asMSX

Cross assembleur pour MSX

v.0.18.5

– Edition 2018 –

[août 2018]

Index

1. Introduction

1.1. Description

AsMSX est un assembleur pour le processeur Z80, développé à l'origine par Eduardo A. Robsy Petrus, qui intègre un certain nombre de caractéristiques facilitant grandement la programmation sur les ordinateurs MSX. Il s'agit d'un cross-assembleur (c'est-à-dire qu'il fonctionne sur une machine différente que MSX mais produit un code exécutable sur MSX) qui fonctionne sur PC avec un système d'exploitation Windows ou Linux.

Grâce à la puissance des ordinateurs actuels, l'assemblage est pratiquement instantané, ce qui constitue un avantage considérable par rapport aux assembleurs MSX natifs. De même, il n'y a aucune restriction sur la taille du code source et il peut être modifié avec n'importe quel éditeur ou traitement de texte.

La première version de asMSX (v.0.01) a été publiée en 2000, et n'était alors qu'une esquisse de ce qu'est dorénavant asMSX. Les erreurs de cette première version ont été corrigées, des améliorations ont été apportées et des caractéristiques plus avancées mises en place.

1.2. Caractéristiques

Parmi les points forts d'asMSX, on peut citer les suivants :

- supporte toutes les instructions officielles du Z80 avec la syntaxe de ZILOG.
- assemble toutes les instructions non-officielles documentées par Sean Young
- comprend la syntaxe de la norme Z80 (accumulateur implicite)
- compatible avec différents systèmes de numération (décimal, hexadécimal, octal et
- binaire)
- opérations arithmétiques et logiques, y compris au niveau des codes source
- prise en charge des nombres décimaux à virgule flottante, converti en virgule fixe
- fonctions mathématiques réelles : trigonométriques, puissances, etc.
- accepte de multiples fichiers via des inclusions
- inclusion de tout ou partie de ces fichiers binaires
- étiquettes locales ou générales pour le code source
- routines du Bios du MSX prédéfinies avec leur nom officiel
- génération automatisée de fichiers binaires exécutables depuis le Basic
- production automatisée de fichiers ROM
- supporte 4 types distincts de megaROM : Konami, Konami SCC, ASCII 8 bits et 16 bits
- utilisation de variables numériques à l'intérieur de l'assembleur
- assemblage de fichiers COM (pour MSX-DOS)
- export de la table des symboles
- impression de textes indicatifs depuis le code
- intégration avec le débogueur de BlueMSX
- amélioration de l'assemblage conditionnel
- génération de fichiers CAS ou WAV pouvant être chargés sur MSX

1.3. Justification

Y-a-t-il vraiment besoin d'un autre assembleur pour Z80 ? Probablement pas. Les annexes comprennent une liste d'assembleurs pour MSX, tant cross-assembleurs qu'assembleurs natifs, ainsi que leur disponibilité. Parmi eux, il existe d'excellents outils de développement qui ont été utilisés par de nombreux programmeurs.

AsMSX a pour objectif de fournir un assembleur totalement compatible avec les instructions du Z80, flexible et confortable, fiable et largement orienté vers le développement de logiciels pour MSX. Ces objectifs ont été atteints avec la version actuelle. C'est l'assembleur le plus confortable pour le développement spécifique de ROM et de MegaROM pour MSX.

Le résultat est une application qui s'utilise en ligne de commande avec une vitesse d'assemblage très élevée et très stable, lors de l'utilisation de bibliothèques standard présentes sur tous les ordinateurs.

1.4. Syntaxe

L'implémentation du langage d'assemblage pour microprocesseur Z80 qu'utilise asMSX diverge sous certains aspects de la syntaxe officielle de ZILOG. La plus grande différence est que les adressages indirects sont écrits entre crochets [] et non entre parenthèses (). La raison de ce changement s'explique par l'intégration dans asMSX d'un puissant langage mathématique qui permet des calculs très complexes, de sorte que les parenthèses sont utilisées dans les évaluations numériques.

Dans tous les cas, et pour améliorer la compatibilité et l'importation du code source ajusté à la syntaxe d'origine, la directive `ZILOG` a été incluse afin que les adressages indirects soient indiquées entre parenthèses, et les crochets utilisés comme séparateurs dans les opérations mathématiques.

1.5. Utilisation

Il s'agit d'une application en ligne de commande pour Windows ou Linux (le code source d'asMSX peut être compilé sur les deux plateformes), de ce fait la manière la plus simple de l'utiliser est d'ouvrir une fenêtre d'interpréteur de commandes MS-DOS ou console Linux.

Pour assembler un fichier, procédez comme suit :

```
asmsx [Paramètres optionnels] fichier.ext
```

Si l'extension du fichier n'est pas spécifiée, l'extension « .asm » sera utilisée par défaut.

Les fichiers de code source doivent être des fichiers en texte brut, codés au format ASCII 8 bits. Dans tous les cas, l'assembleur prend en charge les fichiers en texte brut générés sous Windows et Linux sans avoir à effectuer de conversion préalable. Il est également possible d'assembler le code source en faisant glisser l'icône contenant le programme sur l'icône asMSX elle-même, bien que cela ne soit pas recommandé car

les noms longs ou comprenant des espaces peuvent ne pas être interprétés correctement.

Il est donc suggéré de ne pas inclure de points ou de caractères spéciaux dans le nom du fichier, sauf en ce qui concerne le début de l'extension.

