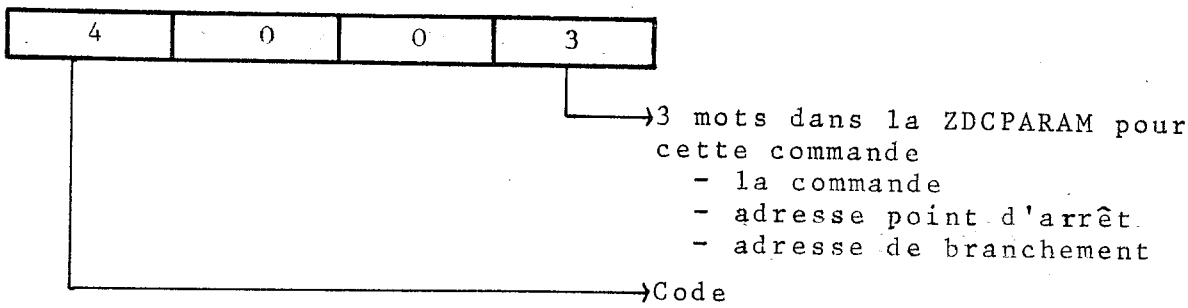
1 - Commande de visualisation



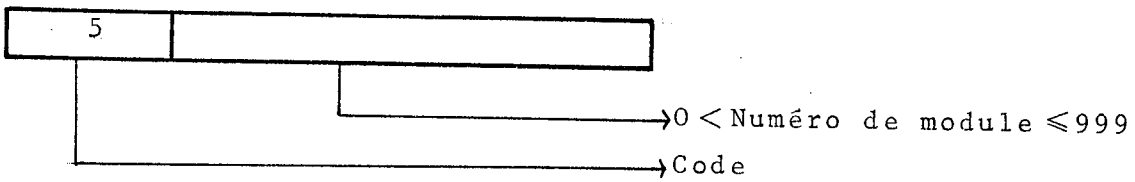
2 - Commande de modification



3 - Commande de branchement



4 - Commande d'enchaînement de module



Cette commande implique aussi 3 mots dans la ZDCPARAM :

- la commande
- adresse point d'arrêt
- numéro de FU bibliothécaire.

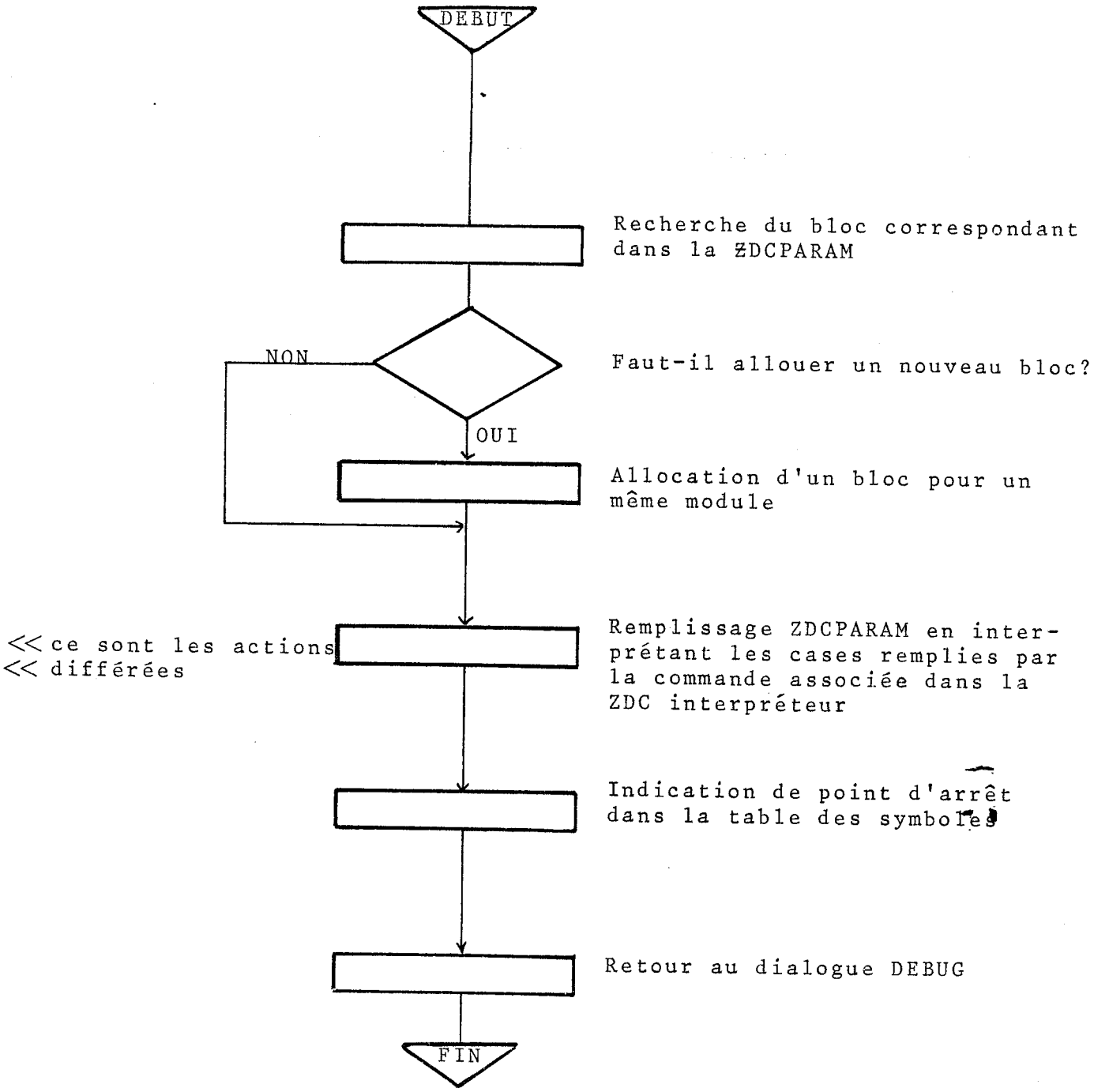
Remarque :

La nature dans le mot de commande a la signification suivante :

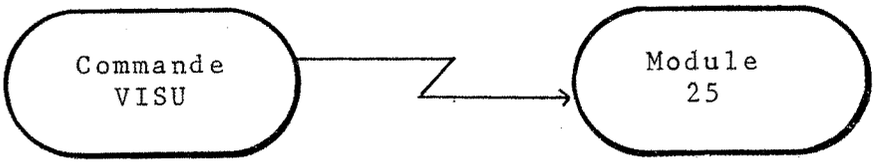
1 ⟶ variable
2 ⟶ tableau
3 ⟶ pile des modules
4 ⟶ échange
5 ⟶ registres

Le format de sorties dans la commande visu a la signification suivante :

1 ⟶ hexadécimal
2 ⟶ binaire
3 ⟶ ascii
4 ⟶ décimal


DEBUT
Recherche du bloc correspondant dans la ZDCPARAM
NON
Faut-il allouer un nouveau bloc?
OUI
Allocation d'un bloc pour un même module
<< ce sont les actions
<< différées
Remplissage ZDCPARAM en interprétant les cases remplies par la commande associée dans la ZDC interpréteur
Indication de point d'arrêt dans la table des symboles
Retour au dialogue DEBUG
FIN

F - Module activé par la commande "VISU" :



Deux cas de figure :

1) Le module est activé par la commande "VISU" :

Ici l'utilisateur demande la réalisation de l'action de visualisation :

- soit de variables
- soit d'éléments de tableau
- soit de la pile des modules
- soit du dernier échange (entrée/sortie)
- soit des registres programmables

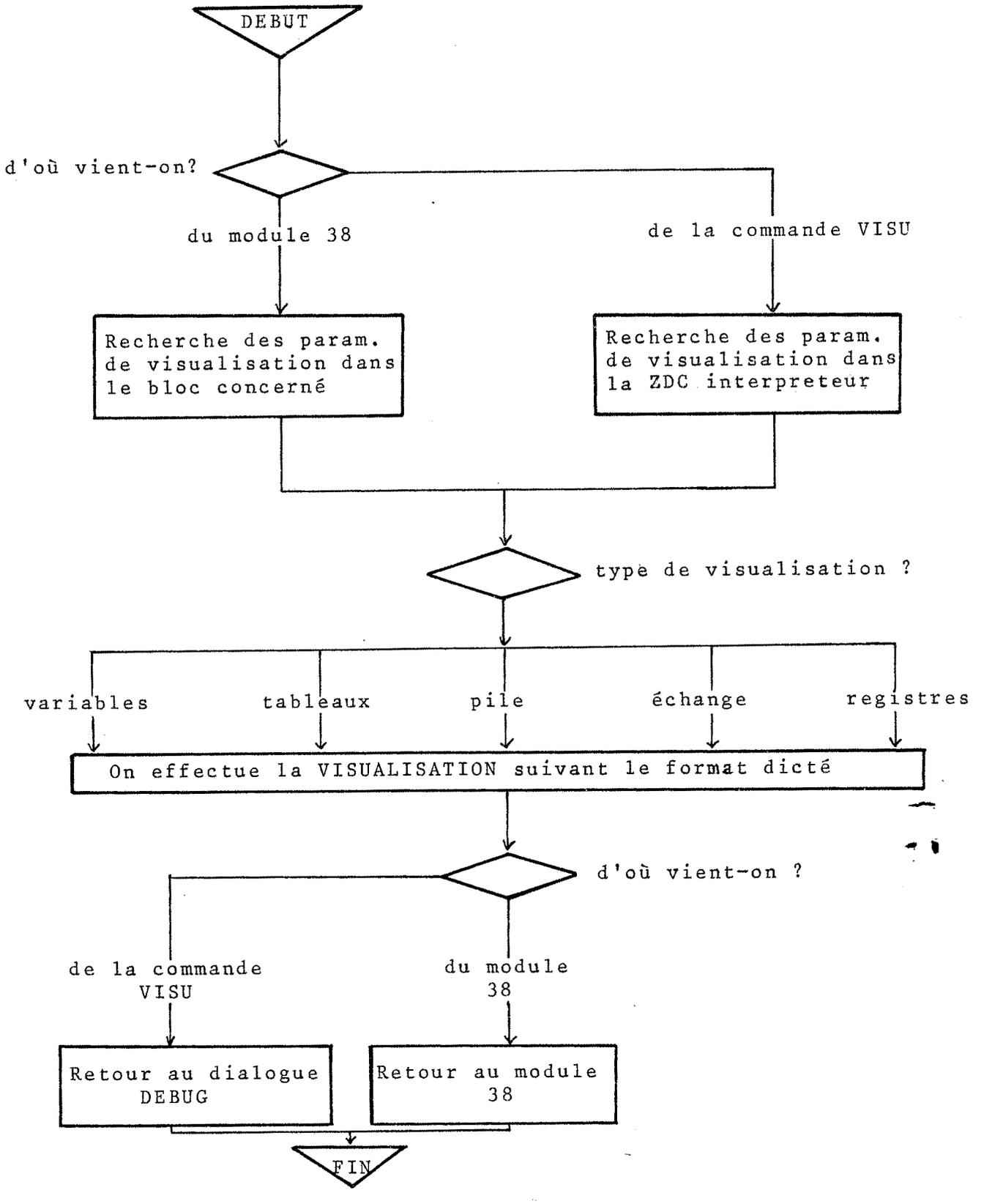
Les paramètres de visualisation sont définis dans la ŽDC de l'interpréteur sur l'émission de la commande lors du dialogue DEBUG.

2) Le module est activé par le module 38 :

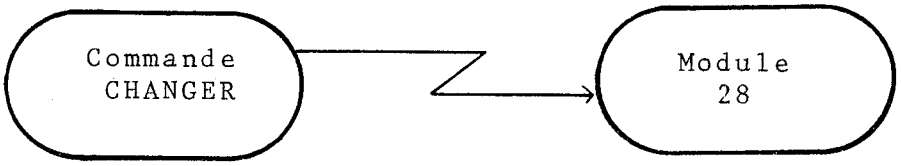
Dans ce cas l'utilisateur a demandé sur une commande "STOP" la réalisation de l'action différée de visualisation. Sur rencontre du point d'arrêt, le module a été mis en attente et le système a activé le module 38.

Ce dernier a pour rôle d'activer le module relatif à l'action demandée. Les paramètres de visualisation sont définis dans le bloc relatif au module et au point d'arrêt.

Organigramme de principe :



G - Module activé par la commande "CHANGER" :



Le principe est le même que pour la commande VISU.

