

PROCEDE ET DISPOSITIF PERMETTANT A UN PROGRAMME FIGE DE POUVOIR EVOLUER

5 La présente invention concerne les programmes d'ordinateur et notamment ceux incorporés dans des supports à microcircuit(s) et, plus généralement les objets portatifs dotés de circuits intégrés comportant au moins une unité centrale, une mémoire morte de programme, une mémoire non volatile programmable et une mémoire de travail. Cette mémoire non
10 volatile est capable de stocker des données et du code, et l'unité centrale peut faire exécuter ce code par l'unité centrale.

 Dans un même objet il est connu jusqu'à ce jour de faire cohabiter deux types très différents de programme, l'un est inscrit dès la fabrication dans le circuit intégré et non modifiable, l'autre est introduit de l'extérieur au
15 cours de l'utilisation normale de l'objet selon les besoins.

 De nos jours, les supports à microcircuit(s) peuvent répondre techniquement à de nombreux besoins. Cependant, la réalisation d'un "masque", c'est à dire le programme dans la mémoire ROM, reste encore très cher et cet investissement décourage bon nombre de "petits" clients qui
20 désireraient acheter quelques milliers de cartes. Une solution consisterait à utiliser un masque existant et rajouter les fonctions que demande le client dans la mémoire programmable.

 La possibilité d'introduire et d'exécuter un code supplémentaire en mémoire programmable offre l'avantage de pouvoir facilement rajouter à un
25 programme existant de nouvelles fonctions ou d'adapter ce programme à des besoins spécifiques.

 Le brevet FR 2 655 170 décrit un dispositif électronique portatif capable d'enregistrer des séquences de codes en mémoire non volatile. La mémoire programmable non volatile comprend une partie particulière
30 configurée en vue de contenir des séquences de codes, cette partie est elle-même divisée en une zone d'indicateurs, une zone de données de séquences de codes et une zone de caractères de contrôle par bloc. Une instruction de test est implémentée dans le programme en mémoire morte pour lire et tester la zone des indicateurs, et dans le cas favorable réaliser le
35 saut. A chaque saut doit donc correspondre une instruction de test et une zone mémoire allouée en mémoire programmable.

Il est également connu par la demande de brevet français 2 667 417 une carte à microprocesseur conçue pour recevoir des programmes multiples en mémoire programmable dans laquelle des instructions dites de filtrage sont associées à chacun des emplacements réservés en mémoire programmable pour constituer un moyen de mémorisation de l'adresse du code d'un sous programme d'instructions accessible par exécution de l'instruction de filtrage mémorisée dans la mémoire morte de la carte. Chaque instruction de filtrage doit connaître l'adresse de la zone mémoire contenant l'adresse du code associé dans la mémoire programmable, ce qui impose une hiérarchie d'accès figée une fois pour toute.

Les deux solutions évoquées ci-dessus présentent les inconvénients ci-après.

Un homme du métier désirant implémenter plusieurs instructions de saut doit multiplier les instructions de saut dans le programme en mémoire morte et les zones spécifiques en mémoire programmable. Si un grand nombre d'instructions de ce type doivent être installées dans le programme principal, la place totale qu'elles occupent en mémoire morte devient importante au détriment de celle du programme. De plus, dans la mémoire programmable non volatile, même si le sous-programme n'est pas enregistré, sa place est réservée ainsi que celle des indicateurs, ce qui occupe inutilement une place non négligeable. Enfin, une fois écrit, le sous-programme ne peut plus être modifié. En résumé, cette solution convient à un nombre très restreint d'instructions de saut et quelques sous-programmes, en aucun cas le problème d'un nombre important de sauts et de sous-programmes n'est résolu.

La présente invention a pour objet un procédé et un dispositif permettant de faire évoluer les fonctionnalités d'une carte à puce sans les inconvénients des brevets cités et notamment en permettant un nombre plus important d'appels à des sous programmes. En effet, le dispositif selon l'invention offre les avantages d'être facile à mettre en oeuvre dans un programme, ou d'utiliser peu de mémoire code, ou de minimiser la taille mémoire utilisée, ou d'être évolutif et d'autoriser toutes sortes de modifications, ou d'optimiser le temps d'exécution.

Pour ce faire, le dispositif d'exécution de séquences de code dans un support comprenant un circuit intégré capable d'exécuter des séquences de code et, d'une part, une première mémoire contenant un programme

principal et éventuellement d'autres séquences de codes exécutables par le circuit intégré et, d'autre part, une seconde mémoire programmable non volatile contenant éventuellement des séquences de codes exécutables par le circuit intégré et une troisième mémoire de travail, est caractérisé en ce qu'une table d'orientation contenue dans la seconde mémoire contient au

moins un champ contenant une donnée de référence de code :

des premiers moyens (INS_INT) permettent :

- de vérifier la présence d'une référence de code,
- de mémoriser en mémoire de travail la donnée d'adresse associée à la référence du code et de positionner un indicateur de déroutement DI et,

des seconds moyens (INS_ORI) permettent :

- de tester l'indicateur de déroutement DI et,
- d'effectuer le saut à l'adresse pointée par le contenu de la mémoire de travail.

Selon une autre particularité la donnée d'adresse d'une séquence de code contenue dans l'une des trois mémoires associée à la référence de code est calculée à partir de la référence de code.

Selon une autre particularité un deuxième champ contient la donnée d'adresse d'une séquence de code contenue dans l'une des trois mémoires.

Selon une autre particularité le dispositif comporte dans une zone éventuellement protégée de la mémoire programmable une information définissant la première adresse et éventuellement le sens de recherche pour les premiers moyens de vérification.

Selon une autre particularité la table d'orientation comporte un champ supplémentaire indiquant l'adresse suivante dans la table où se trouvent les informations correspondantes à la référence de code suivant.

Selon une autre particularité la table d'orientation comporte un champ supplémentaire permettant de s'assurer de l'intégrité des champs précédents et/ou de séquence de code rattachée par comparaison avec le résultat fourni par un moyen de calcul d'une somme de contrôle.

Selon une autre particularité les seconds moyens comportent des moyens de mémoriser en mémoire de travail une adresse de retour après un saut, lorsque plusieurs seconds moyens sont disponibles pour effectuer un saut, à destination de la même adresse de séquence.

Selon une autre particularité le dispositif comporte des moyens CAI d'empêcher l'enregistrement d'un code additionnel, et des moyens RCI d'empêcher une régénération de la carte.

Un autre but est de proposer un procédé d'enregistrement de
5 nouvelles fonctionnalités.

Ce but est atteint par le fait que le procédé d'enregistrement d'un nouveau code dans un dispositif selon une des revendications précédentes, est caractérisé en ce qu'il consiste :

- à recevoir un ordre d'enregistrement ;
- 10 - à vérifier la présence d'un premier champ constituant la référence de code A, B, C,... et d'un second champ AD_Cod_A, AD_Cod_B, AD_Cod_C... représentant l'adresse associée à la référence de code ;
- à vérifier que la valeur de la première adresse constituant l'adresse de table d'orientation AD_TAB est cohérente ;
- 15 - à vérifier qu'un drapeau ECA n'est pas actif ;
- à contrôler que la référence fournie par le premier champ n'existe pas dans la table ;
- à positionner le drapeau ECA dans la zone de mémoire programmable protégée à l'état actif ;
- 20 - à recevoir et écrire les informations correspondantes au nouveau code ;
- à vérifier que l'opération s'est déroulée correctement et mettre à jour la table d'orientation ;
- à positionner le drapeau ECA à l'état inactif.

25 Selon une autre particularité l'opération de mise à jour de la table comporte une écriture du champ d'adresse suivante dans la table d'orientation à l'adresse correspondant au mot d'orientation du code qui précède le code écrit.

D'autres particularités et avantages de la présente invention
30 apparaîtront plus clairement à la lecture de la description ci-après d'un exemple de réalisation du dispositif et de mise en oeuvre du procédé, illustrés à l'aide des figures ci-après dans lesquelles :

- la figure 1 représente une partition de la mémoire programmable de la carte utilisant une variante de procédé de gestion de la table
35 d'orientation ;

- la figure 2 représente un exemple de partition de la mémoire morte de la carte ;

- la figure 3 représente dans un programme principal un exemple d'enchaînement d'une instruction d'interrogation exécutée dans le cours du programme principal et ultérieurement d'une instruction d'orientation ;

- la figure 4 représente un exemple d'enchaînement possible d'instructions d'interrogation et d'orientation ;

- la figure 5 représente un exemple d'algorithme de déroulement d'une séquence de code constituant l'instruction d'interrogation ;

- la figure 6 représente un exemple de configuration de la partition de la mémoire programmable incorporant la table d'orientation et les zones de séquences de code ;

- la figure 7 représente un deuxième exemple de réalisation de la partition de la mémoire incorporant la table d'orientation et les zones de séquences de code ;

- la figure 8 représente un exemple d'organigramme de déroulement d'une séquence de code constituant une instruction d'orientation ;

- la figure 9 représente un exemple des champs constitutifs d'une instruction d'écriture d'une séquence ;

- la figure 10 représente l'organigramme de déroulement d'une séquence d'enregistrement déclenchée par une instruction d'écriture ;

- la figure 11 représente les éléments constitutifs d'un support à microcircuit(s) nécessaire à la mise en oeuvre du dispositif et du procédé de l'invention.

L'invention sera mieux comprise par la description de l'exemple d'utilisation ci-dessous, dans une carte à microprocesseur selon la figure 11.

Une carte à microprocesseur possède une Unité Centrale (10) de type microprocesseur ou à circuit intégré capable d'exécuter des séquences d'instructions, une mémoire de travail RAM (14), une mémoire non volatile ROM (12) contenant le programme principal, une mémoire non volatile (11) EPROM, EEPROM ou FRAM programmable et exécutable par l'U.C. contenant des données et, des organes d'entrées/sorties (13) permettant le dialogue avec un lecteur par une liaison (15) avec ou sans contact. Le programme est gravé en ROM lors de la fabrication du composant, il ne peut donc pas être modifié par la suite.

La mémoire programmable (11) est divisée en plusieurs parties, comme le montre la figure 1. Une première partie (110) notée zone système contient des informations systèmes non lisibles de l'extérieur, permettant de contrôler la validité (authenticité et/ou intégrité) et la localisation des informations ou groupe de données contenus dans le reste de la mémoire (pointeurs...). Une seconde partie (111) notée zone de données est accessible de l'extérieur et sert au stockage des données utilisateurs. Une troisième partie (112) notée table d'orientation contient la table d'orientation, cette dernière est formée d'éléments identiques, chaque élément contient plusieurs champs dont deux principaux : le premier est une donnée de référence de code : "A", "B", ... et le second est l'adresse de début de la séquence de code correspondante mémorisée dans une quatrième partie (113) de la mémoire : AD_Cod_A, AD_Cod_B... La quatrième zone notée zone de séquence de code contient les séquences de code pouvant être appelées par le programme principal. Il est à noter que dans une variante de l'invention la table d'orientation ou la séquence de code peut être chargée en mémoire de travail (RAM) au lieu de l'EEPROM.

L'adresse de début de la table d'orientation appelée AD_TAB, est mémorisée dans la zone système, une seconde information notée SENS, détermine dans quel sens le pointeur lit cette table, soit dans le sens des adresses croissantes, soit dans le sens des adresses décroissantes. Les adresses des valeurs AD_TAB et SENS peuvent être ainsi les seules données relatives à l'invention que le programme en ROM connaît. Il peut être utile également de mémoriser la taille en octets ou le nombre d'éléments maximum (Ta Ta) de la table d'orientation.

Voici un exemple illustrant l'intérêt du procédé de déroutement selon l'invention. Le programme principal peut contenir un algorithme tel le D.E.S. (Data Encryption Standard) capable d'effectuer du chiffrement/déchiffrement d'informations ou certaines fonctions d'authentications. Il peut être intéressant de remplacer cet algorithme par un autre, ou de changer complètement son implémentation, afin de le rendre par exemple plus rapide. Le code de l'algorithme constitue un bloc dans la mémoire ROM de l'Operating System, ce code n'est pas modifiable et il peut être appelé de divers endroits du programme principal, .

L'exemple sera mieux compris en regardant la figure 2. La mémoire ROM qui contient entre autres le programme principal, est divisée en trois

parties. La première réalise l'initialisation de la carte après une mise sous tension : le test de la mémoire de travail, la lecture de l'état de la carte, l'émission des octets de la réponse au RESET. La seconde partie contient le programme d'application qui exécute les différents ordres demandés par le terminal (la première partie de code est toujours exécutée avant la seconde).
5 Les première et seconde parties constituent le programme principal. La troisième partie contient le code "dormant" dont le rôle est expliqué par la suite.

L'algorithme appelé ALGO_1 dont on veut modifier le
10 fonctionnement est contenu dans la partie application. Dans l'exemple, cet algorithme est appelé depuis cinq endroits différents du code. Le nouvel algorithme a été préalablement chargé dans la zone de séquence de codes dans la mémoire programmable et la table d'orientation a été mise à jour, l'adresse d'exécution de sa séquence de code est AD_Cod_A.

15 A la mise sous tension, le programme principal exécute la partie initialisation, on trouve dans cette partie le premier saut à ALGO_1 noté "APPEL_0", aucun déroutement n'est prévu donc ce saut n'est conditionné par aucune instruction de déroutement. On y trouve également une séquence d'instruction d'interrogation: INS_INT_A que le programmeur a
20 pris soin d'implémenter dans cette partie afin d'être sûr de l'exécuter au moins une fois avant l'exécution du programme de la partie application. Dans l'exemple, la référence de code associée à cette séquence d'instruction d'interrogation est "A", elle est inscrite dans le programme principal. Dans un premier temps, l'instruction d'interrogation cherche dans
25 la zone système l'adresse de la table d'orientation AD_TAB. Puis, à partir de cette adresse et dans le sens des adresses croissantes ou décroissantes selon la valeur de "SENS", chaque élément de la table est lu. La valeur des premiers champs correspondant aux références de code est comparée avec la valeur "A". Si la valeur "A" est trouvée dans un premier champ de la table
30 d'orientation, la valeur du second champ correspondante (AD_Cod_A) est lue et stockée dans le registre AD_SAUT en mémoire de travail (RAM) et un indicateur de déroutement (DI) en mémoire de travail est mis en position active.

Le sous programme ALGO_1 est appelé cinq fois par APPEL_0,
35 APPEL_1, APPEL_4 dans le programme principal lorsque DI n'est pas actif. Pour APPEL_1, APPEL_2 et APPEL_3, le déroutement vers un autre

programme ALGO_2 qui peut être en mémoire morte ou en mémoire programmable est réalisé lorsque l'indicateur de déroutement DI est actif car le programmeur a pris soin d'implémenter dans le code du programme principal une instruction d'orientation : INS_ORT_1, INS_ORT_2 et
5 INS_ORT_3 avant chaque saut APPEL_1, APPEL_2, APPEL_3 effectif dans ALGO_1. Comme le montre la figure 3, l'instruction d'orientation consiste à tester tout d'abord si l'indicateur de déroutement DI est actif. Si l'indicateur est inactif, le programme principal suit son cours normal et exécute l'algorithme en ROM par exemple en effectuant le saut APPEL_1 vers
10 ALGO_1. Si l'indicateur de déroutement DI est actif, l'instruction d'orientation déroute le programme à l'adresse indiquée par le contenu du registre AD_SAUT. La valeur AD_Cod_A stockée préalablement dans le registre AD_SAUT est chargée dans le compteur ordinal de l'Unité Centrale et, la séquence de code à l'adresse chargée est exécutée. Par contre, le sous
15 programme, ALGO_1, par la volonté du programmeur, est appelé par APPEL_4 sans possibilité de déroutement, car aucune instruction d'orientation ne lui est associée.

Il est à noter que les expressions d'instruction d'interrogation et d'orientation sont en fait des macro instructions constituées par un ensemble
20 d'instructions élémentaires exécutables par l'unité centrale.

Une même instruction d'orientation peut être exécutée un nombre illimitée de fois, tant que l'indicateur reste actif, le saut s'effectuera toujours à l'adresse mémorisée dans AD_SAUT. De même, comme le montre la figure 2, plusieurs instructions d'orientation " INS_ORT_1", INS_ORT_2,
25 INS-ORT_3 avec le même code de référence "A", peuvent être implémentées dans le programme principal, leurs exécutions entraîneront le même effet. Dans ce cas, il convient d'étudier avec un soin tout particulier, l'adresse de retour au programme principal, des exemples de réalisation d'instruction de retour seront expliqués dans la suite de ce document.

30 En règle générale, le programmeur doit prendre soin qu'au moins une séquence d'instruction d'interrogation soit exécutée avant toute instruction d'orientation. Si la séquence de code ne doit être déroulée qu'une seule fois, une instruction d'effacement de l'indicateur de déroutement DI, visant à rendre inactif l'indicateur de déroutement DI, est
35 exécutée par la séquence de code. Ainsi, une fois le retour dans le

programme principal, aucun déroutement ne pourra plus s'effectuer lors des instructions d'orientation suivantes.

A la lumière de cet exemple, le premier avantage de séparer les deux séquences d'instructions d'interrogation et d'orientation apparaît. La
5 séquence d'instruction d'interrogation, qui est plus élaborée et donc plus coûteuse en code, n'est mise dans le programme qu'une fois. Les séquences d'instruction d'orientation qui sont très courtes et peu coûteuses en place mémoire sont mises en place chaque fois que cela est nécessaire, d'où un gain de place dans la mémoire programme.

10 L'indicateur de déroutement DI peut très bien être remplacé par une valeur particulière du registre AD_SAUT. Si le registre AD_SAUT contient la valeur "NULL" qui n'est pas une valeur significative d'adresse de la mémoire non volatile pour un composant donné, cela signifie qu'aucun déroutement du programme principal n'est à prendre en compte. De même, si le contenu
15 de la mémoire AD_TAB est une valeur non cohérente avec celle de début de la table d'orientation ("NULL" par exemple), on peut considérer qu'aucune séquence n'est disponible et par voie de conséquence, aucun déroutement envisagé.

Comme le montre la figure 4, plusieurs instructions d'interrogation
20 (INS_INT_A, INS_INT_B) peuvent être introduites dans le programme principal. Chacune de ces instructions possède son propre code de référence inscrit dans la mémoire programme, et recherche la présence de ce code parmi les premières valeurs de la table d'orientation en mémoire programme. Si la valeur de ce code est bien en table, l'adresse de la
25 séquence de code associé est mise dans le registre de travail AD_SAUT. Ainsi, l'exécution d'une instruction d'interrogation efface et remplace la valeur du registre AD_SAUT mise à jour par la précédente instruction d'interrogation.

