# APM

# Introduction au 8051 - Programmation et Interfaces

| | |
|---|---|
| **Version** | **2.4** |
| **Status** | ***Terminé (pour le moment)*** |
| **Date** | **07/09/2013** |
| **Auteur** | **Jean-Michel Gonet** |

1. Le microprocesseur....5
1.1 Qu'est-ce qu'un ordinateur?....5
1.2 Les premiers ordinateurs....7
1.2.1 La machine d'accès aléatoire....7
1.2.2 Les machines de Konrad Zuse....8
1.2.3 L'architecture Harvard....9
1.2.4 L'architecture von Neumann....10
1.3 Mémoire et périphériques....11
2. Le 8051 en 5 minutes....13
2.1 8051 - Spécifications....13
2.2 8051 - Structure de la mémoire de données interne....14
2.3 8051 – Les ports d'entrée / sortie numériques....15
2.3.1 Diagramme des ports 1, 2 et 3....15
2.3.2 Diagramme du port 0....16
2.3.3 Le port 0 et les entrées digitales....17
2.4 8051 – Configuration technique....17
2.5 Le cycle d'instruction et la fréquence Fosc/12....18
3. Quelques composants de base....19
3.1 Le « DIP switch »....19
3.2 Interrupteurs et résistances de tirage....19
3.3 Les afficheurs LED 7 segments....21
3.3.1 La résistance de protection....21
3.3.2 Les afficheurs multiples....22
3.3.3 Les convertisseurs BCD à 7 segments....23
4. Pratique – Utilisation des ports d'entrée / sortie....25
4.1 Le circuit....25
4.2 Le programme....26
4.3 Le résultat....26
4.4 Quelques questions....26
4.5 Relation entre software et hardware....26
5. 8051 – Le temps réel....29
5.1 8051 – Les temporisateurs – Timers....29
5.1.1 Fonctionnement des temporisateurs....29
5.1.2 Configuration des temporisateurs....30
5.1.3 Calcul du temps....31
5.2 Les interruptions....32
5.2.1 8051 – Les interruptions sur le 8051....33
5.2.2 Configuration des interruptions....35
5.2.3 C51 – Mise en place des interruptions depuis Keil uVision C....35
6. Pratique – Temporisateurs et interruptions....37
6.1 Le circuit....37
6.2 Le programme....37
6.3 Le résultat....39
6.4 Quelques questions....39
7. 8051 – La UART....41
7.1 La communication série....41
7.2 La UART du 8051....42
7.3 Configuration de la UART....42
7.4 Calcul de la vitesse de transmission....43
7.4.1 Fréquence de transmission basée sur le timer1....44
7.4.2 Fréquence de transmission basée sur le cycle d'instruction....44
7.5 Exemple de configuration de la UART....44

7.6 Pratique – Utiliser la UART comme sortie standard – printf ..... 45
7.6.1 Le circuit ..... 45
7.6.2 Le programme – redéfinition de putchar ..... 46
7.6.3 Le résultat ..... 47
7.6.4 Quelques questions ..... 48
8. Le C en 5 minutes ..... 49
8.1 Programmation C – int, char et le modificateur unsigned ..... 49
8.2 Programmation C – Opérations booléennes ..... 49
8.3 Programmation C – Précédence des opérateurs ..... 50
8.4 Programmation C – Variables locales et variables globales ..... 51
8.5 Programmation C – Pointeurs et tableaux ..... 52
8.6 Programmation C – Les énumérations ..... 53
8.7 Structures ..... 54
8.7.1 Collections de structures ..... 55
8.8 Les unions ..... 56
8.8.1 La virgule fixe ..... 57
8.9 Code distribué sur plusieurs fichiers ..... 58
9. Les composants TTL ..... 61
9.1 Concept – « Chip Select » et l'état de « Haute Impédance » ..... 61
9.2 Composants – Les séries TTL ..... 62
9.2.1 Composants – La famille TTL-HC ..... 63
9.2.2 Composants – Identifiants de circuits TTL à usage courant ..... 63
9.3 Composants – La bascule ..... 64
9.3.1 Composants – Le 74HC373 ..... 65
9.4 Composants – Le multiplexeur et le décodeur ..... 66
9.4.1 Composants – Décodeurs / démultiplexeurs TTL ..... 67
10. La mémoire externe ..... 69
10.1 8051 – L'accès à la mémoire de données externe ..... 69
10.1.1 8051 – Le port 0 en mode mémoire externe ..... 69
10.1.2 8051 – Cycles de lecture et d'écriture de la mémoire externe ..... 70
10.1.3 Programmation C pour 8051 – xdata ..... 71
10.2 Pratique – Lire et écrire dans la mémoire externe ..... 72
10.2.1 Le circuit ..... 72
10.2.2 Le connecteur CE ..... 72
10.2.3 Le programme ..... 73
10.2.4 Le résultat ..... 73
10.2.5 Quelques questions ..... 74
11. Pratique - Une mémoire d'affichage ..... 75
11.1 Le circuit ..... 75
11.2 Le programme ..... 76
11.3 Le résultat ..... 76
11.4 Quelques questions ..... 76
12. Le clavier ..... 77
12.1 Composant – Le clavier ..... 77
12.1.1 Proteus – Les claviers virtuels ..... 78
12.2 Pratique - Le clavier comme une mémoire externe ..... 79
12.2.1 Le circuit ..... 79
12.2.2 Le programme ..... 80
12.2.3 Le résultat ..... 81
12.2.4 Quelques questions ..... 81
13. Le Bus ..... 83
13.1 Concept – Le bus ..... 83

13.2 Pratique – Contrôleur de bus....84
13.2.1 Le contrôleur de bus....85
13.2.2 Les trois périphériques....85
13.2.3 Le programme....86
14. Les test unitaires....89
14.1 Qu'est-ce qu'un test unitaire....89
14.2 Programmation C – Structure pour tests unitaires....89
14.2.1 Programmation C – Le préprocesseur....89
14.2.2 Keil uVision – Créer un environnement de test....90
14.2.3 Programmation C – Structure de base pour les tests unitaires....92
14.2.4 Keil uVision – Lancer les tests unitaires directement....96
14.3 Concept – Le développement piloté par les tests....96
14.4 Exemple pratique de test unitaire....96
15. Keil uVision....99
15.1 Introduction....99
15.2 Création et configuration du projet....99
15.3 Configurer un projet pour Proteus VSM....101
15.4 Ajouter des fichiers dans le projet....102
15.5 Quelques particularités du C pour 8051....104
15.6 Compilation....104
16. Proteus....107
16.1 Introduction....107
16.2 ISIS, ARES et VSM....107
16.3 Placer les composants....108
16.3.1 Choisir les composants du circuit....108
16.3.2 Placer les composants dans le circuit....109
16.3.3 Remplacer un composant par un autre....110
16.4 Connecter les composants....110
16.4.1 Les simples conducteurs....110
16.4.2 Les bus....111
16.4.3 Les terminaux....112
16.4.4 L'alimentation et la terre....114
16.5 Naviguer sur le circuit....114
16.5.1 Gérer la taille de la feuille....114
16.5.2 Distribuer un circuit sur plusieurs feuilles....115
16.6 Simuler le circuit....115
16.6.1 Chargement du programme avec MPLABX....116
16.6.2 Chargement de programmes avec Keil uVision....117
16.6.3 Lancement de la simulation....117
16.6.4 Exécution pas à pas....118
Appendice A. Table des illustrations....121

# 1. Le microprocesseur

Un microprocesseur est une petite machine qui se trouve à l'intérieur des ordinateurs. Cette petite machine lit ou exécute les programmes, puis commande les différentes parties de l'ordinateur pour leur faire accomplir ce qui est demandé.

