Description et Organisation du Stamp2

Table des Matières

Description matérielle des STAMP2 et STAMP2SX

Le STAMP2 est un circuit hybride, présenté sous la forme d'un circuit DIL24, et les éléments suivants (Voir Schéma N°1):

1 - Le circuit interpréteur PBASIC (U1)

C'est le cerveau du STAMP2, il est composé d'un microcontroleur PIC16C57 pour le STAMP2 ou d'un SX28AC pour le STAMP2SX.
Lorsque vous programmez le STAMP2, c'est ce circuit qui interprète les instructions et qui les stocke dans l'EEPROM (IC2). Et lors de l'exécution du programme, c'est lui qui vas rechercher les instructions stockées dans l'EEPROM et qui vas les exécuter en séquence.

Le PIC16C57 peut exécuter 5 millions d'opérations par seconde, alors que sur une seconde le SX28AC en exécute un peu plus de 12 millions. Mais comme les instructions exécutées sont en fait écrite en BASIC et non en language machine, l'interpréteur doit les reconnaître avant de les exécuter, et c'est pourquoi le STAMP2 exécute environ 4000 opérations BASIC par seconde, et le STAMP2SX en exécute 10000 sur le même temps.

Ce circuit possède 20 lignes d'entrée sortie, 16 sont utilisée comme entrées-sorties du STAMP2, (RB0 à RB7 et RC0 à RC7), 2 sont utilisées pour interfacer la mémoire EEPROM, (RA0 et RA1), et les 2 dernières, (RA2 et RA3) sont utilisées pour réaliser l'interface de communication RS232 avec le PC ou avec un périphérique RS232.
Chaque ligne d'entrée-sortie peut avoir 4 états.
L'état de sortie haut (1), la ligne est alors à +5V, et peut fournir un courant maximal de 20 mA.
L'état de sortie bas (0), la ligne est alors à 0V, et peut absorber un courant maximal de 25 mA.
Le troisième état est l'état haute impédance. Dans cet état, la ligne présente une énorme impédance et absorbe ou fournit un courant de 1 micro-ampère.
Enfin, le quatrième état est l'état d'entrée. Dans cet état, la ligne est une entrée qui influence très peu le circuit sur laquelle elle est connectée. Le courant consommé est içi de 1 micro-ampère, comme pour l'état de haute impédance.
Notez encore une autre limitation. Pour chaque groupe de 8 lignes (P0 à P7 et P8 à P15), le total du courant fournit ne peut dépasser 40 mA et le total du courant absorbé ne peut dépasser 50 mA.

Enfin, il faut noter que ce circuit possède aussi une mémoire RAM. Cette mémoire RAM a une taille de 72 octets dont 32 octets sont réservés pour l'utilisation des variables PBASIC, le reste étant utilisé par l'interpréteur du STAMP2. Cette mémoire est en fait la mémoire de travail du STAMP. C'est celle que vous utilisez pour stocker vos variables lors de l'écriture de vos programmes en PBASIC. Notez que cette mémoire est dite volatile. Elle ne conserve pas son état lors d'une coupure d'alimentation, et dans ce cas elle est donc éffacée.

Dans le Stamp2sx, cette mémoire est plus importante, et en plus des 32 octets qui vous sont réservés, il y en a une "Scratch Pad Ram" de 64 octets, dont les 63 premiers peuvent-être utilisés par les instructions spécifiques GET et PUT pour pouvoir "passer" des arguments d'un programme à l'autre. La dernière case mémoire contient le numéro du programme en cours d'exécution.

Description matérielle des STAMP2 et STAMP2SX

2 - La mémoire EEprom (U2).

Contrairement à la RAM contenue dans le circuit interpréteur, l'EEPROM est une mémoire permanente. C'est à dire que son contenu n'est pas éffacé lors d'une rupture d'alimentation. Sa taille est de 2048 octets pour le STAMP2 et de 16384 octets pour le STAMP2SX.
Cette mémoire contient les intructions de votre programme et éventuellement des données que vous pouvez y stocker de manière permanente. Dans le Stamp2, il y a de la place pour un programme, alors que dans le Stamp2SX, vous pouvez y stocker jusqu'a huit programmes, mais chaque programme est indépendant, et la seule manière de faire passer des arguments d'un programme à l'autre est d'utiliser la "Scratch Pad Ram" définie plus haut.

L'EEprom a 2 limitations. La première est que comme elle est située à l'extérieur du circuit interpréteur, son temps d'accès en lecture et en écriture est donc assez lent. Il est alors préférable d'utiliser la RAM comme mémoire de travail, votre programme n'en sera que plus rapide. La deuxième est qu'une EEPROM ne peut supporter qu'environ 10 millions de cycles de lecture-écriture. S'il parait impenssable de programmer 10 millions de fois de STAMP2, par contre, chaque écriture dans l'EEPROM par votre programme consommera un cycle de lecture-écriture. Ayez quand-même cette limitation en tête lorsque vous programmerez votre STAMP2.

3 - Le circuit de RESET (U3).

Comme le circuit interpréteur ne fonctionne bien qu'avec une tension d'alimentation correcte, le circuit U3 surveille en permanance la ligne d'alimentation et si celle-çi descend en dessous de 4 Volts, le circuit U3 éffectuera un RESET sur l'interpréteur PBASIC, et ceci durera tant que la tension d'alimentation ne sera pas redevenue correcte. Ce système garantit un fonctionnement irréprochable du STAMP2 ou du STAMP2SX.

4 - Une alimentation stabilisée 5V (U4).

Pour permettre l'alimentation du STAMP2 ou du STAMP2SX avec une large gamme de tension d'alimentation, le circuit est équipé d'un régulateur 5 Volts à faible chute de tension. Ce régulateur admet des tensions d'entrée (Pin 24 du circuit STAMP2 ou STAMP2SX) comprise entre 5,5 Volts et 15 Volts. Il peut fournir une tension d'alimentation parfaitement régulée de 5 Volts avec un courant maximal de 50 mA. Comme le circuit interpréteur consomme environ 8 mA, la sortie du régulateur (Pin 21 du circuit STAMP2 ou STAMP2SX) peut aussi alimenter les circuits extérieurs jusqu'à un maximum de 42 mA. Si votre circuit extérieur consomme plus, il vous faudra réaliser une alimentation externe qui alimentera votre circuit et éventuellement le STAMP2 ou le STAMP2SX par la pin 21.

Description matérielle des STAMP2 et STAMP2SX

5 - L'interface RS232 (Q1, Q2 et Q3)

Le STAMP2 et le STAMP2SX n'ayant ni clavier, ni écran, il faut les relier à un autre ordinateur pour pouvoir les programmer. C'est l'interface RS232 qui éffectue cette liaison. L'interface est constituée de trois connexions, SIN (Serial IN), SOUT (serial OUT) et ATN (attention).

La norme RS232 utilise des niveaux de +12V (pour le 0) et -12V (pour le 1). Comme le STAMP ne possède pas de tension de -12V, le circuit composé de Q1 et Q3 génère la tension de -12V à partir de la connexion avec le PC connecté grace à SIN et SOUT. Le circuit RS232 du STAMP génère donc du 5V et -12V. Cette petite différence par rapport à la norme RS232 est en général parfaitement accepté par les équipement distants raccordés sur cette interface.
La connexion ATN est utilisée par le PC pour éffectuer un reset du STAMP via le transistor Q2.