Pendant l'assemblage, une série de messages apparaîtra et s'il n'y a pas d'erreur dans le code, une série de fichiers sera générée :

- `fichier.sym` : il s'agit d'un fichier texte en mode ASCII contenant tous les symboles définis dans le programme ainsi que leur valeur exprimée en décimal. Ce fichier est compatible avec le débogueur de l'émulateur BlueMSX et le débogueur de openMSX qui permettent de charger des tables de symboles externes pour faciliter le débogage des programmes. Si aucune balise, constante ou variable n'a été définie, ce fichier ne sera pas généré.
- `fichier.txt` : il s'agit d'un fichier texte en mode ASCII avec les impressions effectuées à partir des directives PRINT incluses dans le fichier source. Si aucune directive de type PRINT n'est utilisée, ce fichier ne sera pas généré.
- `fichier[.z80/.bin/.com/.rom]`: le résultat de l'assemblage du code source sera enregistré avec le même nom que le fichier d'entrée, mais avec l'extension appropriée en fonction des directives utilisées. Ainsi, le fichier aura l'extension ;
  - Z80 s'il s'agit de code sans en-tête,
  - BIN pour les fichiers binaires chargeables à partir du MSX-BASIC avec BLOAD,
  - COM pour les exécutables MSX-DOS,
  - ROM pour les ROM et megaROM MSX.
- `fichier.cas` : Si la directive CAS est utilisée et que le format de sortie le permet, un fichier CAS sera généré, destiné à être chargé dans tout émulateur MSX via une émulation de cassette et pouvant être chargé à partir du MSX-BASIC avec l'instruction BLOAD "CAS:",R
- `fichier.wav` : Lors de l'utilisation de la directive WAV, un fichier audio stéréo 16 bits à 44 100 Hz (qualité CD) destiné à être chargé sur un MSX réel sera généré via le port audio de MSX-BASIC avec l'instruction BLOAD "CAS:",R

1.5.1 Paramètre optionnel

- -z : indique que la syntaxe ZILOG Z80 est utilisée dans le fichier source, sans avoir besoin d'indiquer .ZILOG dans celui-ci.

```
asmsx -z fichier.asm
```

2. Langage assembleur

2.1. Syntaxe du langage

Le format de ligne de code qu'accepte asMSX est le suivant :

```
[étiquette:] [directive[paramètres]] [;commentaire]
[étiquette:] [code opération[paramètres]] [;commentaire]
```

Chacun de ces blocs est optionnel et sera expliqué plus en détail dans les paragraphes suivants. Il n'y a pas de limitation en ce qui concerne les longueurs de lignes, les espaces ou tabulations, car le préprocesseur s'occupe d'ajuster ces éléments.

Les directives et les codes opération peuvent être écrits indifféremment en majuscules ou en minuscules.

Exemples :

```
boucle:
     ld   a,[0ffffh]
; lit le registre maître
points:
     db   20,30,40,50
; table des points
     .ORG 08000h; positionne le début du code en page 2
     nop
```

2.2. Étiquettes

Les étiquettes sont un moyen d'identifier une certaine position en mémoire. En réalité, cela revient à donner un nom à une valeur 16 bits. Leurs noms peuvent être alphanumériques, mais le premier caractère doit être une lettre ou le caractère « _ » (underscore). Pour être défini comme une étiquette, le nom doit être suivi des deux points (« : »). Attention toutefois, les majuscules et minuscules sont distinguées, ce qui signifie que etiquette: ETIQUETTE: ou EtIqUeTtE: ne sont pas identiques.

Exemple :

```
ETIQUETTE:
etiquette:
_alphanumerique:
boucle001:
```

Les deux points permettent de définir une étiquette, dont la valeur correspondra à la position mémoire à occuper par l'instruction suivante. Pour faire appel à une étiquette il suffit d'écrire son nom sans inclure les deux points.

En outre, il est possible de définir des étiquettes locales en faisant précéder leur nom par @@ ou par un point. La particularité des étiquettes locales est qu'elles ne sont disponibles qu'entre une étiquette globale et une autre. Ceci permet d'utiliser plusieurs fois le même nom d'étiquette du moment qu'il s'agit d'une étiquette locale dont le nom est unique entre deux étiquettes globales.

Par conséquent, un code comme celui-ci est parfaitement correct :

```
ROUTINE1:
...
@@BOUCLE:
...
ROUTINE2:
...
@@BOUCLE:
...
```

ou

```
ROUTINE1:
...
.BOUCLE:
...
ROUTINE2:
...
.BOUCLE:
...
```

2.3. Expressions numériques

Les expressions numériques peuvent être des nombres ou le résultat d'opérations plus ou moins complexes effectuées avec ces nombres.

2.3.1. Formats numériques

La liste suivante donne la syntaxe des systèmes de formats numériques interprétés par le compilateur ainsi que quelques exemples :

- **DÉCIMAL** : les nombres en base 10 sont exprimés sous la forme d'un groupe d'un ou de plusieurs chiffres décimaux. La seule restriction étant qu'il en faut pas qu'ils commencent par 0 (voir OCTAL).

  Exemples :

  `0  10  25  1  255  2048  ~~09~~` (non valide)

- **DECIMAL NON ENTIER** : les nombres décimaux sont exprimés par deux groupes d'un ou plusieurs chiffres entiers séparés par un point. Il est nécessaire que ce point soit toujours présent comme séparateur, car sinon il sera interprété comme un nombre entier.

  Exemples :

  `3.14  1.25  0.0  32768.0  -0.50  15.2`

- **OCTAL** : Les nombres en base 8 peuvent être exprimés de deux manières différentes. La première est le format utilisé dans les langages C, C ++ ou Java, c'est-à-dire toujours précédés d'un 0, puis de chiffres octaux (c'est-à-dire de 0 à 7). L'autre forme est celle habituelle des langages d'assemblage, c'est-à-dire le groupe de chiffres octaux et suivi de la lettre O, que ce soit en minuscule ou en majuscule. Ils ont été inclus pour la compatibilité, mais ce n'est pas un système de numérotation pratique pour une architecture comme le Z80.

