Le développement avec la BasicCard (Auteur : Didier DONSEZ)

Java n'a pas été le seul langage qui a été choisi pour le développement d'applications carte. Ces concurrents basés sur l'interprétation d'instructions intermédiaires (i.e. bytecode) sont et ont été Forth, Visual Basic (SCW), C#/CLI (.NET SmartCard), ZC-Basic (BasicCard), … . Ces concurrents basés sur l'exécution d'instructions natives (i.e. code machine) sont le C incluant parfois l'assembleur du processeur cible. Dans ce chapitre, nous nous intéresseront au langage ZC-Basic, le langage de développement de la BasicCard. Cette carte a la vertu d'être très simple à mettre en œuvre notanment dans un contexte pédagogique. Cependant la BasicCard reste une carte mono-applicative qui ne permet pas d'installer et d'exécuter et de desinstaller plusieurs applications (isolées les unes des autres) de plusieurs prestataires comme la JavaCard ou la SCW.
Le langage ZC-Basic est un dialecte du Basic qui doit être compilé vers un langage intermédiaire, le P-Code, pour être exécuter par un interpréteur de P-Code. La BasicCard contient un interprète de P-Code. Ainsi une application carte est donc écrite en ZC-Basic, puis compilée en P-Code et chargée dans la carte pour y être initialisée et personnalisée. Le langage ZC-Basic est un langage propriétaire de la seule société Zeit-Control.
D'autre part, les BasicCards (sauf Compact) possèdent un système de fichiers hiérarchique avec une racine des noms longs semblable à celui de DOS. Les fichiers sont aussi bien accessibles par l'application carte que par l'application terminal.
Les applications peuvent également développer en ZC-Basic. Celles-ci interprètent alors un programme écrit en P-Code. Les applications terminal peuvent être également écrites dans d'autres langages comme C, C++, C#, VB, Java, Delphi via les interfaces de programmation standards PC/SC, MUSCLE, OCF, eOCF ou propriétaires et sur un certain nombre de plate-formes (Windows32, Unix, Linux, MacOSX, OCF, eOCF …).
Dans ce chapitre, nous présenterons successivement l'environnement de développement d'une application, le langage BasicCard, les principales fonctions de bibliothèque et des commandes prédéfinies. Nous terminerons ce chapître avec notre exemple de porte monnaie électronique.

1.1 Environnement de Développement

L'environnement de développement comporte un ensemble d'outils pour le développement des applications coté carte et des applications coté terminal. Le compilateur ZC-Basic vers P-Code est complété par un chargeur d'application carte, un générateur de clés, un chargeur de clés, un simulateur/déboggeur d'application carte et un simulateur/déboggeur d'application terminal.
L'environnement de développement est fourni avec le kit de développement BasicCard comportant aussi un lecteur série ou un lecteur USB, un lecteur portable BalanceReader et un set de cartes BasicCard. L'environnement de développement et ses mises à jour peuvent être téléchargés gratuitement sur le site www.basiccard.com.

Remarquons que le simulateur/deboggeur d'application carte permet de développer des applications carte et terminal en se passant de lecteur et de carte.
Remarquons enfin que un certain nombre d'utilitaires pour manipuler les fichiers d'une BasicCard sont livrés parmi les exemples.

1.1.1 Type de fichiers

L'environnement de développement manipule et produit un certain de fichiers dont les contenus sont les suivants.

- BAS contient un source ZC-Basic pour une application carte ou une application terminal.
- DEF contient un source ZC-Basic déclarant des constantes, des clés ou des commandes et destiné à être inclus dans un fichier BAS
- IMG contient les bytecodes P-Code d'application carte ou terminal destinés à l'interpréteur de la BasicCard ou du Terminal. Il est utilisé par le simulateur de carte ZCMSIM et par le chargeur d'application BCLOAD.
- DBG contient les bytecodes P-Code d'application carte ou terminal destinés à l'interpréteur de la BasicCard ou du Terminal ainsi que les informations nécessaires au deboggage symbolique des applications. Ces fichiers sont utilisés par les déboggeur ZCMDTERM et ZCMDCARD.
- EXE contient un exécutable DOS contenant l'interprète de P-Code et le P-Code d'une application terminal. Il permet de distribuer les applications terminal sans l'environnement de développement.
- LST contient la version lisible des instructions P-Code produit par la compilation.
- MAP contient la version lisible de la table des symboles (variables, fonctions, ...) et de leur allocation dans les régions de mémoire volatile et non volatile.
- LIB contient une bibliothèque additionnelles de fonctions pour être incluse dans une application ZC-Basic. La déclaration des fonctions de cette bibliothèque est dans le fichier DEF du même préfixe.
- ZCC contient le contenu de l'EEPROM du BasicCard simulée par le simulateur/déboggeur ZCMDCARD. Il permet de conserver l'état (données, fichiers, type de la carte, option de compilation) de la carte entre 2 simulations.
- ZCT contient les options de compilation et d'exécution de l'application terminal.
- LOG contient le journal des traces des échanges (ATR, APDU) entre une application terminal et l'application carte
- ZCF contient la description d'un modèle de BasicCard (capacité EEPROM, algorithmes cryptographiques supportés en natif, …). Remarquons qu'il est nécessaire de recharger et de réinstaller les mises à jour de l'environnement de développement pour récupérer les fichiers de configuration ZCF des derniers modèles de BasicCard.
- ZCP contient la description d'un projet regroupant plusieurs applications carte et terminal. Il est utilisé par l'IDE BasicCard.

1.1.2 Outils

L'environnement de développement comporte les outils communs et des outils spécifiques au développement des applications carte (Figure 1) et des applications terminal (Figure 2):

- Le compilateur ZCMBASIC compile le source BAS et les déclarations DEF produit un fichier IMG contenant le programme P-Code. Le compilateur peut générer les fichiers additionnels DBG, LST et MAP. Il peut générer également un exécutable DOS EXE qui inclut l'interprète de P-Code et le programme P-Code de l'application terminal.