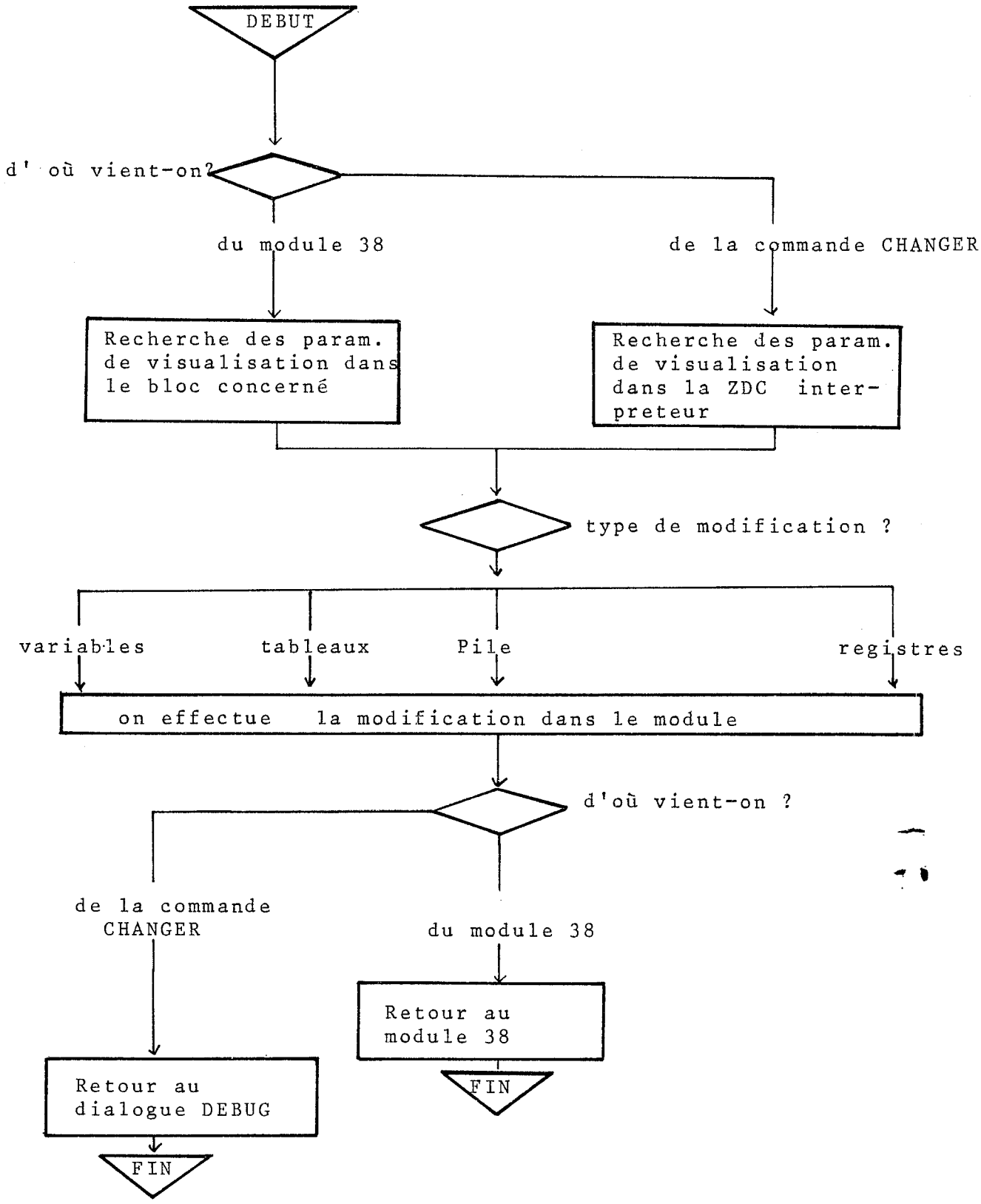
L'activation du module 28 provient :

- soit d'une demande de réalisation d'action immédiate (commande CHANGER)
- soit d'une demande de réalisation d'action différée (module 38).

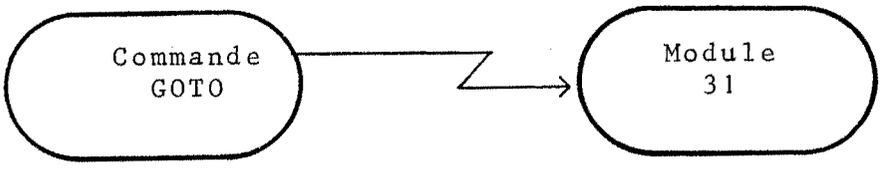
Il s'agit ici d'une action de modification :

- soit d'une variable
- soit d'éléments de tableau
- soit d'une modification de la pile des modules
- soit de registres autres que RP, RS, RSLO, RSLE.

Organigramme de principe :



H - Module activé par la commande "GOTO" :



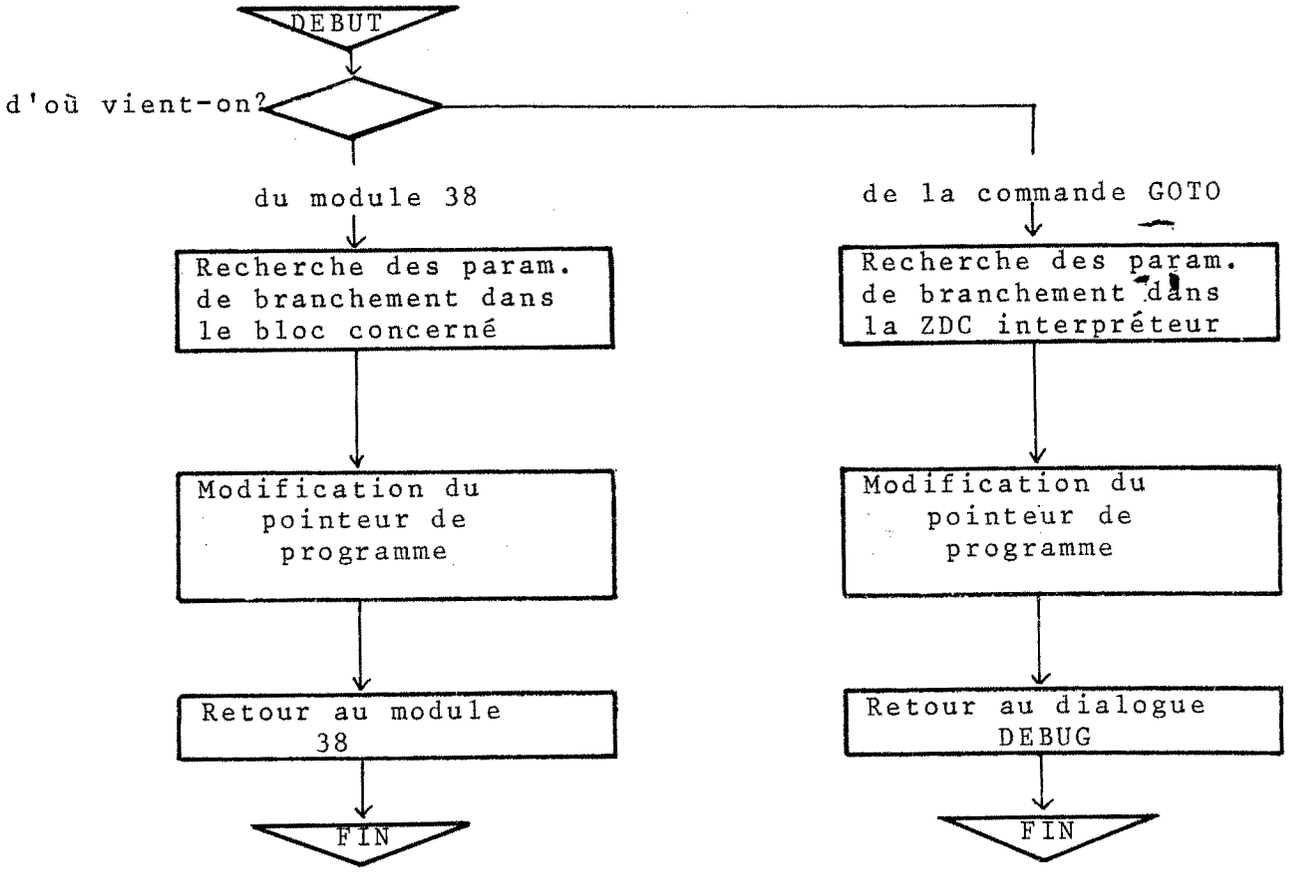
Le principe est le même que pour la commande "VISU".

L'activation du module 31 provient :

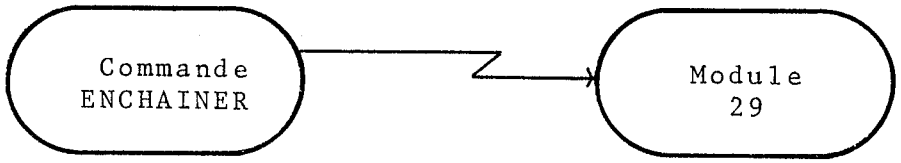
- soit d'une demande de réalisation d'action immédiate (commande "GOTO")
- soit d'une demande de réalisation d'action différée (module 38).

Il s'agit ici d'une action de branchement inconditionnel à une adresse de programme.

Organigramme de principe :



I - Module activé par la commande "ENCHAINER" :



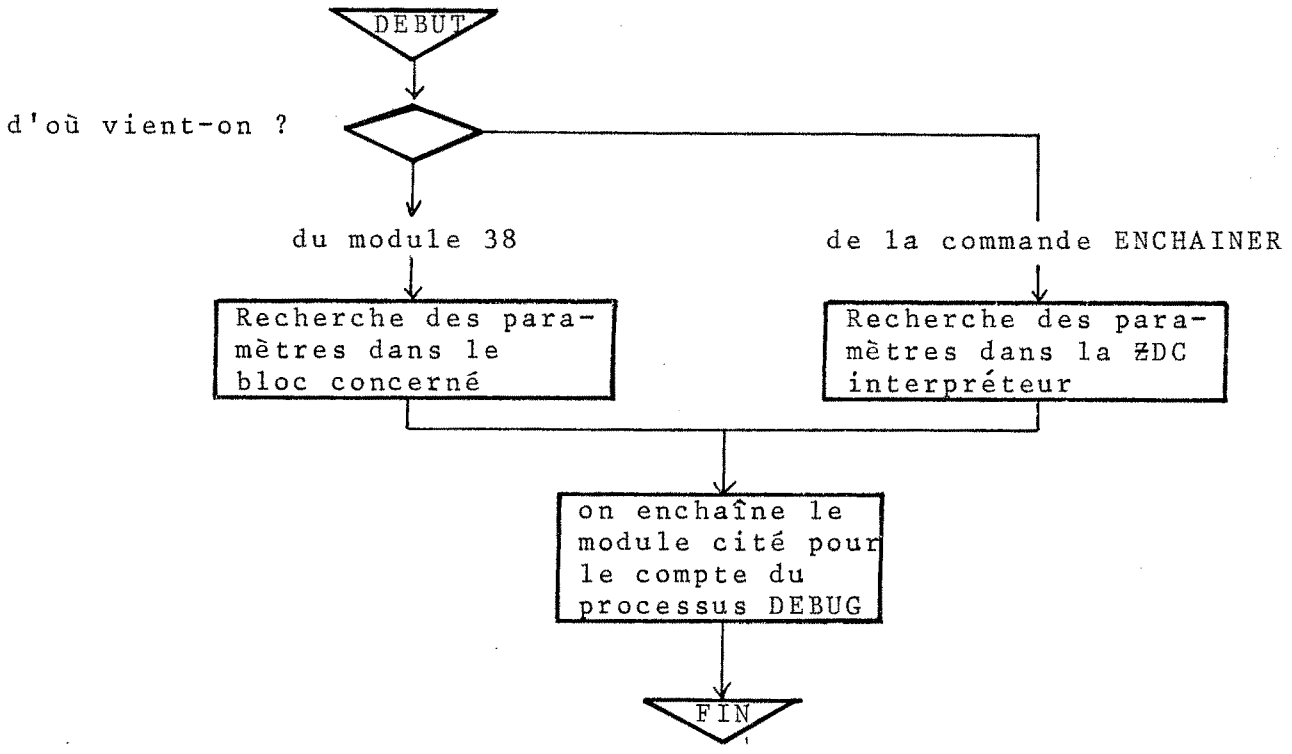
Le principe est le même que pour la commande "VISU".

L'activation du module 29 provient :

- soit d'une demande de réalisation d'action immédiate (commande "ENCHAINER")
- soit d'une demande de réalisation d'action différée (module 38).

Il s'agit ici d'une action d'enchaînement de module. Le module enchaîné s'exécute pour le compte du processus DEBUG. Il doit obligatoirement se terminer par une primitive avec retour au dialogue. Le retour sera effectif si l'exécution est demandée en mode pas-à-pas, sinon il y aura retour au module précédemment mis en attente sur point d'arrêt.

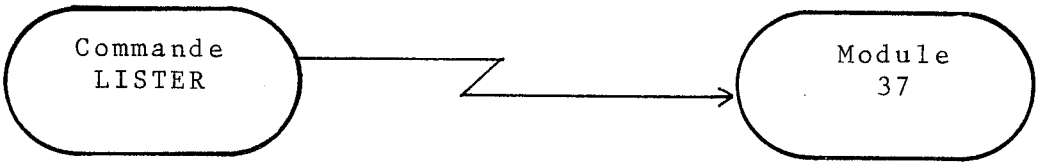
Organigramme de principe :



N.B. Le module que l'on enchaîne est un module écrit par l'utilisateur et ne peut s'exécuter que sous le contrôle du processus DEBUG.