Organisation mémoire des STAMP2 et STAMP2SX

La mémoire du STAMP2 est composé de deux éléments différents. Une RAM de 32 octets, et une EEPROM dont la taille fait 2K ou 16k et qui contient le programme et dont l'espace restant peut-être utilisé pour stocker des valeurs.

La différence fondamentale entre la RAM et l'EEPROM est que le contenu de la RAM est éffacé lors d'une coupure d'alimentation alors que le contenu de l'EEPROM est concervé jusqu'a une autre écriture.

Mémoire RAM du STAMP2

La mémoire RAM est organisée en 16 mots de 16 bits. Les 3 premiers mots (les 6 p bytes) sont utilisés pour définir et utiliser les entrées/sorties (P0 à P15)

Les 13 mots (26 octets) restants sont utilisés par les variables et constantes de votre programme.

Espace variable du STAMP2 - Noms de variables réservés

Mémoire		Noms des variables réservées			
Mot	Octet	Mot	Byte	Nibble	Bit
0	0	INS	INL	INA	IN0 à IN3
				INB	IN4 à IN7
	1		INH	INC	IN8 à IN11
				IND	IN12 à IN15
1	2	OUTS	OUTL	OUTA	OUT0 à OUT3
				OUTB	OUT4 à OUT7
	3		OUTH	OUTC	OUT8 à OUT11
				OUTD	OUT12 à OUT15
2	4	DIRS	DIRL	DIRA	DIR0 à DIR3
				DIRB	DIR4 à DIR7
	5		DIRH	DIRC	DIR8 à DIR11
				DIRD	DRI12 à DIR15
3	6	W0	B0		
	7		B1		
4	8	W1	B2		
	9		B3		
5	10	W2	B4		
	11		B5		
6	12	W3	B6		
	13		B7		
7	14	W4	B8		
	15		B9		
8	16	W5	B10		
	17		B11		
9	18	W6	B12		
	19		B13		
10	20	W7	B14		
	21		B15		
11	22	W8	B16		
	23		B17		
12	24	W9	B18		
	25		B19		
13	26	W10	B20		
	27		B21		
14	28	W11	B22		
	29		B23		
15	30	W12	B24		
	31		B25		

Organisation mémoire des STAMP2 et STAMP2SX

Espace d'entrée/sortie

Comme indiqué dans le tableau çi-dessus, les variables INS, OUTS et DIRS sont utilisées pour contrôler les entrées/sorties du STAMP2. Ils s'utilisent en regard des entrées/sorties du STAMP2 (P0 à P15). Par exemple, à la pin P6 est attaché les variables bits IN6, OUT6 et DIR6, ainsi que les parties de variables mot, byte et nibble correspondant.

Les variables INS et dérivées s'utilisent uniquement en lecture et permettent de connaitre l'état des entrées du STAMP2. Si la pin correspondante est définie en sortie, le lecture éffectuée sera l'état de sortie. Notez que si la pin est définie en entrée et qu'elle est laissée en l'air, la lecture de cette entrée donnera alors une valeur aléatoire (0 ou 1).

Les variables OUTS et dérivées peuvent s'utiliser en lecture (pour lire l'état d'une sortie) ou en écriture (pour forcer une sortie à 0 ou à 1). Lorsque la pin correspondante est définie en entrée, la lecture est aléatoire (mais cela importe peu), et l'écriture de cette mémoire ne modifiera pas l'état de l'entrée correspondante. Par contre, la valeur écrite dans ce registre apparaitra sur la pin correspondante lorsque cette pin sera mise en sortie.

Les variables DIRS et dérivées s'utilisent en écriture ou en lecture.
Un 0 définit la pin correspondante en entrée et un 1 la définit en sortie.
A l'allumage et au reset du STAMP2, ce registre est mis à zéro, définissant ains pins en entrée.

Organisation mémoire des STAMP2 et STAMP2SX

Définition et utilisation des variables

L'éditeur du STAMP2 reconnait les variables décrites plus haut mais vous pouvez en utiliser d'autre pour faciliter la lecture de votre programme.

En tout cas, une regle générale dans l'utilisation des variables, est de ne pas mélanger les variables définies par défaut avec des variables définies par vous même. En effet, les variables que vous définissez, remplissent la mémoire RAM en commençant par le bas (de B0 à B25). D'ailleur, l'utilisation des touches Alt-M vous montrera l'occupation de vos variables. Par contre, si vous utilisez les variables définies par le STAMP2, celles-çi n'apparaîtrons pas lorsque vous utiliserez les touches Alt-M (en effet, pour le STAMP2, toutes les variables par défaut sont définies).

Un petit exemple nous permettra d'éclaircir tout ça. Imaginez que vous définissiez une variable de type Byte appellée "relais". Comme cette variable est définie en premier lieu dans votre programme, le STAMP2 lui allouera la place du premier octet en mémoire RAM. Or, cette place est aussi celle définie par défaut pour B0 ou pour l'octet de poids faible de W0. Donc, si dans le corps de votre programme, vous utilisez B0 ou W0, ces variables modifieront la valeur de votre variable "Relais" puisque en fait ce sont les mêmes variables aux mêmes emplacements mémoire.

Donc, dans notre cours, nous prendrons systématiquement l'habitude de définir réellement toutes les variables que nous utiliserons, à l'exeption des variables INS, OUTS et DIRS et de leurs dérivées. De plus, la lisibilité du programme s'en trouvera améliorée.

Il existe trois commandes qui sont utilisée pour modifier la mémoire de donnée.

VAR : Pour définir les variables en RAM
CON : Pour définir les constantes en RAM
DATA : Pour définir les données stockées dans l'EEPROM

Définition des variables : La commande VAR

La commande VAR s'utilise selon la syntaxe suivante :

```
<Nom>     VAR  <Type>
ou
<Nom2>    VAR  <Nom1>.<Modifiant>
```

Où <Nom> représente le nom de la variable. Ce nom peut-être composé de n'importe quel enchaînement de lettres, de chiffres et du caractère souligné (_). Toutefois, il faut noter quelques restrictions.

- Le nom de la variable ne peut commencer par un chiffre.
- Le nom de la variable ne peut-être un nom de commande reconnu par l'interpréteur du STAMP2. Comme par exemple SERIN, GOTO, etc...
- Le nom de la variable ne peut-être définit qu'une fois.

<Type> représente le type de variable qui peut-être un des type suivant :

bit : Définit une variable Bit
nib : Définit une variable Nibble (4 bits)
byte : Définit une variable Octet (8 bits)
word : Définit une variable Mot (16 bits)

Organisation mémoire des STAMP2 et STAMP2SX

Définition et utilisation des variables (suite)