- L'outil BCLOAD permet de charger une application carte sur une carte ou sur un lot de cartes.
- L'utilitaire ZCMSIM est un interprète de P-Code qui peut être utilisé pour simulé une BasicCard ou exécuter une application terminal.
- L'utilitaire KEYGEN est un générateur de clés pour les divers algorithmes de cryptographie installée dans la carte. Le fichier produit contient les déclarations ZC-Basic des clés. Il peut être soit inclus dans une application carte, soit chargé dans la (ou les) carte avec l'utilitaire BCLOAD.
- Le débogueur symbolique ZCMDCARD permet graphiquement d'exécuter en pas à pas une application carte dont le programme est dans le fichier DBG. Il permet d'inspecter l'état des variables, le contenu des répertoires et des fichiers. Le débogueur sauvegarde l'état de la mémoire (EEPROM) de la carte simulée dans un fichier ZCC pour pouvoir reprendre l'exécution à partir de cet état.
- Le débogueur symbolique ZCMDTERM permet graphiquement d'exécuter en pas à pas une application terminal dont le programme est dans le fichier DBG.

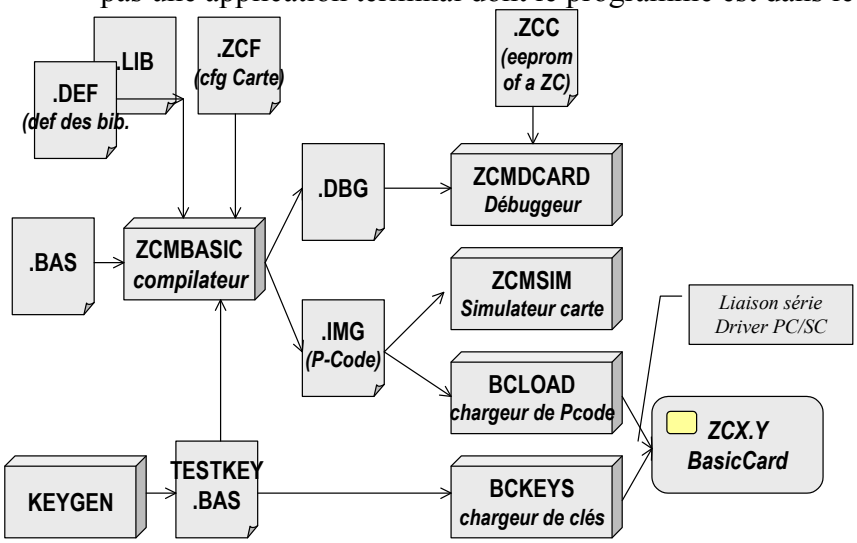

Figure 1 : Outils de développement d'applications carte

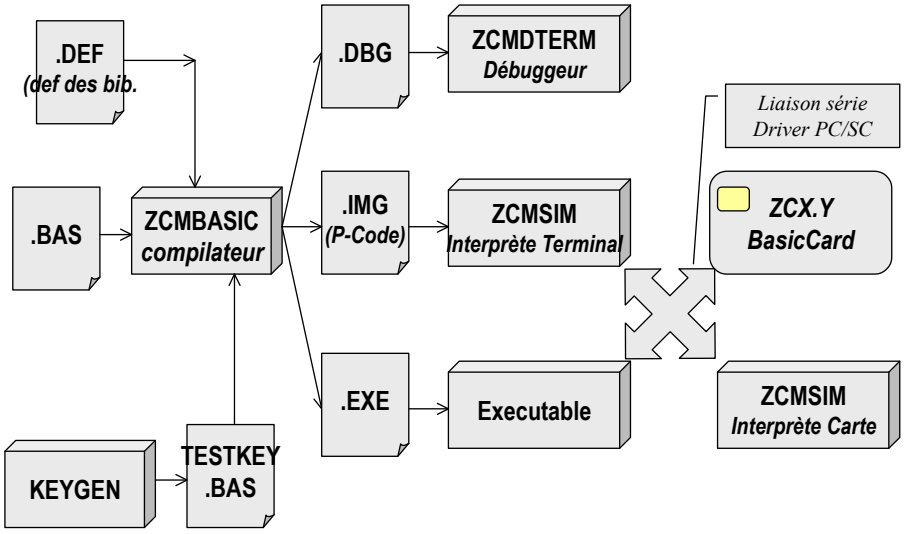

Figure 2 : Outils de développement d'applications terminal

1.2 Cycle de vie et états d'une BasicCard

Les BasicCards Compact et Enhanced ont 4 états possibles : NEW, LOAD, TEST, RUN. La BasicCard Professional en possède un de plus : PERS. La Figure 3 représente les transitions possibles entre ces états. La signification de ces états est la suivante :
NEW est l'état dans lequel est la carte lors de sa fabrication. Elle passe dans l'état LOAD lors de sa configuration par le fabricant. La configuration consiste entre autre à lui affecter un numéro de série unique, une graine (seed) unique pour le générateur de nombres aléatoire, à positionner son ATR par défaut. La carte alors peut être remis aux émetteurs de carte qui y installeront une application et la personnaliseront.

LOAD est l'état dans lequel est la carte quand elle est reçue par l'émetteur. Elle peut être chargée alors d'une nouvelle et unique application. Elle passe alors dans l'état PERS ou dans l'état TEST.

L'état PERS correspond à la phase de personnalisation pendant laquelle certains commandes de l'application sont utilisée pour personnaliser la carte (exemple : positionner le nom du porteur, fixer le montant initial du solde d'un porte monnaie, ...). Comme il est absent des cartes Compact et Enhanced, il peut être émulé avec un drapeau booléen et un test en prologue de chaque commande de l'application.

TEST est l'état d'une carte chargée et personnalisée. Les commandes peuvent lui être envoyées. Cependant l'application courante peut être remplacée par une autre ou par une nouvelle version. La carte peut transiter entre les états LOAD, PERS, TEST. Ces états sont utilisés lors des phases de développement.

Avant d'être remis au porteur, la carte est passée dans l'état RUN. La carte ne peut plus alors être chargée d'une nouvelle application, ni être personnalisée. Une carte ne doit jamais être remis au porteur en mode TEST pour prévenir les fraudes par clonage, lecture des clés privées, réinitialisation des compteurs d'usage, ...