Selon cette description, dans un autobus, le microprocesseur serait le chauffeur: Il lit les signes de trafic, puis il contrôle le bus pour ne pas sortir de la route ni effrayer les autres automobilistes. En même temps, il a un horaire à suivre, des arrêts où les passagers montent et descendent. Le conducteur suit un programme.



*Illustration 1: Le conducteur est le microprocesseur de l'autobus*

Pour décrire avec plus de précision ce qu'est un microprocesseur, il faut commencer par définir ce qu'est un ordinateur.

## 1.1 Qu'est-ce qu'un ordinateur?

Voici la définition d'ordinateur donnée par le dictionnaire:

> *Un ordinateur est une machine capable de transformer des informations selon un programme.*

Ceci laisse quelques questions ouvertes:

- Qu'est-ce qu'une machine?
- Qu'est-ce qu'une information?
- Qu'est-ce qu'un programme?

Une machine est un outil doté d'un mécanisme.

Par exemple, un simple tournevis n'est pas une machine. Mais un tournevis électrique en est un. Le fait qu'un objet soit une machine ou pas ne dépend pas de son utilité, mais de son principe de fonctionnement. On parle d'une cafetière, ou d'une machine à café.

- Une machine consomme de l'énergie pour réaliser un travail.
- Une machine a un propos. Elle sert à quelque chose. Son existence a été planifiée.
- Une machine a été construite de toutes pièces. Normalement par les hommes. Par exemple, un cochon sert à quelque chose, mais ce n'est pas une machine.

On parle d'automatisme lorsqu'une machine a un certain degré d'initiative, voir d'indépendance. Par exemple, une machine à laver peut travailler toute seule, une fois qu'on l'a chargée de linge sale, mis le détergent, et sélectionné le programme.



*Illustration 2: Sélection du programme, sur une machine à laver*

Pour qu'une machine ait de l'initiative elle doit être capable de prendre des décisions. Elle a besoin de:

- Informations sur les quelles baser ses choix.
- Règles, qui transforment les informations en choix.

La machine à laver ne prend aucune décision tant que son propriétaire n'a pas choisi et lancé un des programmes de lavage. Mais, dès que le programme démarre, la machine vérifie si la porte est fermée. Pour cela elle a besoin de:

- Information: Oui ou non, la porte est-elle fermée? Pour cela la machine à laver possède un senseur qui se trouve collé au loquet de la porte.
- Règle: Si la porte est ouverte, alors attendre.

Une fois que la porte est fermée, la machine à laver passe au stade suivant du programme, qui consiste à faire entrer de l'eau. De nouveau elle a besoin de:

- Information: Quel est le niveau de l'eau dans le tambour? Pour cela un senseur est collé à un flotteur.
- Règle: Tant que le niveau est plus bas qu'une quantité, alors attendre.

Dans chaque cas, l'information est arrivée par le biais d'un senseur. Un senseur transforme un phénomène physique en signal électrique. Ce signal électrique arrive dans le cœur décisionnel de la machine qui lit le programme. Le programme contient les règles. Le résultat d'appliquer une règle à une information est une action. Il existe deux types d'actions:

- Des actions qui affectent le déroulement du programme. Ou, dit autrement, sauter à une autre section du programme. Par exemple: "*tant que la porte est ouverte, attendre. Dès qu'elle se ferme, alors passer au pas suivant*". Ou encore: "*tant que le niveau d'eau n'est pas suffisant, attendre*"
- Des vraies actions. Par exemple: "*ouvrir la vanne d'eau*". Ou encore: "*faire tourner le tambour*"

Les vraies actions sont initiées par un signal électrique. Ce signal est transformé en mouvement par un moteur.

Il y a donc, au cœur de la machine à laver, un engin qui reçoit des signaux électriques en provenance de senseurs, et les transforme, au moyen de règles (le programme), en d'autres signaux électriques, qui actionnent des moteurs, des relais magnétiques, des alarmes sonores, etc.

Cet engin vit dans un monde où tout est représenté par des signaux électriques. Cet engin ne travaille pas avec la réalité. Il travaille avec une représentation de la réalité. Il vit dans le monde de l'information.

Cet engin est un ordinateur.

## 1.2 Les premiers ordinateurs

Les algorithmes sont connus depuis l'antiquité. Le premier, et plus fameux, est le calcul du plus grand commun diviseur (PGDC), inventé dans l'antiquité par Euclide.

> *Pour calculer le PGCD de deux nombres, A et B:*
> *1. Effectuer la division euclidienne de A par B. Soit R le reste.*
> *2. Remplacer A par B et B par R.*
> *3. Recommencer.*
> *Le PGCD est le dernier reste R non nul.*

L'ennui de cet algorithme est qu'il ne peut être interprété que par des humains. Pour tenter de comprendre comment une machine pensante pourrait exécuter cet algorithme, il faut d'abord trouver un moyen de le réduire en *opérations élémentaires*. Même *opération élémentaire* est un concept flou. Pour un adulte, la division euclidienne peut paraître élémentaire, mais un enfant de 5 ans ne saurait pas la réaliser.

### 1.2.1 La machine d'accès aléatoire

En 1936, deux chercheurs, M. *Alain Turing*, et M. *Emil Post* publièrent des études sur la formalisation des algorithmes. On déduit de leurs théories que tout algorithme peut être décomposé en une série ordonnée d'opérations élémentaires de trois types:

- Opérations arithmétiques sur des données, qui donnent comme résultat d'autres données (pour ceci il est nécessaire d'avoir un support d'où lire et écrire les données).
- Sauts d'un point de la série d'opérations à un autre.
- Sauts d'un point de la série d'opérations à un autre, suivant le résultat d'une opération arithmétique.

La machine capable d'interpréter ces algorithmes s'appelle « *machine d'accès aléatoire* »:

> *Une machine abstraite qui possède un nombre indéterminé d'éléments de mémoire qu'elle peut accéder dans le désordre, et qui suit pas à pas un programme composé d'un nombre indéterminé d'instructions simples (des accès aux éléments de mémoire, des opérations arithmétiques basiques, des sauts à l'intérieur du programme, éventuellement conditionnés au résultat de la dernière opération).*

Alan Turing fut retrouvé mort dans son lit, empoisonné au cyanure. Une pomme à moitié croquée se trouvait à côté du lit. L'enquête n'estima pas opportun de vérifier si la pomme était contaminée. Il est possible que Turing ait trempé la pomme volontairement dans le poison, mais il est également possible qu'il s'agisse d'un accident.



