ÉCOLE POLYTECHNIQUE
DE MONTRÉAL

DÉPARTEMENT DE GÉNIE INFORMATIQUE

Implantation d'un interpréteur BASIC pour un micro-ordinateur basé sur un microprocesseur 8085 d'Intel

Rapport de projet de fin d'études soumis

comme condition partielle à l'obtention du

diplôme de baccalauréat en ingénierie.

Présenté par: Dominic Thibodeau

Matricule: 61657

Directeur de projet: Georges-Émile April

Date: Mercredi le 5 décembre 2001

Sommaire

Ce rapport présente les étapes d'analyse, du développement et des résultats d'un projet de fin d'études. Ce projet consiste à implanter, sur une plate-forme déjà existante, un interpréteur du langage BASIC.

La plate-forme matérielle est un micro-ordinateur construit à des fins personnelles. Il s'agit d'un système 8 bits basé sur le microprocesseur Intel 8085.

Le projet est divisé en deux parties, qui sont analysées en détail dans le rapport. La partie matérielle est assez restreinte. Elle consiste à doter l'ordinateur de périphériques de sorties adéquats pour l'usage d'un interpréteur BASIC. La partie logicielle, quant à elle, se concentre sur le design et l'implantation de l'interpréteur qui sera chargé en ROM sur la machine.

La partie logicielle est découpée en plusieurs modules, qui sont analysés et implantés de manière indépendante. Chaque module est décrit, tout d'abord de façon sommaire, ensuite de manière plus détaillée. Le texte présente aussi l'étape d'intégration ainsi qu'une discussion des résultats obtenus.

Table des matières

Remerciements

Je tiens en premier lieu à remercier Monsieur Georges-Émile April, le directeur de ce projet, pour son temps ainsi que pour son intérêt porté aux sujets abordés par ce projet.

Je remercie aussi Fujitsu Softek Canada Inc., mon employeur, pour m'avoir gracieusement donné une quantité appréciable de composantes électroniques, de circuits intégrés ainsi qu'un programmeur d'EPROM /PLD sans lesquels ce projet n'aurait pas vu le jour.

Liste des tableaux

Liste des figures

Liste des symboles et abréviations

ANSI:	Protocole d'affichage dérivé du standard ANSI X3.64, utilisé sur les terminaux. Le protocole ANSI a été popularisé par les babillards électroniques (BBS).
ASCII:	Acronyme de "American Standard Code for Information Interchange". Code qui permet de restituer les lettres de l'alphabet romain, les chiffres et les principaux caractères spéciaux utilisés en informatique.
BASIC:	Acronyme de "Beginner's All-purpose Symbolic Instruction Code". Langage de programmation simple inventé en 1963 au College Dartmouth par les mathématiciens John George Kemeny et Tom Kurtzas comme un outil d'apprentissage pour les étudiants. [1]
Bit:	Abréviation de "Binary digit". C'est la plus petite unité de donnée traitée par un ordinateur. Les bits sont utilisés en combinaisons variées pour représenter différents types de données. Le bit peut prendre la valeur 0 ou 1.
Byte (octet):	Ensemble de huit bits, pouvant prendre 256 valeurs différentes. La capacité de stockage de données d'un ordinateur s'exprime en octets.
DRAM:	Acronyme de "Dynamic RAM". Mémoire vive dont le contenu doit être rafraîchi à intervalles réguliers, sinon les données sont perdues.
EPROM:	Acronyme de "Erasable Programmable ROM". Mémoire morte programmable électriquement et effaçable par l'application de rayons ultra-violets.
KiloByte (KB):	1024 bytes (octets).
LCD:	Acronyme de "Liquid Crystal Display". Écran/afficheur à cristaux liquides.
PLD:	Acronyme de "Programmable Logic Device". Circuit intégré dont le comportement (relations entre entrées et sorties) est programmable.
RAM:	Acronyme de "Random Access Memory". Mémoire vive qui peut être lue ou écrite plusieurs fois (habituellement par un processeur). Le contenu de la RAM est perdu lorsque l'alimentation est coupée.
ROM:	Acronyme de "Read Only Memory". Mémoire morte qui est programmée une fois et dont le contenu ne peut être que lu par la suite. Le contenu de la ROM n'est pas perdu lorsque l'alimentation est coupée

SRAM: Acronyme de “Static RAM”. Mémoire vive qui ne nécessite pas de rafraîchissement. Son contenu reste valide tant que l’alimentation est soutenue.

UART: Acronyme de “Universal Asynchronous Receiver Transmitter”. Interface de conversion de données parallèles en données séries (et vice-versa) transmises de manière asynchrone.

Introduction

Ce rapport est le fruit de centaines d'heures de travail soutenu. Il présente la démarche suivie dans le cadre d'un projet de fin d'études en Génie Informatique. Le projet est la suite logique d'un projet personnel de design et d'implantation d'un micro-ordinateur 8 bits basé sur un microprocesseur 8085. Ce système ressemble en plusieurs points aux premiers micro-ordinateurs personnels des années 1980. La tâche effectuée dans ce projet consiste à compléter la machine en la dotant d'un interpréteur du langage BASIC. Ce langage était fréquemment implanté sur les machines de l'époque. Il s'agit d'un langage interprété simple et familier pour de nombreux informaticiens qui ont fait leurs débuts avec celui-ci.

Le rapport est divisé en quatre grands chapitres, en plus de trois annexes. Voici une présentation sommaire de chaque section de ce texte.

Le premier chapitre situe le projet dans son contexte. La première section du chapitre, l'historique, situe le projet par rapport aux ordinateurs du passé. Ensuite, on énonce les travaux antérieurs: le design et l'implantation de la plate-forme sur laquelle repose ce travail. On présente par la suite une description plus détaillée du projet, suivie d'une section décrivant les difficultés à surmonter. Finalement, on explique l'intérêt porté au projet.

Le deuxième chapitre énonce les moyens mis en oeuvre pour réaliser le projet avec succès. On y explique les étapes préliminaires d'étude et de design: le découpage logique du projet en plusieurs modules, suivi d'une description fonctionnelle de chaque module. La majeure partie de ce travail est effectuée avant l'exécution, c'est-à-dire avant le codage des modules. Il est essentiel de bien comprendre le rôle de chaque partie du projet avant de se lancer dans la construction ou la programmation. Il faut aussi décider des outils qui seront utilisés tout au long du travail.

Le chapitre trois présente la partie exécution du travail. On y explique plusieurs concepts et certains détails de l'implantation. On présente certaines structures internes et les liens qu'elles ont entre

elles. Le fonctionnement des fonctions majeures est aussi expliqué. Une section est réservée à chaque module, en plus des sections sur l'intégration du système et la partie matérielle du projet.

Le dernier chapitre présente une discussion du travail réalisé dans le cadre de ce projet. On y retrouve une critique de la manière de procéder et des résultats obtenus. Ce chapitre se termine en indiquant des avenues possibles pour l'amélioration et la complétion du projet.

L'annexe A présente les bases du langage BASIC implanté dans ce projet. On explique les modes de fonctionnement, la syntaxe des expressions, les types de variables supportés ainsi qu'une description de chaque instruction du langage.

L'annexe B liste les mots-clés réservés par l'interpréteur BASIC. Cette liste est importante puisqu'un mot clé ne doit pas être contenu dans un nom de variable - il s'agit d'une erreur fréquente.

Finalement, l'annexe C présente une liste complète des messages d'erreur possibles, avec une description détaillée lorsque nécessaire. Dans tous les cas, on présente une ou plusieurs lignes de codes qui produisent l'erreur dont il est question.

1. Problématique

1.1 Historique

Le début des années 1980 marque l'explosion de la micro-informatique, avec l'avènement d'une panoplie d'ordinateurs personnels qui font leur apparition dans les foyers. La plupart de ces ordinateurs ont plusieurs points en commun:

- Ils sont souvent basés sur un microprocesseur 8 bits (Intel 8080, Zilog Z-80, Motorola 6800, MOS 6502, et plusieurs dérivés)
- La plupart de ces machines contiennent moins de 64KB de mémoire
- Ils ont souvent un interpréteur Basic intégré en ROM, qui s'exécute au démarrage de la machine
- Leur architecture est simple, comparativement aux machines performantes et ultra-complexes d'aujourd'hui

J'ai moi-même grandi avec une de ces machines (un Commodore 64). J'y ai appris les bases de la programmation et de l'informatique en général. Depuis ce temps, mon intérêt pour l'électronique et l'informatique a grandi, et j'ai eu l'idée d'utiliser ma nostalgie du 'bon vieux temps' comme base pour mon projet de fin d'études.

1.2 Travaux antérieurs

Pendant des années, j'ai acquis des connaissances en électronique qui m'ont donné le goût de concevoir et d'exécuter un projet d'une bonne envergure. Avec les années, j'ai amassé une bonne quantité de composantes électroniques. Lorsque j'ai eu l'équipement nécessaire (les plus gros morceaux étant un programmeur d'EPROM/PLA, et un oscilloscope), j'ai décidé de construire mon propre micro-ordinateur 8 bits. Avec des composantes 'modernes', il m'a été relativement facile de designer et de bâtir cette machine. Voici quelques exemples de simplicité du design:

- La mémoire vive (RAM) est statique (SRAM), ce qui simplifie beaucoup l'interface avec le reste du système. Les 32 KB de RAM tiennent sur un seul circuit intégré. Les machines des années 1980 n'avaient pas ce luxe; elles contenaient souvent de la mémoire dynamique (DRAM), qui nécessite des circuits de rafraîchissement complexes.

- La plupart de la logique (décodage des adresses ou des signaux du microprocesseur) est implantée en logique programmable, ce qui permet d'éliminer plusieurs composantes discrètes. Presque tout est 'fait sur mesure'.

- Le processeur utilisé (Intel 8085) a été choisi pour deux raisons bien simples. Premièrement, j'en avais un à ma disposition. Deuxièmement, ce processeur dispose d'espaces mémoire et adresse séparés, ce qui permet une implantation on ne peut plus simple du décodage d'adresse: la moitié de l'espace d'adressage correspond à la ROM (bit A15 = 0), la moitié supérieure à la RAM (bit A15 = 1).

J'ai donc développé un système dont on peut apercevoir l'architecture à la Figure 1.1. On note la simplicité du design. Après l'implantation, un programme simple de type 'Memory monitor' à été développé, mais son usage était très limité. Il manquait quelque chose à la machine.