Si une précédente instruction d'interrogation a mis à jour le registre
30 AD_SAUT avec l'adresse de déroutement, et si une seconde instruction d'interrogation avec un code de référence différent s'exécute en ne trouvant pas son code de référence dans la table d'orientation, il est suffisant et judicieux d'effacer la précédente valeur d'AD_SAUT et ainsi, d'annuler tout déroutement jusqu'à la prochaine instruction d'interrogation. Cependant, on
35 ne s'écarte pas de l'esprit de l'invention si on garde la précédente valeur en conservant ainsi le déroutement.

Un second avantage d'utiliser des références de code "A", "B"... inscrits dans une table d'orientation apparaît alors : cette table ne contient que des données utiles, ne se référant qu'aux déroutements effectifs voulus dans le programme principal. Ainsi, à tout moment, sa taille est optimisée, d'où un gain d'occupation mémoire.

Avantageusement, il est possible de stocker dans le programme principal, des séquences de code qui ne sont pas appelées par lui. A l'aide du mécanisme selon l'invention, il est possible d'associer à chacune de ces séquences une référence de code qui est mémorisée dans la table d'orientation avec l'adresse de début de la séquence. Ainsi, il est possible d'activer un code "dormant" dans la mémoire programme du dispositif portable.

Avantageusement, il est aussi possible par les instructions d'orientation de définir plusieurs points d'entrée à une même séquence de code et de les associer à des codes de référence différents. De même, plusieurs séquences "A", "B" et "C" par exemple peuvent être regroupées en une seule séquence écrite d'un seul coup en mémoire programmable avec le code de référence "A" et le point d'entrée AD_Cod_A, les deux autres codes de référence et les points d'entrées sont écrits par la suite. Un exemple de table d'orientation permettant cette opération est présenté plus loin (figure 7).

Le déroulement des instructions d'interrogation va être maintenant explicité, avec l'aide de la figure 5. Un paramètre d'entrée de cette instruction est la valeur notée "X" du code de référence correspondant à la séquence de code qui sera exécutée lors d'une prochaine instruction d'orientation, dans l'exemple, cette valeur est "X". A l'étape 1, le programme principal, en exécutant la séquence d'instruction d'interrogation, lit dans la zone système à l'adresse @ qui est connue par le programme en ROM le contenu AD_TAB qui est l'adresse de début de la table d'orientation. A l'étape 2, la séquence d'instruction d'interrogation teste le mot lu. Si le mot lu n'est pas significatif, la table d'orientation n'existe pas et aucun déroutement de code n'est envisageable. La séquence d'instruction d'interrogation est terminée et l'indicateur de déroutement DI est inactif (étape 10). Si le mot lu est significatif, la séquence d'instruction d'interrogation met à jour à l'étape 3 un pointeur tel que par exemple AD_SAUT en mémoire de travail avec la valeur AD_TAB lue. Puis à l'étape 4, la séquence d'interrogation lit un

élément de la table pointé par ce pointeur. A l'étape 5 de la séquence d'interrogation la valeur du champ correspondant à la référence de code est comparée à la valeur du code de référence "X" fournie comme paramètre d'entrée. Si les deux valeurs sont identiques, la séquence d'interrogation positionne à l'étape 8 l'indicateur de déroutement DI à l'état actif. A l'aide du pointeur, la séquence d'interrogation lit à l'étape 9 la valeur d'adresse "AD_Cod_X" située aussitôt après le code de référence dans la table d'orientation. Cette valeur est stockée dans le registre AD_SAUT en mémoire de travail RAM. Si les deux valeurs de référence de code ne sont pas identiques, à l'étape 6 le pointeur est mis à jour pour pointer le code de référence suivant dans la table d'orientation. Cette mise à jour s'effectue en prenant en compte la taille de chaque élément de la table, si, par exemple, la table est composée de mots de 32 bits, la valeur du pointeur qui adresse des octets (8 bits) est augmentée de 4. La séquence d'interrogation teste à l'étape 7 si le mot de la mémoire programmable adressé par le pointeur est encore dans la table d'orientation. Ce test dépend de l'organisation de la table dont plusieurs exemples seront donnés par la suite. Si le mot pointé n'appartient plus à la table, la séquence d'instruction d'interrogation est terminée et l'indicateur de déroutement DI est positionné à l'état inactif par l'étape 10. Si le mot pointé est encore en table d'orientation, la séquence d'interrogation reboucle à l'étape 4.

En sortie de la séquence d'instruction d'interrogation, l'indicateur de déroutement DI est toujours positionné, s'il est actif, la valeur dans AD_SAUT est significative. Avantageusement, la séquence d'instructions d'interrogation peut être écrite par le programmeur qui réalise le programme principal comme un sous-programme avec d'une part, pour paramètres d'entrée : le code de référence, par exemple X, à chercher dans la table et éventuellement une adresse de retour et d'autre part, pour paramètre de sortie, un indicateur de déroutement DI spécifiant si le déroutement doit être effectivement réalisé. Il est possible lors de l'initialisation, d'effectuer une lecture de la table d'orientation et de stocker en mémoire de travail toutes les adresses correspondantes aux codes de référence existants, cela évite à chaque instruction d'interrogation de balayer systématiquement la table et, de rechercher inutilement un code qui n'existe pas.

La figure 6 montre un exemple de table d'orientation. Cette table est composée de mots d'orientation de 32 bits, par exemple, le premier champ,

par exemple d'un octet, contient le code de référence, le deuxième champ, par exemple de deux octets suivants, contient l'adresse de début de la séquence de code en mémoire non volatile, le dernier champ, par exemple d'un octet, est optionnel, et peut contenir un code de somme de contrôle (Checksum) : (CHK_A) permettant de contrôler l'intégrité de la valeur des deux premiers champs et/ou de la séquence de code attachée. Dans cette table d'orientation, les éléments se suivent de manière contiguë. La fin de table est atteinte lorsque le mot suivant est vierge. La valeur SENS lue dans la zone système indique le sens d'évolution du pointeur de table : par adresses croissantes ou décroissantes. Dans la figure 6, c'est le sens des adresses croissantes qui est choisi.

La disposition de cette table est intéressante lorsqu'elle contient un grand nombre d'éléments. En effet, les codes de références de ces éléments peuvent se classer selon un certain ordre, l'organisation de ces éléments dans la table peut donc se faire selon des références croissantes : "A", "B", "C", "D", "E" ... ou décroissantes "Z", "Y", "X", "W", ... La taille en octets de la table est déterminée. Elle est soit mémorisée dans une zone (Ta Ta figure 1) de la zone système, soit par une pré-lecture de la table lors de l'initialisation. Lors de la recherche d'un certain code de référence, il est alors possible d'appliquer des algorithmes de tri, par exemple, le pointeur est initialisé avec l'adresse de l'élément situé en milieu de la table en ajoutant à la valeur contenue dans AD_TAB la moitié du nombre d'octets de la table précédemment déterminé puis, le code de référence lu par le pointeur est comparé avec celui à rechercher, selon le résultat le pointeur se déplace en avant ou en arrière et pointe alors au milieu de la zone restante à explorer. Ce mécanisme permet de gagner du temps dans la recherche d'une référence de code en table. Ainsi, si la table possède n éléments ordonnés, la façon classique de recherche en passant d'un élément au suivant aboutit à la lecture en moyenne de $n/2$ éléments pour trouver le bon, Une recherche comme indiquée précédemment permet de rechercher en lisant ($\log n$) éléments, ce qui est un avantage lorsque la valeur n est grande.

Si, après avoir balayé toute la table, aucune valeur lue dans la table ne possède la référence "A" recherchée, aucun déroutement du programme principal vers la séquence de code "A" ne sera réalisé, un indicateur de

déroutement DI en mémoire de travail est dans ce cas mis en position inactive.

Cette disposition oblige le programmeur à prévoir dès la première inscription d'une séquence de codes la taille de la table d'orientation de manière à laisser suffisamment de place mémoire après la valeur de
5 AD_TAB pour compléter cette table et, éventuellement, écrire d'autres séquences de codes.

Une variante de réalisation de table d'orientation peut contenir uniquement la référence de code et dans ce cas le programme d'instruction
10 d'interrogation calculera l'adresse de saut en appliquant un algorithme ayant comme paramètre d'entrée la référence de code.

La figure 7 montre un autre exemple de table d'orientation qui évite au programmeur de prévoir en mémoire Ta Ta l'information correspondant à sa taille ou de faire une pré-lecture de table pour déterminer sa taille. Cette
15 table contient des mots d'orientation ayant un champ de plus que les trois champs précédents : valeur de code de référence, adresse de début des séquences de code et somme de contrôle (Check-Sum). Le champ supplémentaire, qui occupe par exemple les quatrième et cinquième octets, contient la valeur d'adresse de l'élément suivant dans la table d'orientation.
20 La table se lit de la façon suivante : le mot d'orientation correspondant à la première séquence de code se situe à l'adresse AD_TAB, le code de référence est "A" et la séquence commence à l'adresse AD_Cod_A, le mot d'orientation suivant se situe à l'adresse AD_TT2. A cette adresse AD_TT2, se trouve le mot correspondant à la seconde séquence de code qui a pour
25 code de référence "B" et qui commence à l'adresse AD_Cod_B. Le mot suivant se situe à l'adresse AD_TT3 ; et ainsi de suite. La fin de table est atteinte lorsque la valeur d'adresse du mot d'orientation suivant est par exemple "NULL". A noter dans cet exemple : les séquences de code référencées "C" et "D" appartiennent à un même bloc. Ce bloc possède en
30 fait deux points d'entrée : AD_Cod_C, et AD_Cod_D.

On peut parfaitement concevoir qu'une ou plusieurs séquences de code ou parties de séquences soient écrites dans la zone de données de la mémoire programmable. Dans ce cas, il est conseillé de l'introduire dans une zone bien délimitée et connue en tant que telle par la partie applicative
35 du programme principal et, éventuellement par souci de sécurité d'en interdire la lecture par l'extérieur.

Le déroulement d'une séquence d'instructions d'orientation va être maintenant explicité avec l'aide de la figure 8. A l'étape 11, la séquence d'orientation teste l'indicateur de déroutement DI. S'il est inactif le programme principal continue en séquence. Si l'indicateur de déroutement

5 DI est actif, la séquence se poursuit à l'étape 12, par une mémorisation de l'adresse de retour dans le programme principal. Cette mémorisation est optionnelle car de toutes les façons, c'est la séquence qui décide de l'adresse de retour dans le programme principal ou ailleurs. Si plusieurs

10 séquences d'instructions d'orientation peuvent appeler une même séquence de code, il est intéressant de mémoriser l'adresse de retour spécifique à chaque séquence d'instructions d'orientation, de manière à pouvoir revenir à proximité de l'instruction qui a provoqué le déroutement, cette mémorisation peut s'effectuer dans un registre de travail AD_RET ou dans la pile, dans ce

15 dernier cas, la séquence de code mémorisée dans la mémoire programmable est considérée comme un sous-programme qui se termine par exemple par un code spécifique de "retour de sous-programme" ("RTS"). Puis, à l'étape 13 la séquence d'orientation se termine par un saut à la séquence de code commençant à l'adresse mémorisée dans le registre de la mémoire de travail AD_SAUT par une précédente séquence d'interrogation.

20 L'inscription de nouvelles séquences de codes et la mise à jour de la table d'orientation vont maintenant être explicitées.

Un ordre ECR_SEQ_E d'enregistrement de séquence de code est envoyé de l'extérieur par le dispositif d'entrée sortie (13) à la carte. Les données de cet ordre se décomposent en plusieurs champs comme le

25 montre la figure 9. L'ordre ECR_SEQ_E comporte :

- dans un premier champ le code de référence : "A", "B", ...,
- dans un second champ l'adresse de début de la séquence,
- dans un troisième champ facultatif la taille de la séquence, c'est-à-dire le nombre d'octets à écrire,
- 30 - et dans un quatrième champ facultatif, la séquence, c'est-à-dire les octets à écrire.

Les troisième et quatrième champs sont inutiles lorsque la séquence de code est déjà inscrite en mémoire non volatile (FRAM, EPROM, EEPROM ou ROM).

35 La figure 10 montre un organigramme illustrant le déroulement du programme PENR d'exécution de l'instruction ECR_SEQ_E d'enregistrement

d'une nouvelle séquence E. A l'étape 21, l'ordre est envoyé à la carte et les deux premiers champs de l'ordre sont analysés. Une fois le type d'instruction envoyé à la carte reconnu par celle-ci, le programme de gestion de l'enregistrement peut vérifier que la valeur dans AD_TAB est cohérente, si
5 non, la table d'orientation n'est pas opérationnelle, il n'est donc pas possible d'écrire une nouvelle séquence de code et l'ordre est interrompu.

A l'étape 211, si un drapeau ECA est inactif, le programme se poursuit à l'étape, sinon l'exécution du programme est interrompue.

A l'étape 22, le programme d'enregistrement contrôle si une
10 référence identique est déjà inscrite. Si la table contient déjà une référence de code identique à celle reçue, l'ordre est interrompu. Ce contrôle particulier à la présente invention offre un réel avantage par rapport à l'art antérieur décrit précédemment, car il permet de ne pas écrire par erreur deux séquences de codes avec la même référence. A l'étape 23, le
15 programme de gestion de l'enregistrement rend actif un drapeau en mémoire non volatile : ECA_en_cours (Ecriture de Code Additionnel en cours) indiquant qu'un enregistrement d'une nouvelle séquence est en cours. Puis, à l'étape 24, le programme PENR teste le nombre d'octets de codes à écrire. Si celui-ci est nul, aucune nouvelle séquence de code n'est à écrire et seule
20 une nouvelle référence de code doit être rajoutée dans la table, c'est le cas d'activation de code "dormant" en ROM ou le rajout d'un point d'entrée à une séquence de code déjà écrite en mémoire programmable.

Si le nombre d'octets de codes à écrire n'est pas nul la carte est mise en mode réception, à l'étape 25. Les octets sont reçus, éventuellement
25 déchiffrée et/ou signée si c'est le mode choisi, puis écrits "dans la foulée" à l'adresse de début de la séquence définie dans le deuxième champ de l'ordre ECR_SEQ_E. Dans tous les cas, l'adresse d'écriture doit être vierge, sinon l'ordre est interrompu en laissant le drapeau ECA_en_cours actif, ce qui a pour effet de bloquer définitivement la carte, à l'exception du
30 mécanisme de régénération décrit plus loin. Une fois tous les octets de codes reçus et la séquence correspondante écrite, le programme (PENR) vérifie à l'étape (26) si l'écriture s'est correctement déroulée. Un moyen pour cela consiste à élaborer un calcul de somme de contrôle (Checksum) ou une signature à partir de la valeur des octets écrits et leurs adresses. A
35 l'étape 27, le programme (PENR) teste la validité du code enregistré. Si

l'écriture ne s'est pas bien déroulée, l'ordre est interrompu en laissant le drapeau ECA_en_cours actif.

Si l'écriture s'est bien déroulée ou, si il n'y a pas de séquence de code à écrire, le programme PENR met à jour, à l'étape 28, la table d'orientation, en écrivant le nouveau mot d'orientation. Dans le schéma de table décrit par la figure 6, le nouveau mot d'orientation est écrit immédiatement après le dernier mot. Le programme PENR détermine le dernier élément en parcourant la table depuis l'adresse AD_TAB, en progressant suivant le sens indiqué dans SENS. La somme de contrôle (Checksum) écrite dans le troisième champ de mot d'orientation peut être indifféremment envoyée par l'extérieur ou calculée par la carte. Cette somme de contrôle (Checksum) peut être de type simple somme ou OU exclusif (XOR) des octets ou de type cryptographique, elle peut porter sur le mot de la table ou peut inclure la séquence de code rattachée à ce mot. Dans le schéma de table décrit par la figure 7, le nouveau mot est écrit dans un emplacement vierge de la zone des séquences et mots d'orientation de la table. Le programme PENR doit aussi modifier le troisième champ de l'avant dernier champ de la table pour qu'il contienne l'adresse AD_Cod_E de cette nouvelle séquence de code "E".

Lorsque toutes ces opérations se sont bien déroulées, à l'étape 29 du programme PENR, le drapeau ECA_en_cours est rendu inactif, ce qui a pour principal résultat de valider la table d'orientation et les séquences. Si l'ordre ne s'est pas correctement déroulé, un message d'erreur envoyé par le programme PENR à l'extérieur renseigne l'utilisateur sur le type d'erreur.

Il faut signaler le cas particulier pour inscrire le contenu de AD_TAB et éventuellement de SENS. En effet, la description précédente n'est valable que pour une table d'orientation déjà commencée. On rappelle que AD_TAB contient l'adresse de début de la table d'orientation et que si celle-ci est organisée comme le décrit la figure 6 le champ SENS indique le sens de parcours dans cette table.

En utilisant le format des données décrits à la figure 9, on peut dédier une référence de code particulière pour inscrire le contenu de AD_TAB et éventuellement de SENS. En se référant toujours à la figure 9, pour cette référence particulière, le champ AD_Cod_A ne serait pas significatif, tandis que le champ contenant le code de la séquence contiendrait le contenu à mettre dans AD_TAB et éventuellement SENS.

Une autre façon de faire, et qui serait valable pour l'inscription de toute donnée (séquences de code, éléments de la table d'orientation...) serait d'utiliser le format suivant :

Adresse de début d'écriture	Nombre d'octets	Données
-----------------------------	-----------------	---------

- 5 Cette méthode offre l'avantage essentiel de permettre d'écrire n'importe quelle donnée sous la responsabilité du concepteur de code additionnel.

- 10 Pour identifier un système d'exploitation ou certaines de ses particularités telle que l'utilisation d'un code additionnel il suffit de faire envoyer par la carte des octets d'identification, lorsque l'on met la carte sous tension. La mise sous tension est une opération indispensable avant de pouvoir dialoguer avec une carte. De manière interne, le système d'exploitation peut bâtir sa réponse à la remise à zéro en analysant le pointeur AD_SAUT de la table d'orientation et la table elle-même.