  Exemples :

  `01  077  010  1o  77o  10o`

- **HEXADÉCIMAL** : Les chiffres de la base 16, les plus courants lors de la programmation en assemblage, peuvent être exprimés de trois manières différentes. Le premier est, à nouveau, compatible avec celui utilisé dans les langages C, C ++ ou Java, c'est-à-dire toujours précédé de 0x puis du groupe de chiffres hexadécimaux (de 0 à 9 et de A à F). La seconde façon de les exprimer est celle utilisée dans le langage Pascal ; Le groupe des chiffres hexadécimaux est précédé du symbole $. Enfin, le format le plus commun des nombres hexadécimaux dans l'assembleur a également été inclus ; Le groupe de chiffres hexadécimaux suivi de la lettre H, minuscule ou majuscule. Notez que dans ce cas, le premier chiffre doit toujours être numérique, c'est-à-dire que si le nombre hexadécimal commence par une lettre, un zéro supplémentaire doit être ajouté à gauche.

Exemples :

```
0x8a   0xff   0x10   $8a   $ff   $10   8ah   0ffh   10h
```

- **BINAIRE** : les nombres en base 2 sont d'une forme classique. Après le groupe de chiffres binaires (zéro et un), la lettre B doit être indiquée soit en minuscule, soit en majuscule.

  Exemples :

  ```
  1000000B    11110000b    0010101b    1001b
  ```

2.3.2. Opérateurs

Il est possible d'effectuer des opérations avec les nombres, exprimés selon n'importe quelle forme vue précédemment. La notation des opérateurs est la notation habituelle et pour les opérations courantes, il a été choisi de suivre la syntaxe du langage C / C ++, qui correspond également à la convention utilisée en Java, JavaScript, PHP et bien d'autres.

+	Somme
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste de la division entière)

En outre, les opérateurs suivants sont disponibles :

<< Décalage vers la gauche. Déplace le nombre de bits indiqué vers la gauche, ce qui équivaut à multiplier par deux le même nombre de fois.

>> Décalage vers la droite. Déplace le nombre de bits indiqué vers la droite, ce qui équivaut à diviser par deux le nombre de fois indiqué.

En plus des opérations arithmétiques, sont disponibles des opérations logiques qui fonctionnent au niveau du bit. Encore une fois, la syntaxe utilisée est celle du langage C et de ses descendants. Les opérations binaires sont les suivantes :

|	OU bit à bit (*OR*).
&	ET bit à bit (*AND*).
^	OU exclusif bit à bit (*XOR*).

Et l'opérateur unaire :

~ NON logique (*NOT*) qui réalise un complément à un.

Des opérations de comparaison logique sont également définies. Celles-ci utilisent également la même syntaxe de langage C :

`||`	OU logique (OR)
`&&`	ET logique (AND)
`==`	égalité
`!=`	différent
`<`	inférieur
`<=`	inférieur ou égal
`>`	supérieur
`>=`	supérieur ou égal

L'ordre d'interprétation des opérateurs est le même que celui utilisé en C / C ++. De plus, les parenthèses peuvent être utilisées pour regrouper des opérations, comme cela se fait généralement avec les opérations arithmétiques.

Exemple :

```
((2*8)/(1+3))<<2
```

Comme pour les nombres décimaux non entiers, vous pouvez utiliser les mêmes opérations que celles indiquées ci-dessus et, en plus, d'autres fonctions réelles :

`SIN(X)`	Fonction sinus. Les unités d'entrée sont les radians.
`COS(X)`	Fonction cosinus.
`TAN(X)`	Fonction tangente.
`ACOS(X)`	Fonction arc cosinus.
`ASIN(X)`	Fonction arc sinus.
`ATAN(X)`	Fonction arc tangente.
`SQR(X)`	Fonction carrée, X élevé à la puissance 2.
`SQRT(X)`	Fonction racine carrée.
`EXP(X)`	Fonction exponentielle, e puissance X.
`POW(X,Y)`	Fonction puissance, X élevé à la puissance Y.
`LOG(X)`	Logarithme décimal.
`LN(X)`	Logarithme népérien.
`ABS(X)`	Valeur absolue de X.

La valeur de PI a été définie par défaut en double précision, de telle sorte à pouvoir être utilisée directement avec les fonctions réelles de la valeur π.

Exemple :

```
sin(pi*45.0/180.0)
```

Pour utiliser ces valeurs non entières dans les programmes assembleur Z80 de manière simple, il est essentiel de convertir les nombres à virgule flottante en nombre à virgule fixe. Pour cela, les fonctions de conversion suivantes sont disponibles :

`FIX(X)`	Convertir un nombre à virgule flottante en nombre à virgule fixe.
`FIXMUL(X,Y)`	Calcule le produit de 2 nombres à virgule fixe.
`FIXDIV(X,Y)`	Calcule la division de nombres à virgule fixe X par Y.
`INT(X)`	Convertit un nombre non entier en nombre entier.

Il existe un symbole dont la valeur change en fonction de la position dans le code source du programme, il s'agit de $. La valeur du symbole $ correspond à l'adresse d'assemblage en cours, ou, en termes d'exécution, à la valeur du registre PC.

2.4. Définition de données

Les données peuvent être incluses dans le source par le biais des directives suivantes :

Données 8 bits :

```
db        donnée 8 bits,[donnée 8 bits...]
defb      donnée 8 bits,[donnée 8 bits...]
dt        "texte"
deft      "texte"
```

Ces instructions contiennent des informations telles que des octets ou des données 8 bits. La directive « dt » a été incluse par souci de compatibilité avec d'autres assembleurs, mais en réalité ces quatre instructions sont équivalentes.

Données 16 bits :

```
dw        donnée 16 bits,[donnée 16 bits...]
defw      donnée 16 bits,[donnée 16 bits...]
```

Ces instructions prennent en compte des données de 16 bits, étant donné que le Z80 parle “petit endien” (little endian), les 8 premiers bits correspondent à l’octet de poids faible du mot de 16 bits et les 8 derniers bits sont ceux de l’octet de poids fort dans la mémoire de l’ordinateur.