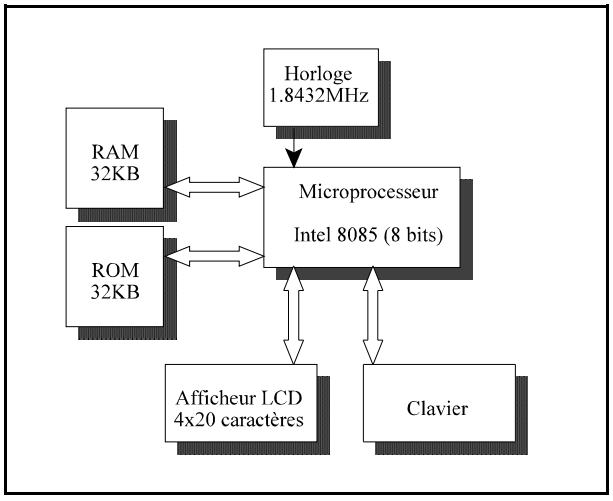

Figure 1.1 - Diagramme système - design original

1.3 Description du projet

Le design et la construction de l'ordinateur ayant été complétés avant d'en arriver à mon projet de fin d'études, je devais donc trouver un autre sujet. J'ai donc choisi de 'compléter' le système en le dotant d'un interpréteur de langage BASIC intégré. Le projet est divisé en deux parties:

- Matérielle: doter l'ordinateur de meilleurs périphériques d'entrée/sortie. L'afficheur LCD actuel ne permet que 4 lignes de 20 caractères, ce qui est nettement insuffisant pour entrer et exécuter des programmes BASIC.
- Logicielle: le coeur du projet, le design et l'implantation de l'interpréteur BASIC

1.4 Difficultés à surmonter

La partie matérielle se doit d'être la plus restreinte possible. En effet, le coeur du projet étant l'interpréteur en soi, il serait illogique de passer trop de temps à doter l'ordinateur d'une sortie VGA ou télé. Ce point pourrait à lui seul équivaloir à un projet de fin d'études. Il faut donc se concentrer à doter l'ordinateur d'entrées-sorties "temporaires", simples mais efficaces.

La partie logicielle est remplie d'embûches. Il est important de bien définir les limites du projet. Il est aussi crucial de bien établir le design sur papier avant de commencer l'implantation, sinon on risque de se retrouver à la fin du projet avec des milliers de lignes de code 'spaghetti' dont le fonctionnement est plutôt aléatoire. Un projet de cette envergure se doit d'être découpé en modules logiques et l'interaction entre ces modules doit être bien définie sinon on court à la catastrophe.

1.5 Intérêts et objectifs

Je termine ce chapitre en soulignant que ce projet n'est pas directement relié à mon travail ou à mes cours. Il découle directement de mon intérêt pour les sujets abordés. Je suis d'avis que ce type de projet est une excellente façon d'acquérir de nouvelles connaissances, ainsi que d'enfin pouvoir appliquer les notions apprises au fil des ans.

2. Méthodologie

La première étape de ce projet consiste à le découper en modules. Si le découpage est bien fait, on peut ensuite travailler sur chaque module de façon indépendante. Chaque partie est considérée comme une "boîte noire", sur laquelle on peut effectuer des tests. Les modules doivent être développés dans un ordre précis, puisqu'il existe certaines dépendances intrinsèques. Par exemple, le module d'évaluation d'expression dépend en partie du module de nombres entiers pour effectuer les opérations mathématiques. Les dépendances les plus évidentes doivent être prévues d'avance.

Pour l'implantation et les tests, il est nécessaire de disposer d'une panoplie d'outils (assembleur, émulateur, matériel électronique). Une partie de ce chapitre est donc réservée à la présentation des outils ainsi qu'à la justification de leur choix.

2.1 Partie matérielle

La majeure partie du projet étant l'interpréteur BASIC en soi, il est crucial de trouver une solution simple et efficace au problème matériel de l'affichage. Comme mentionné au chapitre précédent, il est irréaliste de vouloir développer une solution complète d'affichage pour ce projet. La solution logique est de déléguer la gestion de l'affichage à un tiers - par exemple un terminal.

Un terminal est composé de deux parties: un clavier et un écran. La communication s'effectue de manière sérielle dans les deux sens. Cette solution est la plus économique, car elle ne nécessite pas de support complexe du côté de l'ordinateur:

- La mémoire vidéo se situe du côté du terminal. On économise de la mémoire.
- Le terminal gère le curseur, le défilement, les couleurs. Encore une fois, on enlève beaucoup de charge au matériel. Le défilement à lui seul est assez complexe et demande beaucoup de temps processeur.

- Simplification du matériel au maximum. On évite d'avoir à concevoir des circuits complexes d'affichage, qui fonctionnent à une vitesse élevée par rapport au micro-ordinateur. On évite aussi des problèmes de synchronisation (accès à la mémoire vidéo, par exemple).

Du côté ordinateur, on élimine donc l'afficheur LCD et le clavier, qui sont remplacés par l'écran et le clavier du terminal. Pour la communication série, on doit ajouter un UART au système. En plus de l'UART, on ajoute un PIT (Programmable Infernal Tirer, un compteur programmable), qui sert d'horloge secondaire au système (pour le calcul de délais) et génère des sons simples (onde carrée). La version mise à jour du diagramme système se trouve à la figure 2.1.

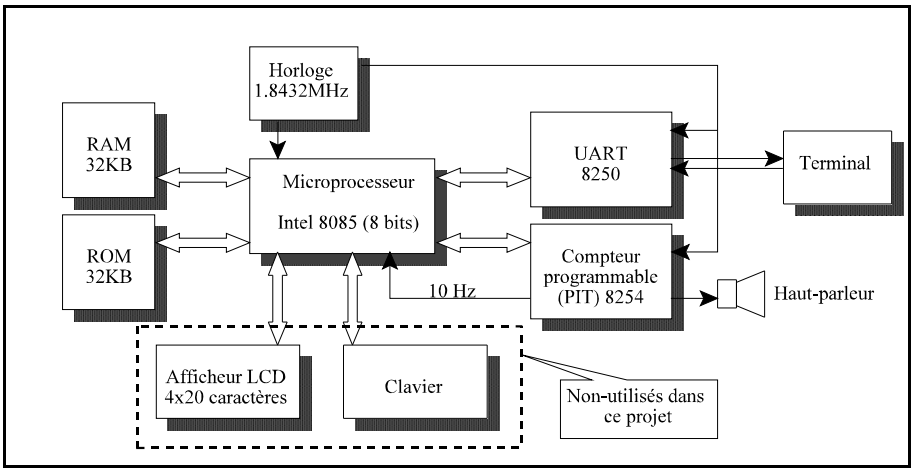

Figure 2.1 - Diagramme système - version révisée

L'UART employé est un 8250. C'est un composant standard des cartes séries trouvées autrefois sur les PCs. L'UART qu'on retrouve sur les PCs modernes est toujours compatible au 8250. Le compteur est un 8254, aussi un composant standard des premiers PCs (avant que tous les circuits soient intégrés sur une puce VLSI). Aujourd'hui encore, les PCs 'contiennent' un 8254, qui sert au rafraîchissement de la mémoire ainsi qu'à produire les sons du haut-parleur intégré. Ces deux composants sont décrits en détail dans plusieurs *Databooks*[2].

Une fois le mode d'affichage établi, il faut déterminer le protocole utilisé pour la communication avec le terminal. Envoyer du code ASCII "pur" n'est pas suffisant pour obtenir un affichage interactif intéressant. Il faut un moyen de positionner le curseur, d'effacer l'écran ou de changer les couleurs. La première idée envisagée est d'utiliser un protocole existant: le protocole ANSI[3]. Je me suis familiarisé avec ce protocole à l'époque des babillards électroniques (BBS), où l'ANSI était fréquemment utilisé pour "agrémenter" l'affichage des menus. Néanmoins, ce protocole a été rejeté pour plusieurs raisons:

- Coûts en caractères à transmettre. Pour positionner le curseur à la position (20,20), par exemple, la séquence à envoyer est "**<Esc>[20;20H**" (8 caractères). Pour changer la couleur de fond ou des caractères, la séquence est longue de 5 caractères.

- Coûts en conversions à effectuer. Le protocole exige que les nombres soient envoyés en notation texte. La couleur '34' doit être envoyée comme le caractère ASCII '3' suivi du caractère ASCII '4'. À l'interne, les nombres ne sont pas conservés sous forme de chaîne ASCII, mais bien sous la forme d'octets (8 bits) ou de mots (16 bits). La conversion d'entiers en chaîne de caractères ASCII est coûteuse au niveau processeur.

Vu les besoins limités pour ce projet, un protocole sur-mesure a été établi pour l'occasion. Évidemment, le fait d'utiliser un protocole non standard implique que le terminal comprenne ce langage. Un programme (en langage C) a donc été produit sur le PC qui tient lieu de terminal. Ce programme s'occupe de la communication série et de la gestion des caractères à l'écran. Le code de ce programme n'est pas présent dans ce rapport parce qu'il n'est pas considéré comme faisant partie de ce projet.

2.2 Découpage en modules

Après analyse, la partie logicielle de ce projet à été découpée en 9 modules. Voici une description de chacun de ces modules et une explication de leur raison d'être. Les détails d'implantation seront présentés au chapitre suivant.

2.2.1 Entrées/sorties (I/O)

Ce module sert d'interface avec le monde extérieur. On y retrouve des fonctions de haut niveau, qui sont utilisés par tous les autres modules:

Fonctions liées au terminal (UART):

- Afficher un caractère
- Afficher une chaîne de caractères
- Effacer l'écran
- Positionner le curseur
- Sélectionner une couleur
- Lire un caractère

Fonctions liées au compteur programmable (PIT):

- Émettre un son
- Gérer les délais

Il comporte aussi des fonctions de bas niveau, qui servent à l'initialisation des périphériques d'entrées/sorties:

- Initialisation des compteurs (PIT)
- Initialisation de l'UART
- Initialisation du tampon d'entrée (input buffer)

2.2.2 Gestion des erreurs

Un programme BASIC doit réagir de manière civilisée à une condition d'erreur. Une erreur peut se produire à plusieurs niveaux, dans la plupart des modules. Dans tous les cas, l'exécution du programme doit se terminer et un message d'erreur doit être affiché. La solution envisagée est similaire aux exceptions en C++. Lorsqu'une erreur se produit, une fonction d'erreur est appelée (équivalent à un *throw* en C++). Le module d'erreur prend alors la relève (*catch*). Il affiche le message d'erreur, vide la pile matérielle et la pile d'expression (voir plus bas), et saute à un point déterminé du programme principal. Le programme se retrouve donc en mode immédiat et tout peut continuer normalement.