Cependant, l'émission de la commande ENCHAINER 24 provoque la sortie d'un DUMP module sur un des périphériques du pool (le module 24 est un module système).

J - Module activé par la commande "LISTER" :

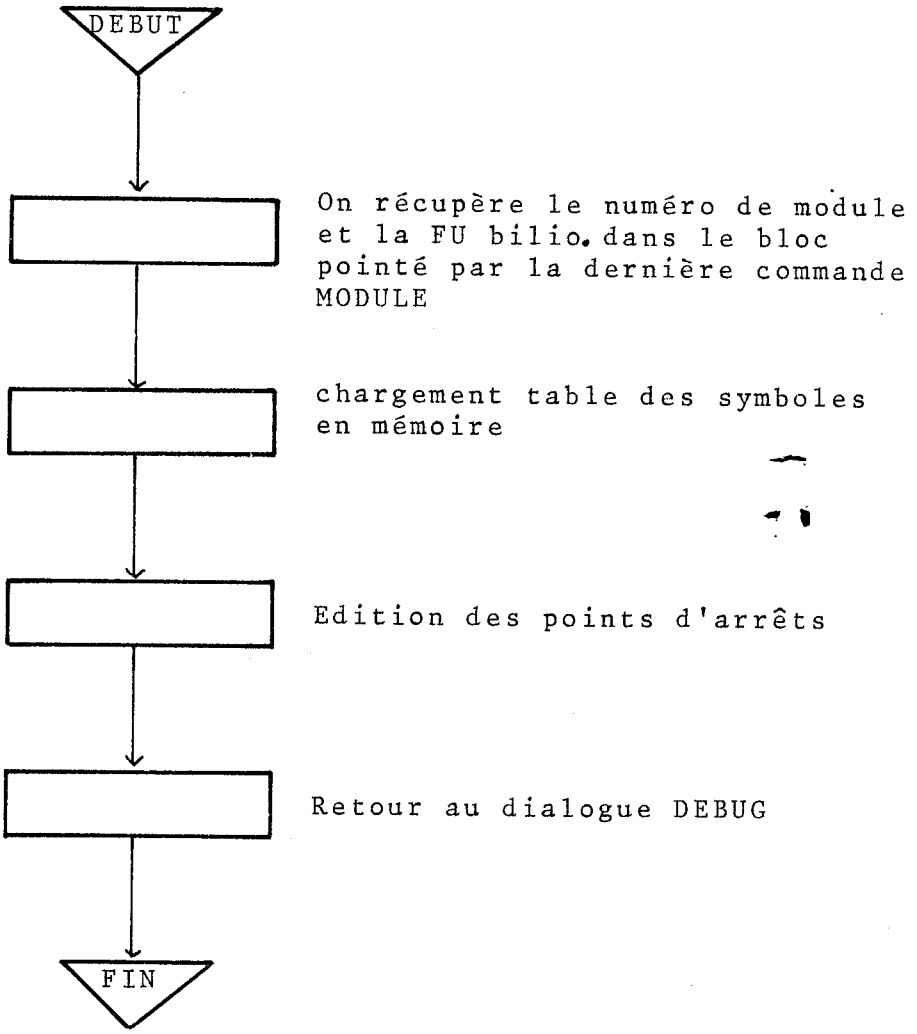


La commande LISTER active le module 37.

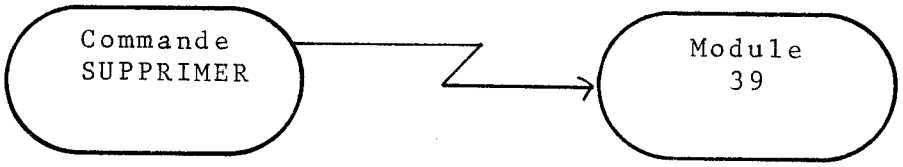
Ce module permet d'éditer, sur le périphérique de sortie, tous les points d'arrêt positionnés sur un module.

La liste des points d'arrêt provient de la table des symboles du module concerné, celui spécifié dans la dernière commande module, ou le dernier mis en attente sur rencontre d'un point d'arrêt.

Organigramme de principe :



K - Module activé par la commande "SUPPRIMER" :



La commande "SUPPRIMER" active le module 39.

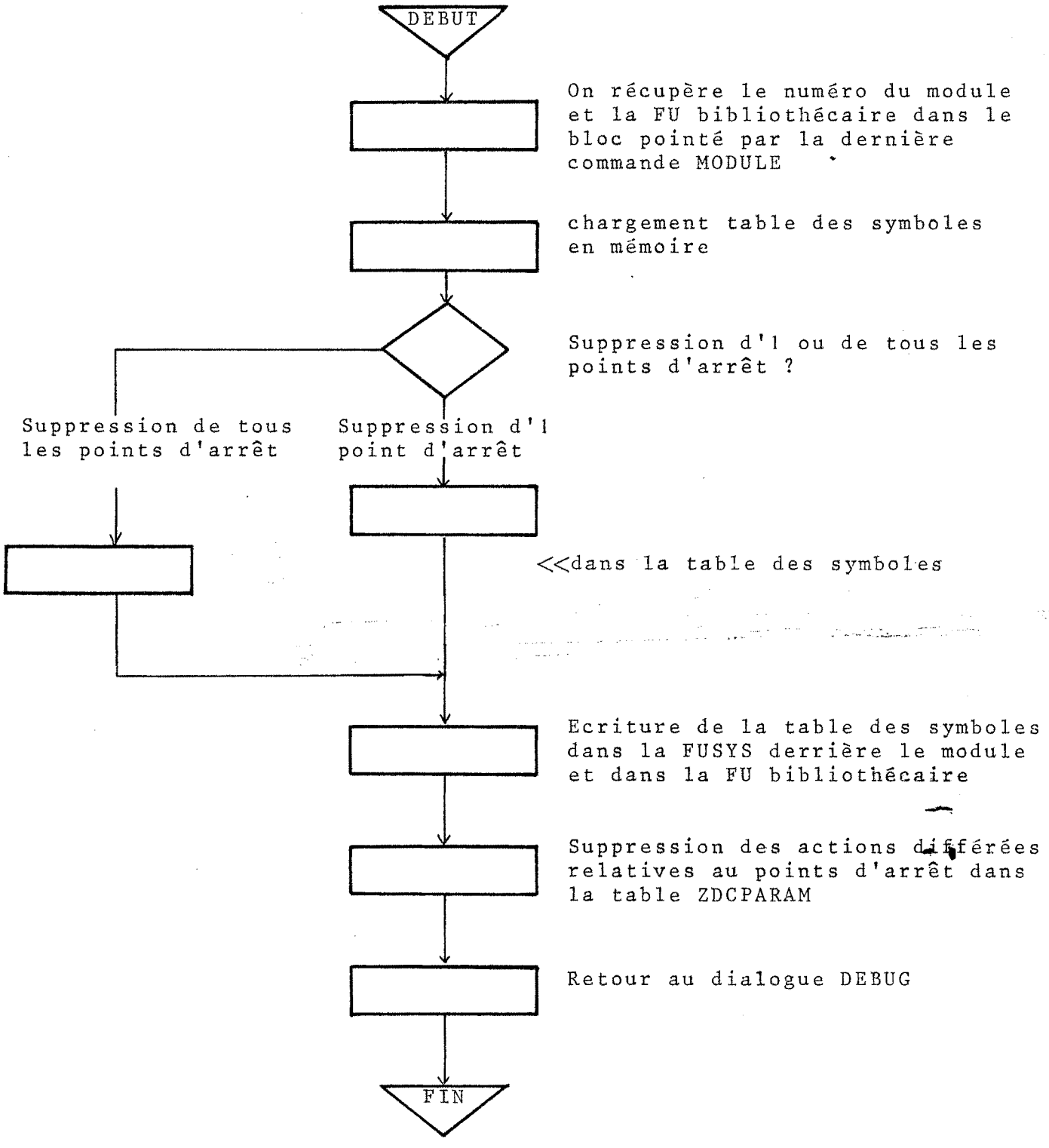
L'exécution du module 39 permet la suppression d'un ou de tous les points d'arrêt positionnés dans la table des symboles du module.

La suppression d'un point d'arrêt implique la remise à jour du bloc dans lequel a été défini le point d'arrêt.

Remarque :

Les points d'arrêt ne sont définis dans les blocs de la table ZDCPARAM que s'ils sont en relation avec des demandes d'action différées. Sinon ces points d'arrêt n'apparaissent que dans la table des symboles du module.

Organigramme de principe :



L - Module activé par le système :

Il s'agit du module 38. Il n'est pas directement associé à une commande. Il est toujours activé sur rencontre d'un point d'arrêt dans un module en test.

Il a pour rôle d'analyser les différentes actions différées à réaliser (visualisation, modification, etc).

Une fois ces actions analysées, le module 38 enchaîne le module réalisant les actions.

En fin de réalisation de ces actions, deux cas de figure peuvent être rencontrés :

- le module s'exécute en mode pas-à-pas, alors il y a retour au dialogue DEBUG.
- le module s'exécute en mode trace, alors il y a retour au module en mise au point.

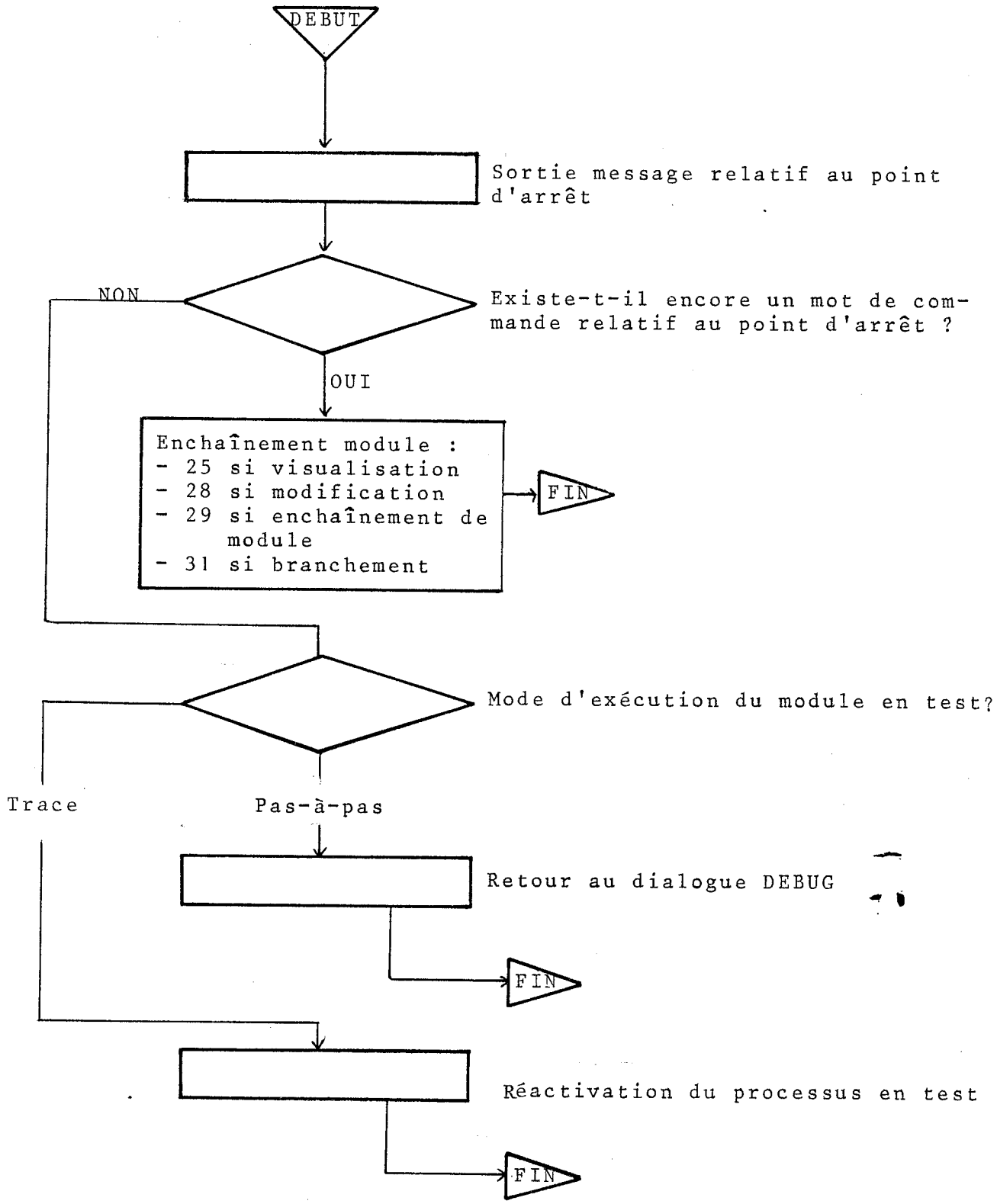
Remarque 1 :

Toutes les applications de type interpréteur disposent d'un module particulier, c'est le module d'enchaînement. C'est lui qui active la plupart des modules de DEBUG.