Pour définir un tableau, il suffit d'ajouter un nombre entre parenthèse après le <Type>. La taille du tableau sera définit par le nombre entre parenthèse plus un Puisque le comptage commence à zéro.
Dans la deuxième syntaxe, la variable est définie comme une partie d'une autre variable. La partie définie, l'est en fonction du modifiant. Voiçi la liste des modifiants reconnus par l'interpréteur du STAMP2.

```
Lowbyte   : Octet de poids faible d'un mot
Highbyte  : Octet de poids fort d'un mot
Byte0     : Octet de poids faible d'un mot
Byte1     : Octet de poids fort d'un mot

Lownib    : Nibble de poids faible d'un octet ou d'un mot
Highnib   : Nibble de poids fort d'un octet ou d'un mot
Nib0      : Nibble 0 (poids faible) d'un octet ou d'un mot
Nib1      : Nibble 1 d'un mot ou nibble de poids fort d'un octet
Nib2      : Nibble 2 d'un mot
Nib3      : Nibble 3 (poids fort) d'un mot

Lowbit    : Bit de poids faible d'un nibble, d'un octet ou d'un mot
Highbit   : Bit de poids fort d'un nibble, d'un octet ou d'un mot
Bit0      : Bit 0 d'un mot, d'un octet ou d'un nibble
Bit1      : Bit 1 d'un mot, d'un octet ou d'un nibble
Bit2      : Bit 2 d'un mot, d'un octet ou d'un nibble
Bit3      : Bit 3 d'un mot, d'un octet ou d'un nibble
Bit4      : Bit 4 d'un mot ou d'un octet
Bit5      : Bit 5 d'un mot ou d'un octet
Bit6      : Bit 6 d'un mot ou d'un octet
Bit7      : Bit 7 d'un mot ou d'un octet
Bit8      : Bit 8 d'un mot
Bit9      : Bit 9 d'un mot
Bit10     : Bit 10 d'un mot
Bit11     : Bit 11 d'un mot
Bit12     : Bit 12 d'un mot
Bit13     : Bit 13 d'un mot
Bit14     : Bit 14 d'un mot
Bit15     : Bit 15 d'un mot
```

Voiçi quelques exemples :

```
souris    VAR  bit       ' Définit la variable "souris" comme étant un bit
chat      VAR  nib       ' Définit la variable "chat" comme étant un nibble
chien     VAR  byte      ' Définit la variable "chien" comme étant un octet
rhino     VAR  word      ' Définit la variable "rhino" comme étant un mot
ver       VAR  bit(10)   ' Définit la variable "ver" comme tableau de 11 bits
tête      VAR  rhino.highbyte ' Définit la variable "tête" comme octet de poids
                         ' fort de la variable "rhino", ce qui implique que
                         ' toute modification de la variable "tête" modifiera
                         ' aussi "rhino".
queue     VAR  rhino.lownib   ' Définit la variable "queue" comme nibble de
                         ' poids faible de "rhino". (Même remarque que pour
                         ' "tête").
langue    VAR  tête.highbit   ' Définit la variable "langue" comme bit de poids
                         ' fort de la variable "tête".
yeux      VAR  tête.bit5 ' Définit la variable "yeux" comme cinquième bit de la
                         ' variable "tête"
```

Organisation mémoire des STAMP2 et STAMP2SX

Définition et utilisation des variables (suite)

Lors de la compilation de votre programme, le STAMP2 regroupera les variables par type pour utiliser le moins de mémoire RAM possible. La commande Alt-M vous permet de voir les résultats de vos déclarations de variables sur la RAM.

Une dernière chose concernant les déclarations de variables : Pour des raison de facilité de compréhension de votre programme, vous pouvez définir des variables qui pointent vers des registres d'entrées/sorties. Voiçi quelques exemples illustrant cela :

```
leds           VAR  OUTL ' Définit la variable "leds" comme étant l'octet de
                         ' poids faible du registre OUT
led3           VAR  OUT3 ' Définit la variable "led3" comme étant le 3ème bit du
                         ' registre OUT
touches        VAR  IND  ' Définit la variable "touches" comme étant le nibble
                         ' de poids fort du registre IN
intérupteur    VAR  IN10 ' Définit la variable "intérupteur" comme étant le
                         ' 10ème bit du registre IN
```

Pour en terminer avec la commande VAR, je ne vous donnerai qu'un conseil : Utilisez des noms de variables qui soient clairs et facile à comprendre et banissez toute abréviation qui pourraient prêter à confusion. L'usage de nom de variables bien choisi facilitera la lecture, la mise au point et la recherche d'erreurs.

Organisation mémoire des STAMP2 et STAMP2SX

Définition des constantes : La commande CON

Comme son nom le laisse entendre, la commande CON permet de définir des constantes. Contrairement aux variables, les constantes sont définies une fois pour toute et ne changerons plus de valeur pendant l'exécution du programme.

En général, ces constantes sont utilisées pour définir des limites dans votre programme, comme par exemple des temporisations, des niveaux de basculement lors de lectures analogiques, des butées lors de divers mouvements, etc... Elles seront en général définies au début de votre programme et permettrons une adaptation de valeurs facile dans votre programme.

La syntaxe générale de la commande CON est la suivante :

```
<Nom>     CON  <expression>
```

Où <Nom> représente le nom de la constante, et répond aux mêmes règles que pour le nom de variables, et <expression> représente une valeur ou une évaluation d'une expression numérique.

Voiçi quelques exemples de l'utilisation de la commande CON :

```
niveau         CON  10        ' La valeur de "niveau" sera fixée à 10
limite         CON  10*4<<2   ' "limite" sera définie à 160
débordement    CON  limite+1  ' et "débordement" sera définit à 161
```

Les opérateurs que vous pouvez utiliser lors de l'évaluation d'expression numérique dans la définition d'une constante sont les suivants :

```
+    Addition
-    Soustraction
*    Multiplication
/    Division
<<   Décalage à gauche
>>   Décalage à droite
&    "ET" Logique
|    "OU" Logique
^    "OU-EXCLUSIF" Logique
```

Notez que lors de l'évaluation d'expression numériques, les expressions sont évaluées strictement de la gauche vers la droite. L'usage des parenthèse pour changer l'ordre de calcul n'est pas permis dans l'utisation de la commande CON et seule l'utilisation des neuf opérateurs définis plus haut est autorisé.

Organisation mémoire des STAMP2 et STAMP2SX

Définition des données stockées dans l'EEPROM : La commande DATA

Lors du chargement de votre programme dans la mémoire EEPROM du STAMP2, l'espace mémoire utilisé commence par la fin pour remplir la mémoire vers le début. La partie du début de la mémoire EEPROM qui reste après chargement de votre programme est libre et vous permet de stocker des valeurs. Vous utiliserez la mémoire EEPROM dans deux cas de figure.

1 - Soit pour y stocker des valeurs qui doivent-être concervée en memoire lors de la coupure de l'alimentation.

2 - Soit parceque l'espace mémoire RAM est saturé, et que vous devez utiliser encore plus de mémoire.

N'oubliez pas que la somme de la mémoire occupée par les données et le programme dans l'EEPROM ne doit pas dépasser 2K octets. Lorsque vous dépasserez cette limite, l'interpréteur du STAMP2 vous enverra un message d'erreur. D'autre part, les valeurs définies dans cette instruction sont chargée dans l'EEPROM du STAMP2 lors du chargement du programme. Elle n'ocupe donc aucune place dans la mémoire programme.

Les syntaxes admises pour l'instruction DATA sont les suivantes :

```
1 : <Etiquette>     DATA <constante1>,<constante2>,...
2 : <Etiquette>     DATA @<adresse>, <constante1>,<constante2>,...
3 : <Etiquette>     DATA (nombre)
4 : <Etiquette>     DATA <constante> (nombre)
5 : <Etiquette>     DATA WORD <constante>
```

Les constantes sont placées en EEPROM à l'adresse indiquée par un pointeur qui est initialisé par défaut à 0. Ce pointeur sera incrémenté d'une unité à chaque donnée mémorisée dans l'EEPROM. L'étiquette qui précède l'instruction se voit allouer la valeur initiale de ce pointeur lors de l'exécution de l'instruction DATA.