*Illustration 3: La pomme de Apple est-elle un hommage à Alan Turing?*

Ce n'est pas officiel, mais il a été spéculé que le logo de Apple est en honneur des contributions de Alan Turing aux sciences de l'information[1]. Il se peut que la fameuse pomme fasse plutôt référence à celle de Newton, ou bien encore celle de Adan et Eve. L'idée de l'arc-en-ciel est de Steve Jobs, pour bien mettre l'accent sur la qualité des couleurs de l'*Apple II*.

### 1.2.2 Les machines de Konrad Zuse

La première réalisation de machine d'accès aléatoire est la *Zuse Z1*, construite par Konrad Zuse dans l'appartement de ses parents[2]. Zuse était un ingénieur civil, et conçu la Z1 entre 1936 et 1938 comme une aide pour ses calculs. Dans sa conception, la *Zuse Z1* était très semblable à un ordinateur moderne: instructions, mémoire, routines, sous-routines, etc. Par contre, elle était entièrement mécanique, elle avait une fréquence d'horloge de 4Hz, et ne pouvait marcher que quelques minutes au maximum. La Z2 (1939) était basée sur des relais. Elle fonctionna juste assez pour impressionner une audience de la *Deutsche Versuchsanstalt für Luftfahrt*, qui lui finança la Z3 (1941) comme un projet ultra-secret pour l'armée de l'air.

1 D'après sa récente biographie, parue en 2011, Steve Jobs a déclaré qu'il aurait bien voulu que ce soit vrai mais qu'il n'y avait jamais pensé lui même.

2 Voir également http://en.wikipedia.org/wiki/Z1_(computer), http://en.wikipedia.org/wiki/Z2_(computer), http://en.wikipedia.org/wiki/Z3_(computer), et http://en.wikipedia.org/wiki/Z4_(computer).

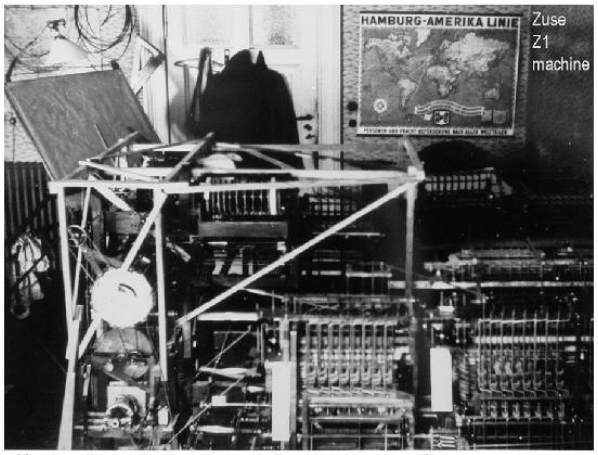



*Illustration 4: La Zuse Z1, construite dans l'appartement des parents de Konrad Zuse*

La Z4 (1945) fut construite par Konrad Zuse dans l'intention de la vendre. L'Allemagne perdant la guerre, il dû la déménager plusieurs fois pour fuir le front et l'armée russe. En 1949, le mathématicien suisse Eduard Stiefel l'acheta et l'amena à l'ETH de Zürich en 1950. La Z4 fut utilisée pour réaliser les calculs de structure de la Grande Dixence.

Par la suite, Konrad Zuse construisit plus de 200 autres machines. Sa compagnie fut finalement achetée par Siemens en 1967.

Les deux clefs du succès des machines de Zuse furent :

- Utiliser un système très simple pour représenter les nombres : le système binaire.
- Utiliser un micro-séquenceur dans le bloc de contrôle : chaque instruction du programme pointe sur une séquence spécifique du micro-séquenceur. Il devient possible de créer des nouvelles instructions en ajoutant des nouvelles micro-séquences, sans toucher le reste du mécanisme.

Cette simplicité lui permit de construire ses premières machines tout seul, et les suivantes avec un staff réduit. Pour la Z4 il avait une équipe de 25 personnes autour de lui, ce qui n'est pas plus grand qu'un modeste bureau d'ingénieurs.

Ces deux concepts sont toujours présents dans les ordinateurs modernes.

### 1.2.3 L'architecture Harvard

le ASCC (*Automatic Sequence Controlled Calculator*), construit par IBM, en association avec l'Université de Harvard, et livré en 1944, est plus connu sous le nom de Harvard Mark I[1].

Beaucoup plus grand et plus cher que ceux de Zuse, il servit à faire des calculs balistiques.

Il lisait le programme d'une bande perforée. En conséquence, le programme était strictement séparé des données.

1 Voir à ce sujet http://www-03.ibm.com/ibm/history/exhibits/markI/markI_intro.html

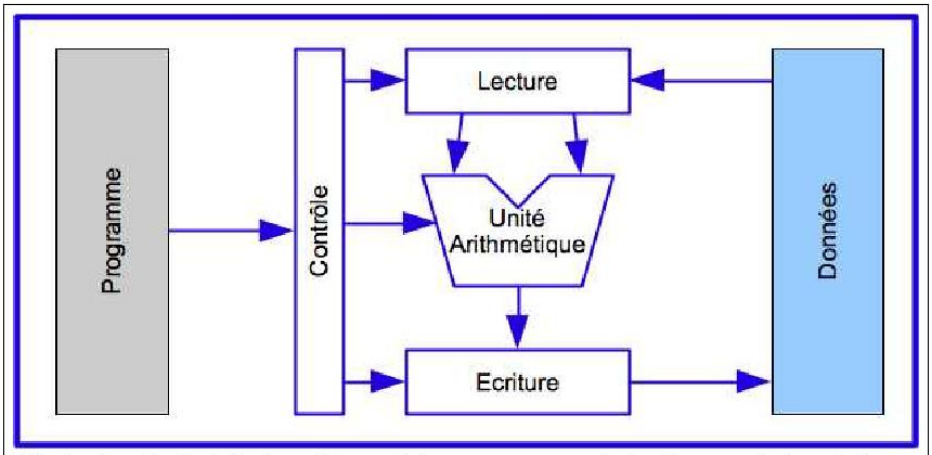



*Illustration 5: Architecture Harvard: le programme est physiquement séparé des données.*

L'architecture de Harvard a l'avantage d'être simple : pour exécuter un programme, il faut alterner des lectures d'instructions et des lectures / écritures de données. Le fait que les deux éléments soient sur des supports différents rend le mécanisme plus simple.