Remarque 2 :

Il peut y avoir plusieurs actions à réaliser sur un même point d'arrêt (exemple : demander une visualisation de registres et de variables). Pour permettre ceci, chaque module réalisant une action différée (modules 25, 28, 29 et 31) se termine par un retour au module 38. Ce dernier teste s'il reste encore une action à réaliser.

Organigramme de principe du module 38 :



IV.2. - INTERFACE AVEC LE SYSTEME MUTEX

La réalisation de ce logiciel de mise au point nous a obligés à apporter un certain nombre de modifications dans le système MUTEX.

A - Gestion des alarmes :

Le principe de positionnement d'un point d'arrêt sur le calculateur SOLAR est la création d'une imparité mémoire centrale. Cette dernière est reconnue par le micro-logiciel en tant qu'alarme. Le module système gérant les alarmes a dû subir des modifications lui permettant :

- la reconnaissance d'un point d'arrêt dans un module d'une application en test.
- la demande d'activation de DEBUG par l'intermédiaire du module d'activation et de SWAPP-OUT des modules.

B - Activation et SWAPP-OUT des modules :

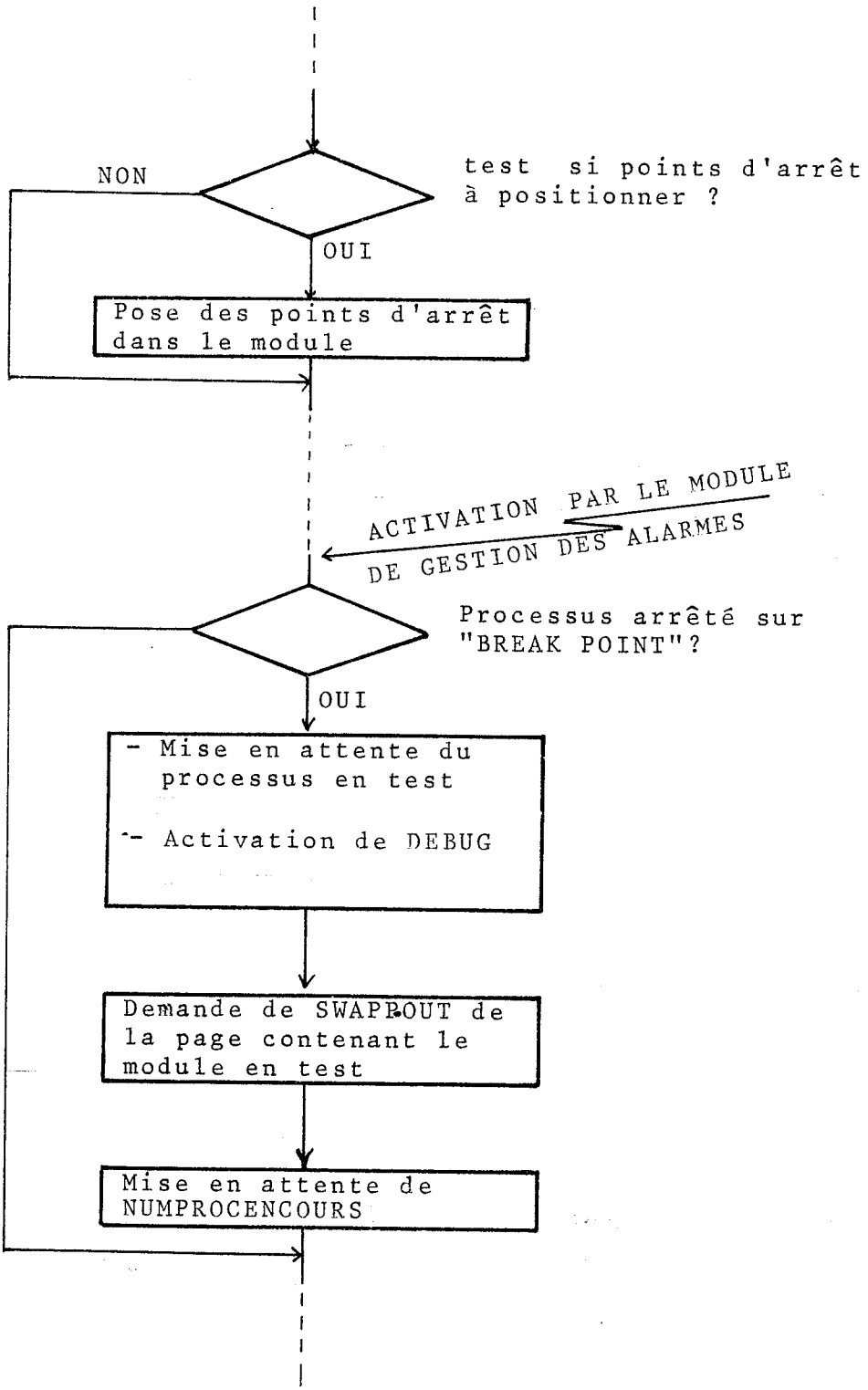
Sur un système multi-consoles comme MUTEX, les points d'arrêt ne peuvent être positionnés directement en mémoire. Le module pouvant être chargé dans une quelconque page, il faut donc attendre de connaître cette page et son chargement avant de positionner des points d'arrêt. Ce positionnement se fait juste avant l'activation du module.

Les points d'arrêt restent positionnés après le passage sur l'un d'eux. Le système devra les supprimer avant la sauvegarde du module dans le fichier système, la page risquant d'être réaffectée à un autre processus.

Le plus apte à subir cette extension de fonctionnement était le module d'activation et de SWAPP-OUT de modules.

Ce dernier a aussi subi des modifications pour la prise en compte d'une demande d'activation du dialogue DEBUG (demande émise par le module de gestion des alarmes) et la mise en attente du processus en test durant la phase du dialogue DEBUG.

Organigramme de principe :



C - Préparation des processus système et reconnaissance des interruptions :

Le processus DEBUG, comme tous les autres processus du système dispose d'un contexte d'exécution initialisé au lancement du système MUTEX (05). Pour intégrer DEBUG il a donc fallu apporter une extension au module de préparation des processus. Ce dernier a pour rôle d'enregistrer certaines interruptions en provenance d'un poste de travail. Une autre extension dans ce module nous a permis d'y inclure l'appel opérateur.

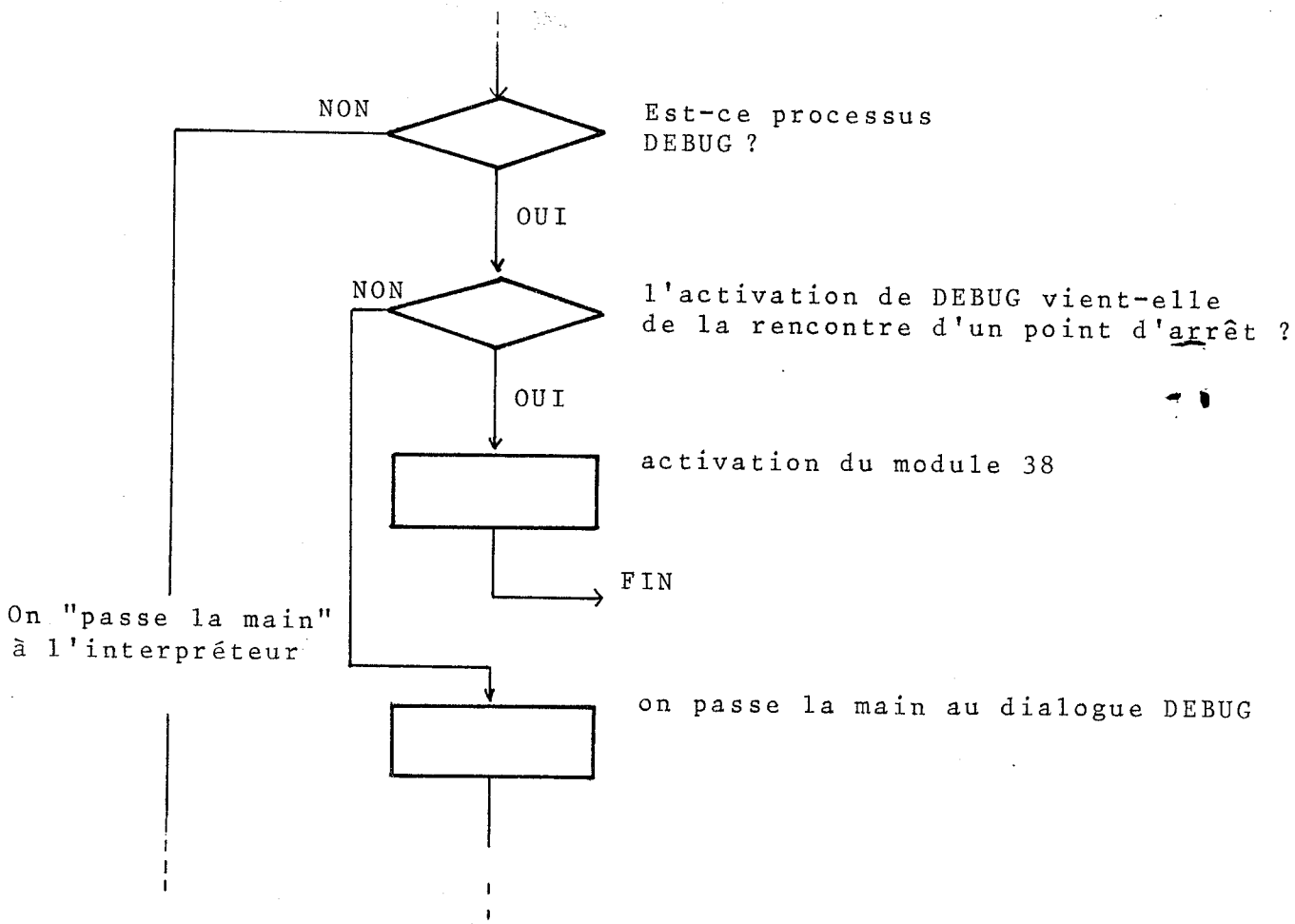
D - Lecture des commandes :

A la rencontre d'un point d'arrêt, nous avons vu qu'il y avait activation du processus DEBUG et plus exactement demande d'activation du module 38 (sortie de messages).

Dans le cas général, le module de lecture des commandes active l'interpréteur.

Sur rencontre d'un point d'arrêt, il lui faut activer le module 38 et non l'interpréteur.

Organigramme de principe :



E - L'interpréteur :

Sur dialogue DEBUG, l'interpréteur ne doit pas considérer certains caractères spéciaux comme séparateurs mais comme caractères alphanumériques , ceci étant dû à la forme des commandes du dialogue DEBUG, qui sont différentes des autres commandes de l'interpréteur.

En effet la commande :

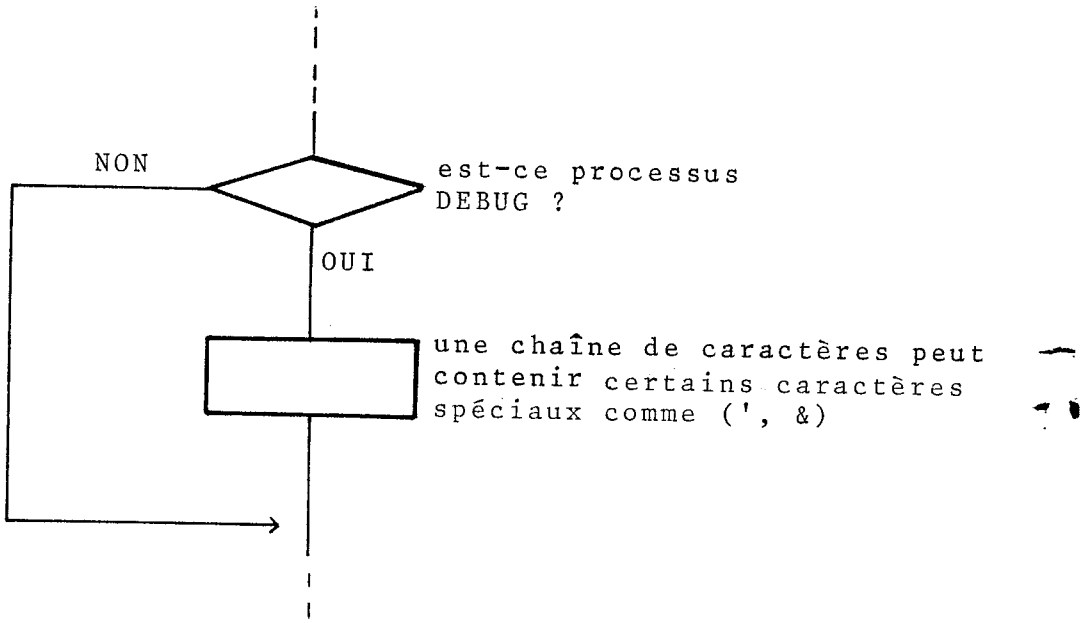
STØP '267 VISU VAR &250 / DEC

signifie qu '267 et &250 sont deux chaînes de caractères.