- 15 Une instruction d'invalidation INS_INA_C d'une séquence de code envoyée de l'extérieur de la carte a pour but, par son programme (PINA) de traitement, d'inactiver un déroutement correspondant à un certain code de référence, par exemple C. Une première façon de faire consiste à effacer toutes les données relatives à ce code : l'élément dans la table d'orientation et toute la séquence correspondante. Puis à mettre à jour la table d'orientation. Une seconde façon de faire, plus rapide consiste à changer dans la table d'orientation le code de référence correspondant à la séquence de code à invalider par une autre valeur : "NULL" par exemple, ce code n'existant plus, aucune instruction d'interrogation ne pourrait mettre à jour la mémoire de travail AD_SAUT pour exécuter cette séquence. Enfin une dernière façon consiste à remplacer la valeur d'adresse en table d'orientation par une valeur d'adresse impossible : "NULL" par exemple ; ou encore diriger l'appelant vers une séquence de code simple qui "redonne la main". Cette dernière manière de faire est parfaitement adaptée aux séquences de code inscrite en ROM, ce code "dormant" peut ainsi très facilement être activé et désactivé.

- 35 Une inscription de régénération INS_REG provenant de l'extérieur de la carte a pour but, par son programme de traitement PREG, d'effacer une ou plusieurs séquences de codes et/ou éléments de table ou, d'une manière générale, une suite de données. Plusieurs méthodes sont possibles : les adresses de début et de fin d'un espace mémoire sont envoyées à la

carte et tous les octets de cet espace sont effacés. Cette méthode un peu "brutale" doit être utilisée avec précaution pour ne pas rendre inexploitable des séquences qui ne doivent pas être effacées. Une autre méthode consiste à préciser à la carte les codes de références à effacer ainsi que, si
5 elles existent, les séquences de codes en mémoire programmable correspondante, à la suite d'un tel ordre, le programme doit déterminer l'adresse des éléments de table correspondant puis, effacer ces éléments et déterminer les adresses de début et de fin des séquences de codes et enfin effacer le contenu. La méthode de recherche de ces adresses puis de mise
10 à jour de la table dépend fortement de la structure de la table d'orientation, voir figures 6 et 7. La méthode ne pose pas de problème pour l'homme de métier et n'a donc pas besoin d'être explicitée.

Un dernier ordre REC_ESP_VRG qui peut faciliter la mise à jour de la table et des séquences de code, est "Recherche Espace Vierge". Envoyé
15 à la carte, sans donnée particulière, cet ordre permet de faire envoyer par le programme PREV de traitement de cet ordre à l'extérieur les emplacements vierges de la mémoire programmable (11) avec les adresses de début et de fin de chacun de ces emplacements. L'utilisateur désireux de rajouter des séquences de codes peut, à la suite de cet ordre, vérifier si la place
20 disponible est suffisante et, définir la séquence de code en fonction de son adresse d'implantation.

Dans une variante de réalisation permettant le contrôle de l'intégrité de la séquence de code additionnel, juste avant d'effectuer le saut dans la séquence d'instruction d'orientation, il est alors nécessaire de prévoir deux
25 champs supplémentaires dans chaque élément de la table d'orientation.

Un premier champ supplémentaire, qui indique la taille de la séquence de code additionnel rattachée à l'élément et, un deuxième champ supplémentaire contenant la somme de contrôle (Checksum) initiale correct.

Si la somme de contrôle (Checksum) calculée diffère de celle
30 initiale, le système d'exploitation arrête l'exécution en signalant par un mot d'état approprié. Cette méthode offre l'avantage qu'un contrôle est effectué avant chaque exécution de la nouvelle séquence de code mais pénalise le temps d'exécution de l'ordre concerné ou peut se limiter à ce contrôle lors uniquement de la première exécution.

35 La deuxième méthode consiste à disposer d'un ordre dédié qui effectue un calcul de la somme de contrôle (Checksum) d'une zone de

mémoire programmable et compare la valeur calculée avec celle fournie ou inscrite préalablement dans le champ CHK_X. Ainsi le gestionnaire du système peut activer périodiquement cet ordre pour s'assurer que tout est ok.

5 Un système d'exploitation qui supporte le mécanisme d'enregistrement et d'exécution de code additionnel peut être bridé afin de réduire ses capacités par des verrous d'interdiction en mémoire programmable tels que :

- CAI : Code Additionnel Interdit qui serait testé par le système
- 10 d'exploitation avant tout enregistrement,
- RCI : Régénère Carte Interdit qui serait testé par le système d'exploitation avant toute régénération.

Ces verrous s'apparentent quant à leur gestion aux verrous (locks) de gestion des phases de vie des systèmes d'exploitation des cartes.

15 Une utilisation de cette possibilité consiste à charger du code additionnel en EEPROM (11) puis de positionner les verrous d'interdictions RCI et CAI, pour figer le code rajouté.

Bien sûr, l'invention n'est nullement limitée à l'exemple de réalisation précédemment décrit mais, au contraire, comprend tous les moyens

20 constituants des équivalents techniques des moyens précisément décrits ainsi que leurs combinaisons, si celles-ci sont exécutées dans l'esprit de l'invention et mises en oeuvre dans le cadre des revendications suivantes.



REVENDICATIONS

1. Dispositif d'exécution de séquences de code dans un support comprenant un circuit intégré (10) capable d'exécuter des séquences de code et d'une part une première mémoire (12) contenant un programme principal et éventuellement d'autres séquences de codes exécutables par le circuit intégré et d'autre part une seconde mémoire (11) programmable non volatile contenant éventuellement des séquences de codes exécutables par le circuit intégré et une troisième mémoire de travail (14), caractérisé en ce qu'une table d'orientation contenue dans la seconde mémoire (11) contient au moins un champ contenant une donnée de référence de code :

des premiers moyens (INS_INT) permettent :

- de vérifier la présence d'une référence de code,
- de mémoriser en mémoire de travail la donnée d'adresse associée à la référence du code et de positionner un indicateur de déroutement DI et,

des seconds moyens (INS_ORT) permettent :

- de tester l'indicateur de déroutement DI et,
- d'effectuer le saut à l'adresse pointée par le contenu de la mémoire de travail AD_SAUT.

2. Dispositif selon la revendication 1, caractérisé en ce que la donnée d'adresse associée d'une séquence de code contenue dans l'une des trois mémoires à la référence de code est calculée à partir de la référence de code.

3. Dispositif selon la revendication 1, caractérisé en ce qu'un deuxième champ contient la donnée d'adresse d'une séquence de code contenue dans l'une des trois mémoires.

4. Dispositif selon la revendication 1, caractérisé en ce qu'il comporte dans une zone éventuellement protégée de la mémoire programmable (11) une information (AD_TAB) définissant la première adresse et éventuellement le sens de recherche (SENS) pour les premiers moyens de vérification.

5. Dispositif selon la revendication 1, caractérisé en ce que la table d'orientation comporte un champ supplémentaire (AD_TT) indiquant l'adresse suivante dans la table où se trouvent les informations correspondant à la référence de code suivante.



6. Dispositif selon la revendication 1, caractérisé en ce que la table d'orientation comporte un champ supplémentaire (CHK) permettant de s'assurer de l'intégrité des champs précédents et/ou de séquence de code rattachée par comparaison avec le résultat fourni par un moyen de calcul d'une somme de contrôle.

7. Dispositif selon la revendication 1, caractérisé en ce que les seconds moyens comportent des moyens de mémoriser en mémoire de travail (AD_RET) une adresse de retour après un saut lorsque plusieurs seconds moyens sont disponibles pour effectuer un saut à destination de la même adresse de séquence.

8. Dispositif selon la revendication 1, caractérisé en ce qu'il comporte des moyens CAI d'empêcher l'enregistrement d'un code additionnel, et des moyens RCI d'empêcher une régénération de la carte.

9. Procédé d'enregistrement d'un nouveau code dans un dispositif selon la revendication 4, caractérisé en ce qu'il consiste :

- à recevoir un ordre d'enregistrement ;
- à vérifier la présence d'un premier champ constituant la référence de code A, B, C... et d'un second champ représentant l'adresse associée à la référence de code AD_Cod_A, AD_Cod_B, AD_Cod_C...
- à vérifier qu'un drapeau ECA n'est pas actif ;
- à vérifier que la valeur de la première adresse AD_TAB est cohérente ;
- à contrôler que la référence fournie par le premier champ n'existe pas dans la table ;
- à positionner le drapeau ECA dans la zone de mémoire programmable protégée à l'état actif ;
- à recevoir et écrire les informations correspondant au nouveau code ;
- à vérifier que l'opération s'est déroulée correctement et mettre à jour la table d'orientation ;
- à positionner le drapeau ECA à l'état inactif.

10. Procédé selon la revendication 9, caractérisé en ce que l'opération de mise à jour de la table comporte une écriture du champ d'adresse suivante dans la table d'orientation à l'adresse correspondant au mode d'orientation du code précédent le code écrit.

PROCESS AND DEVICE ALLOWING A FIXED PROGRAM TO BE CHANGED

The present invention relates to computer programs, particularly those incorporated into microcircuit supports, and more generally portable objects equipped with integrated circuits comprising at least one central processing unit, one read-only program memory, one nonvolatile programmable memory and one working memory. This nonvolatile memory is capable of storing data and code, and the central processing unit can have this code executed by the central processing unit.

At the present time, it is known to have two very different types of programs coexisting in the same object, one of which is written into the integrated circuit upon fabrication and cannot be modified, the other of which is input from the outside during normal utilization of the object, as necessary.

At this time, microcircuit supports are technically capable of meeting numerous needs. However, the production of a "mask," that is, the program in ROM memory, is still very expensive, and this cost discourages many "small" customers who would be interested in purchasing a few thousand cards. One solution would consist of using an existing mask and adding the functions requested by the client into the programmable memory.

The capability to input and execute additional code in programmable memory offers the advantage of being able to easily add new functions to an existing program or to adapt this program to specific needs.

French patent 2 655 170 describes a portable electronic device capable of writing code sequences into a nonvolatile memory. The nonvolatile programmable memory comprises a specific part configured for the purpose of containing code sequences, which part is itself divided into an indicator area, a code sequence data area, and a block check character area. A test instruction is implemented in the program in read-only memory in order to read and test the indicator area, and if possible to execute the jump. Therefore, there must be a test instruction and an allocated storage area in programmable memory corresponding to

each jump.

Also known, from French patent 2 667 417, is a microprocessor card designed to receive multiple programs into programmable memory wherein so-called filtering instructions are associated with each of the reserved locations in programmable memory so as to constitute a means for storing the address of the code of an instruction subroutine accessible by executing the filtering instruction stored in the read-only memory of the card. Each filtering instruction must know the address of the storage area containing the address of the associated code in the programmable memory, which imposes a hierarchy of access that is permanently fixed.

The two solutions described above have the following drawbacks.

One skilled in the art wishing to implement several jump instructions must multiply the jump instructions in the program in read-only memory and the specific areas in programmable memory. If a large number of instructions of this type must be installed into the main program, the total space they occupy in read-only memory becomes substantial, to the detriment of the program space. Moreover, in the nonvolatile programmable memory, even if the subroutine is not written, its space is reserved as is that of the indicators, which takes up a substantial amount of space unnecessarily. Finally, once written, the subroutine cannot be modified. In short, this solution is only suitable for a limited number of jump instructions and a few subroutines, and does not in any way solve the problem of a substantial number of jumps and subroutines.

The subject of the present invention is a process and a device allowing the functionalities of a chip card to be changed without the drawbacks of the patents cited, particularly by allowing a larger number of calls to subroutines. In effect, the device according to the invention offers the advantage of being easy to implement in a program, of using very little code memory, of minimizing the memory size used, of being dynamic and allowing

all kinds of modifications, or of optimizing the execution time.

For this reason, the device for executing code sequences in a support comprising an integrated circuit capable of executing code sequences, as well as a first memory containing a main program and possibly other code sequences executable by the integrated circuit, a second nonvolatile programmable memory possibly containing code sequences executable by the integrated circuit, and a third working memory, is characterized in that an orientation table contained in the second memory contains at least one field containing a code reference data element of first means (INS_INT) which make it possible

- to verify the presence of a code reference,
- to store in working memory the address data element associated with the reference of the code and to set a trap indicator DI, and

of second means (INS_ORI) which make it possible

- to test the trap indicator DI, and
- to execute the jump to the address indicated by the contents of the working memory.

According to another characteristic, the address data element of a code sequence contained in one of the three memories associated with the code reference is calculated from the code reference.

According to another characteristic, a second field contains the address data element of a code sequence contained in one of the three memories.

According to another characteristic, the device comprises, in a possibly protected area of the programmable memory, a data element defining the first address and possibly the search direction for the first verification means.

According to another characteristic, the orientation table comprises an additional field indicating the next address in the table where the information corresponding to the next code reference is located.

According to another characteristic, the orientation table

comprises an additional field which makes it possible to verify the integrity of the preceding fields and/or of the attached code sequence in comparison with the result supplied by a checksum calculating means.

According to another characteristic, the second means comprise means for storing in working memory an address to return to after a jump when a plurality of second means are available to execute a jump directed to the same sequence address.

According to another characteristic, the device comprises means CAI for preventing the writing of additional code, and means RCI for preventing a regeneration of the card.

Another object is to offer a process for recording new functionalities.

This object is achieved due to the fact that the process for writing a new code in a device according to any of the preceding claims, is characterized in that it is comprised of

- receiving a write command;
- verifying the presence of a first field constituting the code reference A, B, C, etc., and a second field representing the address AD_Cod_A, AD_Cod_B, AD_Cod_C, etc., associated with the code reference;
- verifying that the value of the first address constituting the orientation table address AD_TAB is consistent;
- verifying that no flag ECA is active;
- checking that the reference supplied by the first field does not exist in the table;
- setting the flag ECA in the protected area of programmable memory to the active state;
- receiving and writing the information corresponding to the new code;
- verifying that the operation has been executed correctly and updating the orientation table;
- setting the flag ECA to the inactive state.

According to another characteristic, the operation for updating the table comprises a writing of the next address field

in the orientation table with the address corresponding to the orientation word of the code which precedes the code written.

Other characteristics and advantages of the present invention will appear more clearly with a reading of the following description of an exemplary embodiment of the device and implementation of the process, illustrated with the aid of the appended figures, in which:

- Fig. 1 represents a partition of the programmable memory of the card using a variant of a process for managing the orientation table;
- Fig. 2 represents an example of a partition of the read-only memory of the card;
- Fig. 3 represents, in a main program, an example of an interrogation instruction sequence executed during the running of the main program and a subsequent orientation instruction sequence;
- Fig. 4 represents an example of a sequence of possible interrogation and orientation instructions;
- Fig. 5 represents an example of an algorithm for the execution of a code sequence constituting the interrogation instruction;
- Fig. 6 represents an exemplary configuration of the partition of the programmable memory incorporating the orientation table and the code sequence areas;
- Fig. 7 represents a second exemplary embodiment of the partition of the memory incorporating the orientation table and the code sequence areas;
- Fig. 8 represents an example of a flow chart of the execution of a code sequence constituting an orientation instruction;
- Fig. 9 represents an example of the fields constituting a write instruction of a sequence;
- Fig. 10 represents the flow chart of the execution of a recording sequence triggered by a write instruction;
- Fig. 11 represents the elements constituting a

microcircuit support required for implementing the device and the process of the invention.

The invention will be more clearly understood through the description of the exemplary utilization below, in a microprocessor card according to Fig. 11.

A microprocessor card has a Central Processing Unit (10) of the microprocessor or integrated circuit type capable of executing instruction sequences, a working memory RAM (14), a nonvolatile memory ROM (12) containing the main program, a nonvolatile memory (11) EPROM, EEPROM or FRAM programmable and executable by the CPU containing data, and input-output devices (13) which allow dialoguing with a reader through a link (15) with or without contact. The program is burned into ROM during the fabrication of the component and cannot be subsequently modified.

The programmable memory (11) is divided into several parts, as shown in Fig. 1. A first part (110) marked system area contains system information that is not readable from the outside, making it possible to check the validity (authenticity and/or integrity) and the location of the information or data group contained in the rest of the memory (pointers, etc.). A second part (111) marked data area is accessible from the outside and is used to store user data. A third part (112) marked orientation table contains the orientation table, the latter being formed of identical elements, each element containing several fields, including two main fields: the first is a code reference data element "A", "B", etc., and the second is the start address AD_Cod_A, AD_Cod_B, etc., of the corresponding code sequence stored in a fourth part (113) of the memory. The fourth area marked code sequence area contains the code sequences that can be called by the main program. It must be noted that in a variant of the invention the orientation table or the code sequence can be loaded into working memory (RAM) instead of EEPROM.

The start address of the orientation table, called AD_TAB,

is stored in the system area; a second data element marked SENS determines the direction in which the pointer reads this table, either in the direction of the ascending addresses, or in the direction of the descending addresses. Thus, the addresses of the values AD_TAB and SENS can be the only data relative to the invention that the program in ROM knows. It can also be useful to store the size in octets or the maximum number of elements (Ta Ta) of the orientation table.

The following is an example which illustrates the advantage of the trapping process according to the invention. The main program can contain an algorithm such as D.E.S. (Data Encryption Standard) capable of carrying out the encryption/decryption of information or certain authentication functions. It can be advantageous to replace this algorithm with another one, or to completely change its implementation, for example in order to make it faster. The code of the algorithm constitutes a block in the ROM memory of the Operating System; this code cannot be modified, and it can be called from various points in the main program.

The example will be more clearly understood by examining Fig. 2. The ROM memory, which among other things contains the main program, is divided into three parts. The first part implements the initialization of the card after powering up: the testing of the working memory, the reading of the status of the card, and the sending of the octets of the response to the RESET. The second part contains the application program which executes the various commands requested by the terminal (the first code part is always executed before the second). The first and second parts constitute the main program. The third part contains the "dormant" code whose role is explained below.

The algorithm called ALGO_1 whose operation is intended to be modified is contained in the application part. In the example, this algorithm is called from five different points in the code. The new algorithm has been pre-loaded into the code sequence area in the programmable memory and the orientation table has been

updated; the execution address of its code sequence is AD_Cod_A.

At the power up, the main program executes the initialization part, wherein the first jump to ALGO_1 marked "APPEL_0" is found; no trap is provided for, so this jump is not conditioned by any trap instruction. Also found in this part is the interrogation instruction sequence INS_INT_A, which the programmer has been careful to implement in this part so as to be sure to execute it at least once before the execution of the program of the application part. In the example, the code reference associated with this interrogation instruction sequence is "A", and it is written into the main program. During a first period, the interrogation instruction searches in the system area for the orientation table address AD_TAB. Then, starting at this address and proceeding in the direction of the ascending or descending addresses depending on the value of "SENS", each element of the table is read. The value of the first fields corresponding to the code references is compared to the value "A". If the value A is found in a first field of the orientation table, the value of the second corresponding field (AD_Cod_A) is read and stored in the register AD_SAUT in working memory (RAM) and a trap indicator (DI) in working memory is set to the active position.