### 1.2.4 L'architecture *von Neumann*

En 1945, *John von Neumann* publia ce qui s'appelât par la suite architecture de Neumann, et qui est celle de la plupart des ordinateurs modernes:

> *Une machine d'accès aléatoire qui a une unité de traitement séparée d'une unité de stockage. L'unité de stockage peut conserver à la fois des programmes et des données.*

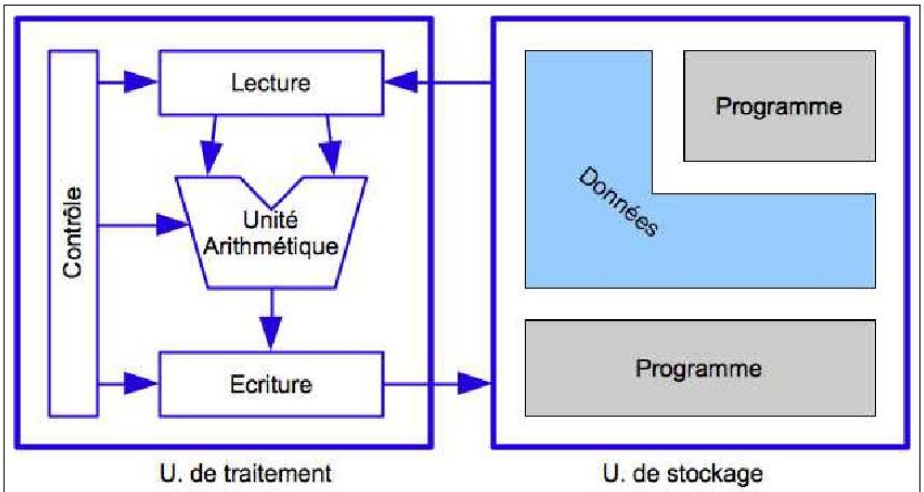



*Illustration 6: Architecture von Neumann: le programme et les données sont sur le même support.*

Le fait que le programme et les données soient sur le même support, permet plusieurs choses importantes :

- Il est possible de « charger » un programme dans la mémoire, puis de l'exécuter.
- Il est possible pour un programme de créer d'autres programmes (assembleurs, compilateurs).
- Il est possible pour un programme de se copier sur d'autres programmes du même ordinateur, et même d'autres ordinateurs si des mécanismes de transmissions existent (virus informatique).
- Il est théoriquement possible pour un programme de créer des versions améliorées de lui même (science fiction ?)

### 1.3 Mémoire et périphériques

Les architectures de Neumann et de Harvard peuvent paraître assez éloignées des réalités domestiques. Cependant, pour permettre à une machine d'accès aléatoire de contrôler les moteurs, thermostats, pompes et verrous d'un lave-linge, il suffit de connecter certains des éléments de mémoire à des sondes et des relais:

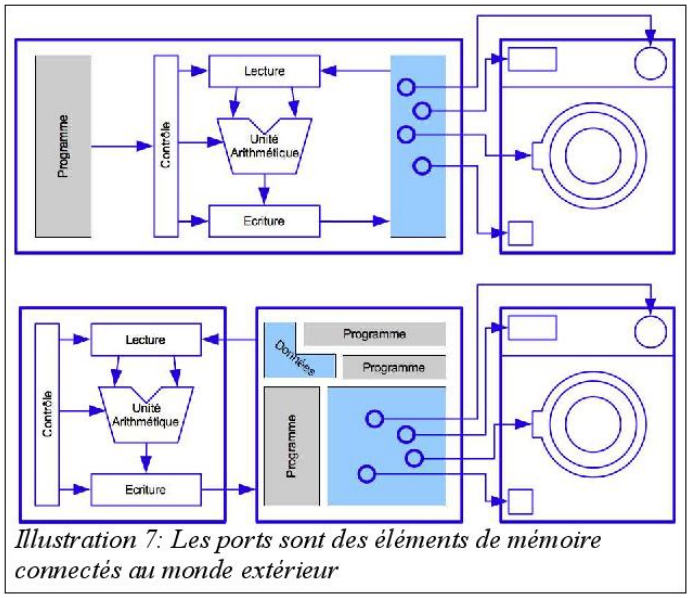



*Illustration 7: Les ports sont des éléments de mémoire connectés au monde extérieur*

Selon leur utilité principale, il y a deux types d'éléments de mémoire:

- Ceux qui ne font que stocker les informations. C'est à ceux-ci qu'on se réfère en général par le terme « *élément de mémoire* ».
- Ceux qui sont connectées à des mécanismes externes. Ces éléments sont des « ports », et les mécanismes aux quels ils sont connectés sont des « périphériques »:
  - Un élément de mémoire relié à une sonde permet à l'unité de contrôle d'effectuer une lecture du monde physique. Cet élément de mémoire est un « port d'entrée ».
  - Un élément de mémoire connecté à un relais permet à l'unité de contrôle d'agir sur le monde physique. Cet élément de mémoire est un « port de sortie ».

# 2. Le 8051 en 5 minutes

## 2.1 8051 - Spécifications

- Le 8051 a quatre ports de 8 bits. Chaque bit peut être configuré comme une entrée ou comme une sortie.
- Le 8051, comme tous les microcontrôleurs, possède des périphériques internes :
  - 4 ports d'entrée / sortie numériques.
  - 2 compteurs / temporisateurs. Ceux-ci sont pratiques pour cadencer des actions dans le programme. Par exemple, pour alterner les temps d'activité et d'arrêt du tambour d'un lave-linge.
  - Un émetteur / récepteur asynchrone universel, appelé couramment UART[1]. Il s'agit d'une unité pour transmettre des données en série. Celui-ci lui permet par exemple de communiquer avec un dispositif de type terminal. Ou également un autre ordinateur, via le port série COM.
- Le 8051 suit l'architecture de Harvard : le stockage du code est séparé du stockage des données.
  - Le 8051 possède une mémoire interne pour les données de 256 octets.
  - Le 8051 possède une mémoire interne pour le code de 2048 bytes[2].
- Le 8051 peut également lire des données d'une mémoire externe (voir 10.1), et lire le code d'une mémoire externe.

1 *Universal Asynchronous Receiver Transmitter*.

2 Le byte (prononcé « *baïte* »), de symbole B, est la plus petite unité adressable d'un ordinateur. Un octet est un ensemble de 8 bits. Dans la majorité des architectures, les bytes sont de 8 bits. Voir également http://fr.wikipedia.org/wiki/Byte

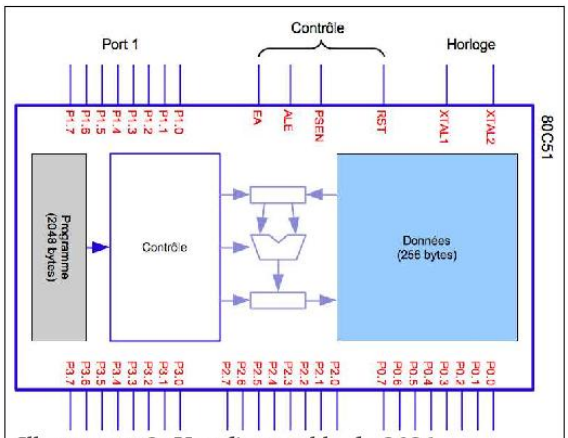


*Illustration 8: Vue d'ensemble du 8081*

## 2.2 8051 - Structure de la mémoire de données interne

La mémoire interne du 8051 est divisée en deux moitiés :

- Les adresses entre 0x00 et 0x7F sont des registres à usage général (des éléments de mémoire). Cette zone est appelée simplement mémoire interne (*Internal RAM*).
- Les adresses entre 0x80 et 0xFF sont des registres à usage spécial (*Special Function Registers*). Ils sont utilisés pour contrôler les périphériques internes : UART, temporisateurs, ports d'entrée / sortie.