Dans la première syntaxe, les données contenues dans les <constante(n)> sont rangées dans l'EEPROM en partant de l'adresse contenue dans le pointeur de l'EEPROM. Si cette syntaxe est utilisée la première, le pointeur vaut 0 et donc <Etiquette> vaut aussi 0. La valeur de <constante1> est stockée à l'adresse 0, puis la valeur de <constante2> est stockée à l'adresse 1 et ainsi de suite pour les autres valeurs éventuelles.

Dans la deuxième syntaxe, <Etiquette> est initialisée à la valeur contenue dans <adresse>. Le pointeur sera mis à cette valeur. La valeur de <constante1> est stockée à l'adresse indiquée par <adresse>, puis la valeur de <constante2> est stockée à l'adresse pointée par <adresse>+1, et ainsi de suite pour les autres valeurs éventuelles.

Dans les deux premiers exemples, le pointeur d'adresse de l'EEPROM est incrémenté à chaque fois qu'une valeur est stockée dans l'EEPROM.

Dans la troisième syntaxe, le but est d'incrémenter le pointeur d'adresse de l'EEPROM d'une valeur égale à (nombre). Ceci aura pour effet de réserver un espace mémoire égal à un nombre d'octets egal à (nombre) dans l'EEPROM. Attention, la seule modification concerne le pointeur d'adresse, le contenu de l'EEPROM ne sera donc pas modifié. Si l'EEPROM est vierge, les valeurs contenues dans ces emplacement mémoire seront des 0, et si l'EEPROM à déjà été utilisée, se seront les valeurs précédentes qui s'y trouverons.

Organisation mémoire des STAMP2 et STAMP2SX

Définition des données stockées dans l'EEPROM : La commande DATA (suite)

La quatrième syntaxe est fort ressemblante à la précédente. La différence est que içi, l'espace mémoire est réservé d'un nombre d'octet égal à (nombre), le pointeur mémoire est incrémenté de la valeur de (nombre), mais les cases mémoires réservées sont initialisée avec la valeur de <constante>.

La dernière syntaxe permet de couper en deux la valeur de <constante> (qui doit être un mot de 16 bits), pour ranger l'octet de poids fort à l'adresse du pointeur de l'EEPROM et l'octet de poids faible à l'adresse du pointeur+1.

Voiçi quelques exemples :

```
liste1    DATA      12,45,38,29     ' liste1 vaut 0, la valeur 12 est stockée à
                                     ' l'adresse 0, la valeur 45 est stockée à
                                     ' l'adresse 1, 38 est mis en 2 et 29 en 3
liste2    DATA      25,85,54        ' Le pointeur d'adresse vaut donc 4 et ce
                                     ' sera la valeur de liste2. Les valeurs 25,85
                                     ' et 54 seront mises aux adresses 4,5 et 6.
liste3    DATA      @10,78          ' Le pointeur d'adresse est mis à 10 et la
                                     ' valeur 78 y sera rangée.
liste4    DATA      16(22)          ' Liste4 vaut 11 (10+1). Les 22 octets
                                     ' suivants seront initialisés a la valeur 16.
liste5    DATA      WORD 1000       ' liste 5 vaut maintenant 33 (11+22). La
                                     ' valeur 1000 est décomposées en 2 valeurs
                                     ' "octets" soit 3 et 232 (3*256+232=1000).
                                     ' La valeur 3 sera stockée à l'adresse 33 et
                                     ' la valeur 232 sera stockée à l'adresse 34.
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Le STAMP2, comme tous les ordinateurs, excelle en math et en logique. Cependant, ayant été conçu pour le contrôle de processus, le STAMP2 réalise les calculs d'une autre façon qu'une calculette ou qu'un tableur. Ce chapitre vous aidera à comprendre comment le STAMP2 se comporte avec les nombres, la mathématique et la logique.

Représentations des nombres

Dans vos programmes, vous pouvez exprimer un nombre de différentes manières, suivant la façon dont il sera utilisé et ce qu'il représente pour vous. Par défaut, le STAMP2 reconnaît les nombres comme 0, 99 ou 62145 comme pour notre système décimal (base 10). Cependant, vous pouvez utiliser le système hexadécimal (base 16 ; aussi appelé hexa) ou le système binaire (base 2).

Etant donné que les symboles utilisés en décimal, hexa et binaire se chevauchent (ex. 1 & 0 sont utilisés par tous ; 0 à 9 sont utilisés par décimal et hexa) le programme du STAMP2 à besoin de préfixes pour différencier les systèmes numériques :

```
        99          Décimal (pas de préfixe)
      $1A6          Hexa
     %1101          Binaire
```

Le STAMP2 converti aussi automatiquement le texte en codes ASCII et vous permet d'appliquer des noms (symboliques) aux constantes à partir de ces systèmes numériques.

Exemples :

```
     lettreA        con  "A"       ' Code ASCII pour A (65).
     santé          con  3
     hex128         con  $80
     desBits        con  %1101
```

A quel moment les expressions numériques sont-elles évaluées ?

Toutes les opérations mathématiques ou logiques dans un programme PBASIC ne sont pas traîtées par le STAMP2.

Les opérations qui définissent des constantes sont évaluées par le programme d'hébergement du STAMP2 (l'éditeur chargé dans le PC) avant que le programme écrit en PBASIC ne soit chargé dans le STAMP2. Cette pré-procédure avant le chargement du programme PBASIC est appellée "Compilation".

Une fois le chargement completé, le STAMP2 commence à exécuter le programme PBASIC. C'est à partir de ce moment que l'on parle d'"exécution".
A l'exécution le STAMP2 évalue les opérations mathématiques et logiques utilisant les variables, ou toutes combinaisons de variables et de constantes.

Parce que les expressions qui sont évaluées pendant la compilation et celles qui sont évaluées pendant l'exécution semblent très similaires, il est difficile de les décrire séparement.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Représentations des nombres (suite)

Voici quelques exemples pour vous aider :

```
Santé          con  3         ' Evaluée à la compilation
Verres         con  Santé*2-1 ' Evaluée à la compilation
CentNonante    con  100+90    ' Evaluée à la compilation
PasTravailleur con  3*b2      ' ERREUR : pas de variables allouées puisque
                              ' cette expression est évaluée à la
                              ' compilation et qu'a ce moment les variables
                              ' prédéfinie n'exitent pas encore.
b1 = Verres                   ' identique à b1 = 5
b0 = 99 + b1                  ' Evaluées à l'exécution
w1 = CentNonante              ' 100 + 90 : Evaluée à l'exécution
w1 = 100 + 90                 ' 100 + 90 : Evaluée à l'exécution
```

Notez que les deux derniers exemples sont résolus à l'Exécution. C'est un peu dommage puisque finalement il s'agit de constantes, et dès lors ce calcul aurait put être résolu à la Compilation. Si vous trouvez quelques choses de similaires dans vos propres programmes, vous pouvez économiser de l'espace sur l'EEPROM en convertissant l'expression d'Exécution 100 + 90 en une expression de compilation comme CentNonante con 100 + 90.

Pour résumer :

- Les expressions de Compilation (Compile-Time) sont celles qui invoquent seulement les constantes ; lorsqu'une variable est impliquée, l'expression doit être évaluée à l'Exécution (Run-Time). C'est pour cela que la ligne "Pas Travailleur con 3*b2" générerait un message d'erreur. La directive CON fonctionne seulement à la Compilation (compile-time), ainsi les variables ne sont pas autorisées.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Ordres des Opérations

Parlons un peu des quatre opérations arithmétiques de base :
Addition (+), soustraction (-), multiplication (*) et division (/).

Vous pouvez vous rappeler que l'ordre dans lequel sont faites une série d'additions et de soustractions n'affecte pas le résultat.
L'expression 12+7-3+22 est identique à 22-3+12+7. Cependant, lorsqu'il s'agit de multiplications et de divisions, c'est une autre histoire ; 12+3*2/4 n'est pas identique à 2*12/4+3. En fait, il est urgent de mettre des parenthèses autour des portions de ces équations pour éclaircir les choses !

Le BS2 résout les problèmes mathématiques dans l'ordre tel qu'ils ont été écrit de la gauche vers la droite. Le résultat de chaque opération est utilisé dans l'opération suivante. Ainsi pour calculer 12+3*2/4, le BS2 va à travers la séquence suivante :