2.2.3 Opérations sur les nombres entiers

L'interpréteur supporte des nombres notés sur 16 bits. Le processeur utilisé (Intel 8085) est un microprocesseur 8 bits dont le vocabulaire 16 bits n'est pas très étendu. Tout au plus, il peut additionner deux nombres de 16 bits. Il faut donc développer un bon nombre de fonctions qui opèrent sur les nombres entiers:

- Négation (Neg(5) = -5, Neg(-5) = 5)
- Addition (à compléter pour détecter les débordements)
- Soustraction (Dérivée de l'addition)
- Multiplication
- Division
- Racine carrée
- Opération binaires ET/OU/OU EXCLUSIF/NON à 'étendre' sur 16 bits
- Conversion entier à chaîne de caractères
- Conversion chaîne de caractères à entier
- Fonctions valeur absolue (ABS) et signe (SGN)
- Comparaison de deux entiers

2.2.4 Gestion des chaînes de caractères

Les chaînes de caractères sont très différentes des nombres entiers. Un entier, peu importe sa valeur, est représenté sur 16 bits. Une chaîne peut contenir de 0 à 255 caractères. Ceci implique une gestion relativement complexe de la mémoire. On doit déterminer à quel endroit les chaînes seront emmagasinées, comment créer ou détruire une chaîne, et comment éviter les 'trous' dans la mémoire. Ce module contient les fonctions suivantes:

- Allocation d'une nouvelle chaîne
- Libération de l'espace occupé par une chaîne
- Comparaison de deux chaînes de caractères
- Copie du contenu d'une chaîne
- Nettoyage de la mémoire (garbage collection)

2.2.5 Gestion des variables

L'utilisation des variables en BASIC est très simple pour l'usager. Une variable peut être utilisée directement, sans avoir été déclarée au préalable (comme en C ou en Pascal). Lorsque utilisée pour la première fois, une variable du type entier est initialisée à zéro. Une variable de type chaîne de caractères donne une chaîne vide. Il faut établir comment les variables seront représentées en mémoire. Il ne semble pas raisonnable de réserver de l'espace pour toutes les variables possibles, puisque même avec 2 caractères par nom de variable, on obtient plus de 1900 noms de variables différents (premier caractère: A-Z (26) * deuxième caractère: A-Z,0-9,nul (37) * variables entières et chaînes (2)). En supposant 5 octets par variable (ce qui est le cas dans ce projet), l'espace variable nécessiterait plus 9 KB, ce qui est peu raisonnable sur une machine de 32 KB de mémoire vive. Il semble donc plus logique de créer les variables au fur et à mesure qu'elles sont utilisées. Le module de gestion des variables doit donc contenir les fonctions suivantes:

- Assigner une valeur à une variable (Elle doit être créée si elle n'existe pas)
- Récupérer la valeur d'une variable (Créée et initialisée si elle n'existe pas)

2.2.6 Encodage/décodage des lignes du programme

Une ligne de programme entrée par l'usager peut être divisée en plusieurs parties: les mots-clés, les constantes numériques, les chaînes de caractères constantes, les variables, les séparateurs (; , :). Pour simplifier le travail du module d'évaluation d'expression, il semble une bonne idée de traiter les lignes de codes avant de les exécuter ou de les emmagasiner en mémoire. Le travail suivant est effectué:

- Extraction des mots-clés: les mots comme FOR, INPUT, PRINT, etc. sont remplacés par des jetons identifiant le mot-clé. Un jeton est un octet dont la valeur est supérieure à 128. Ces jetons peuvent facilement être identifiés par la suite et ils ne risquent pas d'empiéter sur les autres lettres de la ligne (comme les noms de variables).

- Conversion des constantes numériques: les nombres comme "1234" sont convertis en un nombre entier de 16 bits.

- Conversion des chaînes de caractères: le contenu des chaînes n'est pas touché en soi. Leur longueur est par contre calculée et emmagasinée au devant du premier caractère de la chaîne. Les guillemets sont éliminés.

- Conversion des noms de variables: Seuls les deux premiers caractères d'un nom de variable sont significatifs. Le reste est ignoré. Les noms de variables, entier ou chaîne, sont encodés sur deux octets.

Le résultat est une ligne de code 'facile à lire' pour le module d'évaluation d'expression. Ceci rend l'exécution plus efficace. Voici un exemple:

```
10 FOR I = 1 TO 100
20 VARIABLE = VARIABLE + 1
30 NEXT
```

Concentrons-nous sur la ligne 20. Sans encodage de la ligne, le module d'évaluation d'expression doit convertir 100 fois la chaîne "1" en nombre 1, 200 fois la variable VARIABLE en 'VA'. La version encodée est beaucoup plus simple, en plus d'être plus efficace.

Le module contient donc deux fonctions principales:

- Encodage d'une ligne de code source
- Décodage d'une ligne (pour l'affichage lorsque le programme est listé)

2.2.7 Évaluation d'expressions

C'est à ce module que revient les deux tâches les plus importantes: évaluer les expressions, ainsi qu'exécuter les instructions (pour plus de détails sur les expressions et les instructions, consulter l'annexe A).

Le processus d'évaluation d'expression repose sur l'usage d'une pile. Lors d'une addition de deux entiers, par exemple, on retire les deux nombres de la pile, on calcule le résultat et on le place sur la pile. Il en va de même pour la plupart des opérations. À la fin de l'évaluation d'une expression, aussi compliquée soit-elle, on se retrouve avec un seul résultat sur la pile (un résultat entier ou une chaîne, selon le type d'expression).

Une expression à elle seule ne sert pas à grand chose. C'est au coeur d'une instruction qu'elle joue son rôle. Une instruction se sert d'une ou plusieurs expression pour effectuer un travail. Par exemple, une instruction PRINT sert à afficher à l'écran le résultat d'une ou de plusieurs expressions. Une instruction GOTO sert à modifier le flot du programme selon le résultat de l'expression qui la suit - un numéro de ligne. Chaque instruction effectuant un travail différent, il faut prévoir une sous fonction par instruction. Ce module contient donc un grand nombre de fonctions:

- Evaluation d'expression (composée de plusieurs sous fonctions)
- Gestion de la pile d'évaluation (empiler/dépiler)
- Exécution des instructions (composée de plusieurs sous fonctions)

2.2.8 Gestion du programme

Sans ce module, l'interpréteur ne fonctionnerait qu'en mode immédiat - réagissant immédiatement aux instructions entrées par l'usager. C'est ce module qui gère l'espace occupé par le programme en mémoire et qui traite l'ajout et le retrait de lignes à celui-ci. En plus, il exécute les instructions qui on un lien avec le flot d'exécution du programme. Voici les fonctions de ce module:

- Ajout d'une ligne à un programme
- Retrait d'une ligne
- Effacement du programme (NEW)
- Exécution du programme (RUN)
- Fonctions de branchement (GOTO/GOSUB)
- Fonctions de répétition (FOR...NEXT)

2.2.9 Module principal

Ce module a comme rôle l'intégration de tous les modules. C'est celui-ci qui initiative les modules et qui contient la boucle principale. Voici le fonctionnement de cette boucle:

1. Lire une ligne venant du terminal
2. Encoder la ligne (voir section 2.2.6)
3. Déterminer s'il s'agit d'ajout ou de retrait d'une ligne. Si oui, effectuer l'opération et retournr à 1.
4. Sinon, exécuter le contenu de la ligne
5. Retour au point 1

Techniquement, cette boucle est infinie. Une condition d'erreur à un point ou un autre repart l'exécution au point 1.

2.3 Organisation mémoire

L'organisation de la mémoire est très similaire à celle retrouvée sur le Commodore 64, dont je me suis beaucoup inspiré au niveau architecture[4]. Le résultat est présenté à la figure 2.2. Au niveau matériel, l'espace mémoire est déjà découpé en deux parties: la ROM et la RAM. La ROM contient évidemment tous les modules qui composent l'interpréteur BASIC. La RAM doit contenir les éléments suivants:

- Variables programme - les variables utilisées par l'interpréteur
- Pile expression - sert au module d'évaluation d'expression
- Programme BASIC - endroit où est stocké le programme de l'usager
- Variables BASIC - les entiers et chaînes de caractères utilisés par le programme BASIC
- Chaînes de caractères - le contenu des chaînes étant dynamique, celui-ci n'est pas conservé au même endroit que les variables BASIC. Une variable de type chaîne contient simplement un pointeur sur le contenu de la chaîne.
- Pile système - c'est la pile 'matérielle', utilisée par le processeur et l'interpréteur BASIC

Les variables utilisées par l'interpréteur sont de tailles fixes. L'espace qu'elles occupent est connu au moment de la compilation. Il est donc logique que ces variables occupent le début de la mémoire RAM. La pile d'expression est aussi de taille fixe - définie à la compilation. Elle se situe donc à la suite des variables programme.

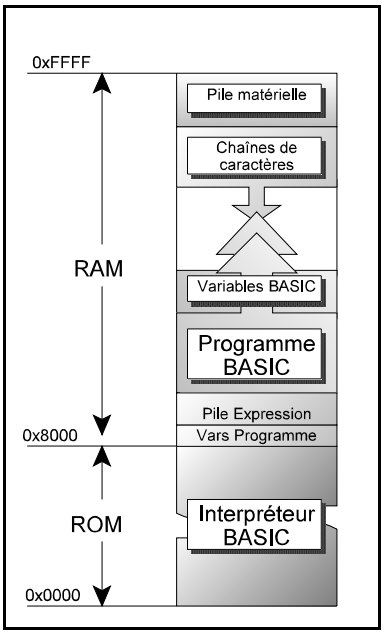

Figure 2.2 - Organisation mémoire du système

Les variables BASIC peuvent être créées pendant l'exécution d'un programme. Autant que possible, il ne faut pas limiter la taille qu'elles peuvent occuper. Ainsi, l'espace variable est placé à la toute fin du programme BASIC. Ce choix est logique, puisque la taille d'un programme BASIC ne peut pas changer pendant son exécution. Le seul inconvénient, qui n'en est pas vraiment un, est que lorsqu'on ajoute ou retire une ligne au programme, toutes les variables sont effacées. C'est un comportement commun à plusieurs interpréteurs BASIC, qui procèdent probablement de la même manière. Donc, après la pile d'expression, la mémoire est occupée de bas en haut par le programme BASIC, suivi de l'espace variables.

Il reste deux blocs de mémoire à allouer: la pile système et les chaînes de caractères dynamiques. La pile système est fixée à une taille raisonnable (ici 1KB), qui ne risque pas de déborder pendant l'exécution du programme. Elle est située au sommet de l'espace mémoire. Il reste à allouer l'espace pour les chaînes dynamiques. On ne peut pas les allouer immédiatement après les variables BASIC, puisqu'on veut conserver ces deux espaces séparés, et que l'espace variable peut grossir pendant l'exécution d'un programme. Les chaînes sont donc allouées du haut de la mémoire, vers le bas.

2.4 Outils

Cette section se concentre sur les outils logiciels choisis pour le développement du projet (les outils matériels n'ont pas vraiment été choisis - ils étaient déjà en ma possession).

2.3.1 Assembleur 8085

En début de projet, le choix d'un nouvel assembleur s'est fait sentir. En effet, l'assembleur utilisé jusqu'à maintenant (asm85[5]) s'avère trop simple pour le développement d'un projet de cette taille. Son plus gros défaut est qu'il ne supporte pas la notion de modules/librairies. Il aurait fallu inclure tout le code du projet dans un seul fichier, ce qui rend presque impossible les tests indépendants des modules, en plus de causer des difficultés à s'y retrouver dans le code.

Après de longues recherches, l'assembleur 'idéal' a été trouvé. Il s'agit de AS8085[6], un assembleur mutliplatforme gratuit qui vient même avec le code source. Cet assembleur est beaucoup plus complet que le précédent, et permet de gérer beaucoup plus facilement les modules. Il est possible d'avoir chaque module dans un répertoire différent. Un répertoire contient le fichier source du module et un fichier source de tests. Cette manière de procéder encourage la modularité et les tests fonctionnels.

2.3.2 Émulateur 8085

Le développement d'un projet de cette envergure est grandement facilité par l'usage d'un émulateur. Sans cet outil, la tâche est plus ardue: pour tester un module ou une modification, il faut transférer le code binaire de la machine de développement (un PC) vers la machine de destination. On travaille alors à l'aveuglette, car le code qui s'exécute n'est pas visible. On ne peut facilement étudier l'état des registres ou des données en mémoire. Bref, tout ce qu'on peut faire, c'est des tests de type 'boîte noire': on donne une entrée, et on regarde le résultat.

Avec un émulateur, on peut examiner le fonctionnement du code directement sur la machine de développement. Il est alors possible de trouver les problèmes et de les régler beaucoup plus rapidement. L'émulateur utilisé dans le cadre de ce projet est **Win85**[7], un émulateur gratuit fonctionnant sous Windows.