Voici un schéma de la mémoire de données du 8051 :

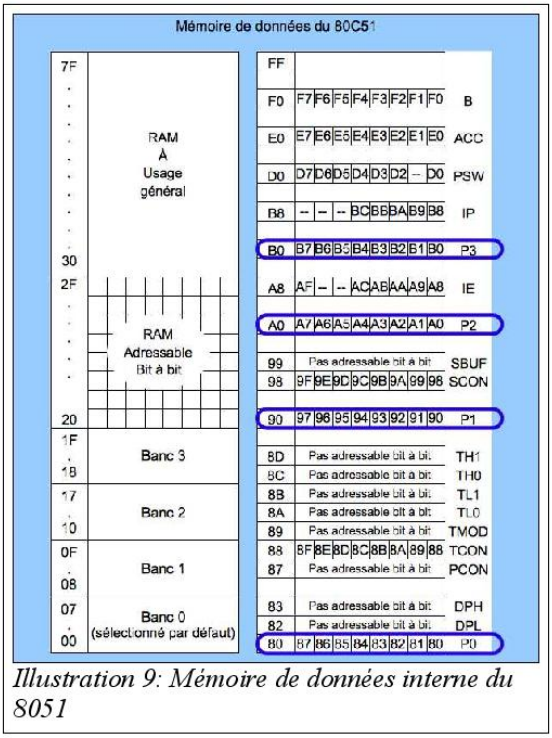


*Illustration 9: Mémoire de données interne du 8051*

## 2.3 8051 – Les ports d'entrée / sortie numériques

Les adresses `0xB0`, `0xA0`, `0x90` et `0x80` de la mémoire interne sont directement reliées aux 4 ports du 8051. Les ports sont bidirectionnels :

- Une lecture de l'adresse correspondante au port permet de lire l'état d'un périphérique branché dessus.
- Une écriture sur l'adresse correspondante au port permet d'établir l'état d'un périphérique branché dessus.

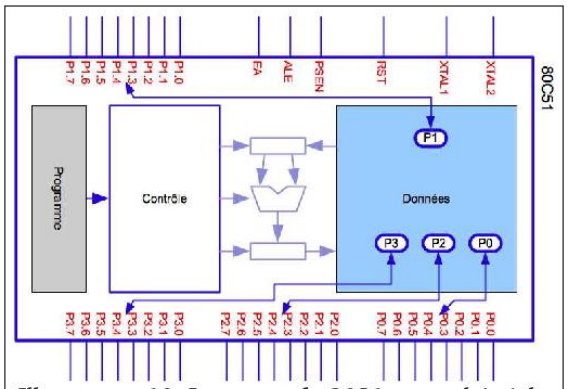



*Illustration 10: Les ports du 8051 sont reliés à la mémoire de données*

### 2.3.1 Diagramme des ports 1, 2 et 3

Les ports 1, 2 et 3 ont huit connecteurs chacun. Chaque connecteur est relié avec le bit correspondant dans la mémoire de données du micro contrôleur par un circuit ressemblant au suivant :

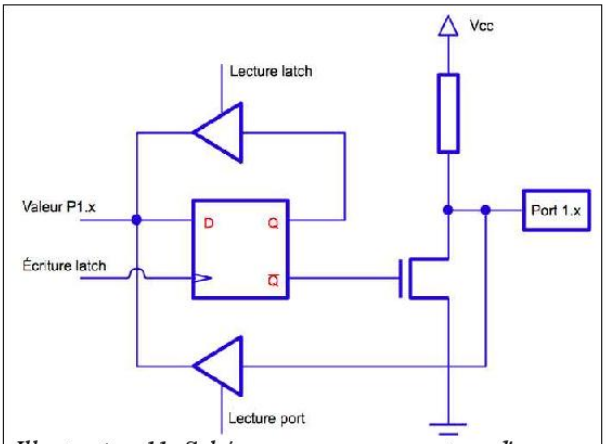



*Illustration 11: Schéma pour un connecteur d'un port. Chaque port a huit connecteurs*

Par exemple, pour configurer le port en sortie, et y écrire un 1 :

1 Le uC écrit un `1` sur la donnée `D` de la bascule (voir point 9.3, des explications supplémentaires sur les bascules de type D).

2 La sortie $\overline{Q}$ de la bascule se met à 0, ce qui bloque le courant à travers le MOS.

3 Le courant qui traverse la résistance (se déversant à travers le port) est très petit.

4 En conséquence le port se met à 1.

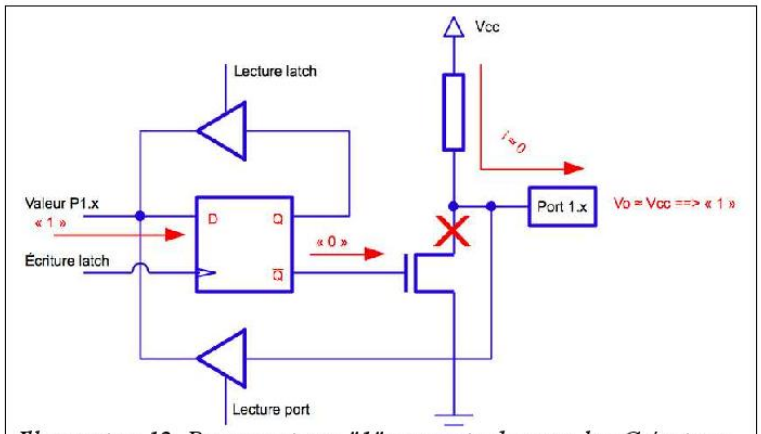



*Illustration 12: Pour avoir un "1" en sortie du port, le uC écrit un "1" sur la bascule*

### 2.3.2 Diagramme du port 0

Le port 0 est légèrement différent des ports 1, 2 et 3 : il est configuré en « collecteur ouvert ». Ou, plutôt, en « drain ouvert », puisque le 8051 est implémenté en technologie CMOS. Ceci veut dire qu'il n'a pas résistance de « tirage » (*pull-up* en anglais) :

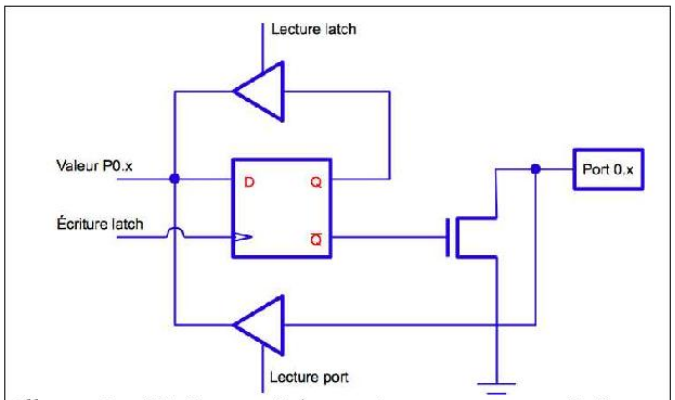



*Illustration 13: Le port 0 (contrairement aux ports 1, 2 et 3) n'a pas de résistance de tirage*

En conséquence, le port 0 ne peut produire de courant. Il peut qu'en consommer :

- À la place de sortir un « 0 », il « consomme » du courant, c'est à dire qu'il se mettre en circuit ouvert.
- A la place de sortir un « 1 », il « bloque » le courant, c'est à dire qu'il se met en circuit fermé.