```
12 + 3 = 5
5 * 2 = 10
10 / 4 = 2
le résultat est 2
```

Du fait que le BS2 travaille sur des nombres entiers, le résultat de 10/4 donne 2, pas 2,5.

D'autres "dialectes" en BASIC calculeraient la même expression en se basant sur les priorités des opérateurs, qui demandent que les multiplications et les divisions s'exécutent avant les additions et les soustractions. Ainsi le résultat du même calcul serait :

```
3 * 2 = 6
6 / 4 = 1
12 + 1 = 13
le résultat est 13
```

Encore une fois, du fait des nombres entiers, la portion de la fraction de 6/4 est abondonnée, nous obtenons 1 à la place de 1,5.

Afin d'éviter toute erreur d'interprétation, nous devons utiliser les parenthèses pour éclaircir nos intentions mathématiques sur le BS2 (pas pour notre besoin ou celui d'autres programmeurs qui liraient notre programme).
En mettant une opération mathématique entre parenthèses, nous lui donnons la priorité sur d'autres opérations. Pour exemple, dans l'expression 1+(3*4), le 3*4 sera traité en premier et ensuite ajouter à 1.

Pour que le BS2 calcul l'expression précédente à la manière du BASIC conventionnel, nous devrions l'écrire 12 + (3*2/4). Sans les parenthèses, le BS2 exécute de la gauche vers la droite. Si vous vouliez être encore plus précis, vous écririez 12 + ((3*2)/4). Lorsqu'il y a des parenthèses à l'intérieur de parenthèses, le BS2 travaille a partir des parenthèses les plus intérieures vers les parenthèses extérieures. Des parenthèses placées à l'intérieur d'autres parenthèses sont dites "nichées" (nested) ou de niveau inférieur. Le BS2 permet l'usage des parenthèses sur huit niveaux de profondeurs.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Calcul des nombres entiers

Le BS2 produit toutes les opérations mathématiques suivant les règles des nombres entiers positifs. De ce fait, il manipule seulement des nombres entiers, et il abandonne toute fraction d'un résultat de calcul.

Cependant, le BS2 peut interpréter correctement les nombres négatifs complémentaires à 2 dans les instructions "DEBUG" et "SEROUT" en utilisant les modifiants comme "SDEC" (pour signature décimal), cela permet de n'avoir que des nombres positifs dans les calculs. Donc avec l'utilisation des nombres négatifs complémentaires à 2, les résultats seront corrects pour l'addition, la soustraction et la multiplication, mais pas pour la division.

Ce sujet est trop long pour être décrits ici.
Si vous avez compris le paragraphe précédent, parfait.
Si vous en avez déduit que manipuler des nombres négatifs est un exercice périlleux qui à défaut d'être évité nécessite un long travail préparatoire c'est pas mal non plus.
Et si vous n'avez rien compris, vous devriez lire des ouvrages orientés vers le calcul mathématique par ordinateur.

Les Opérations a une ou a deux opérandes.

Les opérations les plus courantes comme : +, -, * et /, vous sont familières. Ces opérations travaillent toutes sur deux valeurs, comme dans 1+3 ou 26*144. Les valeurs que ces opérations traitent sont appellées des opérandes. Dans ce cas nous nous trouvons en présence d'opérations à deux opérandes.

Le signe moins (-) peut être utilisé avec une seule opérande, comme dans -4. Evidement, les puristes nous dirons que -4 est l'écriture simplifiée de l'opération 0-4, et qu'il s'agit donc d'une opération à 2 opérandes.
Mais nous, loin de ces considérations philosophiques, nous dirons que (-) à deux rôles : soit il sera une opération de soustraction qui prend deux opérandes, ou soit il sera une opération "inversion de signe" avec une seule opérande.

En classant les opérations mathématiques et logiques du BS2, nous les avons divisés en deux types : à une ou a deux opérandes.

Rappelez-vous la discussion précédente concernant les priorités des opérateurs. A cela, il faut ajouter que les opérations à une opérande sont prioritaires sur les opérations à deux opérandes.
Par exemple, dans l'expression "10 - SQR 16", le BS2 évaluera d'abord la racine 16), qui est une opération à une opérande, puis retranchera le résulat à 10, dan soustraction qui est une opération à deux opérandes.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

L'Espace de travail en 16-bits

Dans les descriptions des opérations qui vont suivrent, nous parlerons régulièrement de valeurs "16 bits" en utilisant les variables 16 bits comme W0, W1 etc... Cela ne veut pas dire que l'opération ne travaille pas avec des valeurs plus petites comme un octet ou un nibble. Cela veut simplement dire que le traitement est exécuté dans un espace de travail à 16 bits. Si la valeur est inférieure à 16 bits, le BS2 ajoute des 0 (zéros) devant la valeur pour en faire une valeur à 16 bits. Par contre, si le résultat d'une opération donne une valeur plus grande qu'un nombre de 16 bits, alors, tous les bits de poids fort sont purement et simplement éliminés pour ramener la taille de la valeur à un nombre de 16 bits.

Ayez toujours en tête cette particularité lors de l'utilisation de nombres négat complémentaires à 2, ou lors de la transformation d'une grande valeur en une val petite.

Pour essayer de rendre cet aspect un peu plus clair, nous allons l'illustrer ave concret.

Soit le petit programme ecrit en BS2 :

```
b2 = -99
DEBUG sdec ?b2
```

Et surprise, lors de l'exécution, l'affichage signé de b2 sera "157" !!!
Mon dieu, mais qu'est ce qu'on vas faire?

En fait, l'explication est relativement simple.
La variable b2 est fatalement définie comme un octet, donc 8 bits, et la valeur signée 99 sera codée %01100011 en binaire. Lors de la négation de 99, le BS2 réalise deux opérations : d'abord, il convertit le nombre en 16 bits, soit %0000000001100011, puis prend le complément à deux, soit %1111111110011101. Mais, le résultat de cette opération est remis dans une variable 8 bits (b2), et donc les huits bits de poids fort sont éliminés et l'on retrouve dans b2 les huit bits de poids faible, soit : %10011101.

C'est ici que ça se devient comique. Notre commande "DEBUG sdec" est aussi prévue pour travailler dans un espace de travail de 16 bits. Donc, en recevant la valeur de b2, qui rappellons-le est définie en huit bits, elle vas y ajouter huit zéros dans les huit bits de poids fort, et nous aurrons alors la valeur %0000000010011101, ce qui donne en décimal signé la valeur de 157, et voilà.

Chacune des descriptions d'opérations ci-dessous est complètée par un exemple. Il serait opportun de tester vos connaissances à propos des opérateurs en modifiant les exemples et voir si vous êtes capable de prévoir les résultats. Te étudiez, et travaillez avec l'instruction Debug jusqu’à n'en plus finir!
La récompense sera une parfaite compréhension du BS2 et de la manière de calculer des microcontroleurs.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à une opérande.

Voiçi la liste des huit opérations à une opérande.

Opération	Description
ABS	Retourne la valeur absolue
SQR	Retourne la valeur de la racine carrée
DCD	Elevation à 2 exposant la valeur
NCD	Encodeur prioritaire d'une valeur 16-bit
-	Négation sur 16 bits (complément à deux)
~	NON logique
SIN	Calcul du sinus
COS	Calcul du cosinus

ABS (Valeur absolue)

Converti un nombre 16 bits signé (complémentaire à deux) en une valeur absolue. La valeur absolue d'un nombre est un nombre positif représentant la différence entre ce nombre et 0 (zéro). Par exemple, la valeur absolue de -99 est 99. La valeur absolue de 99 est aussi 99. On peut dire que ABS débarrasse le signe moins d'un nombre négatif, laissant inchangé les nombres positifs.

ABS fonctionne sur les nombres négatifs complémentaires à 2.

Exemple :