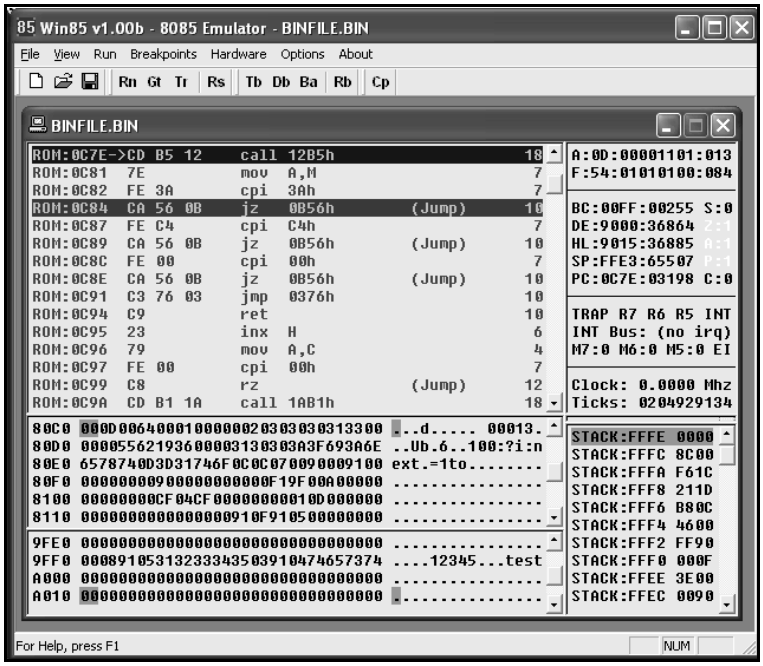

Figure 2.3 - Émulateur Win85

2.3.3 Système de contrôle des versions

À mon travail, nous utilisons l'outil CVS[8] pour le contrôle des versions et des révisions. C'est un outil essentiel lorsqu'on travaille sur des projets d'envergure. Il peut sembler superflu d'utiliser un tel outil pour ce projet, puisque je ne suis qu'un seul développeur à y travailler. Néanmoins, CVS apporte une aide inestimable à plusieurs niveaux:

- Accès au code source à partir de plusieurs machines . Toute machine sur un réseau peut se brancher au serveur CVS et mettre à jour sa version des

fichiers source. Il est ainsi possible de travailler à distance (avec un portable, par exemple). Lorsque les changements sont terminés, il suffit de faire un "check-in" pour que toutes les machines branchées au serveur aient accès à ces nouveaux fichiers.

- La sauvegarde des données du serveur peut facilement être automatisée. On ne peut se permettre de perdre tous les fichiers source en cas de défectuosité matérielle.

- Toutes les versions des fichiers sont conservées. On peut facilement obtenir le sommaire des modifications d'un fichier ou revenir en arrière si l'on se rend contre qu'un changement effectué récemment cause des problèmes. On peut voir exactement quelles lignes ont changé.

3. Réalisation

La réussite du projet dépend d'une implantation soignée de chacun des modules qui le compose. Ce chapitre présente les détails de réalisation de toue les modules. Évidemment, cette section n'est pas une dissertation sur chaque ligne de code, mais bien un résumé des points importants, des concepts et des structures de données employés dans chaque partie du projet.

Le chapitre commence par présenter un survol de l'implantation de la partie matérielle. Ensuite, les détails importants de chaque module sont énoncés. Le chapitre se termine par une section sur l'intégration des modules pour créer le produit fini.

3.1 Niveau matériel

Ce projet se concentrant surtout sur la partie logicielle, cette section sera brève.

3.1.1 Communications avec le terminal (UART)

L'implantation matérielle de l'UART est assez simple. Ce composant s'intègre au système au même titre qu'un circuit de mémoire ou un autre périphérique. L'UART possède trois bits d'adresse, il nécessite donc huit adresses dans l'espace d'adressage. Le microprocesseur dispose de huit bits pour les périphériques d'entrées/sorties. Puisque le nombre de périphériques est limité, seuls les trois bits supérieurs sont décodés, ce qui donne huit blocs disponibles pour les périphériques. L'UART hérite donc de la plage d'adresses 0x60-0x7F, dont seules les adresses 0x60 à 0x67 sont utilisés.

L'UART est programmé pour générer une interruption lors de la réception d'un caractère. Il faut donc implanter une routine de traitement des interruptions du côté du microprocesseur. Cette manière de procéder évite de perdre des caractères pendant l'exécution de certaines fonctions longues. À cette fin, il faut implanter un tampon de réception du côté logiciel (voir section 3.2.1).

Après quelques tests, il a été déterminé que la vitesse idéale de transmission était de 9600 bauds. Au-delà de cette valeur, on assiste à des problèmes de corruption des données ou de caractères manquants. En plus, lors des transferts d'une grande quantité d'information (i.e. lorsque le terminal envoie plusieurs lignes une après l'autre, pour simuler un 'LOAD') il est fréquent de perdre des caractères. Ceci s'explique par le fait qu'un traitement est effectué après l'arrivée de chaque ligne de code. Le terminal commence à envoyer la prochaine ligne avant que le traitement ne soit terminé, ce qui cause parfois un débordement du tampon de réception (16 caractères). La solution la plus simple aurait été d'accroître la taille du tampon, mais le problème risque quand même de se produire dans certains cas.

La solution choisie est d'utiliser une ligne de contrôle de l'UART pour servir de contrôle de flot. Après réception d'une ligne de code, la ligne DTR (Prêt à recevoir) est désactivée. Lorsque le terminal détecte que cette ligne est à l'état désactivé, il cesse d'envoyer des caractères. Aussitôt que l'ordinateur est prêt à recevoir de nouveau, il change l'état de la ligne DTR et, à cet instant, le terminal recommence à envoyer des données.

3.1.2 Horloge secondaire et effets sonores (PIT)

Le composant utilisé pour l'horloge (8254) dispose de trois compteurs programmables. Du côté matériel, son installation est semblable à celle de l'UART. Le compteur dispose de deux lignes d'adresse, donc de quatre registres. Sa plage d'adresse est 0x40 a 0x5F (juste avant l'UART).

Deux des trois compteurs sont utilisés. Le premier est programmé pour générer un signal périodique à tous les 100ms (ou à 10Hz). Ce signal est branché directement à la ligne TRAP du microprocesseur. Un signal sur cette ligne génère une interruption non-masquable (qu'on ne peut ignorer au niveau logiciel). Comme dans le cas de l'UART, il faut implanter une fonction de gestion de l'interruption TRAP. Cette fonction se contente d'incrémenter un compteur de 16 bits à chaque appel. La valeur de ce compteur est utilisée par le module d'entrée/sorties pour calculer des délais.

Le deuxième compteur sert à produire des sons. Il est donc programmé pour générer une onde carrée donc la fréquence est calculée selon la note à produire. La fréquence de sortie est calculée comme une fraction de la fréquence d'horloge (1.8432 MHz). Le compteur peut diviser par un nombre de 16 bits non signé (où zéro correspond à 65536). La plage des valeurs est de 1 à 65536, ce qui permet des fréquences de 28 Hz à 1.8432 MHz, ce qui couvre amplement la plage audible (20 Hz à 20Khz). La sortie de ce compteur (une onde carrée) est directement branchée à un petit haut-parleur.

3.2 Niveau logiciel

Dans cette section, on revient sur chaque module énoncé au chapitre précédent, en portant une attention particulière aux détails d'implantation importants.

3.2.1 Entrées/sorties (I/O)

La plupart des fonctions d'affichage au terminal appellent IO_PUTC. Cette fonction attend que l'UART soit prêt à recevoir un caractère. Ensuite, le caractère à envoyer est placé dans le registre de sortie

Du côté réception, la routine de gestion de l'interruption envoyée par l'UART se charge de récupérer le caractère reçu. Ensuite, le caractère est placé dans un tampon circulaire de 16 octets. Il n'y a pas de gestion des débordements: si le tampon est plein et qu'un caractère est reçu, le caractère le plus 'ancien' est remplacé. La fonction IO_GET retire un caractère de ce tampon. Si ce dernier est vide, la fonction retourne zéro.

La fonction IO_DELAY sert à effectuer une pause d'un temps déterminé par le paramètre d'entrée (1..255). Son fonctionnement est très simple: on lit tout d'abord la valeur du compteur 16 bits

(mentionné à la section 3.1.2). Le paramètre reçu est ensuite additionné à cette valeur et la fonction boucle tant que la valeur du compteur n'est pas arrivée à ce résultat.

La fonction IO_SOUNDON reçoit un numéro de note en paramètre. Ce nombre correspond à un diviseur trouvé dans une table. Cette table est pré-calculée en divisant la fréquence de l'horloge (1.8432 MHz) par la fréquence de chaque note[9]. La table contient 7 octaves à 12 notes par octave (84 nombres de 16 bits). Une fois le diviseur extrait de la table, le compteur est programmé et sa sortie est activée. La fonction IO_SOUNDOFF ne fait que désactiver la sortie pour arrêter le son.

3.2.2 Gestion des erreurs

Ce module est composé de 18 fonctions d'erreurs presque identiques. Chaque fonction ne fait que charger un pointeur sur le message d'erreur qui lui est associé et appelle ensuite la fonction d'erreur principale.

La fonction d'erreur commence par afficher le message d'erreur au terminal. Ensuite, on vérifie si le numéro de la ligne courante du programme est zéro ou non. Dans le cas ou le numéro est non-nul (c'est-à-dire qu'un programme est en train de s'exécuter), le numéro de la ligne est aussi affiché à la droite du message d'erreur. La routine vide ensuite la pile d'expression (voir section 3.2.7) ainsi que la pile système. Ensuite, la fonction fait un saut à un endroit prédéterminé du module principal, où la boucle d'exécution continue normalement.

3.2.3 Opérations sur les nombres entiers

Comme mentionné à la section 2.2.3, le processeur 8085 ne dispose pas de beaucoup de fonctions traitant les nombres de 16 bits. Tout au plus, il peut en additionner deux ensemble, en plus de pouvoir incrémenter (+1) ou décrémenter (-1) un nombre.

Puisque le processeur ne dispose que de quelques registres, le module utilise quatre variables temporaires en RAM (INT_ACC0 à INT_ACC3). Ces variables sont utilisés dans la plupart des opérations sur les entiers.

La fonction de négation (INT_NEG) calcule l'inverse d'un nombre. Il s'agit simplement d'inverser tous les bits du nombre et d'additionner 1 au résultat (complément-à-2).

La fonction d'addition (INT_ADD) 'enrobe' la fonction du processeur pour additionner deux nombres. En effet, il faut vérifier s'il y a débordement (par exemple: 32767 + 1). Une variable (INT_OVERFLOW) est créée à cet effet. Les modules utilisant les fonctions de la librairie doivent s'assurer de vérifier l'état de cette variable après la chaque opération qui risque de causer un débordement (addition, soustraction et multiplication).

La fonction de soustraction (INT_SUB), se contente d'inverser le 2^{e} opérateur (INT_NEG) et d'effectuer une addition (INT_ADD).

Pour comparer deux nombres (INT_CMP), on procède à leur soustraction et on analyse le résultat (signe de la réponse, état de la variable de débordement) pour déterminer si le premier nombre est supérieur, inférieur ou égal au deuxième (Table 3.1)[10][11].