The subroutine ALGO_1 is called five times by APPEL_0, APPEL_1, ..., APPEL_4, in the main program when DI is not active. For APPEL_1, APPEL_2, and APPEL_3, the trap to another program ALGO_2, which can be in read-only memory or in programmable memory, is executed when the trap indicator DI is active, since the programmer has been careful to implement in the code of the main program an orientation instruction INS_ORT_1, INS_ORT_2 and INS_ORT_3 before each jump APPEL_1, APPEL_2, APPEL_3 effective in ALGO_1. As shown in Fig. 3, the orientation instruction is comprised, first of all, of testing whether the trap indicator DI is active. If the indicator is inactive, the main program follows its normal course and executes the algorithm in ROM, for example by executing the jump APPEL_1 to ALGO_1. If the trap indicator DI

is active, the orientation instruction traps the program to the address indicated by the contents of the register AD_SAUT. The value AD_Cod_A previously stored in the register AD_SAUT is loaded into the program counter of the Central Processing Unit, and the code sequence at the loaded address is executed. On the other hand, if desired by the programmer, the subroutine ALGO_1 can be called by APPEL_4 without the possibility of a trap, since no orientation instruction is associated with it.

It must be noted that the expressions interrogation instruction and orientation instruction are actually macroinstructions constituted by a set of elementary instructions executable by the central processing unit.

The same orientation instruction can be executed an unlimited number of times, as long as the indicator remains active; the jump will always go to the address stored in AD_SAUT. Likewise, as shown in Fig. 2, a plurality of orientation instructions INS_OR1_1, INS_OR1_2, INS_OR1_3 with the same code reference "A" can be implemented in the main program; their execution will have the same effect. In this case, it is appropriate to examine the return address to the main program very carefully; exemplary embodiments of return instructions will be explained below.

As a general rule, the programmer make sure that at least one interrogation instruction sequence is executed before each orientation instruction. If the code sequence must not be run more than once, an instruction for clearing the trap indicator DI, in order to render the trap indicator DI inactive, is executed by the code sequence. Thus, once there has been a return to the main program, no further trap can be carried out during the subsequent orientation instructions.

In light of this example, the first advantage of separating the interrogation and orientation instruction sequences is apparent. The interrogation instruction sequence, which is more elaborate and hence more costly in terms of code, is loaded into the program only once. The orientation instruction sequences,

which are very short and not very costly in terms of memory space, are loaded as necessary, resulting in an increase of space in the program memory.

The trap indicator DI can effectively be replaced by a particular value of the register AD_SAUT. If the register AD_SAUT contains the value "NULL", which is not a significant address value of the nonvolatile memory for a given component, it means that there is no trap of the main program to be carried out. Likewise, if the content of the memory AD_TAB is a value that is not consistent with the initial value of the orientation table (for example, "NULL"), it may be presumed that no sequence is available and that consequently no trap is intended.

As shown in Fig. 4, a plurality of interrogation instructions (INS_INT_A, INS_INT_B) can be input into the main program. Each of these instructions has its own reference code written into the program memory, and searches for the presence of this code among the initial values of the orientation table in program memory. If the value of this code is in fact in the table, the address of the associated code sequence is entered into the working register AD_SAUT. Thus, the execution of an interrogation instruction clears and replaces the value of the register AD_SAUT updated by the previous interrogation instruction.

If a previous interrogation instruction has updated the register AD_SAUT with the trap address, and if a second interrogation instruction with a different reference code is executed without finding its reference code in the orientation table, it is sufficient, and wise, to clear the preceding value of AD_SAUT, thus cancelling all traps until the next interrogation instruction. However, it is not outside the scope of the invention to retain the previous value, thus maintaining the trap.

A second advantage of using code references "A", "B", etc., written in an orientation table is apparent: this table contains only necessary data, referring only to the desired traps

effective in the main program. Thus, at any time, its size is optimized, resulting in an increase in memory space.

Advantageously, it is possible to store code sequences in the main program that are not called by it. With the aid of the device according to the invention, it is possible to associate with each of these sequences a code reference that is stored in the orientation table with the start address of the sequence. Thus, it is possible to activate a "dormant" code in the program memory of the portable device.

Advantageously, it is also possible through the orientation instructions to define several entry points to the same code sequence, and to associate them with different reference codes. Likewise, a plurality of sequences "A", "B" and "C", for example, can be grouped into a single sequence written all at once into programmable memory with the reference code "A" and the entry point AD_Cod_A; the other two reference codes and entry points are written subsequently. An example of an orientation table which allows this operation is presented below (Fig. 7).

The execution of the interrogation instructions will now be explained with the aid of Fig. 5. An input parameter of this instruction is the value marked "X" of the reference code corresponding to the code sequence which will be executed during a subsequent orientation instruction; in the example, this value is "X". In step 1, the main program, by executing the interrogation instruction sequence, reads in the system area at the address @, which is known by the program in ROM, the content AD_TAB which is the start address of the orientation table. In step 2, the interrogation instruction sequence tests the word read. If the word read is not significant, the orientation table does not exist, and no trapping of code is possible. The interrogation instruction sequence is terminated and the trap indicator DI is inactive (step 10). If the word read is significant, in step 3 the interrogation instruction sequence updates a pointer, such as for example AD_SAUT, in working memory with the value AD_TAB read. Then in step 4 the interrogation

sequence reads an element of the table indicated by this pointer. In step 5 of the interrogation sequence, the value of the field corresponding to the code reference is compared to the value of the reference code "X" provided as an input parameter. If the two values are identical, the interrogation sequence, in step 8, sets the trap indicator DI to the active state. With the aid of the pointer, the interrogation sequence, in step 9, reads the address value "AD_Cod_X" located immediately after the reference code in the orientation table. This value is stored in the register AD_SAUT in working memory RAM. If the two code reference values are not identical, in step 6 the pointer is updated in order to indicate the next reference code in the orientation table. This update is executed, taking into account the size of each element of the table; if for example the table is composed of words of 32 bits, the value of the pointer, which addresses octets (8 bits), is increased by 4. The interrogation sequence, in step 7, tests whether the word of the programmable memory addressed by the pointer is still in the orientation table. This test depends on the organization of the table, several examples of which will be given below. If the word indicated no longer belongs in the table, the interrogation instruction sequence is terminated and the trap indicator DI is set to the inactive state by step 10. If the word indicated is still in the orientation table, the interrogation sequence loops back to step 4.

At the end of the interrogation instruction sequence, the trap indicator DI is always set; if it is active, the value in AD_SAUT is significant. Advantageously, the interrogation instruction sequence can be written by the programmer, who implements the main program like a subroutine having as input parameters the reference code, for example X, to be searched for in the table and possibly a return address, and as an output parameter a trap indicator DI specifying whether the trap must actually be implemented. It is possible during initialization to execute a reading of the orientation table and to store in working memory all the addresses corresponding to the existing

reference codes, which avoids having to systematically scan the table at each interrogation instruction, and avoids having to search in vain for a code that does not exist.

Fig. 6 shows an example of an orientation table. This table is composed of orientation words of 32 bits, for example; the first field, for example of one octet, contains the reference code, the second field, for example of two subsequent octets, contains the start address of the code sequence in nonvolatile memory, and the last field, for example of one octet, is optional and can contain a checksum code (CHK_A) which makes it possible to check the integrity of the value of the first two fields and/or of the attached code sequence. In this orientation table, the elements follow one another in contiguous fashion. The end of table is reached when the next word is blank. The value SENS read in the system area indicates the direction of progress of the table pointer, by ascending or descending addresses. In Fig. 6, the direction of the ascending addresses has been chosen.

The format of this table is advantageous when it contains a large number of elements. In effect, the reference codes of these elements can be sorted in a certain order; the organization of these elements in the table can therefore be carried out according to ascending references "A", "B", "C", "D", "E", etc., or descending references "Z", "Y", "X", "W", etc. The size in octets of the table is predetermined. It is either stored in an area (Ta Ta in Fig. 1) of the system area, or determined by pre-reading the table during initialization. During the search for a certain reference code, it is possible to apply sorting algorithms; for example, the pointer is initialized with the address of the element located in the center of the table by adding to the value contained in AD_TAB half the previously determined number of octets in the table, then the reference code read by the pointer is compared to the one to be searched for, and in accordance with the result, the pointer is moved forward or backward, thus pointing to the center of the area remaining to be scanned. This device makes it possible to save time in the

search for a code reference in a table. Thus, if the table has n ordered elements, the standard search mode of passing from one element to the next results in the reading, on average, of $n/2$ elements in order to find the right one. A search like the one indicated above makes it possible to search by reading $(\log n)$ elements, which is an advantage when the value n is large.

If, after having scanned the entire table, no value has been read in the table which has the reference "A" sought, no trap of the main program to the code sequence "A" will be executed; in this case, a trap indicator DI in working memory is set to the inactive position.

This format requires the programmer, when writing a code sequence, to arrange the size of the orientation table so as to leave enough memory space after the value of AD_TAB for adding to this table, and possibly for writing other code sequences.

A variant of embodiment of the orientation table can contain only the code reference, and in this case the interrogation instruction program will calculate the jump address by applying an algorithm having the code reference as an input parameter.

Fig. 7 shows another example of an orientation table which avoids the programmer's having to provide in a memory Ta Ta the information corresponding to its size or to carry out a pre-reading of the table in order to determine its size. This table contains orientation words having a field in addition to the three preceding fields: reference code value, start address of the code sequences, and checksum. The additional field, which for example occupies the fourth and fifth octets, contains the address value of the next element in the orientation table. The table is read in the following way: the orientation word corresponding to the first code sequence is located at the address AD_TAB, the reference code is "A", and the sequence starts at the address AD_Cod_A; the next orientation word is located at the address AD_TT2. At this address AD-TT2 is the word corresponding to the second code sequence, which has the reference code "B" and which starts at the address AD_Cod_B. The

next word is located at the address AD_TT3, and so on. The end of table is reached when the address value of the next orientation word is for example "NULL". It must be noted that in this example the code sequences labelled "C" and "D" belong to the same block. This block actually has two entry points: AD_Cod_C and AD_Cod_D.

It is entirely possible for one or more code sequences or parts of sequences to be written in the data area of the programmable memory. In this case, it is advisable to input it into an area that is clearly delimited and known as such by the application part of the main program, and possibly, for security reasons, to prevent its being read from the outside.

The execution of an orientation instruction sequence will now be explained with the aid of Fig. 8. In step 11, the orientation sequence tests the trap indicator DI. If it is inactive, the main program continues in sequence. If the trap indicator DI is active, the sequence proceeds to step 12, by storing the return address in the main program. This storage is optional since in all cases, it is the sequence which chooses the return address in the main program or elsewhere. If several orientation instruction sequences can call the same code sequence, it is advantageous to store the return address specific to each orientation instruction sequence, so as to be able to return to the vicinity of the instruction which caused the trap, which storage can take place in a working register AD_RET or in the stack; in the latter case, the code sequence stored in the programmable memory is considered to be a subroutine which ends with a specific "return to subroutine" ("RTS") code. Then in step 13 the orientation sequence ends with a jump to the code sequence starting at the address stored in the register of the working memory AD_SAUT by a previous interrogation sequence.

The writing of new code sequences and the updating of the orientation table will now be explained.

A code sequence write command ECR_SEQ_E is sent from the outside through the input-output device (13) to the card. The data of this command is broken down into a plurality of fields,

as shown in Fig. 9. The command ECR_SEQ_E comprises:

- in a first field, the reference code: "A", "B", etc.,
- in a second field, the start address of the sequence,
- in a third optional field, the size of the sequence, that is, the number of octets to be written,
- and in a fourth optional field, the sequence, that is the octets to be written.

The third and fourth fields are unnecessary when the code sequence has already been written in nonvolatile memory (FRAM, EPROM, EEPROM or ROM).

Fig. 10 shows a flow chart illustrating the running of the program PENR which executes the instruction ECR_SEQ_E for writing a new sequence E. In step 21, the command is sent to the card and the first two fields of the command are analyzed. Once the type of instruction sent to the card is recognized by the latter, the write control program can verify whether the value in AD_TAB is consistent; if it is not, the orientation table is not operational, so it is not possible to write a new code sequence and the command is interrupted.

In step 211, if a flag ECA is inactive, the program proceeds to the next step; if not, the execution of the program is interrupted.

In step 22, the write control program checks to see whether an identical reference has already been written. If the table already contains a code reference identical to the one received, the command is interrupted. This check, which is unique to the present invention, offers a real advantage over the prior art described above, since it makes it impossible to write two code sequences with the same reference by mistake. In step 23, the write control program renders the flag ECA_in_progress (Writing of Additional Code in progress) active in nonvolatile memory, thus indicating that the writing of a new sequence is in progress. Next, in step 24, the program PENR tests the number of octets of code to be written. If this number is null, there is no new code sequence to be written and only a new code reference is

to be added to the table; this is the case when activating a "dormant" code in ROM or when adding an entry point to a code sequence already written in programmable memory.

If the number of octets of code to be written is not null, the card is placed in the receive mode, in step 25. The octets are received, possibly decrypted and/or signed if that mode has been chosen, then written "automatically" at the start address of the sequence defined in the second field of the command `ECR_SEQ_E`. In all cases, the write address must be blank, otherwise the command is interrupted, leaving the flag `ECA_in_progress` active, which has the effect of definitively blocking the card, except for the regeneration mechanism defined below. Once all the octets of code have been received and the corresponding sequence has been written, the program (PENR) verifies in step 26 whether the write has been executed correctly. One way to do this is to generate a checksum or a signature from the value of the octets written and their addresses. In step 27, the program (PENR) tests the validity of the code written. If the write has not been executed correctly, the command is interrupted, leaving the flag `ECA_in_progress` active.

If the write has been executed correctly, or if there is no code sequence to be written, in step 28 the program PENR updates the orientation table by writing the new orientation word. In the schema of the table described by Fig. 6, the new orientation word is written immediately after the last word. The program PENR determines the last element by scanning the table, starting at the address `AD_TAB` and proceeding in the direction indicated in `SENS`. The checksum written in the third field of the orientation word can either be sent from the outside or calculated by the card. This checksum can be a simple sum or exclusive OR (XOR) of the octets or can be cryptographic, and it can be a function of the word of the table or can include the code sequence attached to this word. In the schema of the table described by Fig. 7, the new word is written in a blank space in the area of the sequences

and orientation words of the table. The program PENR must also modify the third field of the second-to-last field of the table so that it contains the address AD_Cod_E of this new code sequence "E".

When all these operations have been executed correctly, in step 29 of the program PENR the flag ECA_in _progress is rendered inactive, the chief result of which is to validate the orientation table and the sequences. If the command has not been correctly executed, an error message sent by the program PENR to the outside informs the user of the type of error.

It is necessary to indicate the specific case in order to write the contents of AD_TAB and possibly of SENS. In effect, the preceding description is only valid for an orientation table that is already started. It will be recalled that AD_TAB contains the start address of the orientation table and that if the latter is organized as described in Fig. 6, the field SENS indicates the direction in which this table is scanned.

Using the data format described in Fig. 9, it is possible to dedicate a specific code reference for writing the contents of AD_TAB and possibly of SENS. As seen in Fig. 9, for this particular reference, the field AD_Cod_A will not be significant, while the field containing the code of the sequence will contain the contents to be loaded into AD_TAB and possibly SENS.

Another way to do this, which is valid for writing any data element (code sequences, elements of the orientation table, etc.) is to use the following format:

Start address of write	Number of octets	Data
------------------------	------------------	------

This method offers the main advantage of allowing any data element to be written by the developer of the additional code.

To identify an operating system, or some of its characteristics such as the utilization of an additional code, it suffices to send identification octets through the card when the card is powered up. The power up operation which must occur

before it is possible to dialogue with a card. Internally, the operating system can build its response to the reset by analyzing the pointer AD_SAUT of the orientation table and the table itself.

A code sequence invalidation instruction INS_INA_C sent from outside the card through its processing program (PINA) exists for the purpose of inactivating a trap corresponding to a certain reference code, for example C. A first way to do this is comprised of deleting all the data related to this code -- the element in the orientation table and the entire corresponding sequence -- then updating the orientation table. A second, faster way is comprised of changing, in the orientation table, the reference code which corresponds to the code sequence to be invalidated by another value, for example "NULL", in which case the code will no longer exist and no interrogation instruction will be able to update the working memory AD_SAUT in order to execute this sequence. Finally, a last way is comprised of replacing the address value in the orientation table with an impossible address value, for example "NULL", or to direct the calling program to a simple code sequence which "returns control." This last way is well adapted to code sequences written in ROM, wherein this "dormant" code can be very easily activated and deactivated.

A regenerating write INS_REG coming from outside the card, through its processing program PREG, exists for the purpose of deleting one or more code sequences and/or table elements or, in a general way, a data set. Several methods are possible: the start and end addresses of a memory space are sent to the card and all the octets of this space are deleted. This somewhat "rough" method must be used with caution in order not to render inoperative the sequences which must not be deleted. Another method is comprised of indicating to the card the reference codes to be deleted as well as the corresponding code sequences in programmable memory, if any exist; after a command of this type, the program must determine the address of the corresponding table

elements, then delete these elements and determine the start and end addresses of the code sequences and finally delete the contents. The method for searching for these addresses and updating the table is highly dependent on the structure of the orientation table, as seen in Figs. 6 and 7. The method does not pose any problem for one skilled in the art and therefore does not need to be explained.

A final command REC_ESP_VRG, which can facilitate the updating of the table and the code sequences, is "Search Blank Space". Sent to the card without any particular data element, this command makes it possible to send to the outside, through the program PREV for processing this command, the blank spaces of the programmable memory (11) with the start and end addresses of each of these spaces. The user who wishes to add code sequences can, as a result of this command, verify whether the space available is sufficient and define the code sequence as a function of its storage address.

In a variant of embodiment which makes it possible to check the integrity of the additional code sequence, just before executing the jump in the orientation instruction sequence, it is necessary to provide two additional fields in each element of the orientation table: a first additional field, which indicates the size of the additional code sequence attached to the element, and a second additional field containing the initial correct checksum.

If the checksum calculated differs from the initial one, the operating system stops the execution by sending an appropriate status word. This method offers the advantage of performing a check before each execution of a new code sequence, but it increases the execution time of the command in question; it can be limited to performing this check only during the first execution.

The second method is comprised of providing a dedicated command which calculates the checksum of an area of programmable memory and compares the value calculated with the one provided or

written previously in the field CHK_X. Thus, the manager of the system can periodically activate this command in order to make sure that everything is okay.