```
w1 = -99            ' Ecrit -99 (complémentaire à 2 signé) dans w1.
DEBUG sdec ? w1     ' Affiche à l'écran du PC le nombre signé décimal.
w1 = ABS w1         ' Calcule la valeur absolue.
DEBUG sdec ? w1     ' Affiche à l'écran du PC le nombre signé décimal.
```

SQR (Racine carrée)

Calcule la racine carrée entière d'un nombre non signé de 16-bit. (Le nombre ne doit pas être signé, parce que la racine carrée d'un nombre négatif est un nombre "imaginaire"). Rappelez-vous que la plupart des racines carrées ont une partie fractionnaire que le BS2 ignore en faisant son calcul sur uniquement des nombres entiers. Donc, la racine carrée de 100 sera 10 (ce qui est correct), alors que la racine carrée de 99 donnera 9 (alors que le résultat est plus proche de 10 que de 9).

Exemple :

```
DEBUG SQR 100       ' Affiche la racine carrée de 100 (10).
DEBUG SQR 99        ' Affiche la racine carrée de 99 (9).
```

DCD (Elevation à 2 exposant la valeur)

Elève 2 à la puissance d'une valeur de 4 bits. DCD n'accepte qu'une valeur comprise entre 0 et 15, et retourne un nombre de 16 bits dont le bit correspondant à la valeur est mis à 1 (et tous les autres à 0)

Exemple:

```
w1 = DCD 12         ' met bit 12 à 1.
DEBUG bin ? w1      ' Affiche le résultat (%0001000000000000)
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à une opérande (suite).

NCD (Encodeur prioritaire sur une valeur de 16 bits)

NCD prend une valeur 16-bit, trouve le bit de poid le plus fort contenant un 1 et retourne la position du bit plus un (1 à 16). Si il n'y a pas de bit sélectionné, (la valeur de l'Entrée est 0), NCD retourne 0.
NCD est une manière rapide de recevoir une réponse à la question suivante : Quelle est la plus grande puissance de deux que cette valeur peut contenir? La réponse que NCD retourne sera cette puissance, plus un.

Exemple :

```
w1 = %1101          ' le plus haut bit à 1 est le 3ème.
DEBUG ? NCD w1      ' Montre le NCD de w1 (4).
```

\- (Négation sur 16 bits (complément à deux))

Rendre négatif un nombre 16 bits en calculant son complément à deux.

Exemple :

```
w1 = -99            ' Rend le nombre 99 négatif, et l'écris dans W1
DEBUG sdec ? w1     ' Affiche la valeur en décimal signé de W1 à l'écran
w1 = ABS w1         ' Calcule la valeur absolue de W1 et l'écris dans W1
DEBUG sdec ? w1     ' Affiche cette valeur en décimal signé
```

~ (NON logique)

Inverse chaque bit d'un nombre. Tout les bits contenant un 1 sont mis à 0 et tout les bits contenant un 0 sont mis à 1. Ce procédé est aussi appellé complément à un.

Exemple :

```
b1 = %11110001      ' Stocke la valeur %11110001 dans b1
DEBUG bin ? b1      ' Affiche la valeur de b1 en binaire
b1 = ~ b1           ' Calcule le complément à 1 de b1 (l'inverse)
DEBUG bin ? b1      ' Réaffiche la valeur de b1 en binaire
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à une opérande (suite).

SIN (SINUS)

Calcule la valeur du sinus d'un angle exprimé sur une valeur 8 bits (0 à 255), la valeur de ce sinus étant exprimé en un nombre signé complémentaire à deux de 8 bits (-127 à +127).

Voyons d'abord comment s'exprime la valeur du sinus d'un angle dans la mathématique traditionnelle (non informatique). Prenons un cercle avec un rayon de 1 unité, et un système de coordonnées orthogonale au centre du cercle. Le sinus est la coordonnée y, distance depuis le centre du cercle jusqu'à l'intersection d'un angle donné sur le cercle. L'origine de la mesure de l'angle est située à 3 heures sur le cercle (position 0), puis augmente en tournant autour du cercle dans le sens anti-horlogique. Nous avons donc le tableau de valeur suivant :

```
à   0 degrés le sinus vaut   0
à  45 degrés le sinus vaut   0,707
à  90 degrés le sinus vaut   1
à 135 degrés le sinus vaut   0,707
à 180 degrés le sinus vaut   0
à 225 degrés le sinus vaut  -0,707
à 270 degrés le sinus vaut  -1.
à 315 degrés le sinus vaut  -0,707
```

Avec le basic BS2 le cercle n'est pas divisé en 360 parties (degrés), mais en 256 parties. Cette division du cercle en 256 valeurs est aussi appellée "radian binaire" ou BRAD. Chaque brad est donc équivalent à 1,406 degré. De plus, la valeur du sinus qui normalement varie entre -1 et +1, est calculée par le BS2 entre -127 et +127. Il s'agit donc d'un nombre signé complémentaire à deux. Ce choix est en fait guidé par la limitation du BS2 qui est obligé de travailler avec des valeurs entières.

Angle en degré	Valeur sinus	Angle en Brad	Valeur sinus BS2
0	0	0	0
45	0,707	32	89
90	1	64	127
135	0,707	96	89
180	0	128	0
225	-0,707	160	-89
270	-1	192	-127
315	-0,707	224	-89

Pour convertir des brads en degrés, il faut multiplier par 180 et ensuite diviser par 128 ; pour convertir des degrés en brads, multiplier par 128 et diviser par 180.

Voici un petit programme de démonstration de SIN :

```
degr VAR w1                         ' Défini les variables.
sine VAR w2
FOR degr = 0 to 359 STEP 45         ' Utilise les degrés.
sine = SIN (degr * 128 / 180)       ' Converti en brads, exécute SIN.
DEBUG "Angle: ",DEC degr,tab,"Sine: ",SDEC sine,cr
                                    ' Affiche le résultat
Next
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à une opérande (suite).

COS (COSINUS)

Calcule la valeur du cosinus d'un angle exprimé sur une valeur 8 bits (0 à 255), la valeur de ce cosinus étant exprimé en un nombre signé complémentaire à deux de 8 bits (-127 à +127).

Voir les explications de l'opérateur de Sinus (SIN) ci-dessus. Les explications pour Cosinus sont tout a fait les mêmes, avec pour différence que la valeur du cosinus n'est pas la valeur lue sur l'axe X, mais celle qui est lue sur l'axe Y.