Table 3.1 - Comparaison de nombres signés

Résultat	**Condition**
Nombre1 < Nombre2	Résultat différent de zéro Bit de signe égal au bit de débordement
Nombre1 = Nombre2	Résultat égal à zéro
Nombre1 > Nombre2	Bit de signe différent du bit de débordement

La multiplication (INT_MUL) est effectuée par additions et décalages successifs[12]. Pour la division, on utilise un algorithme sans restoration[13].

La conversion d'entier à chaîne de caractères (INT_ITOA) effectue une conversion binaire-à-BCD[14]. Une fois la conversion en BCD effectuée, il est facile de faire la transition en chaîne ASCII.

Finalement, la conversion de chaîne de caractères à entier fonctionne caractère par caractère, avec des multiplications successives du résultat par 10.

3.2.4 Gestion des chaînes de caractères

Les chaînes de caractères sont allouées du haut de la mémoire vers le bas. Chaque chaîne possède un en-tête qui indique sa longueur et un pointeur vers son parent (Figure 3.1). L'espace occupé par une chaîne n'est pas libéré lorsqu'on l'efface, le pointeur vers son parent est simplement remis à zéro. La mémoire est libérée plus tard, lorsque nécessaire.

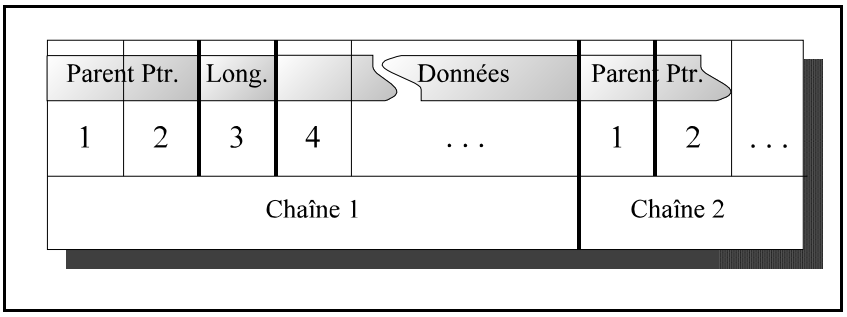

Figure 3.1 - Représentation des chaînes de caractères en mémoire

La fonction d'allocation de mémoire (STR_ALLOC) vérifie s'il y a suffisamment d'espace pour la chaîne à allouer. Si la mémoire est pleine, on appelle la fonction de nettoyage de la mémoire (STR_GARBAGECOLLECTION), qui libère un bloc contigu de chaînes 'orphelines' (dont le pointeur est à zéro). La fonction d'allocation appelle la fonction de nettoyage tant que l'espace libéré n'est pas suffisant pour la nouvelle chaîne. En cas d'échec (plus rien à nettoyer), on signale une erreur d'épuisement de la mémoire.

Comme mentionné plus tôt, la fonction de libération de mémoire (STR_FREE) ne fait que remettre à zéro le pointeur parent de la chaîne à libérer.

La fonction de nettoyage de la mémoire (STR_GARBAGECOLLECTION) libère un bloc contigu de chaînes 'orphelines' à la fois. La fonction procède en partant du bas de la mémoire, en allant vers le haut. On cherche d'abord la première chaîne ayant un parent nul. On marque ce point. On continue ensuite jusqu'à la première chaîne trouvée dont le parent est non nul, et on marque ce point. Nous avons maintenant un bloc d'une ou plusieurs chaînes vides (Si le bloc est vide, la fonction se termine). On décale alors les chaînes se trouvant avant le bloc, écrasant ainsi les chaînes n'étant plus utilisées. Finalement, on doit mettre à jour les variables de type chaîne de caractères dont la position a changé en mémoire. Ceci est fait en itérant parmi toutes les variables, en mettant à jour les pointeurs des blocs ayant été déplacés.

3.2.5 Gestion des variables

Une variable est représentée par un bloc de mémoire de 5 octets. La structure d'une variable est présentée à la figure 3.2. Le format est inspiré de celui utilisé sur le Commodore 64[15]. Évidemment, le format utilisé ici est beaucoup plus simple, puisque seulement deux types de variables sont supportés.

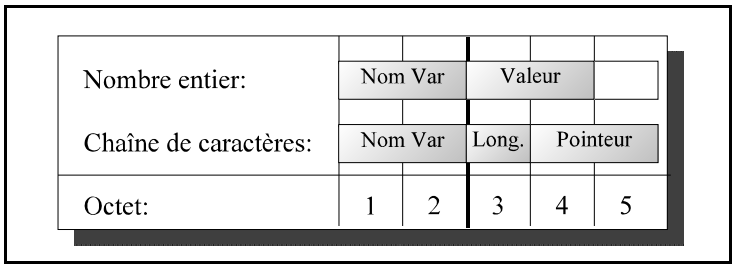

Figure 3.2 - Représentation d'une variable en mémoire

On note qu'une variable de type chaîne de caractères ne contient qu'un pointeur sur son contenu, tandis qu'une variable de type entier comprend sa valeur. Les deux premiers octets contiennent le nom de la variable, en ASCII à quelques détails près:

- Le bit 7 du premier caractère est mis à 1 pour indiquer une chaîne de caractère
- Le deuxième caractère est nul (0) si le nom de la variable ne contient qu'une lettre

Le module de gestion des variables est composé de deux fonctions principales. La fonction d'affectation d'une valeur à une variable (VAR_SET) commence par identifier si la variable mentionnée existe déjà. Sinon, il faut tout d'abord la créer (ce qui l'initialise à 0 ou ""). Ensuite, l'affectation de la valeur est effectuée. La fonction qui retourne la valeur d'une variable procède de la même façon: en créant la variable si elle n'existe pas, puis en retournant sa valeur.

3.2.6 Encodage/décodage des lignes du programme

L'encodage est fait en deux étapes. La première phase (TOK_TOKENIZE1) est effectuée directement sur la chaîne à encoder. Il s'agit d'identifier tous les mots-clés et de les remplacer par le jeton d'identification. Le mot PRINT, par exemple, est remplacé par la séquence suivante:

"P	R	I	N	T"
[0xD1]	[0xFF]	[0xFF]	[0xFF]	[0xFF]

Le premier caractère du mot-clé est remplacé par le jeton, les caractères subséquents par la valeur 255, qui sert de 'remplissage' (ces caractères seront retirés à la phase 2). À cette étape, les variables et les constantes sont ignorées.

La deuxième étape d'encodage (TOK_TOKENIZE2) fonctionne avec une chaîne d'entrée (le résultat de TOK_TOKENIZE1) et une chaîne de sortie. À cette étape, les noms de variables sont convertis

en identificateur de deux octets, les nombres sont encodés, et on retire les guillemets des chaînes de caractères, qu'on précède par leur longueur.

La chaîne encodée est maintenant prête à être insérée dans la section programme (pour plus tard) ou à être exécutée par le module d'évaluation d'expressions.

Il existe aussi une fonction de décodage (TOK_UNTOKENIZE) qui fait le processus inverse. Cette fonction n'étant utilisée que pour lister le programme (LIST), la sortie est envoyée directement au terminal.

3.2.7 Évaluation d'expressions

La fonction d'évaluation d'expression (EXP_EVALUATE) procède à l'analyse syntaxique d'une expression suivant la méthode de récursivité indirecte présentée au chapitre *Complément de programmation: Analyse Syntaxique* du livre Algorithmes et structures de données[16]. Voici un exemple simple tiré du livre qui permet de comprendre le fonctionnement de la fonction d'évaluation d'expression:

Définitions:

- Une expression est une somme ou une différence de deux termes.
- Un terme est un produit ou un quotient de deux facteurs.
- Un facteur est une lettre ou une expression.

Algorithme:

Expression:

- Prendre un premier terme;
- Tant qu'on a un + ou un - entre le premier terme et le deuxième:
  - Prendre un terme;
  - Mettre l'opérateur après ce terme;

Terme:

- Prendre le premier facteur;
- Tant qu'on a un * ou un / entre le premier facteur et le deuxième:
  - Prendre un facteur;
  - Mettre l'opérateur après ce facteur;

Facteur:

- Prendre un caractère
- Si le caractère est une lettre:
  - Retourner le caractère;
- Sinon:
  - Si le caractère est une parenthèse:
  - Prendre une expression;
  - Lire la parenthèse qui reste;

La fonction d'évaluation d'expression implantée dans ce projet fonctionne sur des bases presque identiques à l'exemple présenté ci-haut. Les différences sont les suivantes:

- L'exemple comporte trois niveaux. Cette implantation comporte 8 niveaux différents, qui enforcent la priorité des opérateurs (moins prioritaire en premier):
  - Niveau 0: AND, OR, XOR
  - Niveau 1: NOT
  - Niveau 2: =, <>, < >, <=, >=
  - Niveau 3: +, -
  - Niveau 4: *, /
  - Niveau 5: - (négation - opérateur unaire)
  - Niveau 6: ^ (puissance, présent mais non implanté)
  - Niveau 7: Facteur

- Le dernier niveau (Facteur) ne traite pas que de 'lettre' ou d'expression il traite de:
  - Expression entre parenthèses
  - Variable
  - Constante entière
  - Constante chaîne de caractères
  - Fonction (ex: ABS(expression))

À chaque niveau d'opérateur ou de fonction correspond une fonction qui retire de la pile les paramètres nécessaires, effectue l'opération et place le résultat sur la pile. À la fin de tout le

processus, si l'expression est bien formée, il ne reste qu'un seul élément sur la pile d'expression: le résultat de l'expression.

La pile d'expression est un bloc mémoire de taille fixe. Ce bloc peut contenir un nombre défini d'éléments: entiers, chaînes de caractères ou variables. Chaque élément prend le même espace sur la pile (5 octets). La taille de la pile a été choisie pour pouvoir contenir 16 éléments. Le bloc mémoire réservé a donc une taille de 80 octets. La structure d'un élément sur la pile est présenté à la figure 3.3.

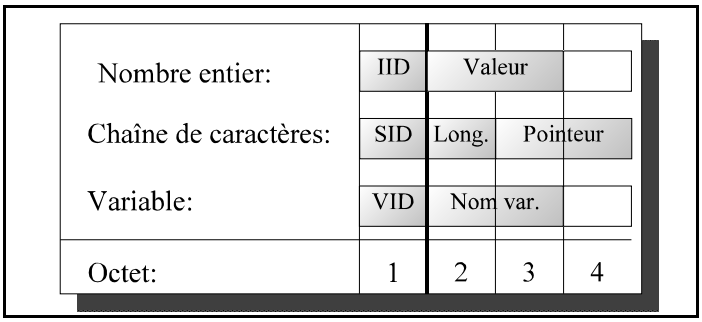

Figure 3.3 - Structure des éléments de la pile d'expression

En soi, une expression ne fait pas de travail utile. Ce sont les instructions BASIC qui font le travail. Ces instructions peuvent nécessiter zéro, un ou plusieurs paramètres, qui sont des expressions. Une ligne BASIC peut contenir une multitude d'instructions, séparées par le symbole ':'. C'est la fonction EXP_EXPREVAL qui parcourt une ligne, identifie les instructions et appelle les fonctions associées. Ces fonctions (DO_PRINT, DO_INPUT, etc.) quant à elles, extraient les paramètres dont ils ont besoin en appelant la fonction d'évaluation d'expression (EXP_EVALUATE) et en retirant le résultat de la pile.