An operating system which supports the device for writing and executing additional code can be restricted in order to reduce its capacities by means of inhibiting locks in programmable memory such as:

- CAI: Additional Code Inhibited, which will be tested by the operating system before any write.

- RCI: Card Regenerate Inhibited, which will be tested by the operating system before any regeneration.

These locks are similar, in terms of their management, to the locks for managing the life cycles of the operating systems of the cards.

One utilization of this capability is comprised of loading additional code into EEPROM (11), then setting the inhibiting locks RCI and CAI so as to freeze the added code.

Of course, the invention is not limited to the exemplary embodiment described above but on the contrary comprises all means constituting technical equivalents of the means specifically described as well as all combinations of same, if executed in the spirit of the invention and implemented within the scope of the following claims.



☐	Description	: paged	to paged
☒	Claim(s)	: paged 22	to paged 24
☐	Drawing(s)	: paged	to paged
☐	Abstract	: paged	to paged

CLAIMS

1. A device for executing code sequences in a support comprising an integrated circuit (10) capable of executing code sequences, as well as a first memory (12) containing a main program and possibly other code sequences executable by the integrated circuit, a second nonvolatile programmable memory (11) possibly containing code sequences executable by the integrated circuit, and a third working memory (14), characterized in that an orientation table contained in the second memory (11) contains at least one field containing a code reference data element of first means (INS_INT) which make it possible

- to verify the presence of a code reference,
- to store in working memory the address data element associated with the reference of the code and to set a trap indicator DI, and

of second means (INS_ORI) which make it possible

- to test the trap indicator DI, and
- to execute the jump to the address indicated by the contents of the working memory AD_SAUT.

2. The device according to claim 1, characterized in that the address data element of a code sequence contained in one of the three memories associated with the code reference is calculated from the code reference.

3. The device according to claim 1, characterized in that a second field contains the address data element of a code sequence contained in one of the three memories.

4. The device according to claim 1, characterized in that it comprises, in a possibly protected area of the programmable memory (11), a data element (AD_TAB) defining the first address and possibly the search direction (SENS) for the first verification means.





5. The device according to claim 1, characterized in that the orientation table comprises an additional field (AD_TT) indicating the next address in the table where the information corresponding to the code reference is located.

6. The device according to claim 1, characterized in that the orientation table comprises an additional field (CHK) which makes it possible to verify the integrity of the preceding fields and/or of the attached code sequence in comparison with the result supplied by a checksum calculating means.

7. The device according to claim 1, characterized in that the second means comprise means for storing in working memory (AD_RET) an address to return to after a jump when a plurality of second means are available to execute a jump directed to the same sequence address.

8. The device according to claim 1, characterized in that it comprises means CAI for preventing the writing of additional code, and means RCI for preventing a regeneration of the card.

9. A process for writing a new code in a device according to claim 4, characterized in that it is comprised of

- receiving a write command,
- verifying the presence of a first field constituting the code reference A, B, C, etc., and a second field representing the address AD_Cod_A, AD_Cod_B, AD_Cod_C, etc., associated with the code reference;
- verifying that no flag ECA is active;
- verifying that the value of the first address AD_TAB is consistent;
- checking that the reference supplied by the first field does not exist in the table;
- setting the flag ECA in the protected area of programmable memory to the active state;





- receiving and writingg the information corresponding to the new code;
- verifying that the operation is executed correctly and updating the orientation table;
- setting the flag ECA to the inactive state.

10. The process according to claim 9, characterized in that the operation for updating the table comprises a writing of the next address field in the orientation table with the address corresponding to the orientation word of the code preceding the code written.

This Sheet Has Been Replaced



ABSTRACT

Applicant: CP8 TRANSAC

Inventor: Azad NASSOR

PROCESS AND DEVICE ALLOWING A FIXED PROGRAM TO BE CHANGED

The present invention relates to a device for executing code sequences in a support comprising an integrated circuit (10) capable of executing code sequences, as well as a first memory (12) containing a main program and possibly other code sequences executable by the integrated circuit, a second nonvolatile programmable memory (11) possibly containing code sequences executable by the integrated circuit, and a third working memory (14), characterized in that an orientation table contained in the second memory (11) contains at least one field containing a code reference data element of first means (INS_INT) which make it possible

- to verify the presence of a code reference,
- to store in working memory the address data element associated with the code reference and to set a trap indicator DI, and of second means (INS_ORI) which make it possible
 - to test the trap indicator DI, and
 - to execute the jump to the address indicated by the contents of the working memory AD_SAUT.

Fig. 1 to be published.

Fig.1

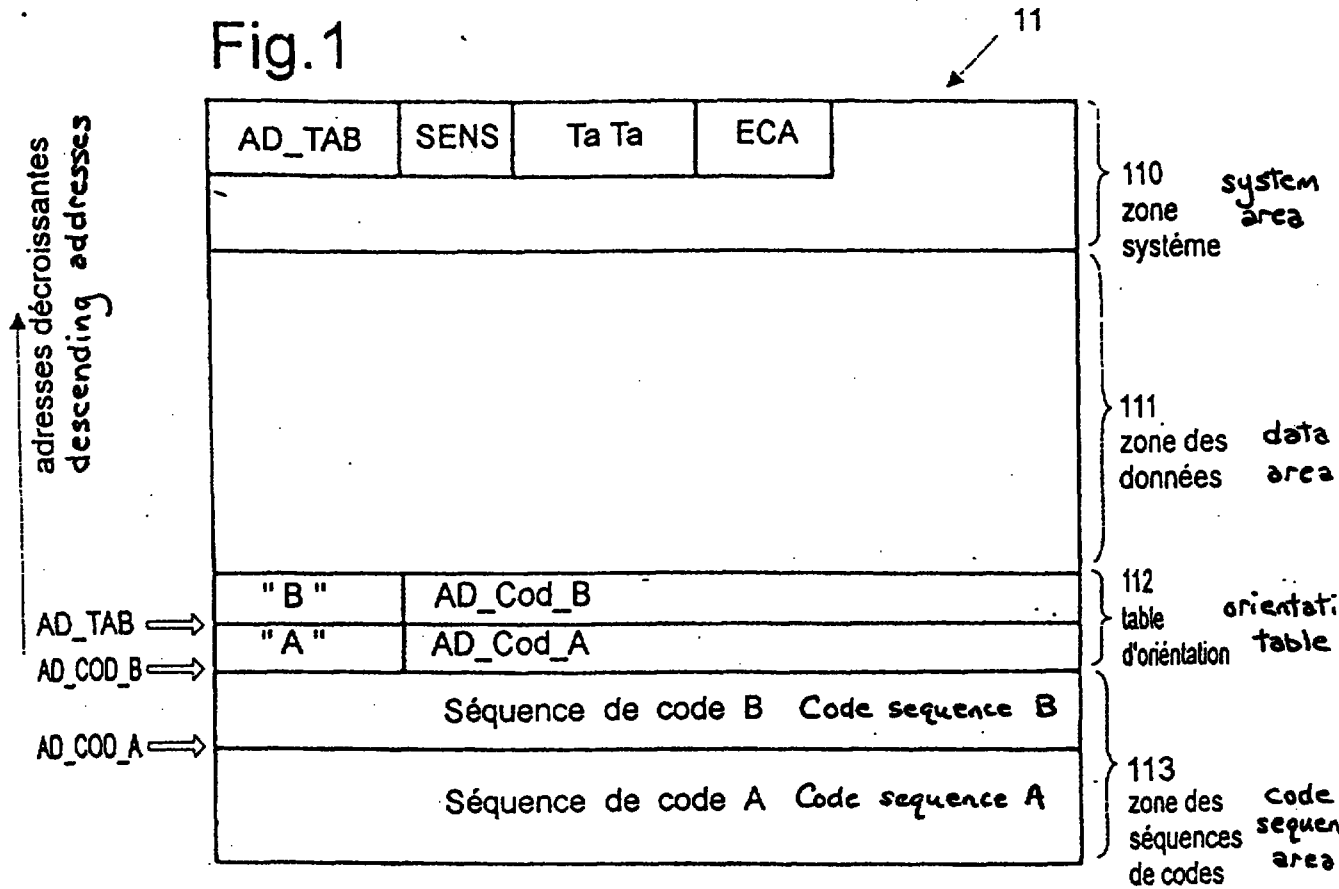
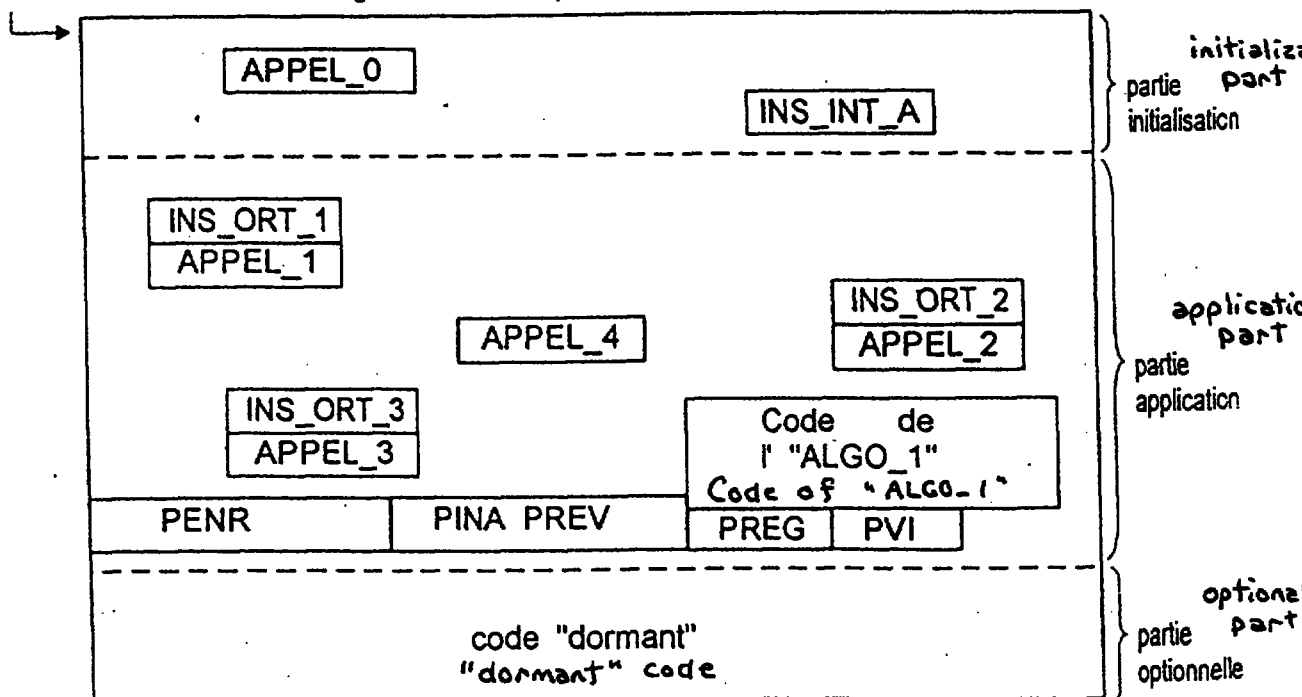


Fig.2

Start address of Main Program
Adresse de début du Programme Principal



Main Program
Programme Principal

Fig. 3

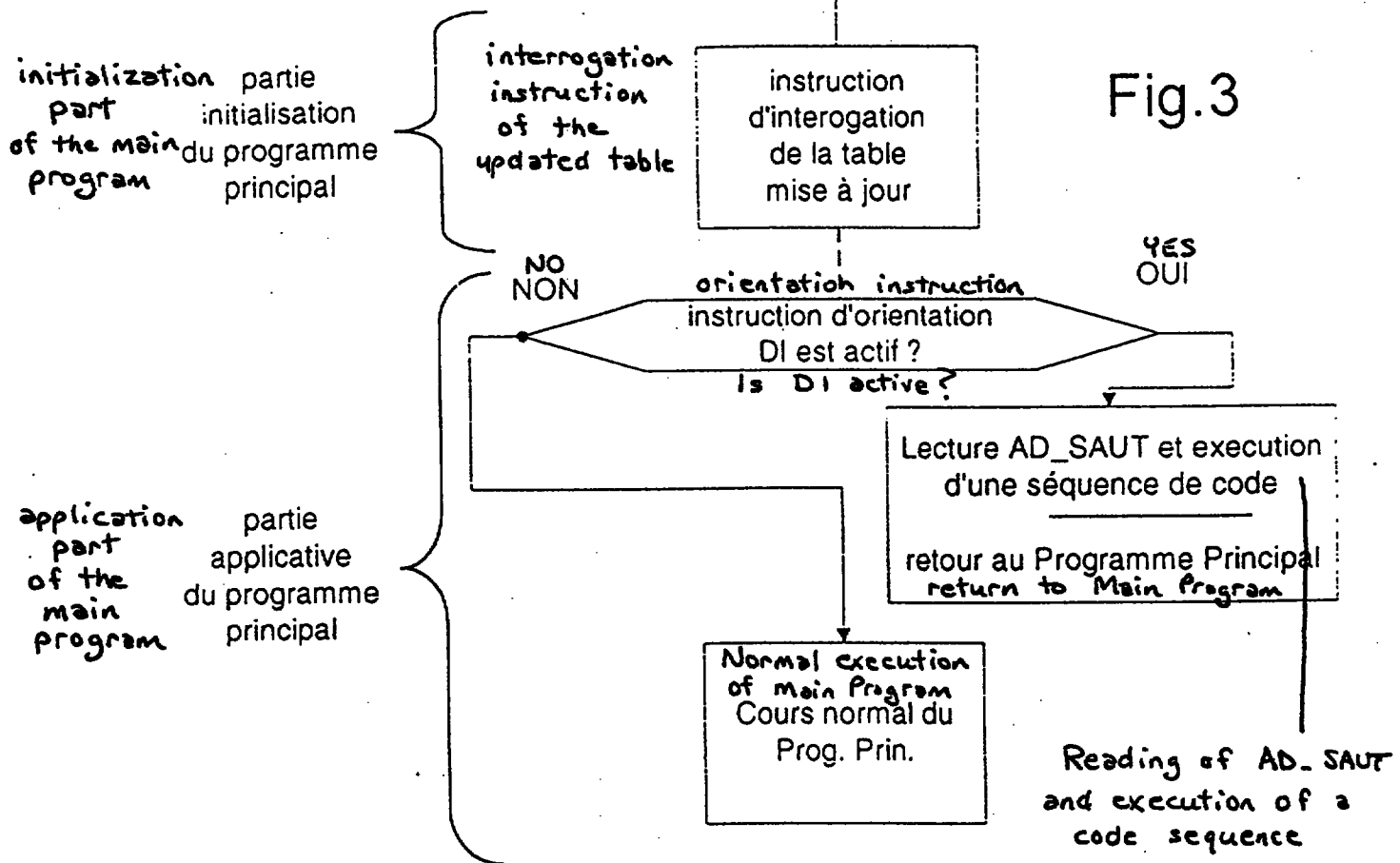
Main Program:
Programme Principal :