Pour tester les particularités de COS, vous pouvez reprendre le programme de démonstration écris pour SIN, et remplacer SIN par COS.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes.

Voici la liste des seize opérations à deux opérandes reconnues par le BS2.

Opération	Description
+	Addition
-	Soustraction
/	Division entière (fournit le quotient)
//	Division entière (fournit le reste)
*	Multiplication (fournit le mot de poids faible)
**	Multiplication (fournit le mot de poids fort)
*/	Multiplication (fournit le mot "central")
MIN	Maintient une valeur supérieure ou égale à une limite
MAX	Maintient une valeur inférieure ou égale à une limite
DIG	Extrait la position d'un chiffre dans un nombre
<<	Décalage à gauche
>>	Décalage à droite
REV	Inverse l'ordre des bits
&	ET logique
|	OU logique
^	OU Exclusif logique

+ (Addition)

Ajoute des variables et/ou des constantes, retourne le résultat en 16 bits. Cette opération travaille exactement comme on pourrait s'y attendre avec les entiers non-signés de 0 à 65535. Si le résultat d'une addition est plus grand que 65535, le bit de report "carry" sera perdu. Si les valeurs a additionner proviennent de nombres 16 bits signés et que le résultat est une variable 16 bits, le résultat de l'addition sera correct en signe et en valeur. Par exemple, l'expression -1575+976 donnera comme résultat une valeur signée -599.

Voyez par vous-même :

```
w1 = -1575
w2 = 976
w1 = w1 + w2                    ' Additionne les nombres.
DEBUG sdec ? w1                 ' Affiche le résultat (-599).
```

- (Soustraction)

Soustraire des variables et/ou des constantes, retourne le résultat en 16 bits. Cette opération travaille exactement comme on pourrait s'y attendre avec les entiers non-signés de 0 à 65535. Si le résultat est négatif, il sera correctement exprimé sous forme d'un nombre de 16 bits signé.

Par exemple :

```
w1 = 1000
w2 = 1999
w1 = w1 - w2                    ' Soustrait les nombres.
debug sdec ? w1                 ' Affiche le résultat (-999).
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

/ (Division)

Diviser des variables et/ou des constantes, le résultat est donné sous la forme d'un nombre de 16 bits.Cette opération travaille exactement comme on pourrait s'y attendre avec les entiers non-signés de 0 à 65535. Par contre, l'utilisation avec des nombre négatifs donne des résultats faux, il ne faut donc utiliser cette opération qu'avec des nombres strictement positifs.

Exemple :

```
w1 = 1000
w2 = 5
w1 = w1 / w2                        ' Divise w1 par w2.
debug dec ? w1                      ' Affiche le résultat (200).
```

Il existe une manière de contourner le problème des résultats faux avec les nombres signés. Pour le faire, il faut faire en sorte que votre programme divise des valeurs absolues, et ensuite rendre le résultat négatif si une (et seulement une) des opérandes est négative. Toutes les valeurs doivent être comprises entre -32767 à +32767.

Exemple :

```
          sign var bit              ' Cette variable contiendra le signe
          w1 = 100
          w2 = -3200
          sign = w1.bit15^w2.bit15  ' Sign = (w1 sign) XOR (w2 sign).
          w2 = abs w2 / abs w1      ' Divise des valeurs absolues.
          IF sign = 0 THEN skip0    ' Rend négatif le résultat si un des
          w2 = -w2                  ' arguments est négatif.
skip0:    DEBUG sdec ? w2           ' Affiche le résultat (-32).
```

// (Reste d'une division)

Retourne le reste laissé après la division d'une valeur par une autre. Certaines divisions n'ont pas de résultats entiers : le résultat se compose d'un nombre entier et d'une fraction. Par exemple, 1000/6 = 166,667. Les mathématiques sur les nombres entiers ne tiennent pas compte de la portion fractionnaire du résultat, ainsi 1000/6=166. Cependant, 166 est une réponse approximative, puisque 166*6=996. L'opération de division a laissé un reste de 4. Le // (double slash) retourne le reste de l'opération de division donnée. Naturellement, les nombres se divisant entièrement, comme 1000/5, produisent un reste de 0.

Exemple :

```
w1 = 1000
w2 = 6
w1 = w1 // w2                       ' Retourne le reste de w1 / w2.
DEBUG dec ? w1                      ' Affiche le résultat (4).
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

* (Multiplication (fournit le mot de poids faible))

Multiplie des variables et/ou des constantes, le résultat est donné sous la forme d'un nombre de 16 bits . Cette opération travaille exactement comme on pourrait s'y attendre avec les entiers non-signés de 0 à 65535. Si le résultat de la multiplication est plus grand que 65535, les bits en excès seront perdus. La multiplication des variables signées seront correctes aussi bien avec les nombres et qu'avec les signes, pour autant que le résultat soit compris entre -32767 à +32767.

Exemple :

```
w1 = 1000
w2 = -19
w1 = w1 * w2                        ' Multiplie w1 par w2.
DEBUG sdec ? w1                     ' Montre le résultat (-19000).
```

** (Multiplication (fournit le mot de poids fort))

Lorsque vous multipliez deux nombres de 16 bits, le résultat est un nombre qui peut atteindre 32 bits. Etant donné que la plus grande variable supportée par BS2 est 16 bits, les 16 bits haut du résultat d'une multiplication 32 bits sont normalement perdus.
L'instruction ** (double étoile) vous donne ces 16 bits haut.

Par exemple, supposons que vous multipliez 65000 ($FDE8) par lui-même. Le résultat est 4.225.000.000 ou $FBD46240. l'instruction * retournera les 16 bits bas (de poids faible), $6240, et l'instruction ** retournera les 16 bits haut(de poids fort), $FBD4 :

```
w1 = $FDE8
w2 = w1 ** w1                       ' Multiplie $FDE8 par lui-même
DEBUG hex ? w2                      ' Affiche le mot de poids fort.
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

*/ (Multiplication (fournit le mot "central"))

Cette opération a pour but de multiplier deux variables et/ou deux constantes, et de fournir dans le résultat (qui est donné sur 32 bits) le mot central, c'est à dire le mot formé de l'octet de poids faible du mot de poids fort et de l'octet de poid fort du mot de poids faible. En d'autres termes, si l'on considère que le résultat d'une multiplication de deux valeurs de 16 bits (2 octets), est une valeur de 32 bits (4 octets), l'opération */ retournera les deux octets centraux.

Cette opération a pour effet de multiplier une valeur par un nombre entier et une fraction. Le nombre entier est l'octet de poids fort du multiplicateur (0 à 255 unités entières) et la fraction est l'octet de poids faible du multiplicateur (0 à 255 unités de 1/256 chacune).

L' instruction */ (star-slash) vous donne une excellente façon de contourner les restrictions mathématiques sur les (nombres entiers seulement) du BS2. Supposons que vous vouliez multiplier une valeur par 1,5 Le nombre entier, et bien sûr l'octet de poids fort du multiplicateur, serait 1, et l'octet de poids faible (fraction) serait 128, étant donné que 128/256=0,5. En fait, il est plus clair d'écrire le */ multiplicateur en hexa ($0180) puisque l'écriture en hexa garde le contenu des octets de poids fort et de poids faible séparément.

Exemple :