Le caractère "/" est pris lui comme séparateur, comme dans les autres commandes de l'interpréteur.

Il suffit donc de prendre en compte ces caractères spéciaux dans la chaîne de caractères lorsque le module 10 se déroule sous le processus DEBUG.

Organigramme de principe :



F - Chargement des modules :

Cet utilitaire est appelé GESMOD. Il fonctionne sous le système BOS-D (système de production de programme) ou sous BACKM-MUTEX (production de programme en background sous MUTEX) ; Il a deux points d'entrée : LMOD pour le chargement de modules et DMOD pour la destruction de modules.

C'est la partie chargement de modules que nous avons modifiée. Aujourd'hui l'utilitaire GESMOD prend en compte une zone utilisée comme table de symboles du module.

Deux cas de figures peuvent se présenter :

1 - Le module est écrit en FORTRAN et il y a création d'une table des symboles dans l'article TSYMB du fichier image mémoire du module. (voir chapitre III - paragraphe 3).

2 - Le module est écrit en PL16 ou ASM (assembleur) et la table des symboles n'existe pas au moment du chargement du module dans le bibliothécaire. Cependant, nous savons qu'un espace est nécessaire au moment de l'utilisation du DEBUG pour générer des points d'arrêt.

Ce que fait GESMOD :

Après chargement du module dans le bibliothécaire, soit l'article TSYMB existe dans le fichier image mémoire du module, et GESMOD charge son contenu à la suite du module (avec alignement sur une frontière de secteur), soit l'article n'existe pas et alors GESMOD initialise (à zéro) une zone équivalente à la longueur de la table des symboles derrière le module (avec alignement sur une frontière de secteur).

Remarque :

- le premier mot de la table des symboles a la signification suivante :

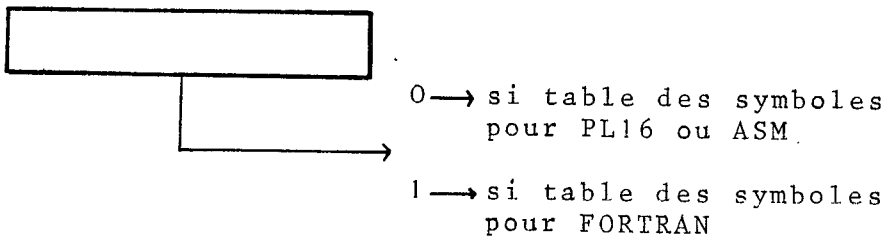
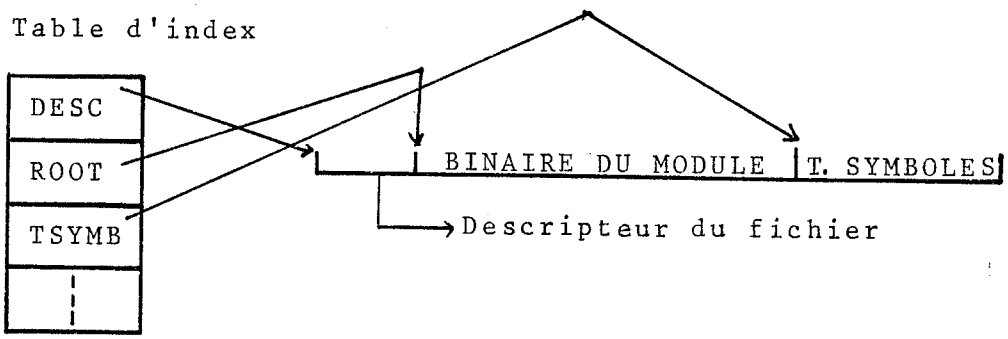


Schéma de principe :

1 - Cas du FORTRAN :

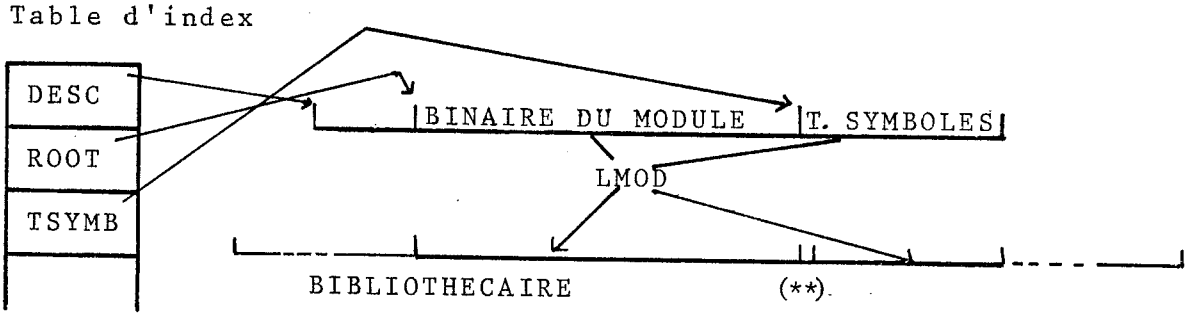
1.1. - Fichier image mémoire (fichier indexé)



1.2. - Appel de l'utilitaire TSFOR

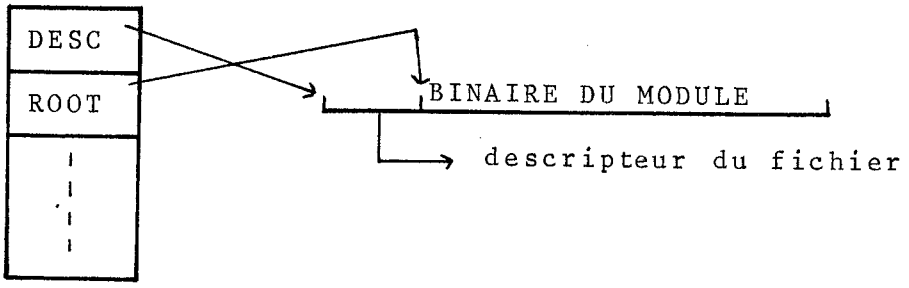
Suppression des symboles inutiles (annexe) de manière à obtenir une table des symboles de longueur LONGTS (*).

1.3. - Appel de l'utilitaire GESMOD

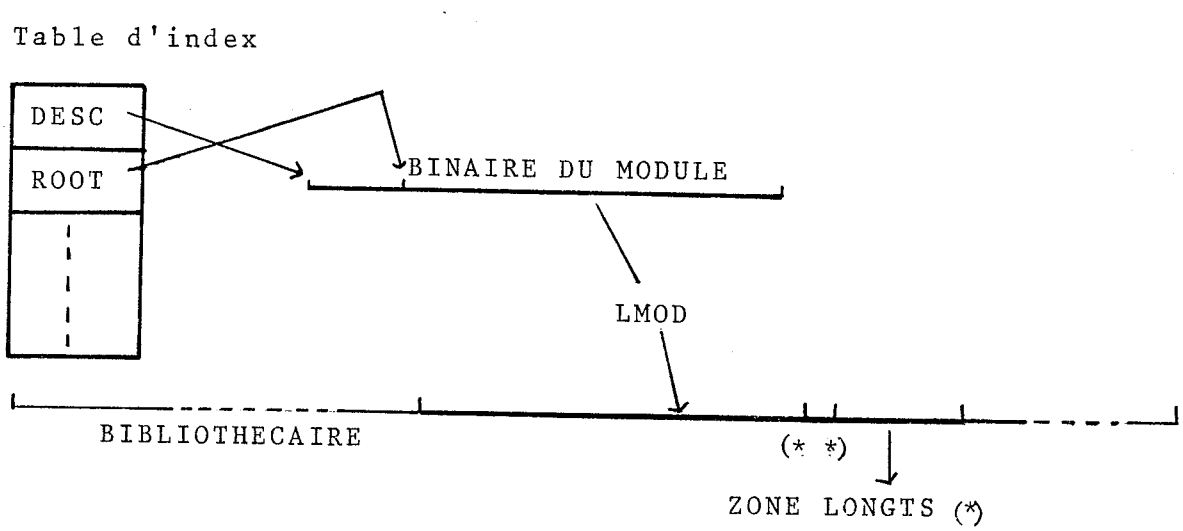


2 - Cas de l'assembleur ou du PL16

1.1. - Fichier image mémoire (fichier indexé)



1.2. - Appel de l'utilitaire GESMOD



(*) LONGTS = zone réservée à la table des symboles et de longueur aujourd'hui égale à deux secteurs.

(**) espace représentant l'alignement de la table des symboles sur une frontière de secteur.

Remarque :

La syntaxe de la commande n'a subi aucune modification. Voir la forme de la commande dans le manuel de référence MUTEX (05)

CONCLUSION

La définition et la réalisation de ce logiciel de mise au point nous ont amenés à un certain nombre de réflexions.

L'inefficacité de beaucoup d'entre eux provient essentiellement d'une mauvaise prise en compte des problèmes posés par les tests de programmes. Ils sont pour la plupart "rattachés" aux systèmes d'exploitation ; ces derniers deviennent de plus en plus sophistiqués, il s'en suit des changements profonds dans le contexte d'exécution des applications. De ce fait, ils doivent être développés en même temps si nous voulons qu'ils soient efficaces et donc utiles.

Pour qu'une mise au point d'application soit rapide, il faut que le programmeur observe une méthodologie des tests. La démarche devrait observer deux phases :

- Un temps de réflexion pendant lequel il pense au comportement attendu de son programme (ou de ses programmes) et détermine les points critiques où il doit définir des conditions d'arrêt.

- Ensuite un passage machine pour vérifier ce comportement et intervenir au moment où il le désire.

Nous constatons que les outils interactifs ne sont pas adaptés à la première phase. Il faudrait qu'ils soient conçus de manière à obliger le programmeur à respecter cette démarche.

La dégradation des temps d'exécution dans la mise au point d'applications fonctionnant en temps réel est un problème qu'il faut se poser tout en sachant que les solutions, s'il y en a, sont en général coûteuses.

Le logiciel de mise au point, quelque soit son degré de perfectionnement, n'offre qu'un moyen d'investigations plus aisé.

Toutefois, pour un programmeur expérimenté, la contribution qu'il apporte dans la recherche d'erreurs est loin d'être négligeable.

GLOSSAIRE

APPLICATION : Ensemble de fonctions à automatiser, ce qui se traduit par la somme des programmes intégrés par l'utilisateur.

ASCII : Code normalisé de "l'Americain Standard Code for Information Interchange".

BATCH : Traitement par lot, traitement différé.

BIBLIOTHECAIRE : Fichier disque où sont stockés les programmes d'une ou plusieurs applications.

BREAK POINT : Désigne un emplacement dans un programme où l'on définit une condition d'arrêt.

CHECKSUM : Information de contrôle représentant le résultat d'un calcul effectué sur les différents éléments d'une information.

DEBUG : Nom donné à l'outil de mise au point réalisé.

DEBUGGER : Outil de mise au point.

Task DEBUGGER : Outil de mise au point de programmes permettant la mise en place de points d'arrêt sur des adresses de programmes.

Memory DEBUGGER : Outil de mise au point de programmes permettant la mise en place de points d'arrêt sur des adresses en mémoire centrale.

FICHIER INDEXE : Fichier disque géré par le système de gestion de fichiers (FMS). Les articles sont de longueur variable. L'accès aux articles se fait par une table d'index contenant le nom des articles et leur adresse physique sur le fichier.

FUSYS : Fichier disque géré par le système d'exploitation MUTEX.

MAP : Carte mémoire : état de la mémoire à un instant donné

MODULE : Voir programme.

MULTI-TACHES : Mode d'exploitation suivant lequel plusieurs tâches ont été exécutées dans le même laps de temps.

MUTEX : Multi User Transaction Executive, système conçu pour l'exécution d'applications de type conversationnel multi-utilisateurs.

OVERLAY : Possibilité de recouvrir une zone mémoire par des codes exécutables différents.

PATCH : Correction du code exécutable.

PL16 : Langage permettant une programmation de haut niveau. Il est utilisé à la SEMS, pour le développement des logiciels sur matériel SOLAR (accès aux registres de la machine, possibilité d'utiliser des instructions assembleur) et par les utilisateurs pour l'écriture d'applications.

POSTE DE TRAVAIL : Terminal de dialogue entre l'utilisateur et le système d'exploitation MUTEX. Il est référencé par un nom.

PROCESSUS : C'est une entité fonctionnelle. Il exécute pour le compte d'un poste de travail les divers modules d'une application.

PROGRAMME : Traduction d'un algorithme dans un langage interprétable sur un calculateur ; c'est un ensemble d'instructions et de données.