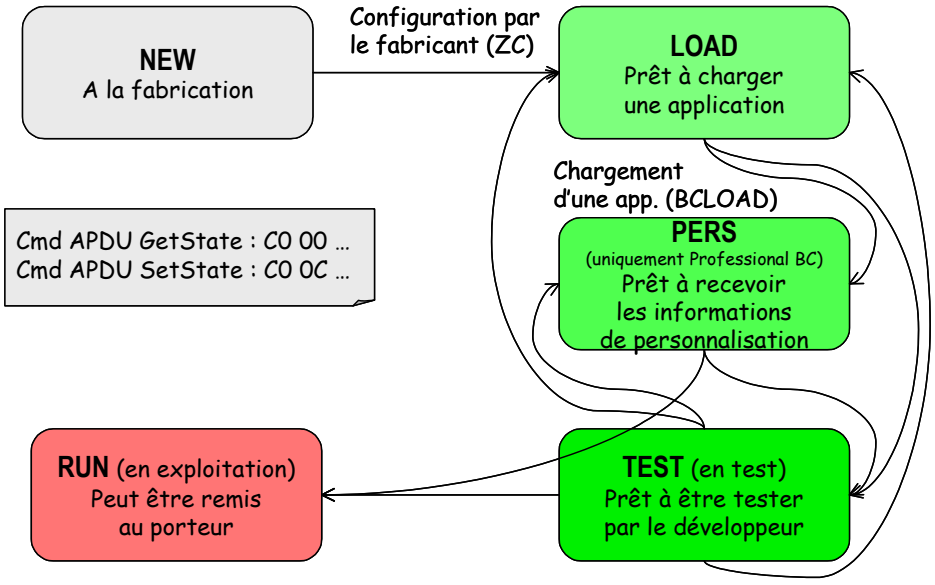

Figure 3 : Etats d'une BasicCard Professional

1.3 Produits

La société allemande ZeitControl, fabricant de la BasicCard, propose 3 gammes de cartes BasicCard. Ces cartes sont commercialisées vierges d'impression sur leur face ou alors personnalisation des faces par impression (Thermosublimation 4 couleurs ou Offset 4

couleurs). Cette société commerciale aussi des stations de personnalisation et de test de BasicCard.
Les 3 gammes, intitulées Compact, Enhanced, Professional, diffèrent par leurs capacités techniques (protocoles, taille de la mémoire RAM et EEPROM, puissance du processeur, fonctions cryptographiques). Le système d'exploitation de ces cartes est installé en grande partie en ROM (17Ko). Certaines fonctions (conversion de flottants vers chaines) sont en EEPROM.

1.3.1 Compact ZC1.1

La carte Compact, dont le nom de produit est ZC1.1, est une carte au format module SIM du GSM11.11. Elle ne supporte que le protocole de communication T=1 (ISO/IEC 7816-3) et la structure des commandes et réponses ISO/IEC 7816-4. Le chiffrage automatique des commande utilise l'algorithme à clé symétrique SG-LFSR (Shrinking Generator – Linear Feedback Shift Register) avec des clés de 64 bits. La carte Compact n'a pas de systèmes de fichier et ne supporte pas les calculs flottants. Sa capacité de RAM est de 256 octets et sa capacité d'EEPROM de 1Kilooctet

1.3.2 Enhanced ZC3.x

La gamme Enhanced, dont le nom de produit est préfixé par ZC3., est commercialisée depuis 1998. Ces cartes ne supportent elles aussi que le protocole de communication T=1 (ISO/IEC 7816-3) et la structure des commandes et réponses ISO/IEC 7816-4. Le chiffrage automatique des commande utilise l'algorithme à clé symétrique DES avec des clés de 56 bits ou 112 bits (Triple DES). Les cartes Enhanced ont un systèmes de fichier hiérarchique (Section 1.5.3) et supporte le calcul flottant à la norme IEEE. Leur capacité memoire est de 256 octets de RAM et une capacité d'EEPROM variant de 2 à 16 Kilooctets.

1.3.3 Professional ZC4.x, ZC5.x

La gamme Professional, dont le nom de produit est préfixé par ZC4. ou ZC5., est commercialisée depuis 2001. Ces cartes supportent les deux protocoles de communication T=0 et T=1 (ISO/IEC 7816-3) et la structure des commandes et réponses ISO/IEC 7816-4. Les cartes Professional ont aussi un systèmes de fichier hiérarchique (Section 1.5.3) et supporte le calcul flottant à la norme IEEE. Leur capacité memoire est de 1024 octets de RAM et une capacité d'EEPROM variant de 16 à 30 Kilooctets. Le chiffrage automatique des commande utilise l'algorithme à clé symétrique DES avec de 56 bits ou 112 bits (Triple DES). ou AES avec des clés de 128 bits. Les cartes Professional sont pourvues également d'algorithmes à clé asymétrique : RSA (clé de 1024 bits) et/ou Elliptic Curve (clé de 167 bits), et de l'algorithme de hachage SHA-1 (FIPS 180-1). Les cartes Professional possèdent un état supplémentaire PERS par rapport aux BasicCards Compact et Enhanced correspondant à l'état de pré-personnalisation (Section 1.2). Les cartes de la gamme Professional sont également annoncées comme 4 fois plus rapide que celle de la gamme Enhanced.

1.3.4 Professional ZC6.x

TODO

1.4Le langage ZC-Basic

Le langage ZC-Basic est un dialecte du Basic, propriété de la seule société Zeit-Control. Un source ZC-Basic doit être compilé vers un langage intermédiaire, le P-Code, pour être exécuter par un interpréteur de P-Code. Ce même P-Code peut être destiné à être exécuté par

interprète de P-Code d'une BasicCard ou bien par ceux des applications terminal (à quelques restrictions près). Cette section présente les différents aspects du ZC-Basic.

1.4.1 Déclaration des variables

La déclaration de variables specifie la durée de vie, la portée, le type et la valeur d'initialisation. En ZC-Basic, la déclaration des variables peut être implicite et explicite. Cependant l'instruction option explicit (que nous recommandons d'utiliser) interdit les déclarations implicites pour rendre détectable, lors de la compilation, les erreurs de frappe dans les identifiants.

1.4.2 Portée et Durée de vie des variables

Les variables peuvent être temporaires ou permanentes. La durée de vie des variables temporaire sera inférieure à la durée de la connexion de la carte dans le lecteur. Les variables permanentes conservent leurs valeurs courantes d'une connexion à l'autre.
Les variables diffèrent également par leur portée. La portée définit la visibilité des variables depuis les sections de code. Quand la portée est globale à l'application, n'importe section de code peut utiliser la variable. Quand la portée est locale à une fonction, procédure ou commande, la valeur de la variable n'est accessible que par les instructions du corps de la fonction,procédure ou commande.
L'instruction de déclaration comporte une classe de stockage (Eeprom, Public, Static, Private) définissant la durée de vie, la portée et la réinitialisation de la variable déclarée.