Exemple :

```
dw      0ABCDh
```

(En mémoire on trouvera d’abord « CD » et à l’adresse suivante « AB ».)

Autre remarque; Il est possible d’indiquer une étiquette à la place d’une valeur 16 bits après la directive « dw ».

Exemple :

```
dw      ETIQUETTE
```

La directive « ds » (ou « defs ») permet de réserver un certain nombre d’octets à l’endroit actuel de la mémoire. Cela peut servir à identifier les variables en RAM par exemple.

```
ds      X
defs    X
```

Pour faciliter les choses, les tailles les plus courantes, c'est à dire octet et mot, ont été prédéfinies, en utilisant les directives suivantes :

`.byte`	réserve un espace d’un octet (8 bits)
`.word`	réserve un espace de deux octets (16 bits), c’est à dire un mot.

2.5. Directives de compilation

Les instructions prédéfinies suivantes servent à organiser le code et permettent d'activer certaines fonctionnalités apportées par asMSX :

`.ZILOG` Cette directive indique au compilateur d'utiliser la notation ZILOG pour tout le code à partir de l'endroit où elle se présente. C'est-à-dire que les parenthèses seront interprétées comme des indirections et les crochets comme regroupement des opérations mathématiques complexes.

`.ORG X` Cette directive sert à indiquer à l'assembleur une position en mémoire. Le code qui suit la directive .ORG sera assemblé à partir de cette adresse, ou s'il s'agit de données, elles seront placées à partir de cette adresse.

`.PAGE X` Équivaut à la directive .ORG, mais au lieu d'indiquer une adresse, indique un numéro de page mémoire. Ainsi :

`.PAGE 0` équivaut à `.ORG 0000h`,

`.PAGE 1` équivaut à `.ORG 4000h`,

`.PAGE 2` équivaut à `.ORG 8000h` et

`.PAGE 3` équivaut à `.ORG C000h`.

```
identificateur .EQU expression
```

Permet de définir une constante. Les règles de nommage sont les mêmes que pour les étiquettes, à ceci près qu'il n'est pas possible de définir des identificateurs locaux.

```
variable = expression
```

AsMSX permet ici l'utilisation de variables entières. Les variables doivent êtres initialisées, puis leurs valeurs numériques affectées, pour pouvoir les utiliser dans des opérations. Par exemple, vous pouvez effectuer les opérations suivantes :

```
Valeur1=0
Valaur1=Valeur1+1
ld   a,Valeur1
Valeur1=Valeur1*2
ld   b,valeur1
```

`.BIOS` Prédéfinit les adresses des routines du Bios, incluant les spécificités des standards MSX, MSX2, MSX2+ et Turbo-R. Il faut employer les noms usuels en majuscules (cf. Annexe II).

`.ROM` Définit l'en-tête pour produire une ROM. Il est impératif d'indiquer d'abord la position, ce qui peut être fait avec la directive .PAGE. Il faut également indiquer à l'aide de la directive .START l'adresse de lancement du programme.

`.MEGAROM [mapper]` Définit l'en-tête pour produire une megaROM. Par défaut, définit aussi la sous-page 0 du mapper, si bien qu'il n'est pas nécessaire d'inclure toutes les instructions .ORG, ni .PAGE ou SUBPAGE après. Les types de mappers supportés sont les suivants :

- `Konami`: taille de sous-page de 8 Ko, limite de 32 pages. La taille maximale de la megaROM sera de 256 Ko (2 Mégabits). La sous-page 0 est nécessairement entre 4000h et 5FFFh , on ne peut donc pas la changer.
- `KonamiSCC`: taille de sous-page de 8 Ko, limite de 64 pages. La taille maximale de la megaROM sera de 512 Ko (4 Mégabits). Prend en charge l'accès au Sound Custom Chip (SCC).
- `ASCII8`: taille de sous-page de 8 Ko, limite de 256 pages. La taille maximale de la megaROM sera de 2048 Ko (16 Mégabits, 2 Mo).
- `ASCII16`: taille de sous-page de 16 Ko, limite de 256 pages. La taille maximale de la megaROM sera de 4096 Ko (32 Mégabits, 4 Mo).

`.BASIC` génère l'en-tête pour produire un fichier binaire qu'on pourra charger sous MSX- Basic. Il est nécessaire d'indiquer avant une position initiale avec la directive « .ORG », et au cas où l'adresse d'exécution du programme ne coïncide pas avec l'adresse de départ, utiliser également la directive « .START ».

L'extension par défaut du fichier produit est « .BIN ».

`.MSXDOS` produit un fichier « .COM » exécutable depuis MSX-DOS. Il n'est pas nécessaire d'utiliser l'instruction « .ORG » parce que le programme commence toujours à l'adresse 0100h.

`.START X` Indique l'adresse de début d'exécution pour que les fichiers ROM, megaROM et BIN dont le point d'entrée (exécution) ne se trouve pas au début du fichier.

`.SEARCH` Pour les ROMs et megaROMs qui démarrent à la page 1 (4000h), cette directive se charge de chercher automatiquement le slot primaire et le slot secondaire de la page 2 correspondante (8000h). La portion de code suivante est alors ajoutée dans le code :

```
        call    0138h ;RSLREG
        rrca
        rrca
        and     03h
; Slot secondaire
        ld      c,a
        ld      hl,0FCC1h
        add     a,l
        ld      l,a
        ld      a,[hl]
        and     80h
        or      c
        ld      c,a
        inc     l
        inc     l
        inc     l
        inc     l
        ld      a,[hl]
; Définit le numéro du slot (ID)
        and     0ch
        or      c
        ld      h,80h
; Activation du slot
        call    0024h ;ENASLT
```

`.SUBPAGE n AT X` Cette macro est utilisée pour définir les sous-pages distinctes d'une megaROM. Dans le modèle de génération de megaROMs d'asMSX, tous les codes et données doivent être inclus en sous-pages, qui équivalent à des blocs logiques d'opérations du mapper. Vous devez indiquer le nombre de sous-pages que vous désirez créer, et à quelle adresse logique commence l'assemblage, et à quelle adresse commence l'exécution.