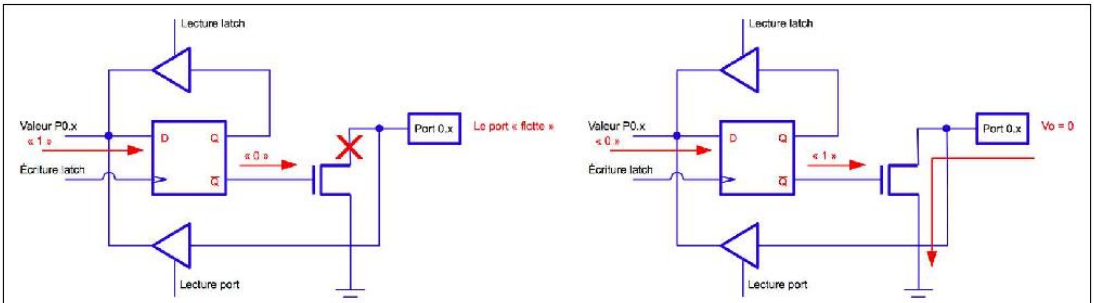



*Illustration 14: Le port 0 peut soit « consommer » du courant, soit « bloquer » le courant.*

Ceci a quelques conséquences sur la manière de connecter des circuits sur le port 0 (voir le chapitre 4 et le point 2.3.3).

### 2.3.3 Le port 0 et les entrées digitales

Comme décrit au point 2.3.2, le port 0 n'a pas de résistances de tirage. En conséquence, il ne peut que consommer du courant, il ne peut pas en produire. Pour connecter des entrées digitales (un latch, par exemple), il faut ajouter extérieurement les résistances de tirage.

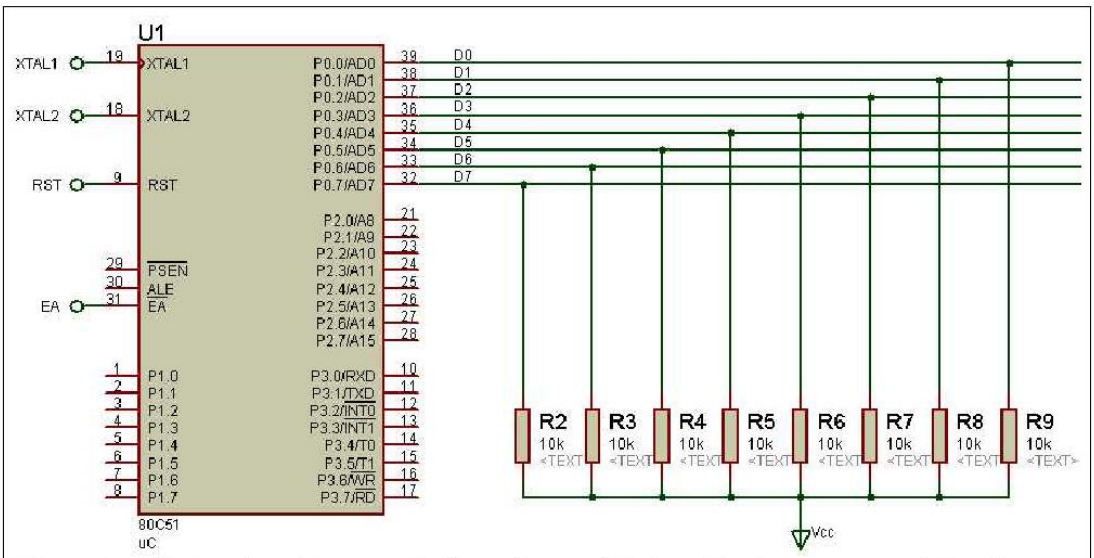



*Illustration 15: Avec les résistances de tirage, le port 0 devient identique aux ports 1, 2 et 3*

## 2.4 8051 – Configuration technique.

Voici les requis techniques minimum pour mettre en fonctionnement un micro-contrôleur 80c51 :

- RST (entrée): Lorsque cette entrée vient à niveau haut, le 8051 est réinitialisé. Ceci veut dire que les registres spéciaux reviennent à leurs valeurs par défaut. Pendant l'allumage il est recommandé de conserver cette entrée à niveau haut pendant au moins 2ms.
- $\overline{EA}$ (entrée): Lorsque cette entrée est à niveau haut, le 8051 exécute le programme de la mémoire programme interne. La mémoire interne est limitée à 2048 bytes.

- `XTAL1` et `XTAL2` : Le 8051 possède un oscillateur interne qui tourne à 12MHz. Pour compléter le circuit interne, il faut brancher entre ces deux bornes un cristal de 12MHz et deux condensateurs de 30pF.

Voici le strict minimum pour que le 8051 puisse fonctionner (les bornes d'alimentation sont omises du circuit) :

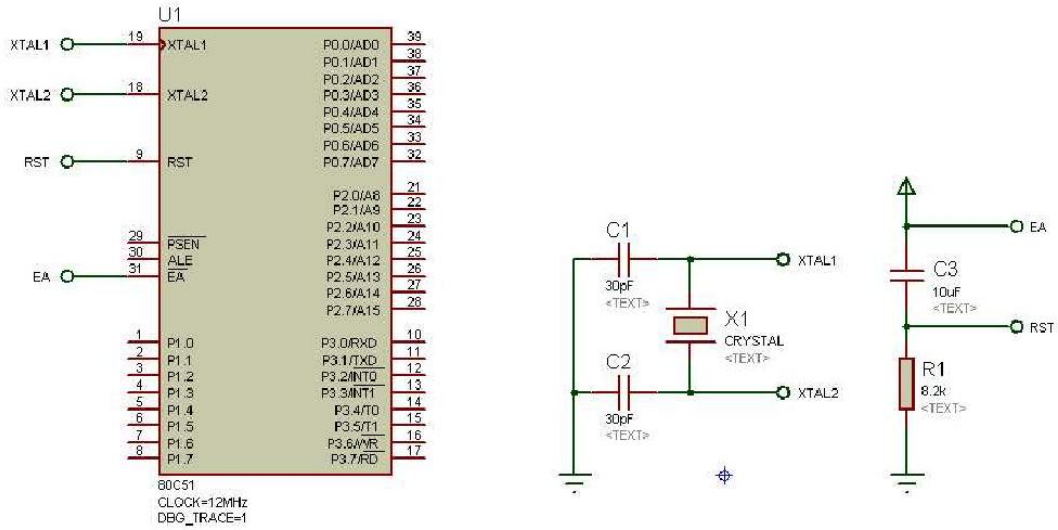



*Illustration 16: Configuration minimum pour le 8051*

## 2.5 Le cycle d'instruction et la fréquence Fosc/12

Le cycle d'instruction[1] est le cycle opérationnel de base d'un microprocesseur ou d'un microcontrôleur. Il consiste en :

- Récupérer l'instruction à exécuter de la mémoire programme.
- Déterminer quelles actions sont requises par l'instruction.
- Eventuellement récupérer des données supplémentaires de la mémoire programme ou de la mémoire de données.
- Exécuter les actions.