REQUETE : Primitive système offerte à l'utilisateur.

SWAPP-IN : Occupation d'une page de la mémoire centrale par écriture du code contenu sur un fichier système.

SWAPP-OUT : Libération d'une page de la mémoire centrale par écriture sur le disque du code qui est contenu dans cette mémoire.

SYSDBG : Application de type interpréteur regroupant l'ensemble des modules de l'outil de mise au point "DEBUG".

VERRUE : Code extension intégrable par l'utilisateur.

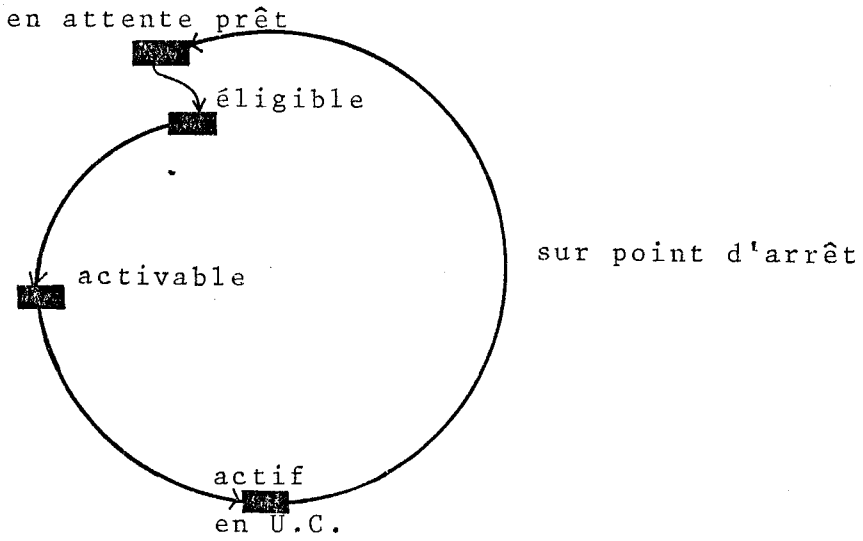
ZDC : Zone des données de communication entre deux modules d'une même application.

ZDCPARAM : Table de stockage des paramètres enregistrés au niveau du dialogue DEBUG.

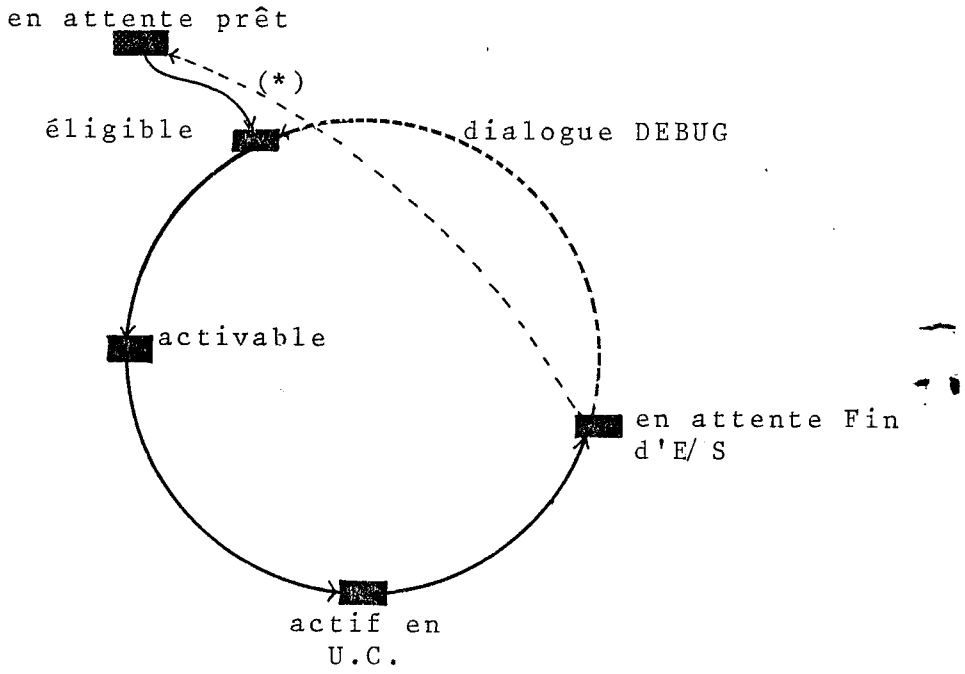
ANNEXES

GRAPHES D'ETAT

GRAPHE D'ETAT DU PROCESSUS EN TEST :



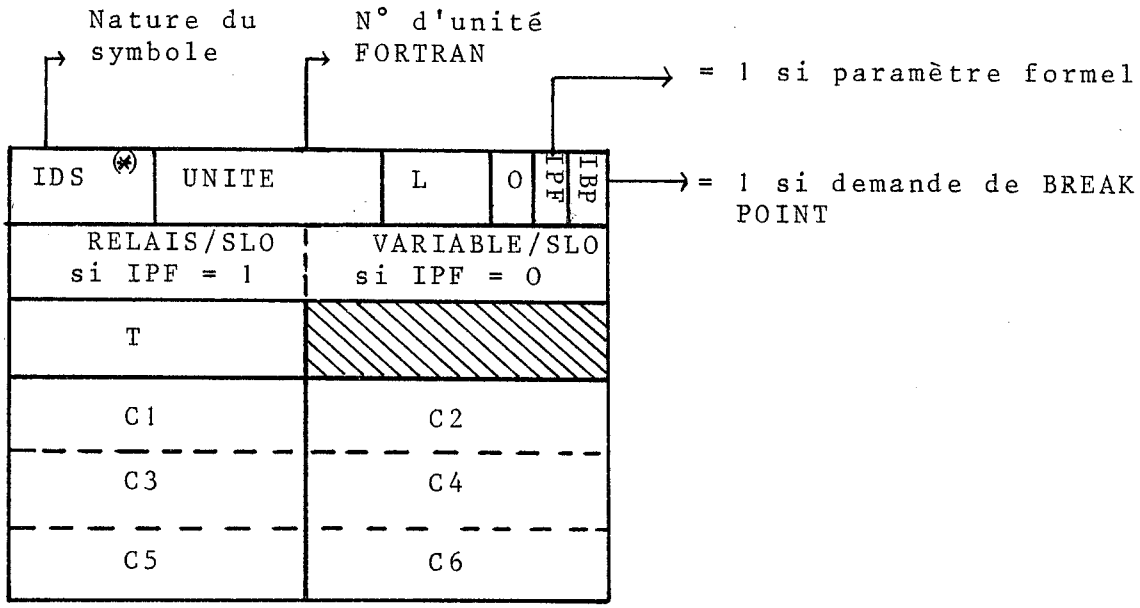
GRAPHE D'ETAT DE L'OUTIL DE MISE AU POINT DEBUG :



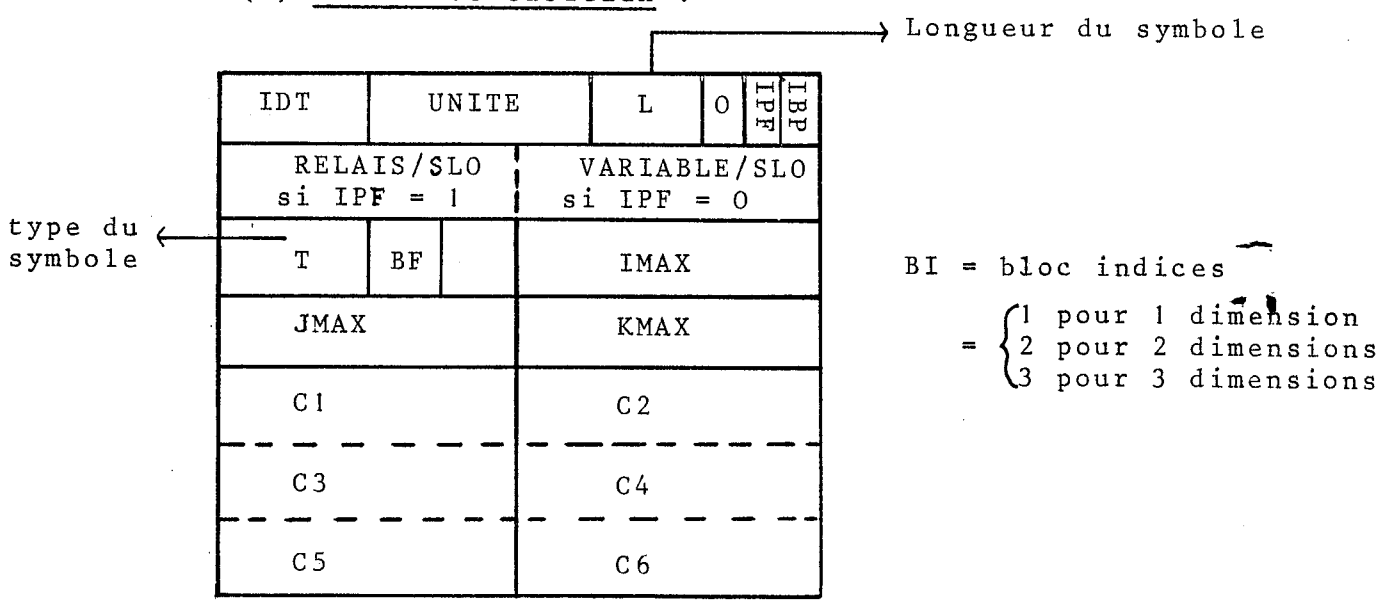
(*) Activation du processus en test et fin de dialogue.

La table est composée de symboles de quatre natures différentes. En voici la structure :

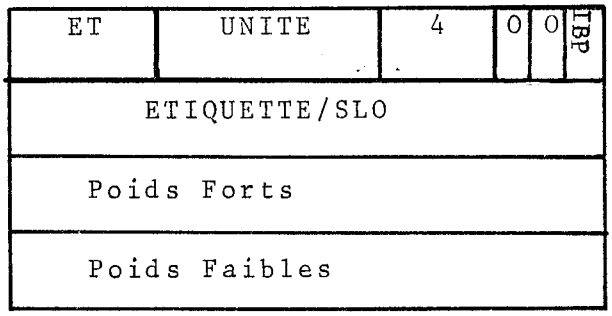
(1) Variables simples :



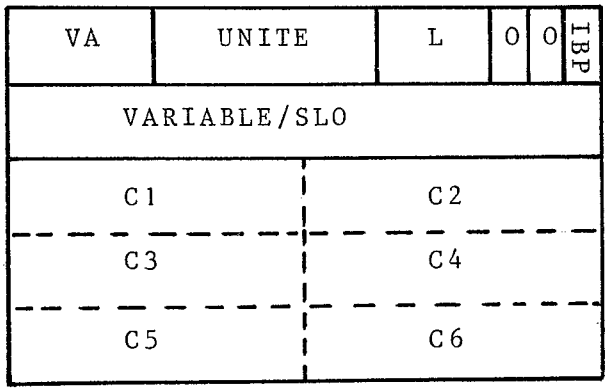
(2) Variables tableaux :



(3) Etiquette de programme :



(4) Variable ASSIGN :



(*) Nature du symbole :

IDS = 1 = Variable simple

IDT = 2 = Variable tableau à 3 dimensions maxi

ET = 3 = Etiquette du programme

VA = 4 = Variable ASSIGN

(**) Type du symbole :

T = 1 = Entier

T = 2 = Réel

T = 3 = Logique

T = 4 = Double précision

T = 5 = Complexe

T = 6 = Hollerith

- RECAPITULATIF DES COMMANDES

COMMANDES DE DEBUG	PAGES
/ LANCER [APPLICATION] SYSDBG [SUR NOMPROC] (*)	41
DEBUG [{SUR NONPROC / SUR TOUS PROCESSUS}]	42
MODULE NUMOD [MODE {TR / PP}] [DANS [BIBLIOTHECAIRE] NONBIB]	44
STOP {ADRPA [[UNITE] NUMUNIT] / {' / &} HEXA}	46
VISU (**) {VARIABLES [{' / &}] NOMELEM [, {' / &} NOMELM...] [/ {HEXA / BIN / ASC / DEC}] / TABLEAU [{' / &}] NOMELEM [(M1)] [LONG] M2 [/ {HEXA / BIN / ASC / DEC}] / PILE / ECHANGE / REGISTRES / [{HEXA / BIN / ASC / DEC}]}	50
GOTO (**) {ADRPA / 'HEXA}	54
ENCHAINER NUMOD (**)	55

COMMANDES DE DEBUG (SUITE)	PAGES
CHANGER (**) { VARIABLES [{' &}] NOMELEM [,{' &}NOMELEM...] /M3 [,...,M3/] ; TABLEAU [{' &} NOMELEM [(M1)] [LONG] (M2)] /M3 [,...M3/] ; PILE [INDICE] M1/M3 ; REGISTRES [RA [,RB [,...,RK] /M3 [,...,M3/]] }	53
ACT	56
LISTER [ARRETS]	47
SUPPRIMER { ADRPA [UNITE] NUMUNIT ; {' &} HEXA ; TOUT [ARRET] }	48
FIN [DEBUG]	43

(*) Commande du langage SYSCTL, toutes les autres appartiennent au langage SYSDBG.