Fig. 4

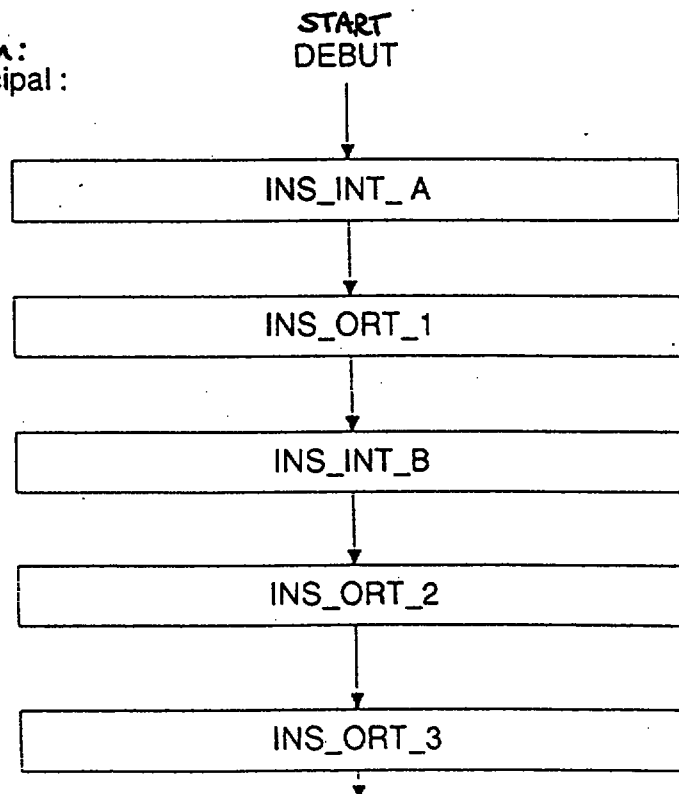


Fig.5

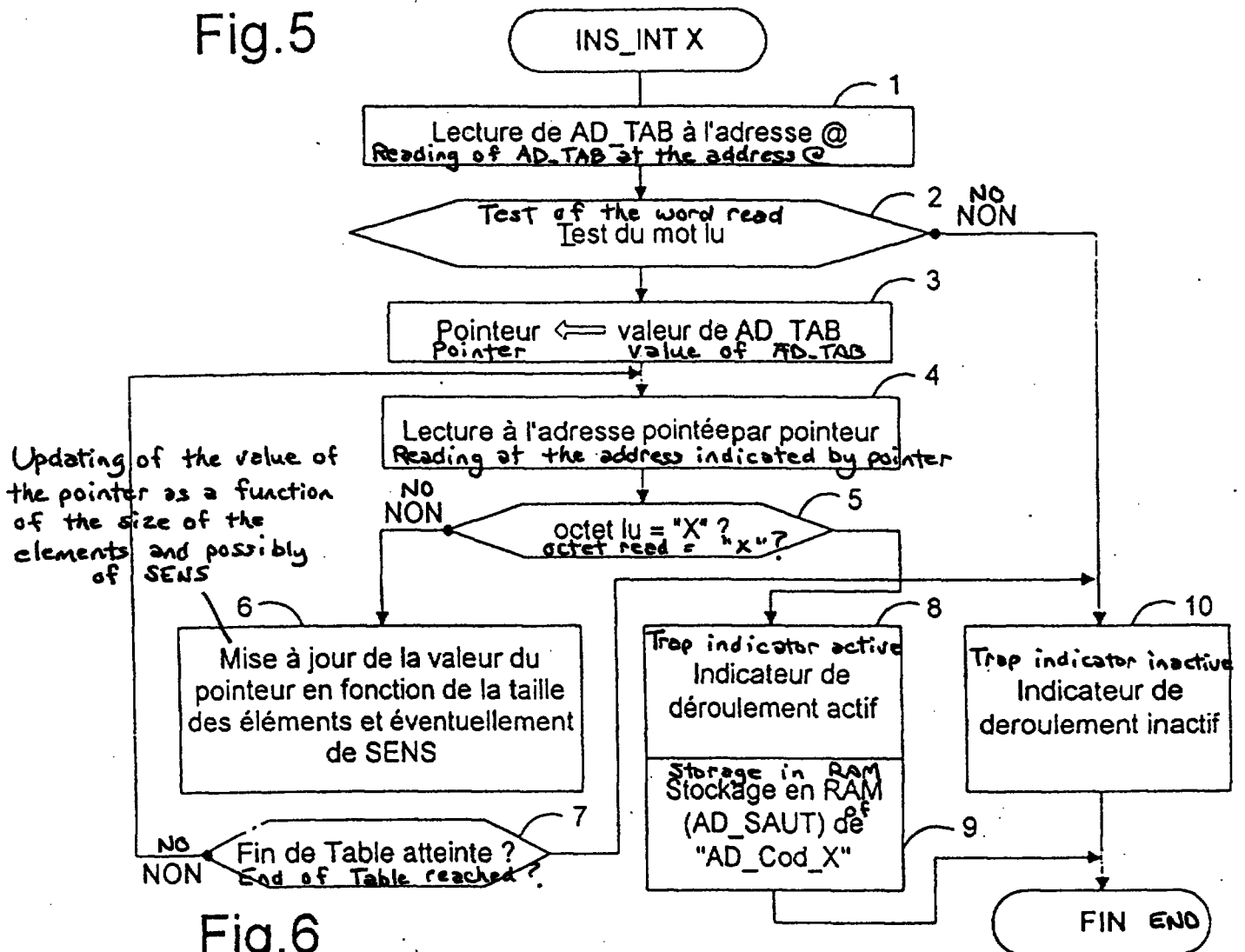


Fig.6

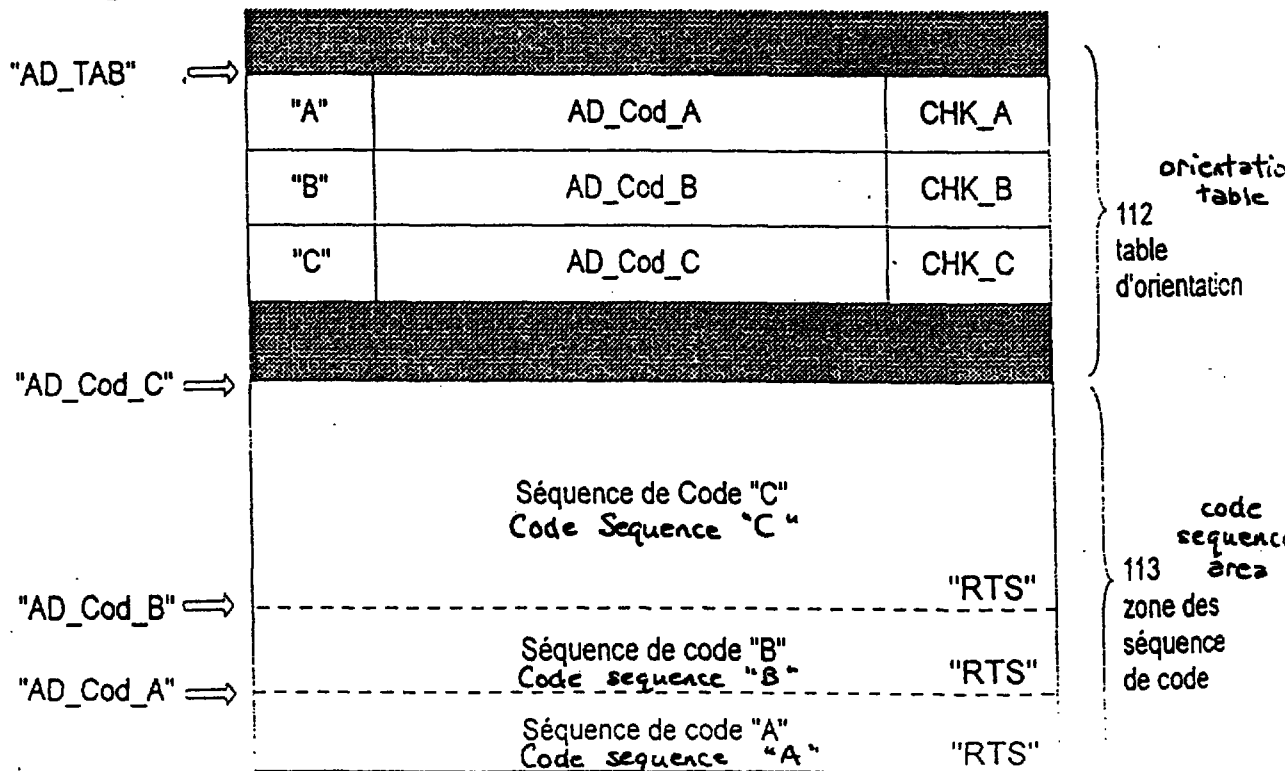


Fig.7

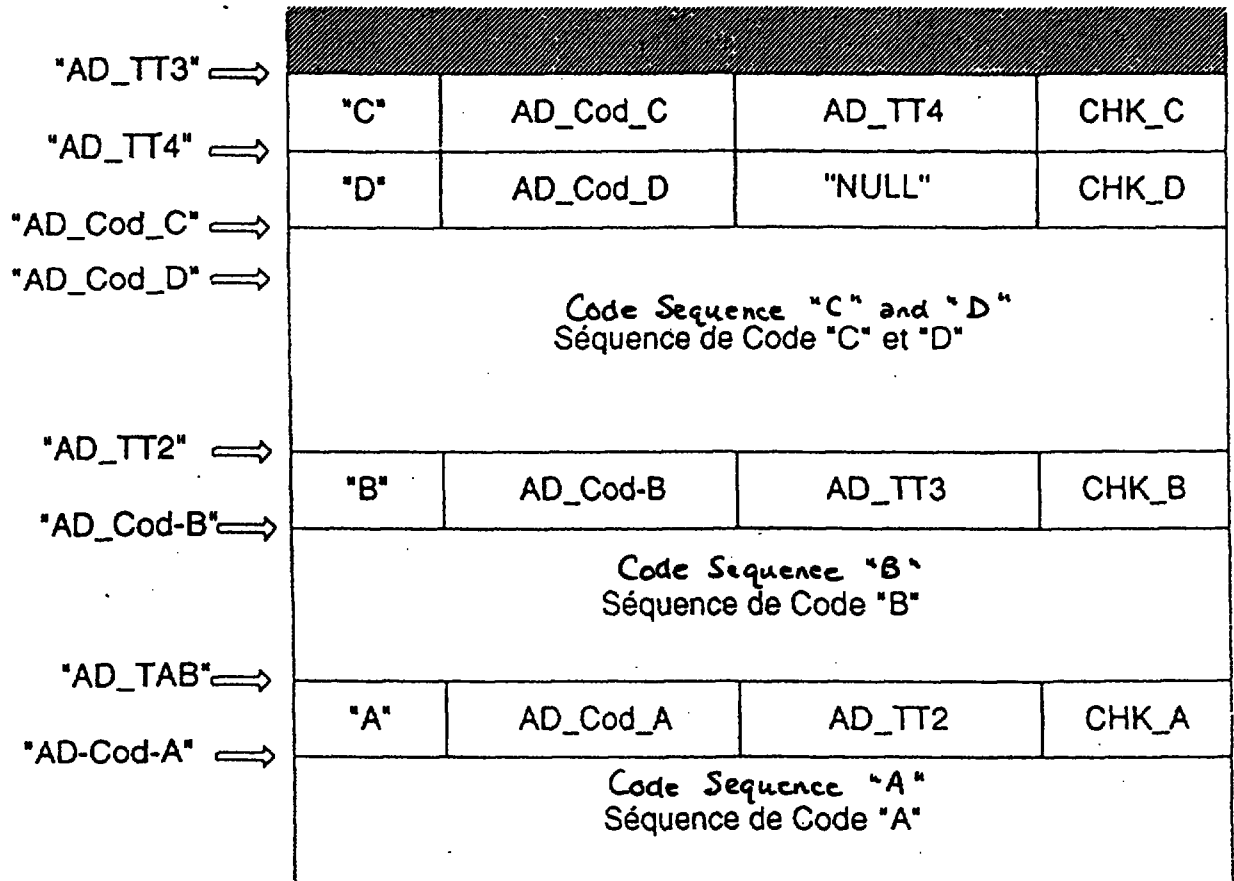
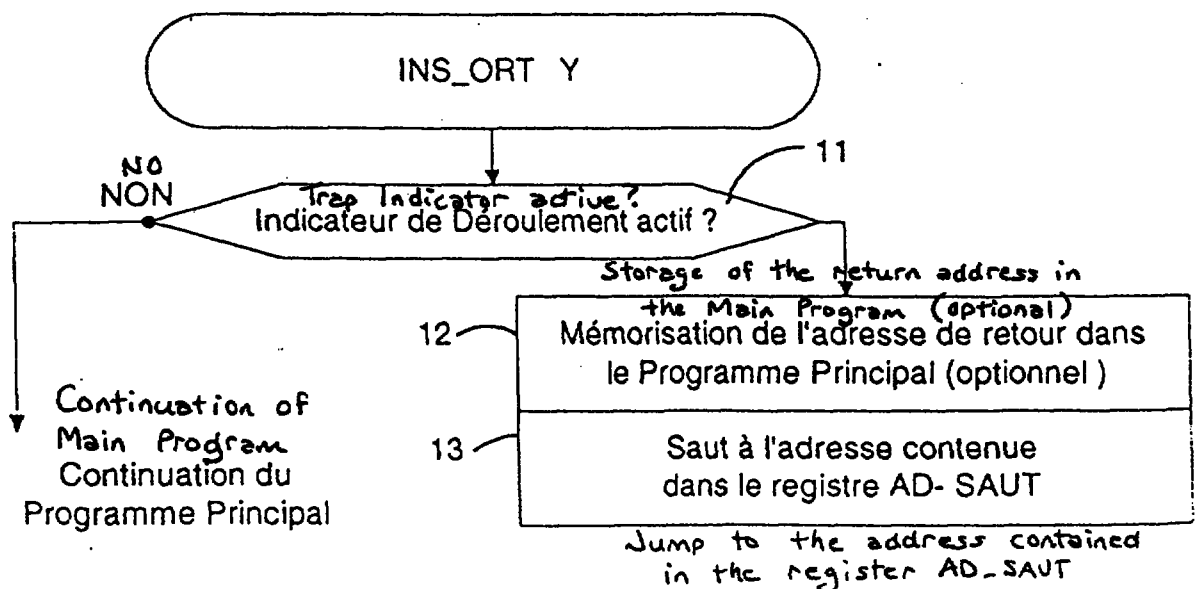


Fig. 8



optional fields
champs facultatifs

Fig. 9

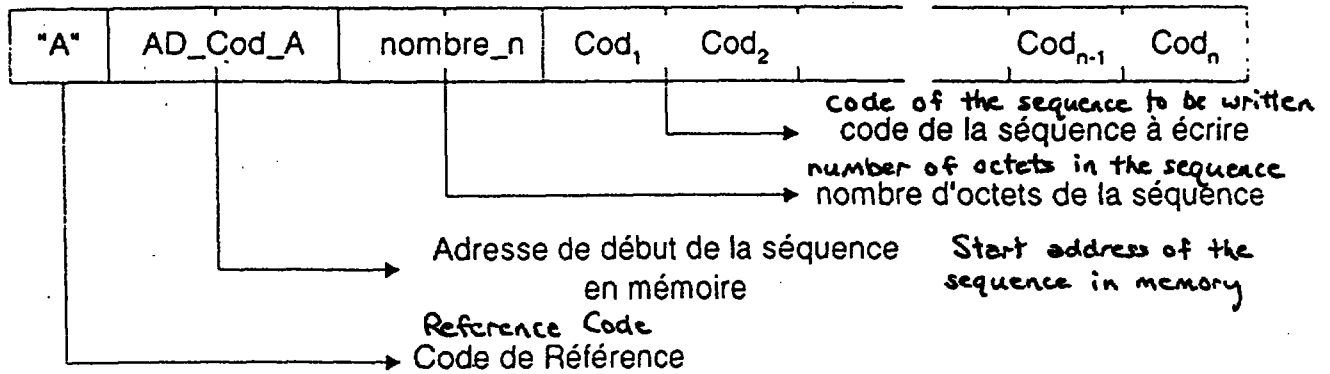


Fig. 10

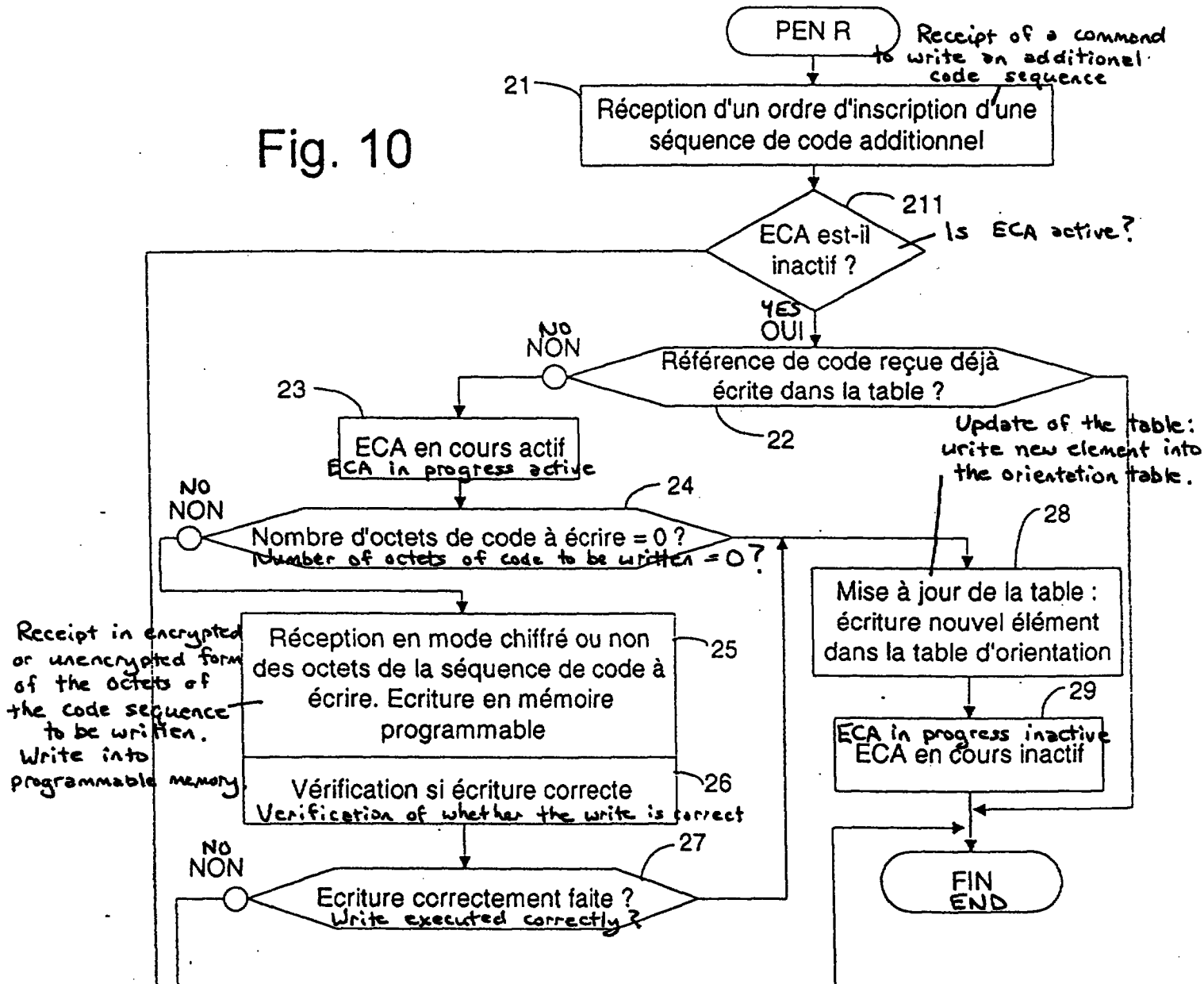
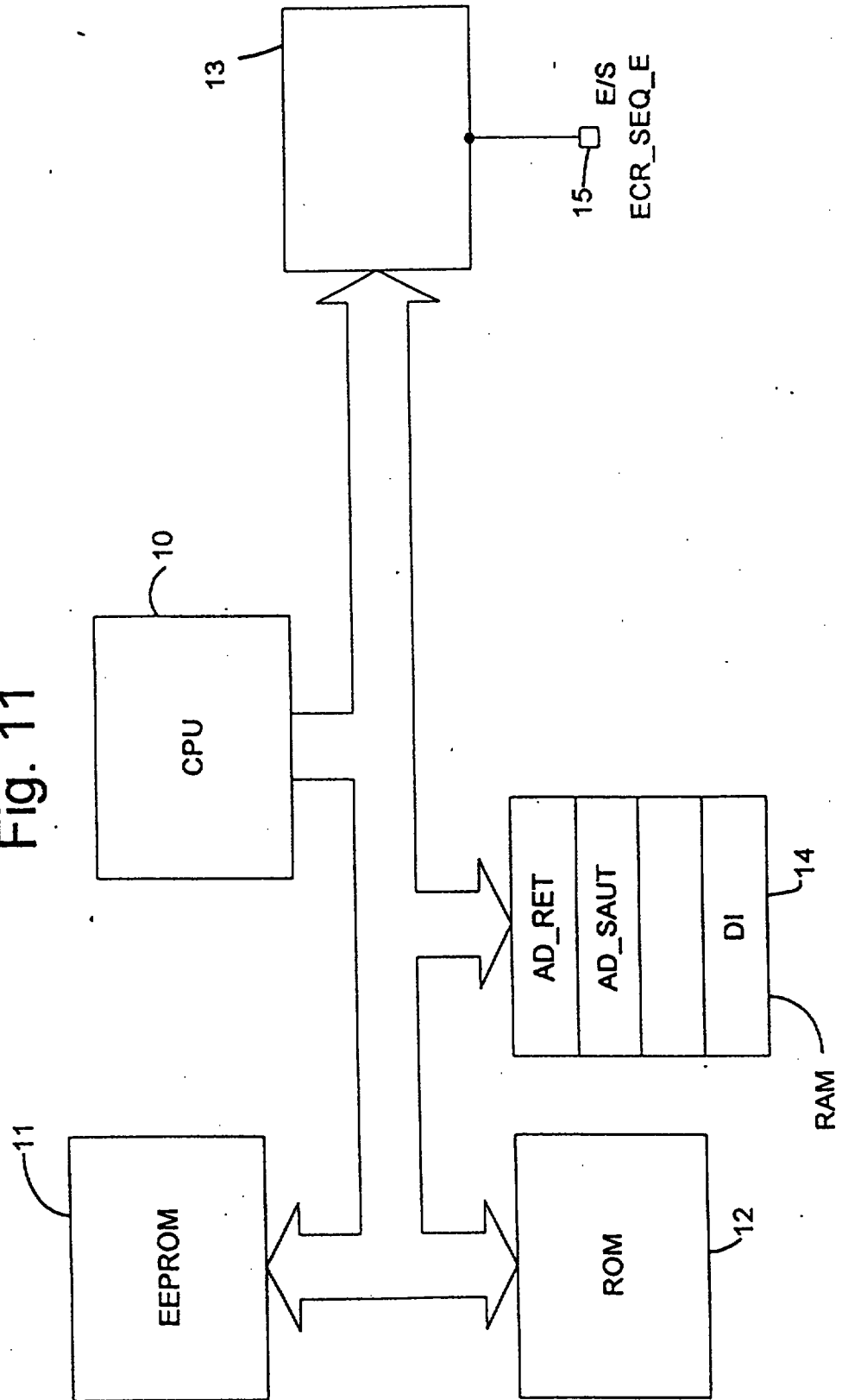
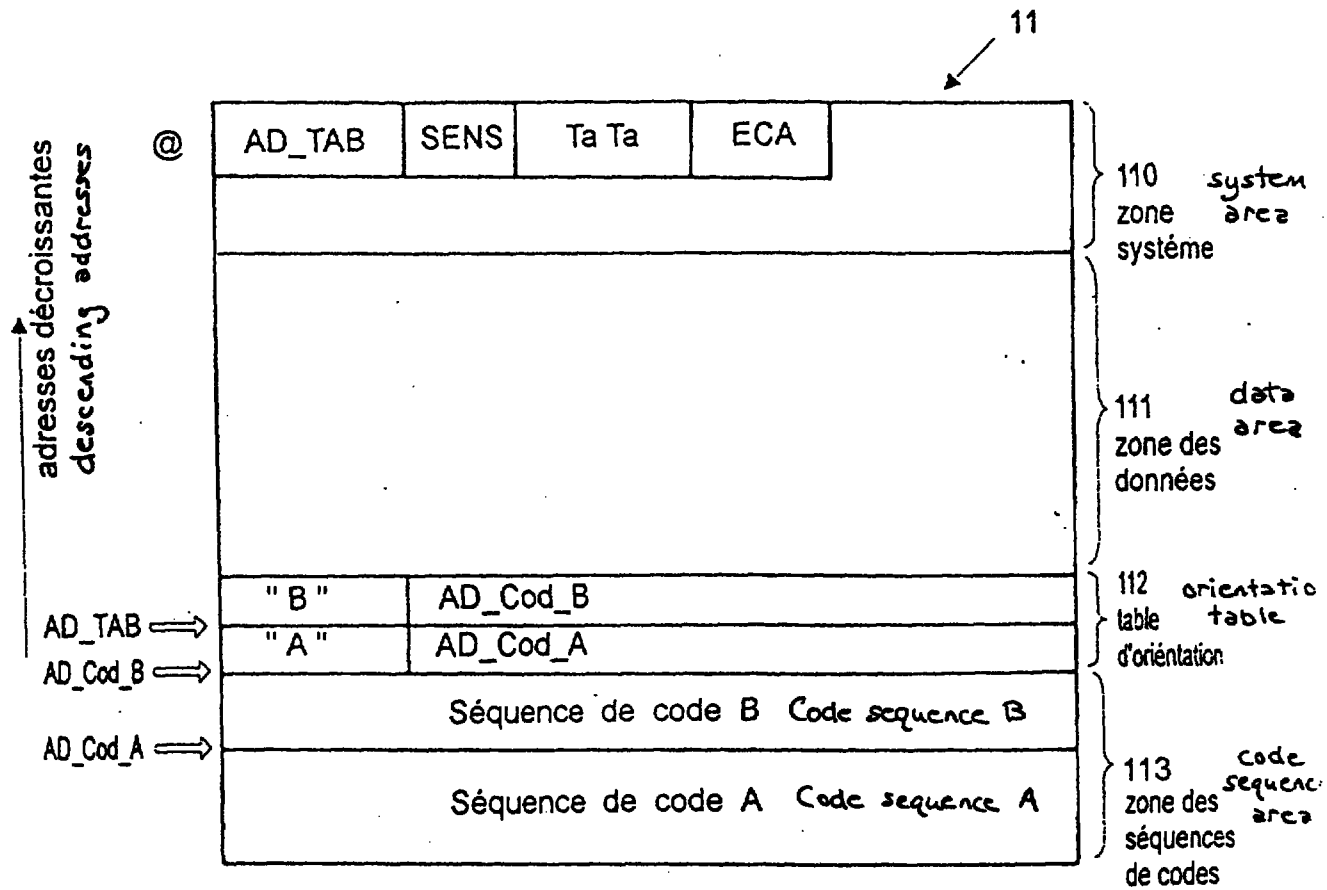