3.2.8 Gestion du programme

Ce module contient les fonctions de gestion et d'exécution d'un programme. La première catégorie de fonctions traite de l'ajout, du retrait ou du remplacement d'une ligne de code (PRG_INSERT, PRG_REMOVE). Un programme est conservé en mémoire selon l'ordre numérique des lignes. Ainsi, s'il existe une ligne 10 et 20, et qu'on désire ajouter la ligne 15, Le bloc allant de la ligne 20 à la fin du programme est déplacé pour faire de l'espace pour l'insertion. Similairement, lorsqu'une ligne est retirée, le bloc de mémoire qui la suit est déplacé pour éliminer le 'vide'. Le format des lignes en mémoire est présenté à la figure 3.4.

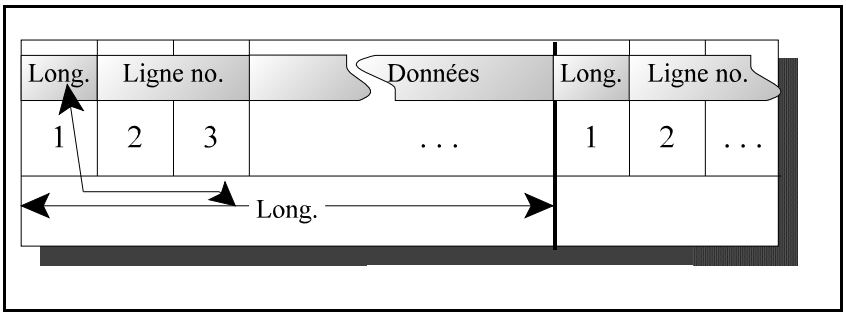

Figure 3.4 - Structure des lignes d'un programme

Ce module implante aussi les instructions relatives au programme: le listing (PRG_LIST), les branchements (PRG_GOTO, PRG_GOSUB), les boucles (PRG_FOR). Le coeur de ce module est la fonction 'do it' (PRG_DOIT), qui exécute les lignes une après l'autre, gère les branchements et les conditions d'arrêt. Le déplacement 'normal', d'une ligne à la suivante, s'effectue en prenant le pointeur sur la ligne courante, et en y additionnant le contenu du premier octet du bloc (sa taille). Le résultat pointe sur le début de la ligne suivante. Il est donc très facile d'avancer linéairement dans le code. Pour les branchements, on doit utiliser une fonction de recherche (PRG_FIND), qui balaye le bloc programme du début à la fin, à la recherche de la ligne en question. Les fonctions GOSUB /RETURN et FOR/NEXT évitent d'avoir à effectuer deux recherches en conservant sur la pile l'adresse de départ. Ainsi, lorsqu'on atteint l'instruction RETURN, on retire de la pile l'adresse de la ligne contenant l'instruction GOSUB.

3.2.9 Module principal

Le fonctionnement de ce module a été décrit en suffisamment de détails à la section 2.2.9. Son implantation étant très simple, elle n'est pas matière à discussion dans ce chapitre.

3.3 Intégration des modules

Au fur et à mesure que les modules sont développés, on voit apparaître les dépendances entre eux. Lors de l'implantation du module d'évaluation d'expression, les modules de gestion des variables, des entiers et des chaînes de caractères sont presque complétés. Tous les morceaux commencent à tomber en place: le programme de test du module d'évaluation d'expression ressemble comme deux gouttes d'eau au module principal (sans la boucle). L'intégration des modules se fait donc tout au long du projet. On se retrouve avec quelques lignes de code à écrire et l'interpréteur est complet!

4. Discussion

Ce dernier chapitre présente une discussion du travail réalisé dans le cadre de ce projet. Celui-ci est divisé en deux sections. La première partie présente une critique des méthodes exploitées. La deuxième partie donne des recommandations pouvant donner de nouvelles orientations au projet.

4.1 Critique des méthodes exploitées

4.1.1 Entrées/sorties

Il est certain que la solution d'utiliser un terminal pour l'affichage est loin d'être optimale. Par contre, compte tenu des contraintes au niveau temps, il s'agit d'une solution adéquate, qu'il est toujours possible d'améliorer dans le futur (voir section 4.2).

4.1.2 Gestion des chaînes de caractères

Le choix de ne nettoyer la mémoire que lorsque celle-ci est pleine est discutable. C'est la méthode employée sur le Commodore 64, et dans certains programmes cela cause des pauses inacceptables. Dans tous les cas, il existe une 'porte de sortie': comme sur le Commodore, la fonction FRE() appelle la fonction de nettoyage de mémoire. Il est possible de l'employer judicieusement dans un programme pour 'étaler' les pauses afin qu'elles soient moins perceptibles.

4.1.3 Gestion des variables

Le choix de ne supporter que les entiers et les chaînes a été effectué en fonction des contraintes de temps. L'absence de tableaux ou de nombre à virgule flottante limite beaucoup la fonctionnalité de l'interpréteur.

4.1.4 Encodage/décodage des lignes du programme

Le fait d'encoder les lignes avant de les conserver en mémoire est une décision qui a été prise pour des raisons de performance et de simplicité. Elle a le net inconvénient de 'modifier' ce que l'usager entre: lorsqu'il liste le programme, certains changements subtils (ou moins subtils) peuvent apparaître:

```
10 MONTANT = VARIABLE * TAUX
```

Devient, après codage/décodage:

```
10 MO = VA * TA
```

D'un autre côté, puisque seuls les deux premiers caractères sont significatifs, on évite probablement des erreurs en n'affichant pas les caractères 'superflus'. Si l'interpréteur supportait les nombres à virgule flottante, on pourrait aussi avoir ce type de changement:

```
10 VA = 1E-2
```

Pourrait devenir, après codage/décodage:

```
10 VA = .01
```

Ce qui est équivalent mais ne correspond pas à la notation utilisée par l'usager.

4.1.5 Gestion du programme

La fonction de recherche d'une ligne de code (utilisée par GOTO/GOSUB) pourrait être plus intelligente. Sous sa forme actuelle, elle démarre toujours sa recherche à partir du début du programme. Il serait possible de comparer le numéro de la ligne à rechercher avec celui de la ligne courante, et d'en servir comme point de départ pour la recherche si la ligne désirée se trouve plus loin.

4.2 Orientations possibles

Si l'intérêt s'y trouve, il existe beaucoup de possibilités de développements au projet dans son état actuel.

La partie matérielle du projet a été mise de côté assez rapidement. Il serait très intéressant en soi de couper la dépendance de l'ordinateur à un terminal. Il serait possible de développer un circuit d'affichage vidéo qui pourrait rendre l'ordinateur indépendant. C'est un projet très intéressant, surtout orienté au niveau matériel. Lorsque j'ai fait le choix de mon projet de fin d'études, c'était presque une question de 'pile ou face' qui m'a fait décider d'aller vers l'interpréteur BASIC (en fait, c'est surtout parce qu'un ordinateur avec affichage mais sans programme est moins intéressant qu'un ordinateur fonctionnel sans affichage).

La partie interpréteur BASIC pourrait aussi être de beaucoup améliorée. Faute de temps, il n'a pas été possible d'intégrer un module de nombre à virgule flottante. Les capacités de calculs de l'ordinateur en sont très réduites. Les structures de données ont été conçues initialement pour supporter ce type. Il serait donc relativement facile d'intégrer une telle librairie à l'interpréteur.

Et puisqu'il reste de la ROM à revendre (l'interpréteur dans sa forme actuelle ne 'consomme' que 7.5KB de ROM, sur 32KB possibles), il serait aussi possible d'ajouter de nouvelles instructions au langage. Par exemple, des fonctions de répétitions du type WHILE...WEND ou REPEAT...UNTIL pourraient s'avérer un ajout intéressant.

Bibliographie

1. The History of BASIC - Beginner's All Purpose Symbolic Instruction Code
http://inventors.about.com/library/inventors/blbasic.htm

2. Harris Semiconductors, Microprocessors Products for Commercial and Military Digital Applications, 1996, pp. 4-198 à 4-214, 5-3 à 5-22.

3. ANSI.SYS -- ansi terminal emulation escape sequences
http://enterprise.aacc.cc.md.us/~rhs/ansi.html

4. Carmichel, D., Key Memory Locations, *Compute!'s Gazette*, vol. 2, no.12, décembre 1984, pp. 154-157.

5. Dunfield Development Systems, XTOOLS Assembler/Disassembler
http://www.dunfield.com/xtools.htm

6. Baldwin, A.R., ASxxxx Cross Assemblers
http://shop-pdp.kent.edu/ashtml/asxxxx.htm

7. Manolaros, G., Win85 - i8085 Emulator for Windows
http://pythagoras.physics.upatras.gr/~gmanol/

8. Concurrent Versions System (CVS)
http://www.cvshome.org/

9. What are the frequencies of musical notes like G and G# in k-hertz?
http://www.physlink.com/Education/AskExperts/ae165.cfm

10. Bellaïche, M., Marchand, M., Talbot, M., Programmation de système, version 2, École Polytechnique de Montréal, août 1994, p.160.

11. Borland, Turbo Assembler 3.0 - Quick Reference Guide, version 3.0, 1991, p.85.

12. Corinthios, M., Arithmétique Informatique, Processeurs Parallèles, École Polytechnique de Montréal, janvier 1990, pp. 7-1 à 7-9.

13. *Idem*. pp. 10.1 à 10.4.

14. *Idem*. pp. 4-7 à 4-9.

15. Basic Variable Format - Knowledge Base Article #183
http://www.floodgap.com/retrobits/ckb/display.cgi?183

16. Laganière, R., Lafranchise, L., Boudreault, Y, Bellaïche, M., Hébert, P., Algorithmes et structures de données, édition corrigée, École Polytechnique de Montréal, septembre 1993, pp.163 à 178

Annexe 1 - Bases du BASIC

Voici une description du langage BASIC, tel qu'implanté dans ce projet.

A1.1 Modes de fonctionnement

A1.1.1 Mode immédiat

Lorsqu'un programme n'est pas entrain de s'exécuter, l'interpréteur est en *mode immédiat*. Dans ce mode, vous pouvez entrer la plupart des commandes BASIC et voir le résultat immédiatement. Vous pouvez aussi ajouter, retirer ou modifier des lignes du programme présentement en mémoire.

Pour ajouter une ligne, entrez son numéro suivi par le contenu de la ligne. Une ligne doit avoir un numéro supérieur à zéro, et inférieur à 32767. Lorsque vous appuyez sur <Enter>, la ligne est insérée dans le programme. Si le programme contenait une ligne correspondant au numéro de la nouvelle ligne, l'ancienne ligne est remplacée.

Ex: 10 PRINT "Bonjour!" <Enter> (Crée ou remplace la ligne 10)

Pour retirer une ligne, entrez son numéro suivi de <Enter>. Si la ligne choisie existe, elle sera retirée du programme.

Ex: 10 <Enter> (Efface la ligne 10)

Pour exécuter un programme, tapez 'RUN' suivi de <Enter>

Ex: RUN <Enter> (Exécute le programme)

A1.1.2 Mode programme

Le mode programme est activé lorsque l'utilisateur exécute un programme (RUN). L'interpréteur procède de la manière suivante:

1. Trouver la première ligne
2. Interpréter et exécuter le contenu de cette ligne
3. Avancer à la ligne suivante
4. Répéter 2 et 3 tant qu'il reste des lignes à exécuter
5. Retourner au mode immédiat

Un programme peut se terminer pour plusieurs raisons:

- L'interpréteur arrive à la dernière ligne du programme
- L'interpréteur exécute l'instruction END
- Une erreur se produit (Voir la liste des messages d'erreur à l'annexe C)

Dans tous ces cas, BASIC retourne en mode immédiat.