- Eeprom spécifie que la variable est permanente. La variable est allouée en mémoire non volatile de la carte et par conséquent sa dernière valeur est retrouvée à la prochaine reconnexion. La variable est initialisée une seule fois lors du chargement de l'application dans la carte.
- Public spécifie que la variabvle est temporaire et que sa portée est globale à l'application et donc accessible depuis toute fonction, procédure, commande. Sa valeur est réinitialisée à chaque reset de la carte et donc elle est conservée entre les invocations de commandes.
- Static spécifie que la variable est temporaire et que la portée est locale à la fonction, procédure ou commande dans laquelle elle est déclarée. Sa valeur est réinitialisée au reset de la carte et donc elle est conservée entre les invocations de commandes.
- Private spécifie que la variable est temporaire et que la portée est locale à la fonction, procédure ou commande dans laquelle elle est déclarée. Sa valeur est réinitialisée à chaque invocation de la fonction, procédure ou commande. Les variables déclarées implicitement sont de cette nature.
- Le Tableau 1 résume la portée et la durée de vie des variables

Nature	Déclaration	Durée de Vie	Portée	Initialisation
Eeprom	Explicite	Permanente	Globale	A l'installation
Public	Explicite	Temporaire	Globale	Au Reset
Static	Explicite	Temporaire	Locale	Au Reset
Private	Explicite ou implicite	Temporaire	Locale	A l'invocation

Tableau 1 : Nature des déclarations de variables

1.4.3 Types primitifs

ZC-Basic propose une variété de types entiers, flottants et chaînes pour optimiser l'utilisation de la mémoire de la carte. Le type peut être déclaré avec l'instruction As ou bien en suffixant la variable d'un caractère spécial. Le Tableau 2 décrit les différents types primitifs en ZC-Basic.

Type	Suffixe	Description	Occupation mémoire
Byte	@	entier 8 bits non signé	1
Integer	%	entier 16 bits signé	2
Long	&	entier 32 bits signés	4
Single	!	flottant IEEE 32 bits	4
String	$	chaîne variable de (<254) car.	3 + Nb caractères
String*N	Sans	chaîne fixe de N caractères N<255	N

Tableau 2 : Types primitifs en ZC-Basic

Les instructions DefByte, DefInt, DefLng, DefSng, DefString sont utilisées pour définir le type des variables implicites en fonction de la première lettre de leur identifiant. Ce sont les dernières instructions Def qui sont utilisés pour le typage implicite. L'instruction est DefInt A-Z et elle définit toute variable implicite comme une variable entière.

```
DefByte I-R ' toute variable commençant par une letter de I à R est de
type Byte
Private Iter ' type Byte
Public Radius! = 1 ' type Single, affectation avec conversion
Public Pistr$ = "3.1416"
Public Pi As Single = Val!(Pistr$)
Eeprom S1 As String*5 = "ABC" ' complété avec des octets NULL
Public S2 As String*3 = &H81, &H82, &H83
Private S3 As String*7 = 3, 4, "XYZ" ’ = 3, 4, 88, 89, 90, 0, 0
Eeprom CustomerName$ = ""      ' We don’t know customer’s name yet
Eeprom Balance& = 500          ' Free 5-euro bonus for new members
For J=1 to 8 Do J=J+1 ' J est declare implicitement et est de type
Byte
```

Listing 1 : Déclarations de variables en ZC-Basic

1.4.4 Types structurés

ZC-Basic supporte la définition de type structuré encore appelé type utilisateur (user-defined). Une variable d'un type structuré comporte un ou plusieurs champs membres de type primitif, structuré ou tableau. Cependant la taille totale des membres ne doit pas excéder 254 caractères.

Un type structuré est défini au moyen de l'instruction suivante :

Type *identifiantDuType*
identifiantDuMembre [**As** *type*] [**,***identifiantDuMembre* [**As** *type*], **. . .**]
identifiantDuMembre [**As** *type*] [**,** *member-name* [**As** *type*], **. . .**]
. . .
End Type

L'assignation d'une variable de type structuré peut être directe ou alors via une liste de valeurs affectées aux membre dans leur ordre de definition. L'accès aux membres d'une variable se fait (comme en C ou en Java) avec l'opérateur . (dot). L'opérateur de comparaison = teste l'égalité membre à menbre de 2 variables de même type structuré. L'opérateur <> teste l'inégalité d'un des champs. Comme il n'existe pas d'ordre défini sur un type utilisateur, les opérateurs <, <=, >, >= ne sont pas autorisés.

```
Type tDate: Jour&: Mois& : Annee% : End Type
Type tOperation
  Montant As Int ' declaration explicite
  Date As tDate
  Commentaire$
End Type

Sub JoursEcoules (D1 As Date, D2 As Date)
Montant = 0.BottomRight.X! - R.TopLeft.X!
Height! = R.BottomRight.Y! - R.TopLeft.Y!
R.Area = Width! * Height!
End Sub

Public O As Operation = -100,31,12,1999, "Carburant" ' Date =
(31,12,99)
```

1.4.5 Tableaux Multidimensionnels

ZC-Basic supporte les tableaux multidimensionnels pour les 4 classes de stockage. Un tableau regroupe des éléments de même type primitif ou d'un même type structuré. Ces éléments sont accessibles avec un indice pour chaque dimension du tableau. Le nombre de dimension est au maximun 32 et un tableau ne peut contenir plus de 16 Koctets de données. Pour les BasicCards Compact et Enhanced, les bornes d'un tableau sont contraintes : la borne inférieure doit être comprise entre –32 et 31, et la borne superieur ne peut dépasser la borne superieur plus 1023.
Un tableau est soit de taille redimentionnable (Dymanic) soit de taille fixe (Fixed). Les bornes des tableaux de taille fixe sont constantes et ne peuvent être redefinies. Les bornes des tableaux de taille dynamique peuvent être redefinies à tout moment avec des expressions entières au moyen de l'instruction ReDim. Cependant le nombre de dimension reste celui fixé à la definition.
Un tableau fixe est déclaré avec l'instruction suivante :

classeDeStockage identifiantDuTableau (*bornes* [, *bornes*, . . .]) [**As** *type*]

Un tableau dynamique est déclaré avec l'instruction suivante :

classeDeStockage **Dynamic** *identifiantDuTableau* (*bornes* [, *bornes*, . . .]) [**As** *type*]