`.SELECT n AT x / .SELECT registre AT x` Cette directive est complémentaire à « `.SUBPAGE n AT X` ». Il s'agit d'une macro qui va sélectionner la sous-page n dans l'adresse X. Le code réel généré pour cette opération dépendra du mapper sélectionné. Ce code n'altère la valeur d'aucun registre, n'affecte pas le mode d'interruption, ni les indicateurs d'état de registre.

`.PHASE X / .DEPHASE` Ces deux routines permettent d'utiliser des adresses virtuelles de mémoire. C'est-à-dire, assembler des instructions dans une position de mémoire avant de les placer dans une autre. Il peut être utile pour introduire du code dans une ROM qui se copiera et s'exécutera depuis la

RAM. Cela a pour effet que les étiquettes s'assemblent en fonction de l'adresse indiquée.

`.CALLBIOS X` Appelle une routine Bios depuis MSX-DOS. Équivaut au bloc de code suivant :

```
LD   IY,[EXPTBL-1]
LD   IX,ROUTINE
CALL CALSLT
```

`.CALLDOS X` Exécute une fonction MSX-DOS. Équivaut au code suivant :

```
LD C,FONCTION
CALL 0005h
```

`.SIZE X` Détermine la taille du fichier binaire résultant en kilo-octets.

`.INCLUDE "fichier"` Inclut du code source en provenance d'un fichier externe.

`.INCBIN "fichier" [SKIP X] [SIZE Y]` Injecte un fichier binaire externe dans le programme. On peut utiliser en plus les paramètres « SKIP » et « SIZE » qui permettent respectivement d'indiquer le nombre d'octets à ignorer depuis le début du fichier et d'indiquer le nombre d'octets à prendre en compte au total (si on ne veut pas inclure tout le fichier).

`RANDOM(n)` Génère une valeur aléatoire entre 0 et n-1 (à priori non fonctionnel car il produit toujours la même valeur sans même prendre en compte la valeur de n, d'autre part cette fonction de génère pas du code pour calculer une valeur aléatoire mais donne une valeur aléatoire à utiliser dans le source, donc peu utile).

Par exemple :

```
LD  A,RANDOM(100)
```

`DEBUG "texte"` Permet de fournir un texte qui sera utilisé par un débogueur tout en étant transparent lors de l'exécution. Le débogueur de BlueMSX supporte cette fonctionnalité depuis la version 2.3 ;

http://www.msxblue.com/manual/debugger_c.htm

La longueur du texte ne peux pas dépasser 125 caractères.

Le code inséré est le suivant :

```
ld d,d
jr longueur[texte]+1
db ["texte"],0
```

`.BREAK [X] / .BREAKPOINT [X]` Permet de définir un point d'arrêt pour le débogueur de l'émulateur BlueMSX. Son exécution est transparente, bien que son élimination soit recommandée dans le fichier final. Si l'adresse n'est pas indiquée, le point d'arrêt sera exécuté à la position à laquelle il a été défini.

Le code inséré est le suivant :

Sans valeur renseignée (point d'arrêt à l'endroit courant)

```
ld b,b
jr 0
```

Avec valeur `[X]`renseignée (point d'arrêt à l'étiquette indiquée)

```
ld b,b
jr 2
dw [adresse_point_d_arret]
```

`REPT n / ENDR` Cette macro-instruction permet de répéter un bloc de code un nombre de fois déterminé. Elle prend en charge l'imbrication pour générer automatiquement des tables complexes.

La seule restriction est que le nombre de répétitions doit être indiqué sous la forme d'un nombre décimal direct et non d'une expression numérique indirecte.

Par exemple :

```
REPT  16
  OUTI
ENDR

X=0
Y=0
REPT 10
   REPT 10
      DB X*Y
      X=X+1
   ENDR
   Y=Y+1
ENDR
```

`.PRINTTEXT "texte" / .PRINTSTRING "texte"` Imprime un texte dans le fichier de sortie.

`.PRINT expression / .PRINTDEC expression` Imprime une valeur numérique au format décimal.

`.PRINTHEX expression` Imprime une valeur numérique au format hexadécimal.

`.PRINTFIX expression` Imprime une valeur numérique à virgule fixe.

`.CAS ["texte"] / .CASSETTE ["texte"]` Génère un fichier « .CAS » avec le résultat de la compilation. Valable uniquement pour le format BASIC, ROM sur la page 2 et direct Z80.

Le texte permet d'indiquer le nom de chargement de la cassette, avec un maximum de 6 caractères. S'il n'est pas indiqué, c'est le nom du fichier de sortie qui sera utilisé.

`.WAV ["texte"]` Identique à la commande « .CAS », mais au lieu de générer un fichier CAS, génère un fichier audio « .WAV » qui peut être chargé directement dans un MSX réel via le port de cassette.

`.FILENAME ["texte"]` Avec cette directive tous les fichiers résultant de l'assemblage recevront le nom du fichier indiqué. S'il n'est pas utilisé, ils recevront le nom par défaut du nom fichier de code source assemblé, à la différence près de l'extension appropriée.

`.SINCLAIR` La directive « Sinclair » rend le fichier assemblé enregistré au format « .TAP » avec les en-têtes appropriés destiné à être chargé dans un émulateur Sinclair ZX Spectrum ou dans du matériel réel si les applications nécessaires sont disponibles [il existe un problème avec la somme de contrôle des en-têtes].