Fig. 11



Abstract Abrégé





REPLACEMENT SHEETS	
☐ Description	: paged to paged
☒ Claim(s)	: paged <u>22</u> to paged <u>35</u>
☐ Drawing(s)	: paged to paged
☐ Abstract	: paged to paged
DATE OF RECEIPT <u>27</u> OCT <u>1999</u>	

CLAIMS

1. A device comprising processing means for executing code instructions, non-programmable memory means and programmable memory means, wherein:

-said non-programmable memory means contain a code including several instructions and at least one code sequence reference;

-said programmable memory means contain an orientation table comprising a field for storing at least one code sequence reference linked with a code sequence for which it is intended to perform a code jump from said code to said code sequence, said code sequence being stored in one of said non-programmable memory means and programmable memory means;

-said processing means successively executes said instructions of said code until it reaches a given code sequence reference, and then reads said orientation table for deciding whether or not said code sequence reference is contained in said orientation table;

-in case of the response being yes, said processing means executes said code jump from said code to said code sequence linked with said code sequence reference then executes said code sequence, and then executes remaining instructions of said code; and

-in case of the response being no, said processing means does not execute said code jump and executes remaining instructions of said code.

This is the replacement sheet
Date: 27 OCT 1999



2. The device according to claim 1, wherein an address data element, associated to the executed code sequence and defining its location in said memory means, is calculated from the code sequence reference.

3. The device according to claim 1, wherein a second field contains an address data element of the code sequence to be executed defining its location in said memory means.

4. The device according to claim 1, characterized in that it comprises in a protected area of the programmable memory (11), a data element defining a first address and a search direction in order that the processing means read the programmable memory means.

5. The device according to claim 1, characterized in that the orientation table comprises an additional field indicating the next address in the table where information corresponding to the code reference is located.

6. The device according to claim 1, characterized in that the orientation table comprises an additional field adapted to verify the integrity of the preceding fields and/or of the attached code sequence in comparison with a result supplied by a checksum calculating means.

7. The device according to claim 1, characterized in that It comprises means for storing in the programmable memory means an address to return to after a jump when it is intended to execute several jumps directed to the same sequence address.



8. A process for writing a new code sequence in a device comprising processing means for executing code instructions, non-programmable memory means and programmable memory means, wherein said non-programmable memory means contain a code including several instructions and at least one code sequence reference, and said programmable memory means contain an orientation table comprising a field for storing at least one code sequence reference linked with a code sequence for which it is intended to perform a code jump from said code to said code sequence, said code sequence being stored in one of said non-programmable memory means and programmable memory means, the process comprising the steps of

- receiving a write command;
- verifying the presence in the write command of a first field constituting the code sequence reference, and a second field representing the address of the code sequence in the memory means;
- verifying that a predetermined flag is inactive;
- checking that the code sequence reference supplied by the first field does not exist in the orientation table;
- setting said flag to an active state;
- receiving and writing in the memory means information associated with the new code sequence, including updating said orientation table;
- setting said flag to the inactive state; and
- in case said flag is inactive or the code sequence reference exists in the orientation table before writing, stopping the writing process.

This is the replacement sheet

Date: 27 OCT 1999



9. A device according to claim 1, wherein it comprises:

-first means for verifying the presence of a code reference and storing in a working memory an address data element associated with the code sequence reference and setting a trap indicator; and

second means for testing the trap indicator and executing a jump to the address indicated by the contents of the working memory.

This is the replacement sheet

Date: 27 OCT 1999

1 / 6

Fig.1

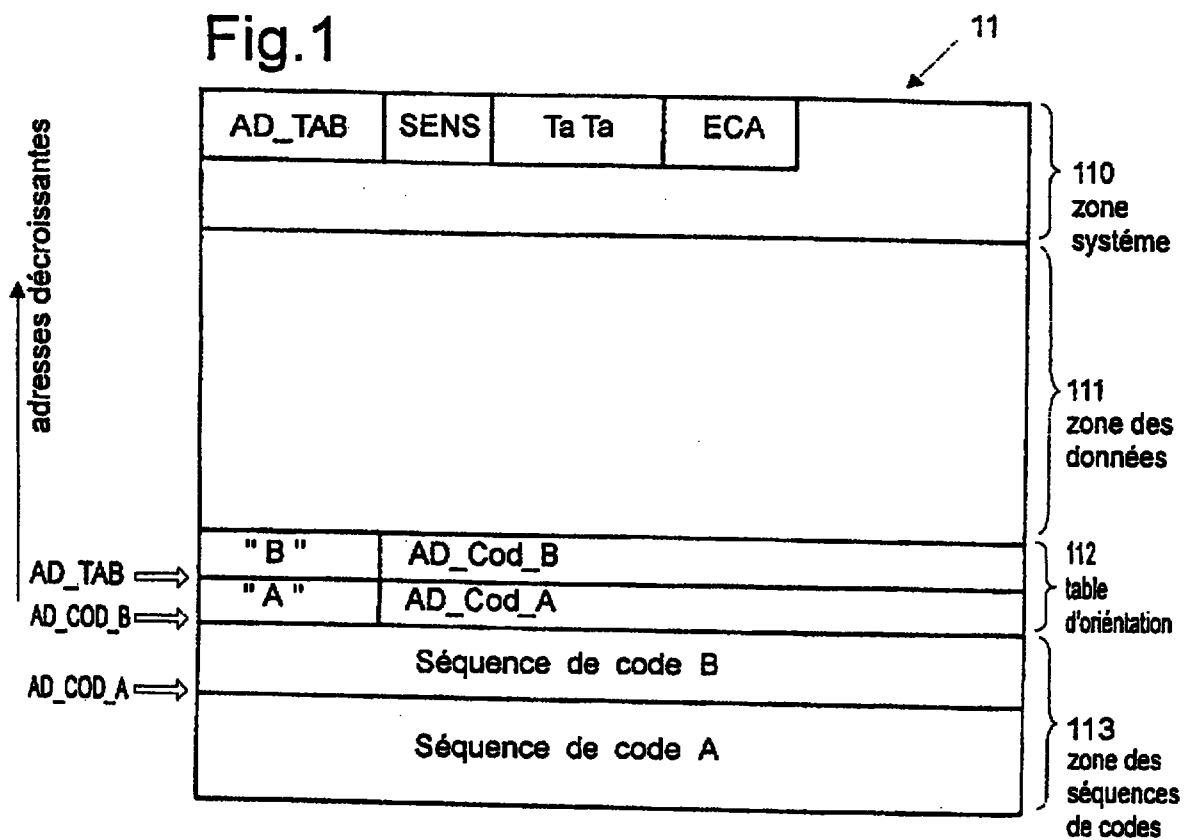
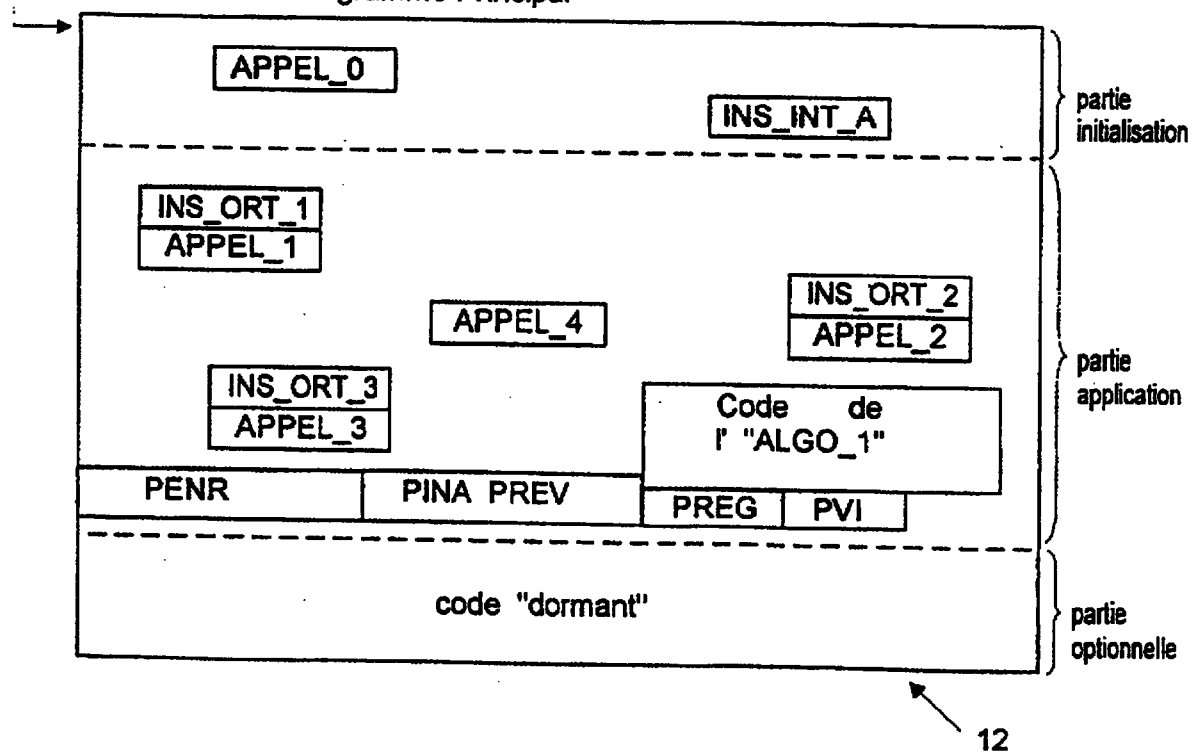
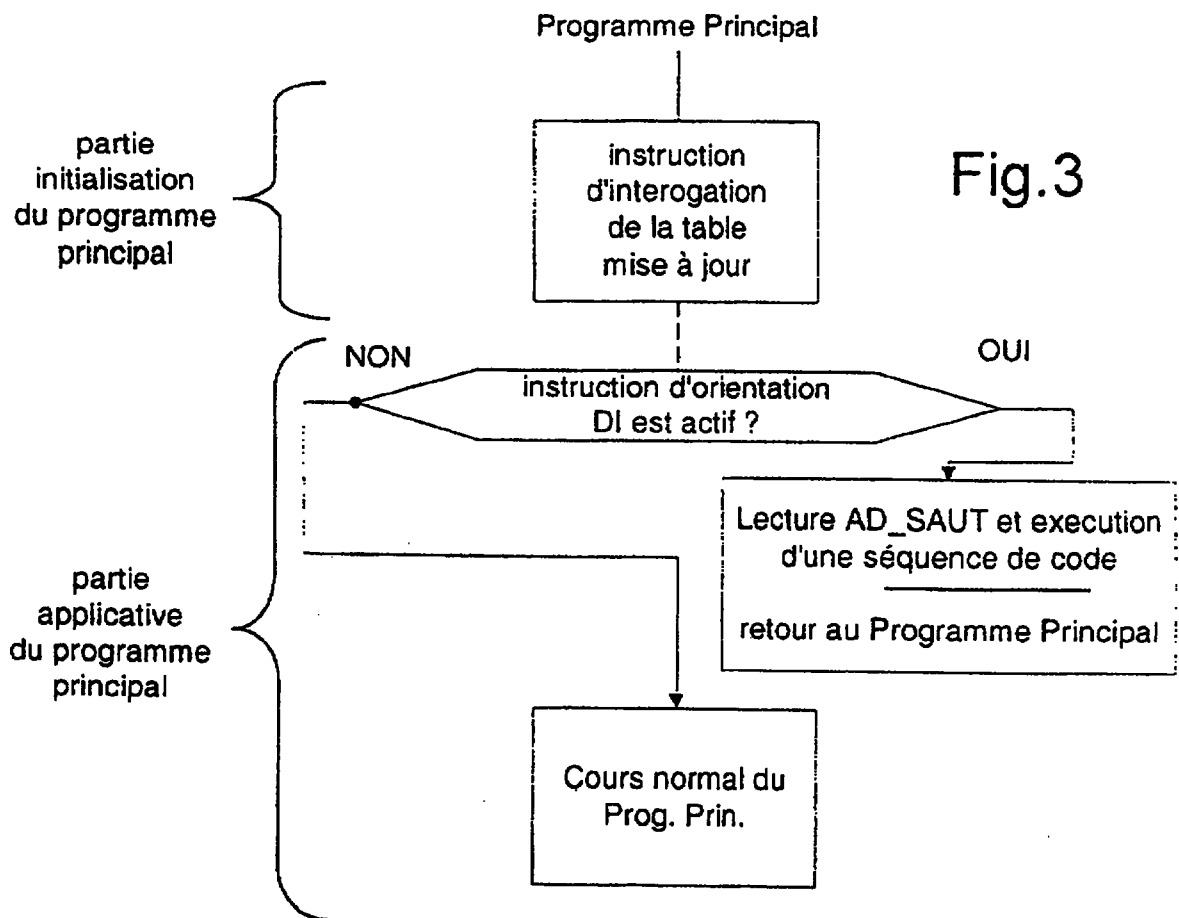