Une ligne est composée d'une ou plusieurs instructions. Pour inclure plus d'une instruction sur une ligne, on utilise le symbole ':' comme séparateur. Par exemple:

```
20      A = 100: PRINT A
```

Lorsque exécutée, cette ligne imprime '100'. L'assignation a lieu, suivie par la commande PRINT, qui imprime à l'écran la valeur de la variable A. Une ligne peut contenir plusieurs instructions. La seule limite est sa longueur maximale: 255 caractères.

A1.2 Variables

Cette version du langage BASIC supporte deux types de variables: entier et chaîne de caractères.

A1.2.1 Entiers

Un entier est un nombre signé dont la valeur se situe entre -32767 et +32767. Pour assigner une variable, vous pouvez utiliser le signe d'égalité (=) ou le mot-clé LET.

Ex:	A = 1234	(Assigne 1234 à la variable A)
	LET VA = 10 * A	(Assigne 1234*10=12340 à VA)

Les règles de nomenclature pour les variables de type entier sont les suivantes:

- Le premier caractère du nom doit être une lettre (A à Z). Les majuscules et minuscules ne sont pas différenciées ('A' équivaut à 'a').
- Les caractères suivants sont optionnels. Si présents, il doivent être des lettres ou des chiffres.
- Un nom de variable ne doit pas contenir un mot clé (voir la liste des mots-clés à l'annexe B).
- Seuls les deux premiers caractères identifient le nom de la variable. Donc VAR1 et VAR2 sont équivalents et correspondent à la variable VA

Exemples de noms de variables valides:

A, AB, F1, F123 (=F1)

Exemples de noms de variables incorrects:

- 1A (Premier caractère doit être une lettre),
- VALUE (contient le mot clé VAL)
- TO (TO est un mot clé)
- A$ (chaîne de caractères, voir section suivante)

A1.2.2 Chaînes de caractères

Une chaîne de caractères peut contenir de 0 à 255 caractères. Le nom d'une variable de type chaîne de caractères doit se terminer par le symbole '$'. Toutes les règles de nomenclature des variables de type entier s'appliquent aussi aux chaînes de caractères. Pour assigner une variable, vous pouvez utiliser le signe d'égalité (=) ou le mot-clé LET. Une chaîne constante doit être entourée de guillemets:

```
Ex:   A$ = "ABCD"                    (Assigne ABCD à la variable A$)
      LET VA$ = "123"+A$             (Assigne 123ABCD à VA$)
```

Exemples de noms de variables valides:

A$, AB$, F1$, F123$ (=F1$)

Exemples de noms de variables invalides:

A (entier), VALUE$ (contient le mot clé VAL)

A1.3 Expressions

Une expression est un 'calcul' effectué par l'interpréteur. Le résultat de l'opération peut être de type entier ou une chaîne de caractères. Une expression peut être assignée à une variable du même type, imprimée à l'écran (PRINT), utilisée dans une expression conditionnelle (IF), etc. Étudions d'abord les deux types d'expressions possibles.

A1.3.1 Expressions de type entier

Résultat d'une fonction:	ABS(-10), SQR(1000), SGN(A), LEN("ABC")
Opération arithmétique:	10 * 10, 100-A, A/B, -A
Opération logique:	A < 2, C = D, B>=C
Opération binaire/logique:	A AND 2, 10 XOR 6, NOT 1,
	(A<2) OR (A>10), (B=3) AND (C<10)
Combinaison d'expressions:	2 * (A+B), SQR(10*Z), (A+B)*(A-B)

Les opérations logiques retournent -1 pour VRAI, et 0 pour FAUX:

Ex:	2 < 10	(VRAI: -1)
	2 = 3	(FAUX: 0)
	(2<10) AND (2=3)	(FAUX: 0)
	(2<10) OR (2=3)	(VRAI: -1)

A1.3.2 Expression de type chaîne de caractères

Concaténation:	A$ + B$ + C$
Fonction retournant une chaîne:	LEFT$("1234567890", 5), STR$(1234)
Combinaisons:	A$ + RIGHT$(B$, I) + CHR$(13)

A1.4 Opérateurs

A1.4.1 Opérateurs arithmétiques

- Addition (entier + entier): 3 + 10, A+B, A+B+C+D

- Addition (chaîne + chaîne): " A B C " + " D E F " , A $ + B $, A$+B$+C$+D$
- Négation (-entier) -10, -A, -(A+B)
- Soustraction (entier - entier): 10 - 20, A - B
- Multiplication (entier * entier): 10*10, B*C, (A+B) * (C+D)
- Division(entier / entier): 10/2, A/B, (2*A)/B

A1.4.2 Opérateurs logiques

Toutes ces opérations retournent un entier, soit -1 pour VRAI, 0 pour FAUX

- Égalité (entier = entier): 2=3, A=B, (A+B) = 2
- Égalité (chaîne = chaîne): "ABC" = "DEF", A$ = "X"
- Inégalité (entier <> entier): 2<>3, A<>B
- Inégalité (chaîne <> chaîne): A<>"OUI"

Autres opérateurs (s'appliquent autant aux entiers qu'aux chaînes):

- Plus petit que (<)
- Plus grand que (>)
- Plus petit ou égal à (<=)
- Plus grand ou égal à (>=)

La comparaison des chaînes s'effectue caractère par caractère, d'après la valeur numérique correspondant à chacun. Le code ASCII est utilisé. Par exemple, "A" (=65) est considéré inférieur à "a" (=97). Dans les cas similaires au suivant:

- "ABC" (op de comparaison) "ABCDEFG"

lorsque les chaînes sont identiques jusqu'à la fin de l'une ou de l'autre, la chaîne la plus courte est considérée comme étant inférieure à l'autre. Dans ce cas-ci, "ABC" < "ABCDEFG".

A1.4.3 Opérateurs binaires/logiques

Les opérateurs AND, OR, XOR et NOT peuvent remplir deux fonctions. Notons que ceux-ci ne travaillent que sur des entiers. Premièrement, on peut s'en servir comme opérateurs binaires:

2 AND 3 = 2	(00000010 AND 00000011 = 00000010)
2 OR 4 = 6	(00000010 OR 00000100 = 00000110)
4 XOR 6 = 2	(00000100 XOR 00000110 = 00000010)
NOT 3 = -4	(NOT 00000011 = 11111100)

Et puisque VRAI et FAUX ne sont que deux valeurs numériques (-1=11111111 et 0 = 00000000), on peut se servir de ces opérateurs comme opérateurs logiques:

(2=3) AND (3=3)	= FAUX AND VRAI = FAUX
(2=3) OR (3=3)	= FAUX OR VRAI = VRAI
(2=2) XOR (3=3)	= VRAI XOR VRAI = FAUX
NOT (2=2)	= NOT VRAI = FAUX

Donc, si on a une variable VA = 33, l'expression (VA>10) AND (VA<40) retourne -1 (VRAI).

A1.5 Fonctions

On peut diviser les fonctions en 2 catégories: celles qui retournent un entier, et celles qui retournent une chaîne de caractères.

A1.5.1 Fonctions qui retournent un entier

ABS(entier): Retourne la valeur absolue de l'entier passé en paramètre.

Ex: ABS(10) = 10, ABS(-10) = 10

ASC(chaîne): Retourne la valeur numérique correspondant au premier caractère ce la chaîne passée en paramètre.

Ex: ASC("A") = 65, ASC("1234567890") = 49

FRE(entier): Retourne le nombre d'octets de mémoire libre pour le programme et les variables. Le paramètre entier est ignoré mais obligatoire.

Ex: FRE(0) = 27216

LEN(chaîne): Retourne la longueur de la chaîne de caractères passée en paramètre.

Ex: LEN("ABCDE") = 5, LEN("") = 0

PEEK(entier): Lit la mémoire à l'adresse passée en paramètre, retourne la valeur de l'octet à cet endroit (0..255)

Ex: PEEK(12345) = 255

RND(entier): Retourne un nombre pseudo-aléatoire entre -32767 et +32767. Le paramètre est ignoré mais obligatoire.

Ex: RND(0) = 16354, RND(0) = -27362, RND(0)=1002

SGN(entier): Donne le signe du paramètre: -1 si <0, +1 si >0, sinon retourne 0.

Ex: SGN(-1000) = -1, SGN(0) = 0, SGN(12345) = 1

SQR(entier): Calcule la partie entière de la racine carrée du nombre passé en paramètre.

Ex: SQR(100) = 10, SQR(12345) = 111

VAL(chaîne): Convertit une chaîne de caractères en nombre, lorsque possible (sinon retourne 0).

Ex: VAL("-100") = -100, VAL("PASNOMBRE") = 0

A1.5.2 Fonctions qui retournent une chaîne de caractères

CHR$(nombre): Retourne une chaîne contenant un caractère, correspondant (en ASCII) au paramètre.

Ex: CHR$(65) = "A", CHR$(32) = " " (espace)

LEFT$(chaîne, n): Où 'n' est un entier. Retourne une chaîne contenant les 'n' premiers caractères de la chaîne passée en paramètre. 'n' doit être positif et non-nul. Si 'n' dépasse la longueur de la chaîne, toute la chaîne est retournée

Ex: LEFT$("ABCDE", 3) = "ABC"
LEFT$("", 10) = ""
LEFT$("ABCDE", 10) = "ABCDE"

MID$(chaîne, p, n): Où ‘p’ et ‘n’ sont des entiers. Retourne une sous chaîne de ‘n’ caractères, Partant du ‘p’ième caractère de la chaîne. ‘p’ doit être positif et non-nul. ‘n’ doit être positif.

Ex: MID$(“ABCDE”, 2, 2) = “BC”
MID$(“ABCDE”, 4, 1) = “D”
MID$(“ABCDE”, 2, 99) = “BCDE”
MID$(“ABCDE”, 99, 2) = “”

RIGHT$(chaîne, n): Où ‘n’ est un entier. Retourne une chaîne contenant les‘n’ derniers caractères de la chaîne passée en paramètre. ‘n’ doit être positif et non-nul. Si ‘n’ dépasse la longueur de la chaîne, toute la chaîne est retournée.

Ex: RIGHT$(“ABCDE”, 3) = “CDE”
RIGHT$(“”, 10) = “”
RIGHT$(“ABCDE”, 10) = “ABCDE”

A1.6 Instructions

Voici une liste des instructions supportées, leur syntaxe, ainsi que leur fonction

A1.6.1 Entrées/sorties

BEEP

Syntaxe: BEEP

Description: Produit un bip sonore pendant une demi-seconde

CLS

Syntaxe: CLS

Description: Efface l'écran (avec la couleur de fond courante) et positionne le curseur à la position (1,1) (en haut à gauche)

COLOR

Syntaxe:
COLOR <expression entier>
COLOR <expression entier> , <expression entier>
COLOR , <expression entier>

Description: Change la couleur courante. Le premier paramètre indique la couleur des caractères, le deuxième indique la couleur du fond. Le premier ou le deuxième paramètre est optionnel, ce qui permet de ne changer qu'un seul des attributs et de laisser l'autre à sa valeur courante. Les valeurs acceptables pour les couleurs sont de 0 à 15 inclusivement.