2.6. Commentaires

Il est fortement recommandé de commenter son code tout au long du code source assembleur. AsMSX vous permet d'entrer des commentaires sous différentes formes :

Pour commenter une seule ligne le point-virgule est le caractère standard des commentaires dans le code assembleur :

```
; Ici un commentaire
```

La ligne complète sera considérée en tant que commentaire, jusqu'au retour à la ligne.

Il est possible d'utiliser le double barres obliques qui est l'indicateur de commentaire d'une ligne en C / C ++.
Cette syntaxe fonctionne comme la précédente :

```
// Ici un commentaire
```

Pour les commentaires sur plusieurs lignes il faut utiliser la combinaison « barre-étoile » pour marquer le début et « étoile-barre » pour la fin.
Il est possible également d'écrire les commentaires entre accolades.
L'ensemble du texte entre ces délimiteurs sera ignoré de l'assembleur.

```
/* Ici un commentaire */
{ Ici un commentaire }
```

2.7. Assemblage conditionnel

AsMSX comporte un principe de pré-compilation pour l'assemblage conditionnel. Le format d'un bloc d'assemblage conditionnel est alors le suivant :

```
IF   condition
     Instructions
ELSE
     Instructions
ENDIF
```

La condition peut être de n'importe quel type, à condition de suivre les règles syntaxiques de C/C++ et Java. Une condition n'est vraie que si son résultat est différent de zéro.

Si la condition est vraie, le code sera assemblé selon les instructions suivant le « IF ».

Si la condition n'est pas vraie, les instructions suivant le « IF » seront ignorées, et seul le bloc suivant le « ELSE » sera assemblé.

Bien entendu, le bloc « ELSE » est facultatif et il n'est pas nécessaire de l'inclure dans chaque instruction « IF ». Cependant, « ENDIF » doit être utilisé impérativement pour fermer chaque bloc conditionnel.

Il est possible d'imbriquer des instructions « IF » sans limitation, comme illustré dans l'exemple suivant :

```
IF (ordinateur==MSX)
     call MSX1_Init
ELSE
     IF (ordinateur==MSX2)
          call MSX2_Init
     ELSE
          IF (ordinateur==MSX2plus)
               call MSX2plus_Init
          ELSE
               call TurboR_Init
          ENDIF
     ENDIF
ENDIF
```

Le code, les directives et les macros à l’intérieur de ces structures seront exécutés en fonction des conditions, il est donc possible de créer des structures de la forme suivante :

```
CARTOUCHE=1
BINAIRE=2

format=CARTOUCHE

IF (format==CARTOUCHE)
     .page     2
     .ROM
ELSE
     .org 8800h
     .Basic
```

```
ENDIF

.START Init
```

La seule restriction porte sur l’écriture des instructions « IF <condition> », « ELSE » et « ENDIF » qui doivent se trouver chacune sur des lignes séparées. C'est-à-dire qu'elles ne peuvent pas être regroupées avec d'autres instructions ou macros dans la même ligne de code.

Le code suivant produirait une erreur lors de l'assemblage :

```
IF (variable) nop ELSE inc a ENDIF
```

Ce code doit être écrit comme suit :

```
IF (variable)
     nop
ELSE
     inc  a
ENDIF
```

De plus, il existe également l'instruction conditionnelle « IFDEF » qui ne vérifie, non pas une valeur ou une expression utilisée, mais si l’étiquette indiquée, la constante ou la variable a été définie ou non. Par exemple, le bloc suivant :

```
IFDEF test
     .WAV "Test"
ENDIF
```

Permet de générer un fichier « WAV » si et seulement si la balise ou le symbole « `test` » a déjà été défini dans le code source. Notez que « IFDEF » ne reconnaîtra l’étiquette, telle que définie, si elle a été incluse avant l'instruction « IFDEF ».

Annexe I : Liste des changements

v.0.01a: [10/09/2000] Première version publique

v.0.01b: [03/05/2001] Corrections de bugs. Ajout de PRINTFIX, FIXMUL, FIXDIV

v.0.10 : [19/08/2004] Amélioration globale. Codes opération vérifiés à 100%

v.0.11 : [31/12/2004] IX, IY acceptent maintenant les décalages nuls ou négatifs

v.0.12 : [11/09/2005] Version de sauvegarde
Ajout de REPT/ENDR, variables/constantes, RANDOM, DEBUG BlueMSX, BREAKPOINT BlueMSX, PHASE/DEPHASE, $ symbol

v.0.12e: [07/10/2006]
Paramètres additionnels pour INCBIN "file" [SKIP num] [SIZE num]
Macro placée en seconde page (32KB ROMs / megaROMs)
Ajout du support expérimental pour les MegaROMs:
* MEGAROM [mapper] - définition du type de mapper
* SUBPAGE [n] AT [adresse] - définition de la page
* SELECT [n] AT [adresse] - affecte la page

v.0.12f: [16/11/2006]
Correction de plusieurs opérateurs binaires
Assemblage conditionnel

v.0.12f1:[17/11/2006]
Assemblage conditionnel inclus et autres conditions

v.0.12g:[18/03/2007]
PHASE/DEPHASE bug corrigé
Support du format CAS initial
Génération fichier WAV en sortie
Amélioration de l’assemblage conditionnel : IFDEF

v.0.14: [UNRELEASED]

Première version Linux fonctionnelle

Stabilité améliorée quelque peu

v.0.15: [UNRELEASED]

Suppression des opérations ADD IX,HL et ADD IY,HL

Problème d'interférence entre Label et Macro résolu

Amélioration globale des pointeurs concernant la stabilité

Les paramètres SKIP et SIZE de INCBIN sont maintenant compatibles 32-bits

v.0.16: [CANDIDATE]