```
w1 = 100
w1 = w1 */ $0180                    ' Multiplie par 1.5 [1 + (128/256)]
DEBUG ? w1                          ' Monter le résultat (150).
```

La manière la plus simple de calculer la valeur du mot à ultiliser dans l'opération */ est la suivante : Prendre la valeur de la partie entière comprise entre 0 et 255, et la convertir en hexa, vous obtenez un nombre sous la forme $HH. Puis, prendre la partie fractionnaire sous la forme 0,xxxx et la multiplier par 256, et prendre la partie entière du résultat (qui sera comprise entre 0 et 255) et la convertir aussi en hexa, vous obtenez un nombre sous la forme $LL. La valeur du mot à utiliser sera alors tout simplement la combinaison du premier et du deuxième octet, respectivement en poids fort et en poids faible, et vous obtenez le nombre Hexa $HHLL. Ceci parrait évidement compliqué, mais avec un petit exemple tout vas s'éclaircir :

Supposont que nous voulons multiplier un nombre par Pi (3,14159). L'octet de poids fort sera 3, ce qui donne $03 en hexa. Puis, l'octet de poids faible sera calculé comme suit : 0,14159*256=36,24704 et nous prenons la partie entière, soit 36, que nous convertissons en Hexa, ce qui nous donne $24. Ainsi la constante Pi à utiliser avec */ sera la combinaison de ces deux nombres, soit $0324. Ceci n'est pas parfait pour Pi, mais l'erreur est seulement de 0,1%.

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

MIN (Maintient une valeur supérieure ou égale à une limite)

Limite une valeur à un minimum spécifié positif de 16 bits.

La syntaxe de MIN est:

```
valeur MIN limite
```

où "valeur" est la valeur sur laquelle l'opération MIN va agir.
et "limite" est la valeur minimum que la valeur puisse atteindre.

La logique de MIN est :
1 - si la valeur est inférieure à limite, alors faire valeur=limite
2 - si la valeur est plus grande que ou égale à la limite, alors, faire valeur=valeur.

MIN travaille seulement en mathématique positive ; ces comparaisons ne sont pas valables lorsqu'on utilise les nombres négatifs signés. Cela vient du fait que la représentation d'un entier positif d'un nombre comme -1 ($FFFF ou 65535 en décimal non-signé) est plus grand qu'un nombre comme 10 ($000A ou 10 décimal).
Utiliser MIN seulement avec des nombres entiers positifs.

Exemple :

```
FOR w1=100 TO 0 STEP -10          ' Itération de w1 à partir de 100
                                  ' jusqu'à 0 par pas de 10.
     DEBUG ? w1 MIN 50            ' Affiche w1 en bloquant sa valeur à 50.
NEXT                              ' Bouclage
```

MAX (Maintient une valeur inférieure ou égale à une limite)

Limite une valeur à un maximum spécifié positif de 16 bits.

La syntaxe de MAX est:

```
valeur MAX limite
```

où "valeur" est la valeur sur laquelle l'opération MAX va agir.
et "limite" est la valeur maximum que la valeur puisse atteindre.

La logique de MAX est :
1 - si la valeur est supérieure à limite, alors faire valeur=limite
2 - si la valeur est plus petite que ou égale à la limite, alors, faire valeur=valeur.

Comme pour l'opération MIN, l'opération MAX ne travaille qu'avec des nombres entiers strictement positifs.

Exemple :

```
FOR w1=0 TO 100 STEP 10           ' Itération de w1 depuis 0 jusque 100
                                  ' par pas de 10.
     DEBUG ? w1 MAX 50            ' Affiche w1 en bloquant sa valeur à 50.
NEXT                              ' Bouclage
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

DIG (Extrait la position d'un chiffre dans un nombre)

Cette opération renvoie le chiffre (digit), dont la position est définie par l'argument de l'opération, contenu dans un nombre positif de 16 bits. La position est définie entre 0 (le chiffre le plus à droite) et 4 (le chiffre le plus à gauche).

Exemple :

```
w1 = 9742
DEBUG ? w1 DIG 2                    ' Affiche le 3ième chiffre (7)
FOR b0=0 TO 4
     DEBUG ? w1 DIG b0              ' Affiche les 5 chiffres qui composent
                                    ' le nombre w1 ( 0 9 7 5 2 )
NEXT
```

<< (Décalage à gauche)

Décale vers la gauche les bits qui composent un nombre de 16 bits d'un nombre de places spécifié. Les bits de poids fort qui sont décalés sont perdus, et à droite, les bits de poids faible qui sont décalés sont remplacés par des zéros.
Cette opération de décalage à gauche revient en fait à la multiplication de ce nombre par 2 exposant le nombre de places. Par exemple, l'opération 100 << 3 (décale les bits du nombre décimal 100 à gauche de trois places) équivalent à 100 * (2 exposant 3) et le résultat est donc 800.

Exemple :

```
w1 = %1111111111111111
FOR b0=1 TO 16                      ' Itération de b0 de 1 à 16.
     DEBUG BIN ? w1 << b0           ' Affiche en binaire la valeur
                                    ' de w1 "décalée" par b0
NEXT
```

>> (Décalage à droite)

Décale vers la droite les bits qui composent un nombre de 16 bits d'un nombre de places spécifié. Les bits de poids faible qui sont décalés sont perdus, et à gauche, les bits de poids fort qui sont décalés sont remplacés par des zéros.
Cette opération de décalage à droite revient en fait à la division de ce nombre par 2 exposant le nombre de places. Par exemple, l'opération 100 >> 3 (décale les bits du nombre décimal 100 à droite de trois places) équivalent à 100 / (2 exposant 3) et le résultat est donc 12.

Exemple :

```
w1 = %1111111111111111
FOR b0=1 TO 16                      ' Itération de b0 de 1 à 16.
     DEBUG BIN ? w1 >> b0           ' Affiche en binaire la valeur
                                    ' de w1 "décalée" par b0
NEXT
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

REV (Inverse l'ordre des bits)

Cette opération inverse l'ordre du nombre définit de bits de poids faible d'un mot de 16 bits.
Par exemple, l'opération %0000000010101101 REV 4 aurrait comme résultat %1011, une image miroir des quatre bits de poids faible de la valeur.

Exemple :

```
DEBUG bin ? %11001011 REV 4        ' Affiche les 4 bits de poids faible.
```

& (ET logique (AND))

Cette opération renvoie la combinaison ET bit à bit de deux mots de 16 bits
Chaque bit des valeurs est sujet à la logique suivante:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Dans le résultat de &, les bits vaudront 1 si les deux bits sont égaux à 1, et vaudront 0 dans tous les autres cas.

Exemple :

```
debug bin ? %00001111 & %10101101  ' Affiche le résultat de & (%00001101)
```

| (OU logique (OR))

Cette opération renvoie la combinaison OU bit à bit de deux mots de 16 bits
Chaque bit des valeurs est sujet à la logique suivante:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Dans le résultat de |, les bits vaudront 0 si les deux bits sont eagux à 0, et vaudront 1 dans tous les autres cas.

Exemple :

```
debug bin ? %00001111 | %10101001  ' Affiche le résultat de | (%10101111)
```

Exécution Mathématique et Logique dans les STAMP2 et STAMP2SX.

Les opérations à deux opérandes (suite).

^ (OU Exclusif logique (XOR))

Cette opération renvoie la combinaison OU exclusif bit à bit de deux mots de 16 bits. Chaque bit des valeurs est sujet à la logique suivante:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Dans le résultat de ^, les bits vaudront 0 si les deux bits sont égaux à 0 ou les deux bits sont égaux à 1, et vaudront 1 dans tous les autres cas.

Exemple :

```
debug bin ? %00001111 ^ %10101001  ' Montre XOR résultat (%10100110).
```