La valeur des bornes peut être omise à la déclaration pour différer l'allocation d'éléments au prochain redimensionnement avec l'instruction Redim.
Les éléments sont par défaut initialisés à zéro pour les numériques et à chaine vide pour les chaines de caractère.
L'instruction Erase remet les éléments d'un tableau fixe à zero et libère l'espace memoire alloué quand il s'agit d'un tableau dynamique.

```
DefByte A-B
Dim A1() 'Tableau dynamique à une dimension de Byte
Dim A2(,)'Tableau dynamique à deux dimensions de Byte
Dim A3(,,) 'Tableau dynamique à trois dimensions de Byte

Eeprom OP(10) As Operation ' Tableau fixe d'Operation
..
For I = 1 To 10
  If OP(I).Montant <0 : TotalDepense=TotalDepense+OP(I).Montant
  Next I
End
```

1.4.6 Représentation mémoire

Le Tableau 2 donne l'occupation mémoire consommée par les variables primitives.
Une chaîne de taille variable est implémentée par 1 pointeur 16 bits référençant une zone allouée pour les caractères. Le premier octet de cette zone contient la longueur de la chaîne et les octets suivants contiennent les caractères de la chaîne.
Un tableau est implémenté par une structure mémoire contenant l'adresse (16 bits) de la zone contenant les éléments, la taille d'un élément (8 bits), un drapeau (1 bit) indiquant si le tableau est fixe (=0) ou dynamique (=1), le nombre de dimension (7 bits), et pour chaque dimension le couple de bornes inferieure (6 bits) et superieures (10 bits).

1.4.7 Expressions

1.4.8 Affectations

L'affectation des variables se fait avec l'instruction Let (dont le mot clé est optionnel).

[Let] var = expression

La variable peut être de type primitif ou structuré, peut être d'élément d'un tableau ou le membre d'une variable de type structuré. Dans le cas de variables entières, l'expression doit être de type entier. Dans le cas de variables flottantes, l'expression doit être de type flottant ou entier. Dans le cas de variables structurées, l'affection ne peut se faire qu'à partir d'une expression du même type qui peut être une variable ou une liste de valeurs affectables membre à membre.
Pour les variables chaines (String et String*N), il existe les 5 instructions d'affectation suivantes :

*[**Let**] identifiantDeChaine = expressionDeChaine*

*[**Let**] **Mid$** (identifiantDeChaine, position [, longueur]) = expressionDeChaine*

*[**Let**] identifiantDeChaine (position) = expressionDeChaine*

*[**Let**] **Left$** (identifiantDeChaine, longueur) = expressionDeChaine*

*[**Let**] **Right$** (identifiantDeChaine, longueur) = expressionDeChaine*

Mid$ écrit les *longueur* caractères de la chaine *identifiantDeChaine* à partir de la position *position* (démarrant à 1) avec les caractères de l'expression. L'instruction *identifiantDeChaine (position)* est équivalente à *Mid$(identifiantDeChaine,position,1)* et n'écrit qu'un seul caractère. Left$ écrit les *longueur* premiers caractères de la chaine *identifiantDeChaine* avec les caractères de l'expression. Right$ écrit les *longueur* derniers caractères de la chaine *identifiantDeChaine* avec les caractères de l'expression.

1.4.9 Fiabilisation de l'affectation

Le temps d'écriture de données en EEPROM (6 millisecondes) est relativement long par rapport au temps d'écriture en RAM. Il peut donc se produire une coupure électrique (arrachement de la carte, coupure du terminal, …) avec que l'écriture en EEPROM soit complète. Il peut être nécessaire dans les applications sensibles que l'information soit toujours cohérente c'est à dire qu'on puisse toujours récupérer l'information présente avant l'affection ou l'information après l'affection. L'incohérence de l'information est recherchée par les attaquants (pirates) pour recupérer des informations confidentielles ou modifier le comportement de l'application présente sur la carte.
Contrairement à la JavaCard qui dispose de mécanismes transactionnels, la fiabilisation de l'application et de ces affections en EEPROM, est à la charge du développeur. Une solution

est d'utiliser une variable ombre qui contient la dernière affectation et un drapeau qui identique que l'écriture était en cours au moment de la coupure.
Le Listing 2 fiabilise le PME dans lequel la ressource sensible est le solde du PME.

```
Eeprom Balance As Long
Eeprom ShadowBalance As Long
Eeprom Uncommited = False

Rem  Instruction exécutée à chaque demarrage (Reset)
If Uncommited Then     ' l'affection Balance = ShadowBalance
                       ' n'a pas été complément terminée
  Balance = ShadowBalance  ' l'affection Balance = ShadowBalance
                           ' de nouveau retentée
  Uncommited = False
End If

Sub ChangeBalance (NewBalance As Long)
  ShadowBalance = NewBalance
  Uncommited = True
  Balance = ShadowBalance
  Uncommited = False       ' l'affection Balance = ShadowBalance
                           ' est complétement terminée
End Sub

Command &H80 &H22 Debit (Amount As Long)
  If Balance < Amount Then SW1SW2 = InsufficientCredit : Exit
  ChangeBalance(Balance – Amount)
End Command
```

Listing 2 : Fiabilisation de l'affectation avec une BasicCard

1.4.10 Structures de contrôle

Le langage ZC-Basic propose à peu près toutes les structures de contrôle classiques (test, boucle, branchement)
Cependant le traitement des exceptions est absent du langage.
A partir de 27

1.4.11 Fonctions et Procédures

1.4.12 Passage des paramètres

1.4.13 Commandes

Remarquons que cette dernière forme permet d'utiliser ZC-Basic pour développer des applications terminal pour des cartes autres que BasicCard (exemple : JavaCard, EMV, GSM …). Pour cela, vous pouvez déclarer la commande SendAPDU dont vous renseignerez CLA, INS, P1, P2, Data, Le à l'exécution comme le présente le Listing 3 : Envoi d'APDU "brute" vers une carte, pour envoyer et recevoir des APDU « brutes » vers la carte.

```
Declare Command SendAPDU(Data as String)
Private PCLA As Byte = &HC8
Private PINS  As Byte = &HA0
Private PP1  As Byte = &H11
Private PP2  As Byte = &H22
Private PData$ = "ABC" ' equivalent à &H40, &H41, &H42
```

```
Private PLe  As Byte = 45
Call WaitForCard() : ResetCard : Call CheckSW1SW2()
Call SendAPDU(CLA=PCLA,INS=PINS,P1=PP1+10,P2=PP1,PData$,Le=PLe)
Print SW1SW2
Print ValH(PData$) ' les données de la réponse sont dans Pdata$
```

Listing 3 : Envoi d'APDU "brute" vers une carte

Il est possible de supprimer l'octet Le de la commande APDU avec la Disable Le.