Première version pleinement développée sur Linux

Correction d'un bug affectant les extensions de noms de fichiers

Suppression du IM 0/1 bizarre – apparemment il s'agit du code opération IM 0 non documenté

Directive FILENAME permettant de désigner le nom de fichier assembleur

Directive ZILOG permettant d'assembler le code ZILOG (Syntaxe officielle)

Entête standard de 16 octets pour les ROM/MEGAROM

Correction d'un problème ennuyeux concernant $DB (donnée) qui était interprété comme DB

Directive SINCLAIR ajoutée pour permettre la génération de fichier TAP (aïe!) --> DOIT ENCORE ETRE TESTE

Évolutions prévues :

- Adapter le BIOS pour les modèles SINCLAIR ?
- Gestion DISK
- Gestion R800/Z80/8080/Gameboy
- Gestion du format de fichier Sinclair ZX Spectrum TAP/TZX

[version "après" Rosby]

v.0.17: [19/12/2013]

[Résolu] Anomalie 1: Crash sur Linux lors de l'inclusion de fichier .asm additionnels (par theNestruo)

[Résolu] Anomalie 5: Codes retour différents de zéro lors d'erreurs (by theNestruo)

v.0.18: [01/02/2017]

Problème résolu concernant .megaflashrom et « defines ».

v.0.18.1: [11/02/2017]

Correction de multiples alertes de compilation en spécifiant les paramètres et le type explicite de la variable de retour.

Correction d'un problème avec la génération du nom du fichier cassette causé par une variable non initialisée (binario).

v.0.18.2: [25/05/2017]

Ajout du paramètre -z. Ce paramètre permet d'utiliser la notation standard ZILOG concernant les indirections, c'est à dire les parenthèses au lieu des crochets tels qu'interprétés par défaut par asMSX. Ce paramètre évide de devoir indiquer « .ZILOG » dans le code source.

De nouvelles étiquettes locales peuvent être utilisées ; .Local_Label en plus de la syntaxe existante @@Local_Label.

Maintenant la directive ".instruction" est correctement interprétée. Par exemple, il était possible d'écrire "azilog", "bzilog"… sans que cela génère d'erreur au lieu de la syntaxe correcte ".zilog" ou "zilog".

v.0.18.3: [10/06/2017]

Correction d'un bug induit dans la version du 5 février lors de l'utilisation de la directive « INCLUDE ». Parser 1 p1_tmpstr n'utilise pas l'allocation dynamique. A la place il utilise l'allocation mémoire par strtok. Cet espace mémoire n'est jamais effacé, il faut tenir compte de cela dans les futurs développements pour éviter les fuites mémoire.

v.0.18.4: [18/06/2017]

Correctif sur les chaines qui ne se terminent pas. Une meilleure solution de correction a été trouvée. Probablement un correctif plus flexible.

v.0.18.5: [31/08/2018] Sylvain

Ajout des mnémoniques système de la « Main ROM » lors de l'utilisation de la directive « .BIOS ».

Ajout des mnémoniques correspondant aux variables du système MSX localisées en RAM, lors de l'utilisation de la directive « .BIOS ».

Correctif d'un bug lors de l'écriture du fichier binaire de sortie dans le cas où il ne peut pas être écrit (droit d'accès insuffisants, fichier bloqué par un antivirus ou répertoire inexistant).

Correctif d'un bug dans la directive DEBUG (0 en fin de texte et calcul saut relatif)

Annexe II : Étiquettes définies lors de l'utilisation de la directive .BIOS

Étiquette	*Adresse associée*
CHKRAM	00000h
SYNCHR	00008h
RDSLT	0000Ch
CHRGTR	00010h
WRSLT	00014h
OUTDO	00018h
CALSLT	0001Ch
DCOMPR	00020h
ENASLT	00024h
GETYPR	00028h
CALLF	00030h
KEYINT	00038h
INITIO	0003Bh
INIFNK	0003Eh
DISSCR	00041h
ENASCR	00044h
WRTVDP	00047h
RDVRM	0004Ah
WRTVRM	0004Dh
SETRD	00050h
SETWRT	00053h
FILVRM	00056h
LDIRMV	00059h
LDIRVM	0005Ch
CHGMOD	0005Fh
CHGCLR	00062h
NMI	00066h
CLRSPR	00069h
INITXT	0006Ch

INIT32	0006Fh
INIGRP	00072h
INIMLT	00075h
SETTXT	00078h
SETT32	0007Bh
SETGRP	0007Eh
SETMLT	00081h
CALPAT	00084h
CALATR	00087h
GSPSIZ	0008Ah
GRPPRT	0008Dh
GICINI	00090h
WRTPSG	00093h
RDPSG	00096h
STRTMS	00099h
CHSNS	0009Ch
CHGET	0009Fh
CHPUT	000A2h
LPTOUT	000A5h
LPTSTT	000A8h
CNVCHR	000ABh
PINLIN	000AEh
INLIN	000B1h
QINLIN	000B4h
BREAKX	000B7h
ISCNTC	000BAh
CKCNTC	000BDh
BEEP	000C0h
CLS	000C3h
POSIT	000C6h
FNKSB	000C9h
ERAFNK	000CCh
DSPFNK	000CFh

TOTEXT	000D2h
GTSTCK	000D5h
GTTRIG	000D8h
GTPAD	000DBh
GTPDL	000DEh
TAPION	000E1h
TAPIN	000E4h
TAPIOF	000E7h
TAPOON	000EAh
TAPOUT	000EDh
TAPOOF	000F0h
STMOTR	000F3h
LFTQ	000F6h
PUTQ	000F9h
RIGHTC	000FCh
LEFTC	000FFh
UPC	00102h
TUPC	00105h
DOWNC	00108h
TDOWNC	0010Bh
SCALXY	0010Eh
MAPXYC	00111h
FETCHC	00114h
STOREC	00117h
SETATR	0011Ah
READC	0011Dh
SETC	00120h
NSETCX	00123h
GTASPC	00126h
PNTINI	00129h
SCANR	0012Ch
SCANL	0012Fh
CHGCAP	00132h