Ex: COLOR 7: COLOR 3,4: COLOR ,5

GOTOXY

Syntaxe: GOTOXY <expression entier>, <expression entier>

Description: Positionne le curseur. Le premier paramètre correspond 'X', le deuxième au 'Y'. (1,1) est le coin supérieur gauche de l'écran. X doit se trouver entre 1 et 80 inclusivement. Y doit se trouver entre 1 et 25 inclusivement.

Ex: GOTOXY 1,1: GOTOXY 10,Y

INPUT

Syntaxe: INPUT {"message";|,} <variable entier|chaîne>

Description: Le programme pause et attend une réponse de l'usager. La réponse est copiée dans la variable passée en paramètre.

Ex: INPUT A (L'usager entre 12345<Enter>. On assigne 12345 à A)

INPUT A$ (L'usager entre TOTO<Enter>. On assigne TOTO à A$)

On peut spécifier un message à afficher avant la pause. Dans ce cas, on indique le message entre guillemets, suivi d'un point-virgule ou d'une virgule, suivi du nom de la variable. La virgule affiche un point d'interrogation à la fin du message. Le point-virgule n'ajoute rien.

Ex: INPUT "Entrez un nombre"; A
Imprime: Entrez un nombre, puis pause.
INPUT "Quel est votre nom", A$
Imprime: Quel est votre nom?, puis pause.

LIST

Syntaxe: LIST

Description: Imprime à l'écran le listing du programme

PRINT / ?

Syntaxe: PRINT <expression entier|ptr> {,|;{<expression entier|chaîne>....}
? <expression entier|ptr> {,|;{<expression entier|chaîne>....}

Description: Imprime à l'écran. On peut spécifier une ou plusieurs expressions de type entier ou chaîne de caractères. Les expressions sont séparées par une virgule (équivaut à un 'TAB': alignement à tous les 8 espaces) ou un point-virgule (pas d'espace). '?' est un raccourci pour PRINT.

Ex:	PRINT 2*2	Imprime:	4	
	PRINT 2*2, 4+4	Imprime:	4 8	
	? 4+4;-1234	Imprime:	8-1234	
	?"AB";"CD",123	Imprime:	ABCD	123

A1.6.2 Branchement

GOTO

Syntaxe: GOTO <expression entier>

Description: Interrompt le flot d'instruction et continue à la ligne spécifiée en paramètre. La ligne doit exister, sinon le programme se termine avec un message d'erreur.

Ex: GOTO 100 (Continue à la ligne 100)
GOTO A (A doit avoir la valeur d'une ligne existante)

GOSUB/RETURN

Syntaxe: GOSUB <expression entier>

Description: Similaire à un GOTO, sauf que GOSUB sauve l'endroit où le programme se situe, pour pouvoir y revenir lorsque l'interpréteur trouve une instruction RETURN.

Ex:
```
10 ?"Avant gosub" : GOSUB 1000: ?"Après gosub"
20 END
1000 ?"Pendant gosub"
1010 RETURN
```

À l'exécution, on obtient:
```
Avant gosub
Pendant gosub
Après gosub
```

A1.6.3 Conditions

IF...THEN
IF...THEN...ELSE

Syntaxe: IF <expression entier> THEN <Instruction>{:Instruction{:...}}
{ELSE <Instruction>{:Instruction{:...}}}

Description: Évalue l'expression passée en paramètre. Si l'expression est VRAIE (différente de zéro), les instructions après le THEN sont exécutées jusqu'à la fin de la ligne ou jusqu'au ELSE. Si l'expression est fausse, les instructions après le ELSE (si présentes) sont exécutées. Les IF...THEN...ELSE ne peuvent pas être imbriqués.

Ex: IF 3=3 THEN ?"VRAI" ELSE ?"FAUX"
Imprime VRAI
IF 2=3 THEN ?"VRAI" ELSE ?"FAUX"
Imprime FAUX
IF A<0 THEN ?"ABCD":GOTO 1000 ELSE GOTO 10
Si A est négatif, imprime ABCD et branche à la ligne 1000. Sinon, branche à la ligne 10

A1.6.4 Répétition

FOR...NEXT

Syntaxe: FOR <variable entier> = <expression entier> TO <expression entier>
{STEP <expression entier>}
...
NEXT

Description: Commence par assigner la première expression à la variable. Ensuite, execute le code jusqu'à l'instruction NEXT. La variable est incrémentée de 1 (ou de STEP, si spécifié). La variable est ensuite comparée à la deuxième

expression (après le TO) pour voir si la boucle est terminée. Sinon, on recommence. Les boucles FOR peuvent être imbriquées.

Ex: FOR I = 1 TO 5 : ?I, : NEXT
Imprime: 1 2 3 4 5
FOR I=0 TO 8 STEP 2: ?I: NEXT
Imprime: 0 2 4 6 8

```
10 FOR X=1 TO 10
20 FOR Y = 1 TO 10
30 PRINT X; " *  " Y; " = "; X * Y
40 NEXT : NEXT
```

(Imprime les tables de multiplications de 1*1 jusqu'à 10*10)

A1.6.5 Divers

CLR

Syntaxe: CLR

Description: Remet toutes les variables à zéro

Ex: A = 666: ?A: CLR: ?A Imprime: 666
0

END

Syntaxe: END

Description: Termine le programme courant

LET / =

Syntaxe: LET <variable entier> = <expression entier>
LET <variable chaîne> = <expression chaîne>
<variable entier> = <expression entier>
<variable chaîne> = <expression chaîne>

Description: Assigne une valeur à une variable. Le LET est optionnel. Les types ne doivent pas être mélangés.

NEW

Syntaxe: NEW

Description: Efface le programme présentement en mémoire, remet les variables à zéro.

POKE

Syntaxe: POKE <expression entier>, <expression entier>

Description: Écrit un octet (2e paramètre) en mémoire, à l'adresse donnée par le premier paramètre. Le deuxième paramètre doit avoir une valeur entre 0 et 255 inclusivement.

Ex: POKE 1024, 255 Écrit 255 à l'adresse 1024

REM

Syntaxe: REM {Commentaire}

Description: Le contenu d'une ligne après le mot REM est ignoré. Sert à inclure des commentaires dans le code.

Ex: 10 ?N$: REM Imprime le nom de l'usager

RUN

Syntaxe: RUN {<expression entier>}

Description: Exécute le programme présentement en mémoire. Si le paramètre est indiqué, démarre le programme à la ligne correspondant à l'expression.

Ex: RUN Démarre le programme
RUN 200 Démarre à la ligne 200

SLEEP

Syntaxe: SLEEP <expression entier>

Description: Pause l'exécution du programme pour un temps proportionnel à la valeur passée en paramètre. Chaque unité est 1/10 de seconde. Le délai doit être supérieur à zéro.

Ex: SLEEP 10 Pause 1 seconde

Annexe B - Liste des mots-clés

La liste suivante contient les mots réservés par l'interpréteur BASIC. Ces mots ne peuvent êtres utilisés comme noms de variables (ou faire partie d'un nom de variable, comme **TO**TAL, qui contient le mot réservé **TO**). La liste est présentée en ordre alphabétique.

ABS
AND
ASC
BEEP
CHR$
CLR
CLS
COLOR
ELSE
END
FOR
FRE
GOSUB
GOTO
GOTOXY
IF
INPUT
LEFT$
LEN
LET
LIST
MID$
NEW
NEXT
NOT
OR
PEEK
POKE
PRINT
REM
RETURN
RIGHT$
RND
RUN
SGN
SLEEP
SQR
STEP
STR$
THEN
TO
VAL
XOR

Annexe C - Messages d'erreur

Cette section contient la liste des messages d'erreur, ainsi qu'une courte explication de chaque message.

Division by zero error
Description: Se produit lorsqu'on tente de diviser un nombre par zéro.
Exemple: PRINT 2/0

ELSE without IF error
Description: L'instruction ELSE doit se trouver sur la même ligne que le IF..THEN.
Exemple: 10 IF A=0 THEN GOSUB 1000
20 ELSE GOSUB 2000

FOR without NEXT error
Description: Se produit si un FOR n'a pas de NEXT correspondant. En mode immédiat, le NEXT doit être sur la même ligne que le FOR.
Exemple: FOR I=1 TO 10: PRINT I (Mode immédiat)

10 FOR I=1 TO 10: FOR J=1 TO 10
20 PRINT I,J
30 NEXT (Le 'FOR I' n'a pas de NEXT)

Illegal argument error
Description: Un paramètre de type entier est hors bornes.
Exemple: GOTOXY 0,1 (X et Y doivent être > 0)
COLOR 32 (0..15)

Invalid symbol error
Description: Un symbole inconnu est présent dans la ligne.
Exemple: A$ = 3% (% n'est pas un symbole connu)

Missing parameter error
Description: Il manque un ou plusieurs paramètres à une fonction.
Exemple: PRINT SQR() (SQR requiert un paramètre de type entier)
PRINT LEFT$(A$) (Il manque un paramètre de type entier)

NEXT without FOR error

Description: Une instruction NEXT ne correspond à aucun FOR.

Exemple: FOR I=1 TO 10: PRINT I: NEXT: NEXT

Out of memory error

Description:
1. Le programme est trop gros ou trop de chaînes sont allouées.
2. La pile d'expression a débordé (expression trop complexe, trop de FOR imbriqués, GOSUBs récursifs, etc...).

Exemple: (Cas 2) 1000: PRINT "Infini...": GOSUB 1000

Overflow error

Description: Le résultat (numérique) d'une opération est supérieur à 32767 ou inférieur à -32767.

Exemple:
```
PRINT 1000*1000
PRINT 32767+1
```

RETURN without GOSUB error

Description: Une instruction RETURN a été trouvée, mais il n'y a pas d'adresse de retour sur la pile. Peut arriver facilement si on oublie un END à la fin d'un programme avec des sous-routines.

Exemple:
```
10   PRINT "Debut"
20   GOSUB 1000
30   PRINT "Fin"
                          (Il manque un END ici)
1000 PRINT "Sous-routine"
1010 RETURN
```

String too long error

Description: Le résultat de l'opération donnerait une chaîne de caractères de plus de 255 caractères.

Exemple: FOR I=1 TO 100: A$ = A$ + "ABCDEF": NEXT

Syntax error

Description: Message d'erreur très général dénotant une erreur de syntaxe.

Exemple:
```
IF A=1 PRINT "A=1"          (THEN manquant)
A$ = "ABCD" - "1234"        (Opération "-" illégale sur les chaînes)
PRINT ABS(-1                (Parenthèse manquante)
```

Type mismatch error

Description:	Le type d'un paramètre est erroné.	
Exemple:	PRINT LEN(1234)	(LEN requiert une chaîne de caractères)
	A = STR$(A$)	(STR$ requiert un nombre)
	A$ = B$ + C	(entier+entier ou chaîne+chaîne)

Undefined line number error

Description:	Tentative de GOTO, GOSUB ou RUN à une ligne inexistante.	
Exemple:	NEW: GOTO 1000	(La ligne 1000 n'existe pas)

Unterminated string constant error

Description:	Une chaîne de caractères doit commencer et se terminer par des guillemets. (Si on veut inclure des guillemets à l'intérieur d'une chaîne, utiliser CHR$(34))
Exemple:	10 PRINT "Chaîne non terminée...