1.4.14 Directives de Précompilation

Le langage ZC-Basic offre des directives de pré-compilation qui permet d'inclure des définitions de procédures externes et de réaliser des compilations conditionnelles.

1.4.14.1 Inclusion de fichiers

L'inclusion de fichier permet d'organiser et de modulariser votre projet Basiccard. Par exemple, une bonne organisation de projet sépare la déclaration des commandes et des clés du source de l'application carte et des sources des applications terminal.
La directive #Include filename inclut un source ZC-Basic (par exemple la déclaration des commandes et des clés) dans le source à compiler.

La directive #Library filename charge la bibliothèque de fonctions ou commandes dans le programme. Cette directive est utilisée par exemple pour ajouter les fonctions cryptographiques optionnelles.

1.4.14.2 Compilation conditionnelle

La compilation conditionnelle permet de produire des programmes P-Code différents à partir du même fichier source. Une utilisation classique est d'avoir des sections de sources différentes en fonction des caractéristiques de la carte a laquelle sera destinée le P-Code. Ces sections permettent par exemple, en compte des différences de capacité mémoire, d'algorithmes cryptographiques disponibles, …
Le bloc de directive #If , #ElseIf #Else, #EndIf spécifie les conditions et les sections de source alternatives. Les conditions sont des expressions booléennes utilisant des constantes de l'application. Ces constantes peuvent être aussi définies avec l'option –D du compilateur ZCMBASIC.

Les directives #IfDef et #ElseIfDef, #IfNotDef et #ElseIfNotDef testent la condition est une constante est définie ou non.

Les constantes prédéfinies TerminalProgram CompactBasicCard EnhancedBasicCard ProfessionalBasicCard, CardMajorVersion et CardMinorVersion renseignent sur le type de carte et sur leur version pour les compilations conditionnelles.

Enfin, les directives #Message message et #Error message affichent à la console le message suivant lors de la compilation. Cependant la directive#Error arrête la compilation.

Le Listing 4 présente l'utilisation de plusieurs directives de précompilation.

```
#IfDef CompactBasicCard
#Error Ce programme nécessite le système de fichiers !
#ElseIf MaxLineLength > 80
#Error La constante MaxLineLength est trop grand (max 80)
#Else
#Message MaxLineLength est correcte
#EndIf
```

Listing 4: Directives de précompilation BasicCard

1.4.14.3 Directives Carte

Les directives carte agissent sur les paramètres d'utilisation de la carte.

La directive #State { LOAD | PERS | TEST | RUN } place la carte dans l'état spécifié lors du chargement de l'application sur la carte. Cette directive est équivalente à l'option –S du compilateur ZCMBASIC.

La directive #Files nFiles limite le nombre (compris entre 0 et 16) de fichiers ouverts simultanement afin de limiter l'usage de la RAM (6 * nFiles + 7 octets). Si nFiles est égal à zéro, le système de fichier n'est pas installé.

La directive #Stack stack-size spécifie la taille maximal de la pile (stack) de l'interprète de P-Code.

La directive #BWT n precise le Block Waiting Time du protocole T=1 dans l'ATR de la carte. La value n qui doit être une puissance de 2 comprise entre 1 et 512 en dixième de seconde, représente le delai maximun que prend l'exécution d'une commande. Passer ce delai, le lecteur de carte retourne une erreur (représente par le mot d'état swCardTimedOut). La valeur par défaut est 16 soit 1,6 seconde pour la Compact et 128 soit 12,8 secondes pour l'Enhanced et la Professional. Le développeur doit ajuster ce paramêtre en fonction du temps maximum de traitement de toutes les commandes. Un BWT trop court provoque un timeout pour les commandes trop longue. Attention, cet ajustement n'est valide que pour une famille utilisant le même cadencement de processeur.

1.4.15 Mots d'état

Le mot d'état (SW : Status Word) d'une réponse APDU représente l'issue de l'exécution de la commande

Coté application carte, le mot d'état de la réponse est affecté aux variables SW1SW2, SW1, SW2. Par défaut, lors que la commande s'est correctement exécutée, les valeurs de mots d'état retournées sont SW1SW2= swCommandOK (&H9000) ou Sw1= sw1LeWarning (&H61). Cette dernière valeur, sw1LeWarning, est un avertissement pour signaler que le nombre d'octets retournés n'est pas égal au champ Le de la commande qui indique le nombre d'octets de données prévus dans la réponse. Cette valeur est retournée par les commandes retournant des données de taille variable non connue à l'avance comme la commande GET APPLICATION (&HC0 &H0E &H00 &H00 &HFE) ou les commandes avec un paramètre de type String.

En cas d'erreur applicative, l'application carte peut affecter une valeur de mot d'état non réservée à SW1SW2. Le Listing 5 illustre les retours de mots d'état quand l'application n'est pas personnalisée et quand le solde est insuffisant lors d'un débit. Lors que SW1SW2= swCommandOK (&H9000) ou Sw1= sw1LeWarning (&H61), la réponse ne contient que le mot d'état et pas de données.

```
Const swNotPersonalised     = &H6B00
Const swInsufficientFunds   = &H6B01
…
Eeprom Personalised = False
Eeprom Balance As Long
Eeprom CustomerName$
…
Command &H80 &H22 Debit (Amount As Long, NewBalance As Long)
  If Not Personalised Then SW1SW2 = swNotPersonalised : Exit
  If Amount > Balance Then SW1SW2 = swInsufficientFunds : Exit
```

```
  ChangeBalance(Balance - Amount)
  NewBalance = Balance
End Command

Command &H80 &H26 GetCustomerName (Name As String)
  If Not Personalised Then SW1SW2 = swNotPersonalised : Exit
  Name = CustomerName
End Command
```