CHGSND	00135h
RSLREG	00138h
WSLREG	0013Bh
RDVDP	0013Eh
SNSMAT	00141h
PHYDIO	00144h
FORMAT	00147h
ISFLIO	0014Ah
OUTDLP	0014Dh
GETVCP	00150h
GETVC2	00153h
KILBUF	00156h
CALBAS	00159h
SUBROM	0015Ch
EXTROM	0015Fh
CHKSLZ	00162h
CHKNEW	00165h
EOL	00168h
BIGFIL	0016Bh
NSETRD	0016Eh
NSTWRT	00171h
NRDVRM	00174h
NWRVRM	00177h
RDBTST	0017Ah
WRBTST	0017Dh
CHGCPU	00180h
GETCPU	00183h
PCMPLY	00186h
PCMREC	00189h

Données MAIN ROM

Étiquette	*Adresse associée*
CGTABL	00004h
VDP_DR	00006h
VDP_DW	00007h
MSXID1	0002Bh
MSXID2	0002Ch
MSXID3	0002Dh

Adresses des variables RAM

Étiquette	*Adresse associée*
RDPRIM	0F380h
WRPRIM	0F385h
CLPRIM	0F38Ch
LINL40	0F3AEh
LINL32	0F3AFh
LINLEN	0F3B0h
CRTCNT	0F3B1h
CLMLST	0F3B2h
TXTNAM	0F3B3h
TXTCOL	0F3B5h
TXTCGP	0F3B7h
TXTATR	0F3B9h
TXTPAT	0F3BBh
T32NAM	0F3BDh
T32COL	0F3BFh
T32CGP	0F3C1h
T32ATR	0F3C3h
T32PAT	0F3C5h
GRPNAM	0F3C7h
GRPCOL	0F3C9h
GRPCGP	0F3CBh

GRPATR	0F3CDh
GRPPAT	0F3CFh
MLTNAM	0F3D1h
MLTCOL	0F3D3h
MLTCGP	0F3D5h
MLTATR	0F3D7h
MLTPAT	0F3D9h
CLIKSW	0F3DBh
CSRY	0F3DCh
CSRX	0F3DDh
CNSDFG	0F3DEh
RG0SAV	0F3DFh
RG1SAV	0F3E0h
RG2SAV	0F3E1h
RG3SAV	0F3E2h
RG4SAV	0F3E3h
RG5SAV	0F3E4h
RG6SAV	0F3E5h
RG7SAV	0F3E6h
STATFL	0F3E7h
TRGFLG	0F3E8h
FORCLR	0F3E9h
BAKCLR	0F3EAh
BDRCLR	0F3EBh
MAXUPD	0F3ECh
MINUPD	0F3EFh
ATRBYT	0F3F2h
QUEUES	0F3F3h
FRCNEW	0F3F5h
SCNCNT	0F3F6h
REPCNT	0F3F7h
PUTPNT	0F3F8h
GETPNT	0F3FAh

CS120	0F3FCh
CS240	0F401h
LOW	0F406h
HIGH	0F408h
HEADER	0F40Ah
ASPCT1	0F40Bh
ASPCT2	0F40Dh
ENDPRG	0F40Fh
ERRFLG	0F414h
LPTPOS	0F415h
PRTFLG	0F416h
NTMSXP	0F417h
RAWPRT	0F418h
VLZADR	0F419h
VLZDAT	0F41Bh
CURLIN	0F41Ch
EXBRSA	0FAF8h
PRSCNT	0FB35h
SAVSP	0FB36h
VOICEN	0FB38h
SAVVOL	0FB39h
MCLLEN	0FB3Bh
MCLPTR	0FB3Ch
QUEUEN	0FB3Eh
MUSICF	0FB3Fh
PLYCNT	0FB40h
VCBA	0FB41h
VCBB	0FB66h
VCBC	0FB8Bh
ENSTOP	0FBB0h
BASROM	0FBB1h
LINTTB	0FBB2h
FSTPOS	0FBCAh

CODSAV	0FBCCh
FNKSWI	0FBCDh
FNKFLG	0FBCEh
ONGSBF	0FBD8h
CLIKFL	0FBD9h
OLDKEY	0FBDAh
NEWKEY	0FBE5h
KEYBUF	0FBF0h
BUFEND	0FC18h
LINWRK	0FC18h
PATWRK	0FC40h
BOTTOM	0FC48h
HIMEM	0FC4Ah
TRPTBL	0FC4Ch
RTYCNT	0FC9Ah
INTFLG	0FC9Bh
PADY	0FC9Ch
PADX	0FC9Dh
JIFFY	0FC9Eh
INTVAL	0FCA0h
INTCNT	0FCA2h
LOWLIM	0FCA4h
WINWID	0FCA5h
GRPHED	0FCA6h
ESCCNT	0FCA7h
INSFLG	0FCA8h
CSRSW	0FCA9h
CSTYLE	0FCAAh
CAPST	0FCABh
KANAST	0FCACh
KANAMD	0FCADh
FLBMEM	0FCAEh
SCRMOD	0FCAFh

OLDSCR	0FCB0h
CASPRV	0FCB1h
BRDATR	0FCB2h
GXPOS	0FCB3h
GYPOS	0FCB5h
GRPACX	0FCB7h
GRPACY	0FCB9h
DRWFLG	0FCBBh
DRWSCL	0FCBCh
DRWANG	0FCBDh
RUNBNF	0FCBEh
SAVENT	0FCBFh
EXPTBL	0FCC1h
SLTTBL	0FCC5h
SLTATR	0FCC9h
SLTWRK	0FD09h
PROCNM	0FD89h
DEVICE	0FD99h