(**) Ces commandes peuvent être précédées du paramètre STOP.

EXEMPLE D'UTILISATION

Nous voulons mettre au point une application de type interpréteur, de nom ESSAI, écrite en PL16.

Cette application a pour rôle : l'ouverture, la fermeture et la destruction d'un fichier disque. Ce dernier est géré par le système de gestion de fichiers FMS intégré au système d'exploitation MUTEX.

Par soucis de simplification, le fichier est créé par une commande opérateur (fichier de type séquentiel).

L'application ESSAI est composée de trois modules :

- Module 500 activé par la commande :

OUVrir [FICHIER] NOMFIC-CT sur Dii

- Module 501 activé par la commande :

FERmer [FICHIER] sur Dii

- Module 502 activé par la commande :

DETruire [FICHIER] sur Dii

Paramètres :

NOMFIC-CT : Nom et catalogue du fichier

Dii : Numéro de repère pris dans la liste 1 à 15.

Nous trouvons dans ce qui suit :

A - L'écriture des commandes de l'application sous le langage D'INTGEN

B - Les listings de programmation

C - La mise au point proprement dite.

A - Ecriture des commandes sous le langage D'INTGEN :

```
/DECL APP ESSAI
/LANCER SYSGEN
 GENERER LANGAGE APPLICATION ESSAI
COMMANDE OUVRIR
     EMPILER 500
     MOT IGNORABLE FICHIER
     RANGER PARAMETRE ALPHA SUR 2 CASES DEPUIS 1
     BRANCHER SI SEPARATEUR "-" ETIQUETTE 1
     BRANCHER ETIQUETTE 2
          BLOC ETIQUETTE 1
     RANGER PARAMETRE ALPHA SUR 1 CASE DEPUIS 3
     MOT IGNORABLE SUR
     RANGER PARAMETRE ENTIER EN 4
     BRANCHER SI NON FIN DE COMMANDE ETIQUETTE 2
          BLOC ETIQUETTE 2
     SIGNALER ERREUR 8
FIN DE COMMANDE
COMMANDE FERMER
     EMPILER 501
     MOT IGNORABLE FICHIER
     MOT IGNORABLE SUR
     RANGER PARAMETRE ENTIER EN 1
     BRANCHER SI NON FIN DE COMMANDE ETIQUETTE 2
          BLOC ETIQUETTE 2
     SIGNALER ERREUR 8
FIN DE COMMANDE
COMMANDE DETRUIRE
     EMPILER 502
     MOT IGNORABLE FICHIER
     MOT IGNORABLE SUR
     RANGER PARAMETRE ENTIER EN 1
     BRANCHER SI NON FIN DE COMMANDE ETIQUETTE 2
          BLOC ETIQUETTE 2
     SIGNALER ERREUR 8
FIN DE COMMANDE
 FIN DE GENERATION
```

```
LIGNE RC  RL  RW  DONNEES   TABL  PROG  PF  BN :*SOURCE

0001  1F  1F  1F  1F  0000  0000  0000  00  00 : << OUVERTURE FICHIER SEQUENTIEL(MODULE 500)
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : <<
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :       MAIN PROCEDURE MOD500
0004  1F  1F  1F  1F  0000  0000  0000  00  00 : .COMMON SECTION ZDC
0005  02  1F  1F  02  0000  0000  0000  01  01 :       RES 70;
0006  02  1F  1F  02  0046  0000  0000  01  01 :       WORD CASE1;       <<NOM FICHIER
0007  02  1F  1F  02  0047  0000  0000  01  01 :       RES 3;
0008  02  1F  1F  02  004A  0000  0000  01  01 :       WORD CASE3;       <<CATALOGUE
0009  02  1F  1F  02  004B  0000  0000  01  01 :       RES 2;
0010  02  1F  1F  02  004D  0000  0000  01  01 :       WORD SU;          <<NUMERO REPERE
0011  02  1F  1F  02  004E  0000  0000  01  01 :       RES 178;
0012  02  1F  1F  02  0100  0000  0000  01  01 : .KSTORE SECTION PILEK
0013  02  1F  1F  03  0000  0000  0000  01  01 :       RES 80;
0014  02  1F  1F  03  0050  0000  0000  01  01 : .LOCAL SECTION LOC500
0015  02  04  1F  04  0000  0000  0000  01  01 :       REF PROCEDURE OPEN,ENCHNE,SORLIB;  <<PROCEDURES EXTERNES
0016  02  04  1F  04  0003  0000  0000  01  01 :       CONSTANT FICINEX=142,ERRFAU=141;
0017  02  04  1F  04  0003  0000  0000  01  01 :       WORD NOMFIC1,NOMFIC2,NOMFIC3,CATAL;
0018  02  04  1F  04  0007  0000  0000  01  01 :       WORD FU       =('0F'),  <<FU D3
0019  02  04  1F  04  0008  0000  0000  01  01 :            W        =(1),     <<ECRITURE AUTORISEE
0020  02  04  1F  04  0009  0000  0000  01  01 :            CRD      =(0),     <<COMPTE RENDU
0021  02  04  1F  04  000A  0000  0000  01  01 :            PA       =(1),     <<CONDITION DE PARTAGE
0022  02  04  1F  04  000B  0000  0000  01  01 :            TIMEOUT =(-1),
0023  02  04  1F  04  000C  0000  0000  01  01 :            CODARRET=('8D00),<<CODE D'ARRET
0024  02  04  1F  04  000D  0000  0000  01  01 :            LIBELLE,          <<POUR MSG D'ERREUR
0025  02  04  1F  04  000E  0000  0000  01  01 :            S00      =(0),    <<SU DE SORTIE(POSTE
0026  02  04  1F  04  000F  0000  0000  01  01 :            ZERO     =(0);    <<SI MODULE D'ENCHAINEMENT
0027  02  04  1F  04  0010  0000  0000  01  01 : .USING RC=ZDC,RL=LOC500,RK=PILEK;
0028  02  04  1F  04  0010  0000  0007  01  01 :       RB,RA:=@NOMFIC2;
0029  02  04  1F  04  0010  0000  0009  01  01 :       RA:=@CASE1;
0030  02  04  1F  04  0010  0000  000A  01  01 :       MOVE(RX:=3)FROM RA TO RB;
0031  02  04  1F  04  0010  0000  000C  01  01 :       CATAL:=CASE3;
0032  02  04  1F  04  0010  0000  000E  01  01 :                  <<OUVERTURE FICHIER
0033  02  04  1F  04  0010  0000  000E  01  01 :       CALL OPEN(@SU,@NOMFIC1,@FU,@CRD,@PA,@TIMEOUT,0,@CODARRET,RA:=8);
0034  02  04  1F  04  0017  0000  0020  01  01 :       GOTO SORMODULE ON(CRD=0);
0035  02  04  1F  04  0017  0000  0022  01  01 :       IF(CRD='600C)THEN          <<FICHIER INEXISTANT
0036  02  04  1F  04  0018  0000  0025  01  01 :                    LIBELLE:=FICINEX;
0037  02  04  1F  04  0018  0000  0027  02  01 :       ELSE IF(CRD='600B)THEN     <<FAU EXISTANTE
0038  02  04  1F  04  0019  0000  002B  02  01 :                          LIBELLE:=ERRFAU;
0039  02  04  1F  04  0019  0000  002D  03  01 :       END; END;
0040  02  04  1F  04  0019  0000  002D  01  01 :       CALL SORLIB(@S00,@LIBELLE,@CRD,RA:=3);
0041  02  04  1F  04  001B  0000  0035  01  01 : SORMODULE:
0042  02  04  1F  04  001B  0000  0035  01  01 :       CALL ENCHNE(@ZERO,RA:=1);
0043  02  04  1F  04  001C  0000  0039  01  01 : END.
FIN DE COMPILATION  0000 ERREUR(S) '01A6 MOTS
```

```
*CC COMPIL-JC,D3
*/CALL LKLOAD
*/LINK MOL1-JC
MOD500 '016C          << Adresse 1ère instruction du programme
ZOC    '0000 }
PILEK  '0100 }        << Adresses des 1ères données du COMMON,
LOC500 '0150 }        << de la KSTORE et du LOCAL.
MODULE 1 :WARN 15
*/LOOK BIBFMS-MX/BIBMUT-MX
OPEN   '01F9
LOPEN  '01AE
 NOM   : MUTEX-0 BIBFMS
 DATE  : 30 JUILLET 1979
 REF   : 1.164.374.10/01.01.64.14
         '0305
         '0306
MODULE 1 :WARN 15
ENCHNE '030A
LOCHNE '0306
SORLIB '035E
LOC    '034C
TIMOUT '03C2
LOCOCT '03C2
NULPV  '03D8
LPASSE '0400
LOCOCT '0400
MODIPV '0416
ALARME '0433
LOCALM '042E
RAZREP '0458
LOCREP '0458
ALAFMS '046E
LOCALA '0468          } << Edition de liens.
 NOM   : MUTEX-0 BIBMUT
 DATE  : 15 NOVEMBRE 1978
 REF   : 1.164.314.13/01.01.64.12
         '04B9
         '04BA
LIRLIB '0552
LOC    '053A
SORTIE '05C6
LOCSOR '05BC
CPTOUT '05DE
LOCOCT '05DA
NFUBTB '060B
LOC    '0607
TRONES '0664
*/STAT
*/ENDL

BEGIN  '0000
END    '06D3
RUN    '016F          } << Chargement du programme
*/CATAL IM,MOD500-JC  << dans le fichier MOD500-JC
*/CLOSE IM
*/SI MOL1-JC
*/DELE SI
*/CALL GESMOD
*/OPEN OLD 1,MOD500-JC
                      } << Chargement dans le
*/LMOD,1,500,BIBCOM,256 << bibliothécaire.
*/EOJ
```

```
LIGNE RC  RL  RW  DONNEES   TABL  PROG  PF  BN :*SOURCE

0001  1F  1F  1F  1F  0000  0000  0000  00  00 : << FERMETURE FICHIER SEQUENTIEL(MODULE 501)
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : <<
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :      MAIN PROCEDURE MOD501
0004  1F  1F  1F  1F  0000  0000  0000  00  00 : .COMMON SECTION ZDC
0005  02  1F  1F  02  0000  0000  0000  01  01 :      RES 70;
0006  02  1F  1F  02  0046  0000  0000  01  01 :      WORD SU;          <<NUMERO REPERE
0007  02  1F  1F  02  0047  0000  0000  01  01 :      RES 1;
0008  02  1F  1F  02  0048  0000  0000  01  01 :      RES 184;
0009  02  1F  1F  02  0100  0000  0000  01  01 : .KSTORE SECTION PILEK
0010  02  1F  1F  03  0000  0000  0000  01  01 :      RES 80;
0011  02  1F  1F  03  0050  0000  0000  01  01 : .LOCAL SECTION LOC501
0012  02  04  1F  04  0000  0000  0000  01  01 :      REF PROCEDURE CLOSE,ENCHNE,SORLIB;  <<PROCEDURES EXTERNES
0013  02  04  1F  04  0003  0000  0000  01  01 :      CONSTANT ERREAU=140;
0014  02  04  1F  04  0003  0000  0000  01  01 :      WORD CRD      =(0),     <<COMPTE RENDU
0015  02  04  1F  04  0004  0000  0000  01  01 :           S00      =(0),     <<SU DE SORTIE(POSTE
0016  02  04  1F  04  0005  0000  0000  01  01 :           LIBELLE,
0017  02  04  1F  04  0006  0000  0000  01  01 :           ZERO     =(0);     <<SI MODULE D'ENCHAINEMENT
0018  02  04  1F  04  0007  0000  0000  01  01 : .USING RC=ZDC,RL=LOC501,RK=PILEK;
0019  02  04  1F  04  0007  0000  0007  01  01 :                  <<FERMETURE FICHIER
0020  02  04  1F  04  0007  0000  0007  01  01 :      CALL CLOSE(@SU,@CRD,RA:=2);
0021  02  04  1F  04  0009  0000  000D  01  01 :      GOTO SORMODULE ON(CRD=0);
0022  02  04  1F  04  0009  0000  000F  01  01 :      IF(CRD='600A)THEN   <<FAU EXISTANTE
0023  02  04  1F  04  000A  0000  0012  01  01 :                     LIBELLE:=ERREAU;
0024  02  04  1F  04  000A  0000  0014  02  01 :                     CALL SORLIB(@S00,@LIBELLE,@CRD,RA:=3);
0025  02  04  1F  04  000C  0000  001C  02  01 :                     END;
0026  02  04  1F  04  000C  0000  001C  01  01 : SORMODULE:
0027  02  04  1F  04  000C  0000  001C  01  01 :      CALL ENCHNE(@ZERO,RA:=1);
0028  02  04  1F  04  000D  0000  0020  01  01 : END.
FIN DE COMPILATION  0000 ERREUR(S) '017E MOTS
```

```
*CC COMPIL-JC,D3
*/CALL LKLOAD
*/LINK MOL1-JC
MOD501 '015D      << Adresse 1ère instruction du programme
ZDC    '0000  }
PILEK  '0100  }   << Adresses des 1ères données du COMMON,
LOC501 '0150  }   << de la KSTORE et du LOCAL.
MODULE 1 :WARN 15
*/LOOK BIBFMS-MX/BIBMUT-MX
CLOSE  '019E
LDELIN '017E
 NOM   : MUTEX-0 BIBFMS
 DATE  : 30 JUILLET 1979
 REF   : 1.164.374.10/01.01.64.14
         '01E3
         '01E4
SULIGN '01E4
MODULE 1 :WARN 15
ENCHNE '0202
LOCHNE '01FE
SORLIB '0256
LOC    '0244
ALAFMS '02C0
LOCALA '028A
ALARME '0310
LOCALM '030B
MODIPV '0335
 NOM   : MUTEX-0 BIBMUT
 DATE  : 15 NOVEMBRE 1978
 REF   : 1.164.314.13/01.01.64.12
         '034D
         '034E
LIRLIB '03E6
LOC    '03CE
SORTIE '045F
LOCSOR '0450
CPTOUT '0472
LOCOCT '046E
NFUBTB '049F
LOC    '049B
TRONES '04F8
TIMOUT '0568
LOCOCT '0568
*/STAT
*/ENDL                    } << Edition de liens

BEGIN  '0000
END    '057D
RUN    '0160
*/CATAL IM,MOD501-JC      } << Chargement du programme
*/CLOSE IM                << dans le fichier MOD501-JC
*/SI MOL1-JC
*/DELE SI
*/CALL GESMOD
*/OPEN OLD 1,MOD501-JC    } << chargement dans le
*/LMOD,1,501,BIBCOM,256   << bibliothécaire
*/EOJ
```

```
LIGNE RC  RL  RW  DONNEES   TABL  PROG  PF  RN :*SOURCE

0001  1F  1F  1F  1F  0000  0000  0000  00  00 : << DESTRUCTION FICHIER SEQUENTIEL(MODULE 502)
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : <<
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :       MAIN PROCEDURE MOD502
0004  1F  1F  1F  1F  0000  0000  0000  00  00 : .COMMON SECTION ZDC
0005  02  1F  1F  02  0000  0000  0000  01  01 :       RES 70;
0006  02  1F  1F  02  0046  0000  0000  01  01 :       WORD SU;          <<NUMERO REPERE
0007  02  1F  1F  02  0047  0000  0000  01  01 :       RES 1;
0008  02  1F  1F  02  0048  0000  0000  01  01 :       RES 184;
0009  02  1F  1F  02  0100  0000  0000  01  01 : .KSTORE SECTION PILEK
0010  02  1F  1F  03  0000  0000  0000  01  01 :       RES 80;
0011  02  1F  1F  03  0050  0000  0000  01  01 : .LOCAL SECTION LOC502
0012  02  04  1F  04  0000  0000  0000  01  01 :       REF PROCEDURE DELET,ENCHNE,SORLIB;  <<PROCEDURES EXTERNES
0013  02  04  1F  04  0003  0000  0000  01  01 :       CONSTANT ERREAU=140;
0014  02  04  1F  04  0003  0000  0000  01  01 :       WORD CRD      =(0),     <<COMPTE RENDU
0015  02  04  1F  04  0004  0000  0000  01  01 :            S00      =(0),     <<SU DE SORTIE(POSTE
0016  02  04  1F  04  0005  0000  0000  01  01 :            LIBELLE,
0017  02  04  1F  04  0006  0000  0000  01  01 :            ZERO     =(0);     <<SI MODULE D'ENCHAINEMENT
0018  02  04  1F  04  0007  0000  0000  01  01 : .USING RC=ZDC,RL=LOC502,RK=PILEK;
0019  02  04  1F  04  0007  0000  0007  01  01 :                  <<DESTRUCTION FICHIER
0020  02  04  1F  04  0007  0000  0007  01  01 :       CALL DELET(@SU,@CRD,RA:=2);
0021  02  04  1F  04  0009  0000  000D  01  01 :       GOTO SORMODULE ON(CRD=0);
0022  02  04  1F  04  0009  0000  000F  01  01 :       IF(CRD='600A)THEN   <<FAU EXISTANTE
0023  02  04  1F  04  000A  0000  0012  01  01 :                  LIBELLE:=ERREAU;
0024  02  04  1F  04  000A  0000  0014  02  01 :                  CALL SORLIB(@S00,@LIBELLE,@CRD,RA:=3);
0025  02  04  1F  04  000C  0000  001C  02  01 :                  END;
0026  02  04  1F  04  000C  0000  001C  01  01 : SORMODULE:
0027  02  04  1F  04  000C  0000  001C  01  01 :       CALL ENCHNE(@ZERO,RA:=1);
0028  02  04  1F  04  000D  0000  0020  01  01 : END.
FIN DE COMPILATION  0000 ERREUR(S) '017E MOTS
```

```
*CC COMPIL-JC,D3
*/CALL LKLOAD
*/LINK MOL1-JC
MOD502 '015D      << Adresse 1ère instruction du programme
ZDC    '0000  }
PILEK  '0100  }   << Adresses des 1ères données du COMMON,
LOC502 '0150  }   << de la KSTORE et du LOCAL
MODULE 1 :WARN 15
*/LOOK BIBFMS-MX/BIBMUT-MX
DELET  '019E
LDELET '017E
SULIGN '01E3
 NOM  : MUTEX-0 BIBFMS
 DATE : 30 JUILLET 1979
 REF  : 1.164.374.10/01.01.64.14
        '01FD
        '01FE
MODULE 1 :WARN 15
ENCHNE '0202
LOCHNE '01FE
SORLIB '0256
LOC    '0244
MODIPV '02BA
ALARME '02D7
LOCALM '02D2
ALAFMS '0302
LOCALA '02FC
 NOM  : MUTEX-0 BIBMUT
 DATE : 15 NOVEMBRE 1978
 REF  : 1.164.314.13/01.01.64.12
        '034D
        '034E
LIRLIB '03E6
LOC    '03CE
SORTIE '045F
LOCSOR '0450
CPTOUT '0472
LOCOCT '046E
NFUBIB '049F
LOC    '049B
TRONES '04F8
TIMOUT '0568
LOCOCT '0568
*/STAT
*/ENDL

BEGIN  '0000
END    '057D
RUN    '0160
*/CATAL IM,MOD502-JC
*/CLOSE IM
*/SI MOL1-JC
*/DELE SI
*/CALL GESMOD
*/OPEN OLD 1,MOD502-JC
*/LMOD,1,502,BIBCOM,256
*/FOJ
```