Ce cycle est répété depuis que le microprocesseur ou microcontrôleur est mis sous tensions, jusqu'à ce qu'il est mis hors tension.

Sur le 8051, un cycle d'instruction s'exécute en 12 cycles d'horloge. Par exemple, si le 8051 est cadencé à 12MHz, il exécute $1x10^6$ instructions par seconde.

Dans la documentation du 8051, la durée du cycle d'instruction est typiquement notée par Fosc/12, où Fosc est la fréquence de l'oscillateur externe ou horloge. La fréquence Fosc/12 est à prendre en compte pour toutes les opérations en temps réel (voir chapitre 5).

1 Extrait de Wikipedia http://en.wikipedia.org/wiki/Instruction_cycle

# 3. Quelques composants de base

## 3.1 Le « *DIP switch* »

Un « *DIP switch* » consiste en un ou plusieurs interrupteurs qui ont exactement la taille d'un DIP.

> *En micro-électronique, un boîtier DIP (Dual Inline Package), est un boîtier de circuit intégré qui connecte un circuit intégré au monde extérieur. On rencontre également le terme DIL pour désigner ces boîtiers. Les boîtiers DIP peuvent être directement soudés sur le circuit imprimé (PCB), ou insérés dans des supports eux-mêmes soudés, permettant un remplacement facile du composant et une réduction des risques de destruction lors de la soudure.*
>
> *Les boîtiers DIP peuvent être aussi bien utilisés pour les circuits intégrés (tels que les microprocesseurs) que pour des rangées de composants électroniques discrets (tels que les résistances).*
>
> *(source : http://fr.wikipedia.org/wiki/Dual_Inline_Package)*

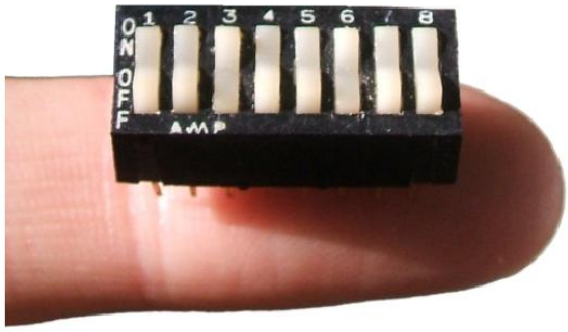



*Illustration 17: Un « DIP switch » de 8 éléments*

## 3.2 Interrupteurs et résistances de tirage

Les entrées digitales ont une impédance d'entrée très élevée. Il faut en tenir compte si on branche un interrupteur directement dessus :

- Lorsque l'interrupteur est fermé (donc, laisse passer le courant), l'entrée digitale est

directement connectée à l'alimentation, et détecte correctement un « 1 » logique.

- Par contre, lorsque l'interrupteur est ouvert (donc, ne laisse pas passer de courant), l'entrée digitale n'est connectée à rien. En conséquence, il n'est pas garanti qu'elle détecte un « 1 ». L'électricité statique ou les parasites peuvent suffire à la faire basculer d'un niveau à l'autre.

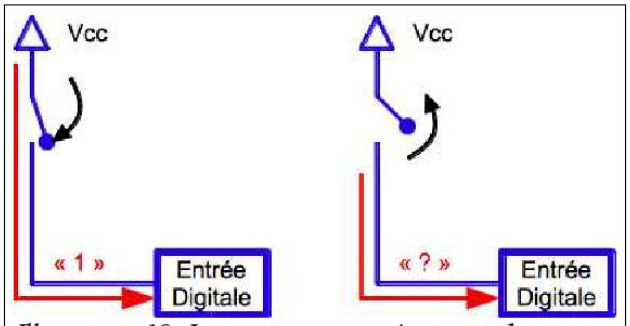


*Illustration 18: Interrupteur sans résistance de tirage*

Pour que, lorsque l'interrupteur est ouvert, l'entrée digitale soit forcée au niveau « 0 », on utilise une résistance de tirage :

- Lorsque l'interrupteur est fermé, le courant passe à travers la résistance. L'entrée digitale se trouvant branchée directement à Vcc, elle perçoit un « 1 » logique.
- Lorsque l'interrupteur est ouvert, l'entrée digitale est branchée à la masse à travers la résistance. Comme la résistance interne de l'entrée digitale est très élevée, la résistance de tirage fait presque l'effet d'un court-circuit. En conséquence, l'entrée digitale perçoit un « 0 ».

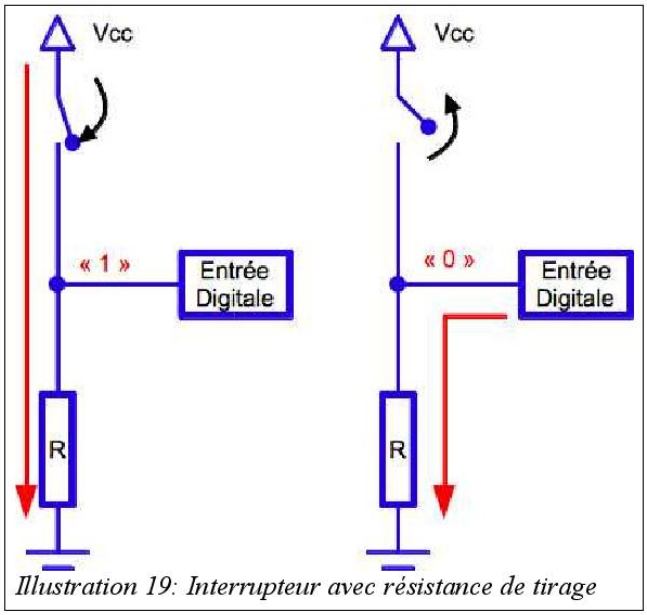


*Illustration 19: Interrupteur avec résistance de tirage*

Pour être effective, la valeur de la résistance de tirage doit être beaucoup plus grande que la résistance interne de l'alimentation, et beaucoup plus petite que la résistance interne de l'entrée digitale.

$$R_{vcc} \ll R \ll R_d$$

### 3.3 Les afficheurs LED 7 segments

L'afficheur LED 7 segments[1] est un des composants électroniques les plus simples pour représenter des chiffres de 0 à 9, et les lettres de A à F.

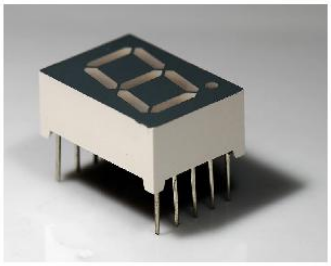


*Illustration 20: Afficheur LED 7 segments*

Il y a deux configurations possibles:

- Cathode commune (*common cathode*) : toutes les cathodes sont reliées entre elles.
- Anode commune (*common anode*) : toutes les anodes sont reliées entre elles.