Listing 5 : Retour de mots d'état par la carte

Coté application terminal, l'appel d'une commande affecte les variables SW1SW2, SW1, SW2 du mot d'état de la réponse. L'application terminal peut tester leur valeur avant de poursuivre l'exécution. Ces variables peuvent aussi contenir des valeurs indiquant un disfonctionnement de la communication entre l'application et le lecteur (swNoCardReader, swCardReaderError, swComPortBusy …) et entre la lecteur et la carte (swT1Error, swCardError, …). Le Listing 6 présente le traitement du mot d'état par l'application terminal.
La signification de l'ensemble des valeurs des mots d'état est détaillée dans [BCM section 7.6.1] et la fonction SWName$() retourne la chaîne correspondante à la constante prédéfinie du mot d'état.

```
Call GetInfo(CurrentBalance,CustName)
If SW1SW2=swNotPersonalised Then
 Print "Card not personalised" : Exit
Elsif SW1SW2<>swCommandOK Or SW1<>sw1LeWarning Then
 Print "Error" SW1SW2 SWName$() : Exit
Endif
Print "name: " CustName " balance: " CurrentBalance

Call Debit(1000,NewBalance)
If SW1SW2=swNotPersonalised Then
 Print "Card not personalised" : Exit
Elsif SW1SW2=swInsufficientFunds then
 Print "Insufficient Funds" : Exit
Elsif SW1SW2<>swCommandOK Or SW1<>sw1LeWarning Then
 Print "Error" SW1SW2 SWName$() : Exit
Endif
Print "new balance : " NewBalance
```

Listing 6 : Traitement du retour d'un mot d'état par le terminal

1.4.16 Correspondance Command et APDU

La connaissance de la structure des commandes et réponses APDU entre l'application terminal et l'application carte, n'est pas requise quand les deux applications sont programmées en ZC-Basic. Cependant, elle est nécessaire quand d'autres langages (Delphi, Java, C#, VB) sont utilisés pour développer l'application terminal comme nous le verrons dans le chapitre Error: Reference source not found sur OpenCard Framework.
L'appel d'une commande BasicCard est encodée sous la forme d'une commande APDU contenant l'octet de classe CLA, l'octet d'instruction INS et les valeurs des paramètres de la commande et d'une réponse APDU contenant les valeurs éventuellement modifiées des paramètres et le mot d'état. Les octets P1 et P2 de la commande APDU sont inutilisés. L'octet Lc contient le nombre d'octets des paramètres. L'octet Le qui indique le nombre d'octets attendus dans la réponse, est égal à Lc que tous les paramètres sont de taille fixe. L'octet Le vaut &HFE quand le dernier paramètre est de taille variable.
L'ordre des octets des paramètres doit être « Big Endian », c'est à dire l'octet de poids fort à l'adresse de poids fort. Rappelons que l'encodage « Big-Endian » est utilisé par Java et la

plupart des processeurs non Intel et que l'ordre Little-Endian est l'ordre utilisé par les processeurs Intel x86 et Pentium.
A REVOIR A PARTIR DES TRACES

Paramêtre encodés dans la commande et la réponse
Encodage des types et des tableaux

```
Declare Command &H80 &H00 PersonaliseCard (
         Balance As Long, PIN As String*4, Name$)
Declare Command &H80 &H02 GetCardInfo (
         Balance As Long, RemainingRetry As Byte, Name$)
Declare Command &H80 &H04 GetBalance (
         Balance As Long)
Declare Command &H80 &H06 VerifyPIN (
         TestPIN As String*4)
Declare Command &H80 &H08 Debit (
         Amount As Long, NewBalance As Long)
Declare Command &H80 &H10 Credit(
         Amount As Long, NewBalance As Long)
Declare Command &H80 &H12 LastOperations (
         Operations As Long[10], Dates As Long[10])
```

Voir fichiers envoyé sur les correspondance
Listing de trace du Listing MOTETATTERM
Chiffrage des APDU (EnableEncryption/DisableEncryption)
Cet encodage se corse quand les échange entre l'application terminal et l'application carte sont chiffrés
EXEMPLE

1.5 La bibliothèque de fonctions ZC-Basic

1.5.1 Fonctions spécifiques à la carte

Déclaration d'un autre ATR

1.5.2 Fonctions Cryptographiques

Générations des clés
Dispositive materiel pour la BasicCard Professional
Chiffrage des échanges APDU

1.5.3 Systèmes de fichiers

1.5.4 Fonctions spécifiques au terminal

Fichiers

1.5.5 Journal de traces

Open Log File nom_fichier_journal
Close Log File

1.6 Commandes BasicCard Prédéfinies

Cf 7.7
CLA=C0
INS= (disponible selon les états)
•

Commande	INS	Signification
GET STATE	00	Get the state and version of the card
EEPROM SIZE	02	Get the address and length of EEPROM
CLEAR EEPROM	04	Set specified bytes to FF
WRITE EEPROM	06	Load data into EEPROM
READ EEPROM	08	Read data from EEPROM[1]
EEPROM CRC	0A	Calculate CRC over a specified EEPROM address range
SET STATE	0C	Set the state of the card
GET APPLICATION ID	0E	Get the Application ID string
START ENCRYPTION	10	Start automatic encryption of cmd/resp. data
•END ENCRYPTION	12	End automatic encryption
•ECHO	14	Echo the commandata
•ASSIGN NAD	16	Assign a Node Address to the card
•FILE IO	18	

1.7La machine virtuelle Basiccard

1.7.1 L'interprète P-Code

Nous avons vu que le compilateur ZCMBASIC transforme le source ZC-Basic en P-Code. Le P-Code est le langage d'instruction de la machine virtuelle (ou interprète P-Code) de la BasicCard. Ces instructions P-Code sont pour la plupart commune à l'interprète de la BasicCard pour l'application carte et à l'interprète du terminal pour l'application terminal.
L'interprète P-Code utilise 3 registres : PC (Program Counter), SP (Stack Pointer), FP (Frame Pointer). Le registre 16 bits PC pointe l'instruction P-Code à exécuter. Le registre SP représente le sommet de pile (Stack) à partir duquel des emplacements mémoire sont libres. Le registre FP représente la zone de travail d'une procédure dans la pile. Il faut noter que SP et FP sont des registres 8 bits sur les BasicCards Compact etEnhanced qui n'ont que 256 octets de RAM. Ce sont des registres 16 bits sur la BasicCard Professional et sur le terminal qui disposent de plus de RAM (1Ko pour la BasicCard Professional et au maximun 64 Ko pour le terminal).
La pile (stack) contient quatre types de données. Les paramètres de la commande APDU réçue du port d'entrée sortie sont localisés dans le bas de la pile. Les paramêtres d'une procédure et l'adresse de retour sont empilés avant l'appel de saut de procédure. Les variables locales déclarées Private (Section 1.4.1), de la procédure constitue la frame de la procédure référencée par le registre FP. Et enfin la pile contient tous les calculs intermédiaires des expressions des procédures.