Fig.2

Adresse de début du Programme Principal

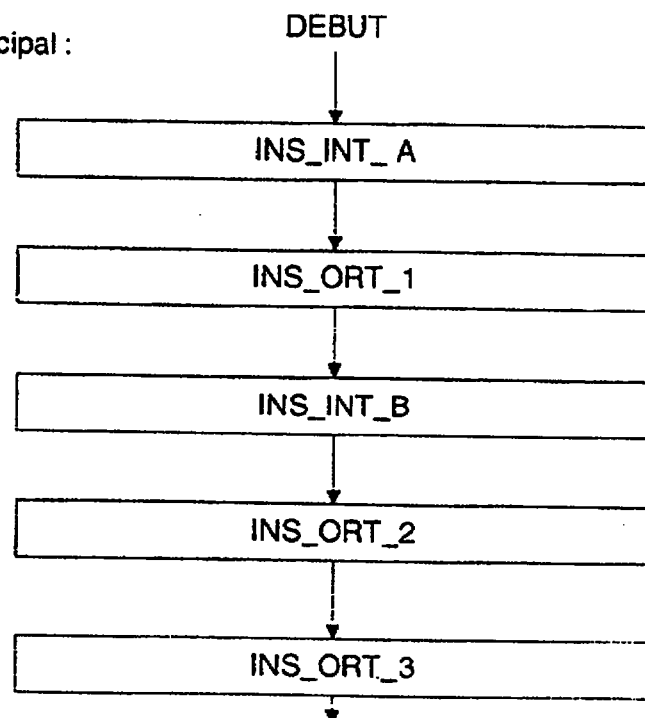


2/6



Programme Principal :

Fig. 4



3 / 6

Fig.5

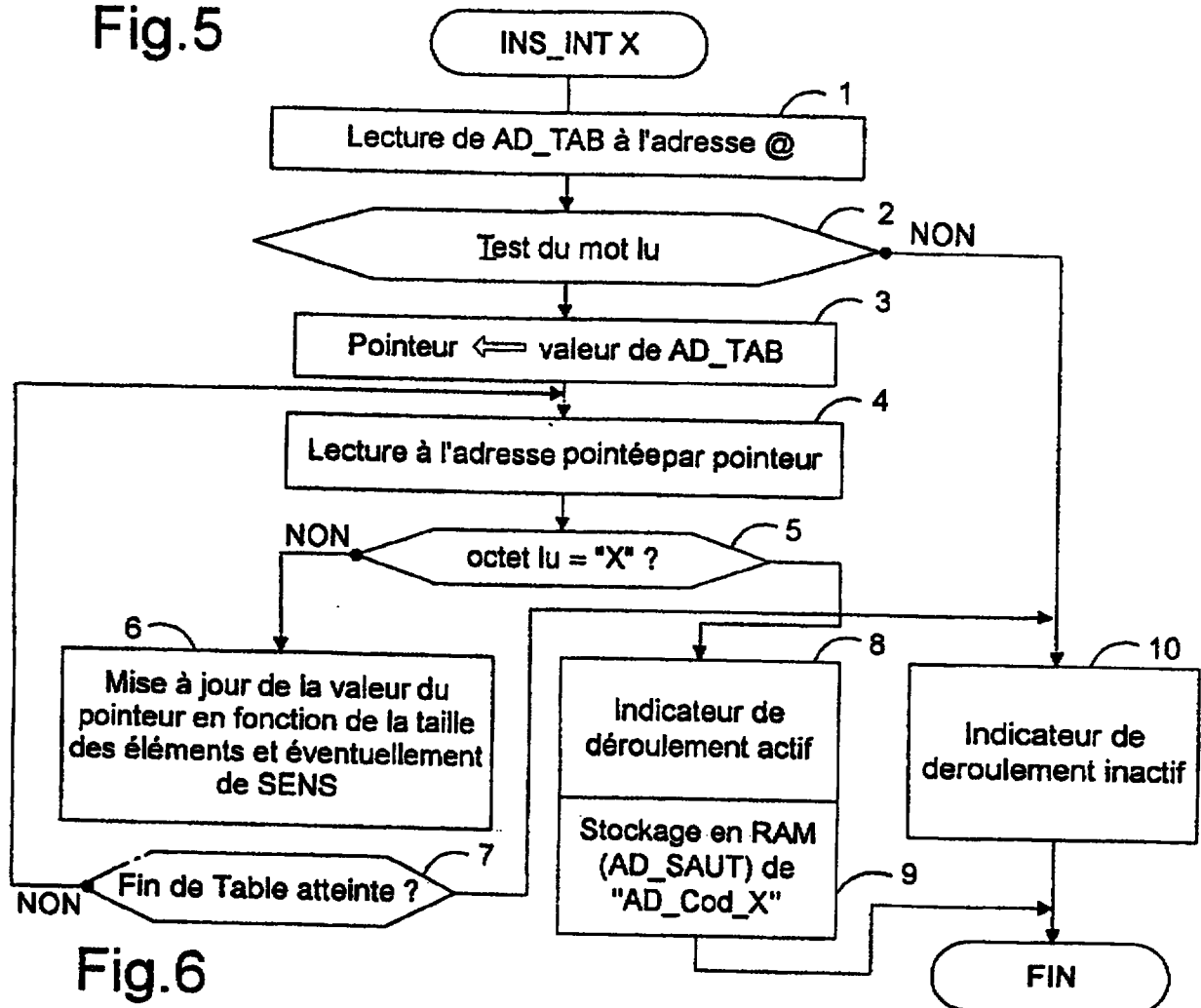
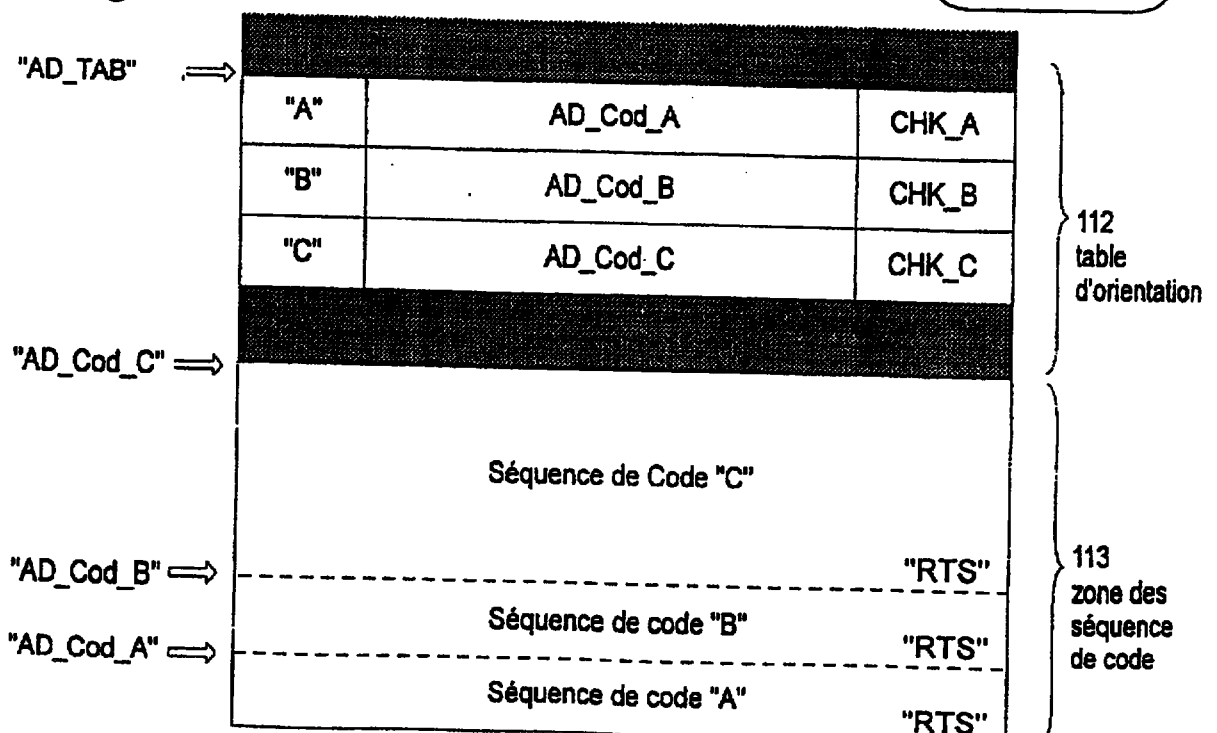


Fig.6



4/6

Fig.7

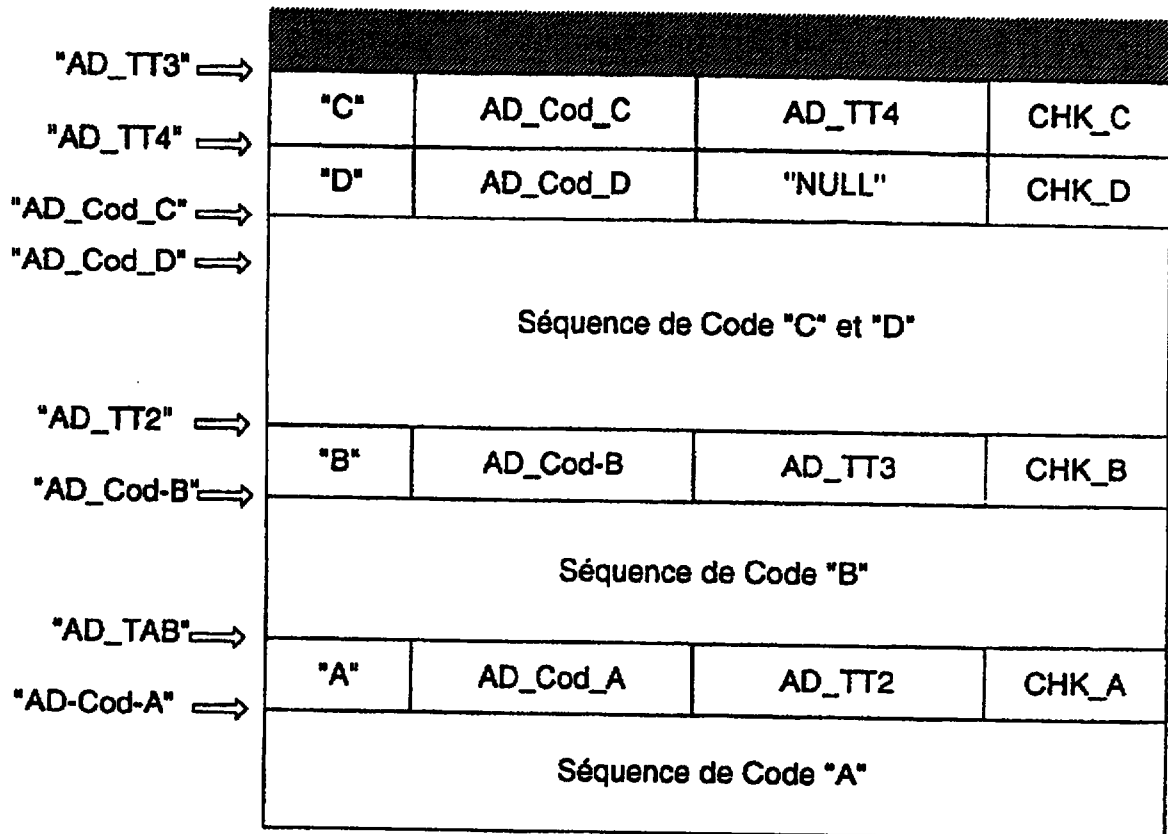
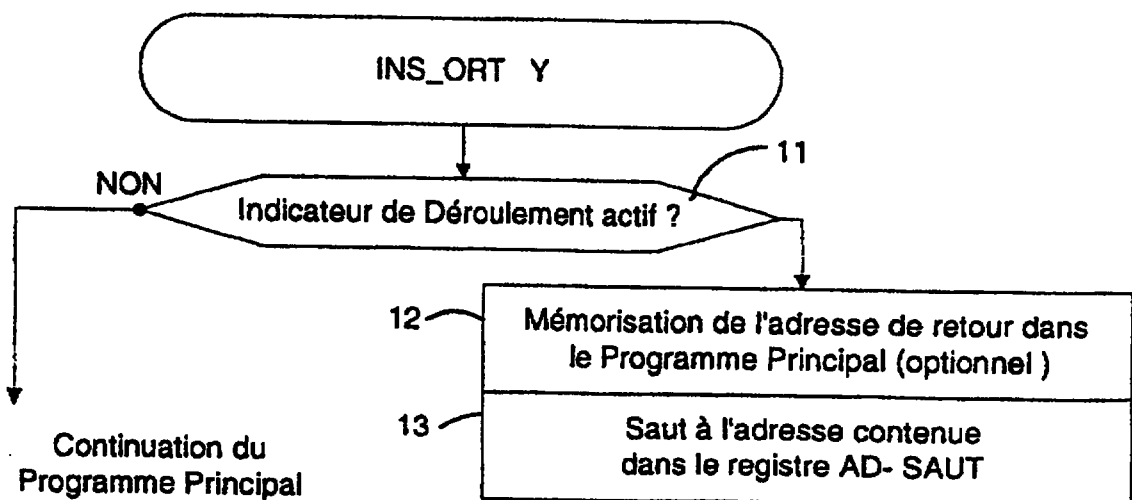


Fig. 8



5/6

Fig. 9

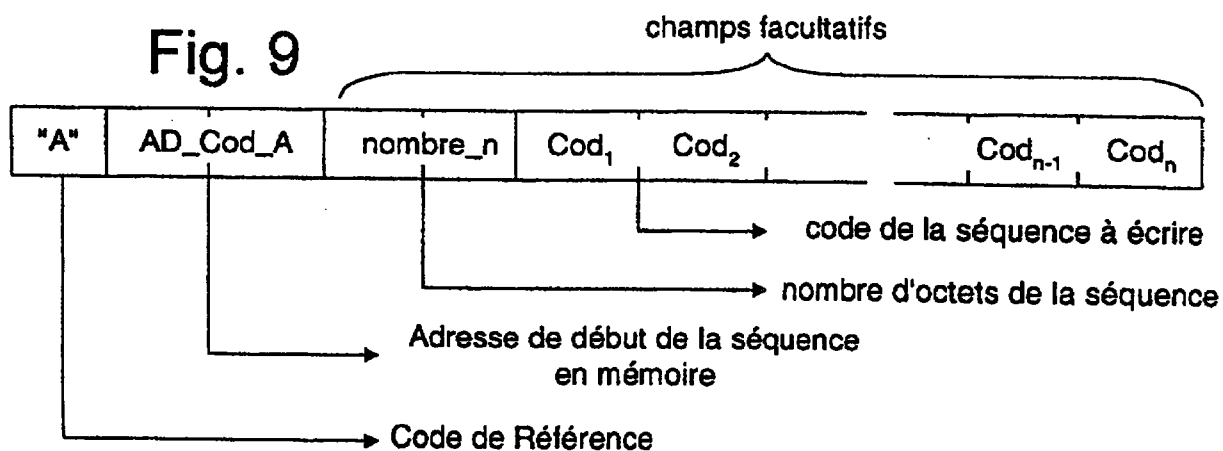
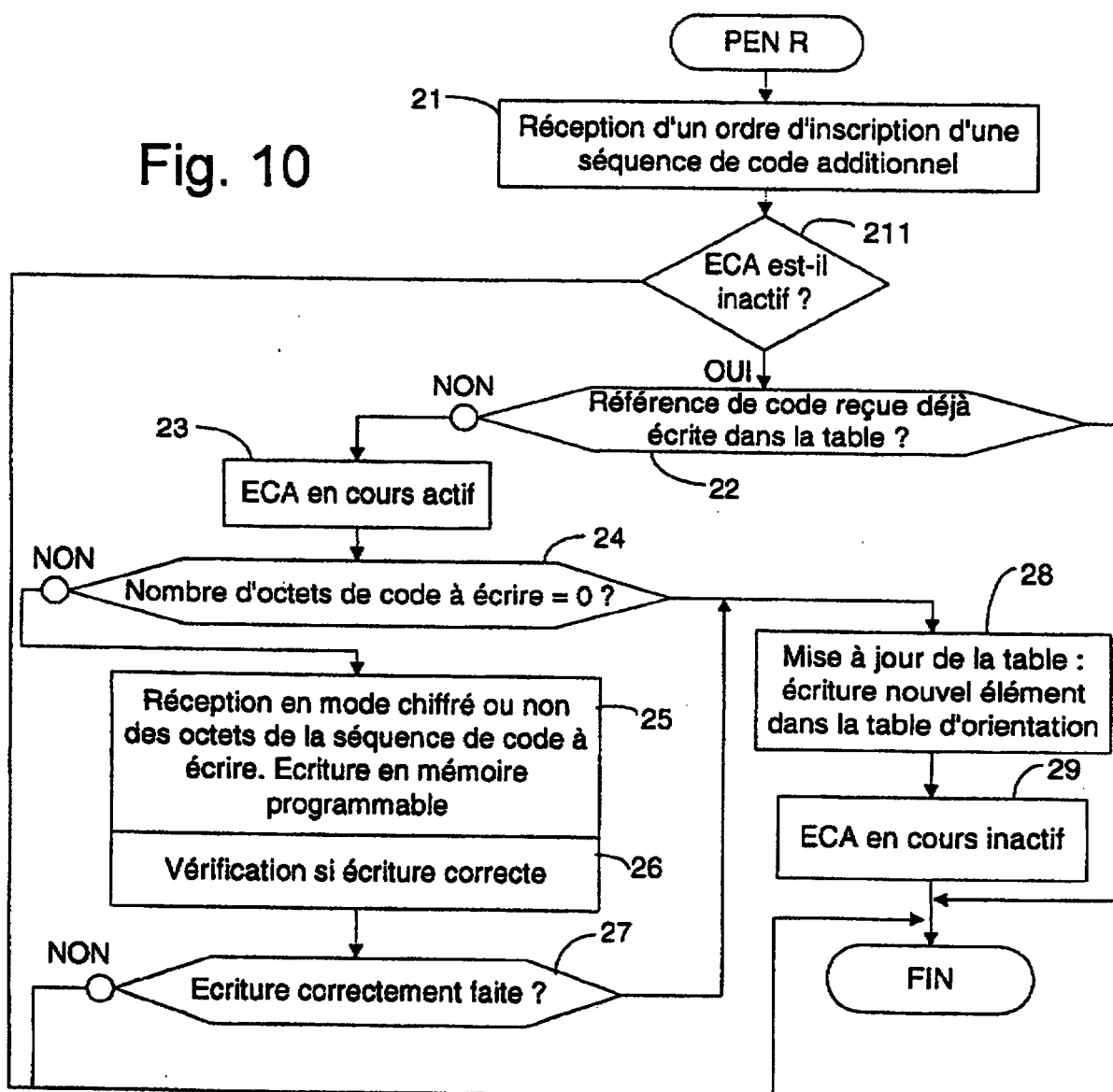


Fig. 10



6/6