<< Edition de liens

<< Chargement du programme
<< dans le fichier MOD502-JC

<< Chargement dans le
<< bibliothécaire

C - MISE AU POINT DE L4APPLICATION ESSAI :

L'application ESSAI est lancée sur POSTO2.

L'application SYSDB (DEBUG) est lancée sur POSTO1.

La création du fichier a été faite auparavant par une commande de MUTEX.

<< signifie un commentaire.

<<Sur POSTO1

```
> /LAN APP SYSDBG
> DEBUG SUR PØSTO2          <<Activation DEBUG
>>MØDULE 500
>>STØP '17A VISU VAR '153,'154,'155,'156/ASCI     <<demande d'actions
>>STØP '17A VISU VAR '4D/DEC                      <<différées à l'a-
>>ACT                                             <<dresse '17A
```

<<Sur POSTO2

```
> /LAN APP ESSAI                    <<application à tester
> ØUV FIC EXEMPL-JC SUR 1           <<commande l'applica-
                                    <<tion ESSAI
```

<< Sur POSTO1

```
PRØCESSUS PØSTO2  MØDULE  500   STØP ADRESSE  017A

VISU  VARIABLES
 '0153 '0154 '0155 '0156                    <<Messages
 ""   "EX"  "EM"  "JC"                      <<de DEBUG

VISU  VARIABLES
 '004D
    +1
```

<< Le MOVE n'est pas bon (RB est initialisé avec une adresse incor-
<< recte), il faut revenir juste avant et mettre les bonnes valeurs
<< dans RA et RB.

```
>>VISU REC
  RA    RB    RX    RY    RC    RL    RI    RK    RP    RS   RSLO  RSLE
 'CAC3 '0154 '0000 '0101 '0030 '01D0 '010F '0100 '017A '013C '1293 '13DE
>>GOTO '176
>>CHAN REC RA,RB/'46,'153/
>>ACT

PROCESSUS POST02  MODULE  500   STOP ADRESSE  017A


VISU  VARIABLES
 '0153 '0154 '0155 '0156
 "EX"  "EM"  "PL"  "JC"

VISU  VARIABLES
 '004D
    +1
```

<< C'est bon, on peut maintenant voir si l'OPEN se passe
<< bien.

```
>>STOP '13C VISU VAR '159
>>ACT

PROCESSUS POST02  MODULE  500   STOP ADRESSE  013C


VISU  VARIABLES
 '0159      <<adresse CRD
 '0000      <<c'est OK
```

<< On peut maintenant tester le module 501

```
>>MODU 501
>>STOP '164 VISU VAR '46,'47/DEC
>>ACT
```

<<Sur POSTO2

```
> FERMER FICHIER SUR 1
```

<<commande de l'application ESSAI

<<Sur POSTO1

```
PROCESSUS POST02  MODULE  501   STOP ADRESSE  0164


VISU  VARIABLES
 '0046 '0047
     0    +1
```

<< Le numéro de repère doit être à l'adresse '46 et non '47

```
>>CHAN VAR '46,'47/+1,0/
>>VISU VAR '46,'47/DEC
 '0046 '0047
    +1     0
```

<<on corrige

<< C'est bon, on peut voir si le CLOSE se passe bien

```
>>STOP '16A VISU VAR '153
>>ACT

PROCESSUS POSTO2  MODULE  501    STOP ADRESSE  016A


VISU  VARIABLES
 '0153        << adresse CRD
 '0000        << c'est OK

  << Le CLOSE est OK, la même erreur existe dans le module 502
  << (DETRUIRE FICHIER). Pour corriger, il faut avant, demander
  << l'activation du module 500 (OUVRIR FICHIER), faire la modi-
  << fication à l'adresse 176

>>MODU 500 MOD TR                    << exécution en mode TRACE
>>STOP ' 176 CHAN REG RA,RB/'46,'153/
>>SUP '17A
>>SUP '13C
>>MODU 502
>>STOP '164 CHAN VAR '46,'47/+1,0
>>ACT

                          << Sur POSTO2

> OUV FICHIER EXEMPL-JC SUR 1        << commande de l'application ESSAI

                          << Sur POSTO1

PROCESSUS POSTO2  MODULE  500    STOP ADRESSE    C176

MODIF REGISTRES
  RA       RB
 '0046    '0153
```

(13) PROGRAID - Outil de mise au point sur système mono console
Fiches techniques - T2000 - Télémécanique - 1973

(14) AID - "Outil interactif d'aide à la mise au point"
Manuel de référence SOLAR - T1600 - Janvier 1976 - SEMS

(15) DRIP16 - "Debugging réal time interactive program"
Manuel de référence - SOLAR - Octobre 1976 - SEMS

(16) ODT (On line Debugging Tool)
RSX-11 M V 3.0 First Printing, December 1976

(17) DEBUG - An extension to current on line Debugging techniques
Thomas G. EVANS, D. Lucille DARLEY - CACM Vol. 8 n° 5- 1965

(18) ON - LINE DEBUGGING TECHNIQUES : A Survey
Thomas G. EVANS and D. Lucille DARLEY - 1966

(19) Exploratory experimental studies comparing on-line and off-line programming performance - Communication of the ACM
Volume 11/ number 1/ Janvier 1968

(20) Perspectives on software engineering - MARTIN V. ZELKOWITZ - Communication of the ACM - Volume 10/ number 2/ Juin 1978

(21) "Aide à la mise au point en langage évolué" - Thèse du Docteur-Ingénieur René ODDOUX - 1973

(22) "Outils de mise au point pour langages de haut niveau : association de modules et contrôle de l'exécution" - Thèse de Docteur de troisième cycle Henry SAVARY - 15 Septembre 1973

(23) "Outils d'aide à la mise au point en programmation système"
Rapport de DEA - Silvia LUMELSKY - Septembre 1975

(24) The mythical man month, Addison-Weslay Publ. co ;
BROOKS, Frederic P. ; Reading Mass. - 1975

(25) Debugging techniques en large systems - Randall RUSTIN
Prentice HALL, Englewoad CLIFFS - 1971

(26) "Technique de programmation et méthodologie de conduite de projet" - Rapport de synthèse d'un groupe travail SEMS - Mars 1979