[1] Cette commande qui ne s'applique que dans l'état TEST, permet de « dumper » la mémoire de la carte en vue d'être cloner !

Chaque instruction P-Code est constitué d'un code opération (OpCode) et de zéro ou un paramètre (adresse, entier, flottant, chaîne). La plupart de ces instructions utilise intensivement la pile en dépilant des opérandes et en réempilant leur résultat. Les appels au système comme l'accès aux fichiers, … passe aussi par une instruction (OpCode=06) qui est complété d'un code d'appel système (SysCode).

1.7.2 Organisation mémoire

La mémoire d'une BasicCard est organisée en plusieurs régions en mémoire volatile et en mémoire non volatile.
Les régions en mémoire volatiles (RAM) sont :

- RAMSYS contient des données nécessaires au système. Elle occupe par exemple 71 octets pour la BC Compact et 107 octets pour la BC Enhanced sur les 256 octets disponibles.
- STACK contient la pile d'exécution.
- RAMDATA contient les variables globales temporaires de l'application. Ces variables ZC-Basic sont déclarées Public et Static (Section 1.4.1).
- RAMHEAP sert à l'allocation des données temporaires de taille dynamique comme les tableaux dynamiques et les chaînes de longueur variable.
- FILEINFO est utilisé pour les fichiers ouverts
- FRAME sont des sous regions de STACK indiquant la base des variables locales des procédures

Les régions en mémoire non volatiles (EEPROM) sont :

- EEPSYS contient des données non volatiles nécessaires au système. Elle occupe par exemple 35 octets pour la BC Compact et 429 octets pour la BC Enhanced.
- STRVAL contient le code de conversion flottant vers chaîne pour les BC Enhanced
- CMDTAB contient la table des descripteurs de commandes.
- PCODE contient les instructions P-Code du programme.
- STRCON contient les chaînes de caractère constantes.
- KEYTAB contient la table des clés de chiffrement/déchiffrement
- EEPDATA contient les variables ZC-Basic permanentes, déclarées Eeprom, qui sont de longueur fixe (entiers, flottant, chaîne de longueur fixe).
- EEPHEAP sert à l'allocation des fichiers et de données permanentes qui sont les variables ZC-Basic déclarées Eeprom (Section 1.4.1). Cette zone sert à stocker des données temporaires quand la région EEPRAM n'est pas assez vaste. Il faut noter que les temps d'écriture sont ralentis par rapport aux écritures en RAM.
- Plug-In contient des bibliothèques additionnelles de fonctions.

La taille de ces régions est assignée à la compilation.
Le programme terminal est organisé en deux segments CODE et DATA. Le segment CODE contient seulement la région PCODE. Le segment DATA contient les autres régions en mémoire volatile et non volatile. On y retrouvera les régions STACK, RAMSYS, RAMDATA , RAMHEAP et STRCOM en mémoire volatile et les deux régions EEPDATA et EEPHEAP en mémoire « non volatile ». La mémoire non volatile de programme terminal correspond à la sauvegarde des variables permanentes déclarées Eeprom, dans le fichier IMG du programme terminal. Cette sauvegarde est réalisée par l'instruction ZC-Basic Write Eeprom et à la terminaison du programme (quand le simulateur ZCMSIM est lancé avec l'option –W ou après confirmation pour le déboggeur ZCMDTERM).
L'organisation mémoire d'une application est décrite par les fichiers LST et MAP que la compilation génère de manière optionnelle.

1.8 Exemple d'application

Nous allons maintenant reprendre l'application de Porte Monnaie Electronique (PME) décrite à la section Error: Reference source not found pour la portée en ZC-Basic. Rappelons que les fonctionnalités de ce PME sont

- La personnalisation avec le nom du porteur, le code PIN et le solde initial.
- La consultation du nom du porteur, du solde, du nombre de ré-essais restants
- La validation d'un code PIN avec blocage du PME au bout de 3 ré-essais (retry),
- le débit d'un montant (montant) à partir du solde (balance) après validation d'un PIN code,
- la rechargement ou crédit du solde après validation d'un code PIN et authentification forte du créditeur.

Les 3 applications terminal écrites en ZC-Basic permettent l'émission de la carte, à sa recharge et à son débit chez un commerçant.

Nous invitons le lecteur à regarder les exemples DebitCredit et BalanceReader de l'environnement de développement BasicCard.

1.8.1 epurse.def

```
Declare Command &H80 &H00 PersonaliseCard (
         Balance As Long, PIN As String*4, Name$)
Declare Command &H80 &H02 GetCardInfo (
         Balance As Long, RemainingRetry As Byte, Name$)
Declare Command &H80 &H04 GetBalance (
         Balance As Long)
Declare Command &H80 &H06 VerifyPIN (
         TestPIN As String*4)
Declare Command &H80 &H08 Debit (
         Amount As Long, NewBalance As Long)
Declare Command &H80 &H10 Credit(
         Amount As Long, NewBalance As Long)
Declare Command &H80 &H12 LastOperations (
         Operations As Long[10], Dates As Long[10])
```

1.8.2 epurse.bas

1.8.3 issuer.bas

1.8.4 bearer.bas

1.9 Exercices

Les opérations du PME doivent être journalisées afin d'apporter une preuve en cas de contestation ou d'offrir la possibilité de lister les opérations. La consultation de l'historique se fait après vérification du PIN Code. Comme les ressources mémoire de la carte sont limitées, l'historique est gérée dans un journal à gestion circulaire. C'est à dire, les plus anciennes opérations sont effacées par les plus récentes.

Ajoutez un historique des opérations au PME en gérant un tableau circulaire.

Ajoutez un historique des opérations au PME en gérant un fichier de manière circulaire.

Ecrivez une application terminal pour consulter l'historique des opérations.

Développez un petit utilitaire terminal en ZC-Basic qui permet d'envoyer des APDU brutes (cf. Listing 3) vers un carte insérée (à la manière de l'outil APDUTool).