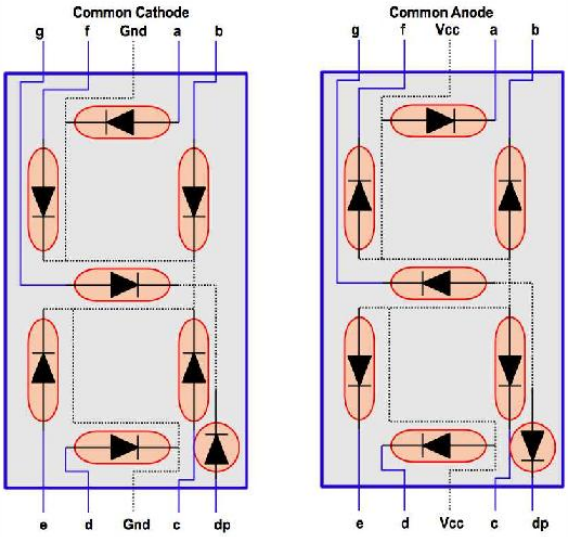



*Illustration 21: Afficheur LED 7 segments en cathode commune et anode commune*

#### 3.3.1 La résistance de protection

Les LED des afficheurs 7 segments on, comme toutes les diodes à émission lumineuse :

- Une très petite résistance interne.
- Un tension de seuil qui dépend fortement de la couleur du LED.

1 Voir également: http://fr.wikipedia.org/wiki/Afficheur_7_segments

Lorsqu'on alimente une diode, il faut faire attention à tenir le courant dans les limites de fonctionnement.

Les valeurs typiques de fonctionnement sont, pour des afficheurs de 10mm rouges :

- Courant d'alimentation typique : $I_{led} = 1mA$
- Tension seuil : $V_{led} = 1.8V$

Les circuits micro-contrôlés ont typiquement une alimentation disponible de 5V. En conséquence, la résistance de protection devrait valoir :

$$R = \frac{V_{alim} - V_{led}}{I_{led}} = \frac{5 - 1.8}{1 \cdot 10^{-3}} = 3.2k$$

### 3.3.2 Les afficheurs multiples

Pour simplifier les circuits, il existe sur le marché[1] des afficheurs LED groupés par 2, 4 et même plus chiffres.




*Illustration 22: Afficheurs 7 segments multiples*

Le circuit interne suit toujours le même principe ; les connecteurs `a`, `b`, `c`, `d`, `e`, `f`, `g` et `dp` sont connectés entre eux, par contre les anodes / cathodes sont séparées. Ceci permet d'allumer les afficheurs indépendamment les uns des autres.

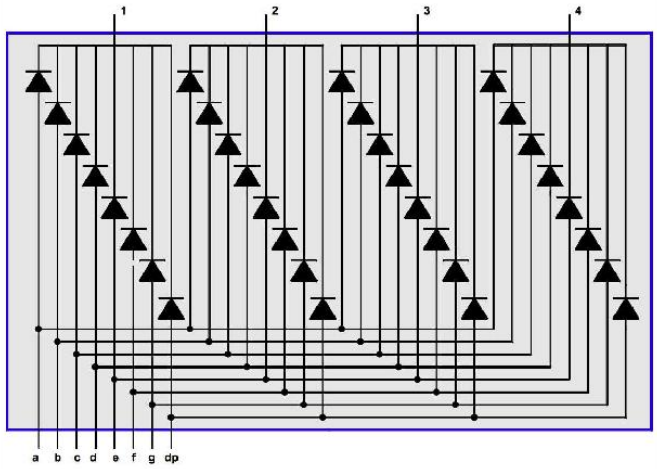



*Illustration 23: Schéma interne d'un afficheur 7 segments multiple en anode commune.*

Par contre, si plusieurs afficheurs sont allumés, ils affichent tous le même nombre. En conséquence, le circuit de contrôle est obligé d'allumer chaque afficheur à tour de rôle, à une vitesse suffisante

1 Voir le catalogue en ligne : http://www.awatronic.fr/afficheurs-multiples-7-segments-leds/cat_159.html

pour donner l'impression qu'ils sont touts allumés à la fois.

### 3.3.3 Les convertisseurs BCD à 7 segments

Un problème classique des afficheurs à 7 segments est qu'ils ne suivent pas une notation standard, et qu'ils utilisent 7 bits pour représenter les chiffres de 0 à 9, qui normalement nécessitent de seulement 4 bits.

Voici la conversion pour les chiffres 0 à 9 :

| **Valeur** | **Affichage** | ***abcdefg*** | **a** | **b** | **c** | **d** | **e** | **f** | **g** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0x3F | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0x06 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 2 | 0x5B | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 3 | 0x4F | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 4 | 0x66 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 5 | 0x6D | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 6 | 6 | 0x7D | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 7 | 0x07 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 8 | 0x7F | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 9 | 0x6F | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 10 | A | 0x77 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 11 | b | 0x7C | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 12 | C | 0x39 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 13 | d | 0x5E | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 14 | E | 0x79 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 15 | F | 0x71 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

Pour pouvoir afficher correctement des valeurs numériques sur l'afficheur 7 segments il faut réaliser la conversion. Les convertisseurs BCD à 7 segments servent justement à ça. Il s'agit de circuits intégrés qui ont 4 entrées et 7 sorties, et qui implémentent la table de vérité ci-dessus.

En général, ces circuits ne gèrent pas le point décimal.

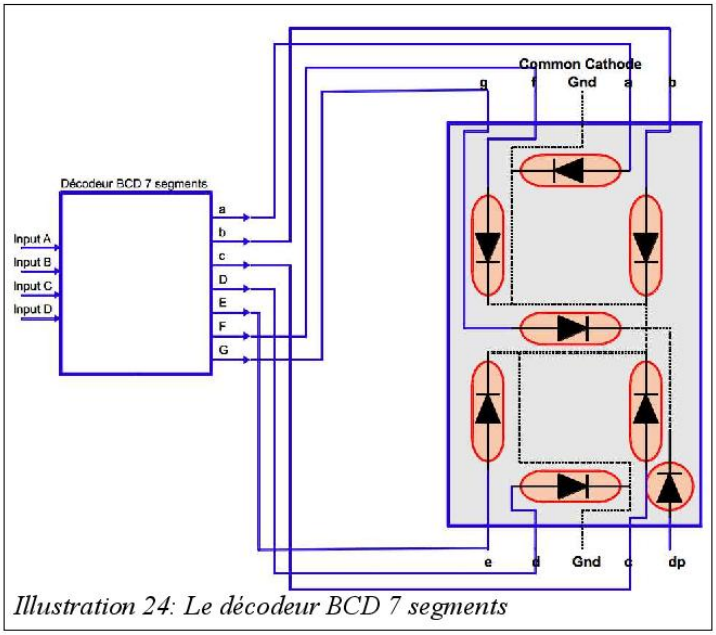



*Illustration 24: Le décodeur BCD 7 segments*

Sur un microcontrôleur, la conversion peut être réalisée plus économiquement depuis le programme.

# 4. Pratique – Utilisation des ports d'entrée / sortie

Au chapitre 2 nous avons vu les ports d'entrée / sortie numériques. Dans ce premier exercice pratique nous démontrons leur fonctionnement.

### 4.1 Le circuit

Les interrupteurs sont connectés au port 1, et un afficheur 7 segments en anode commune sont connectés sur le port 0.

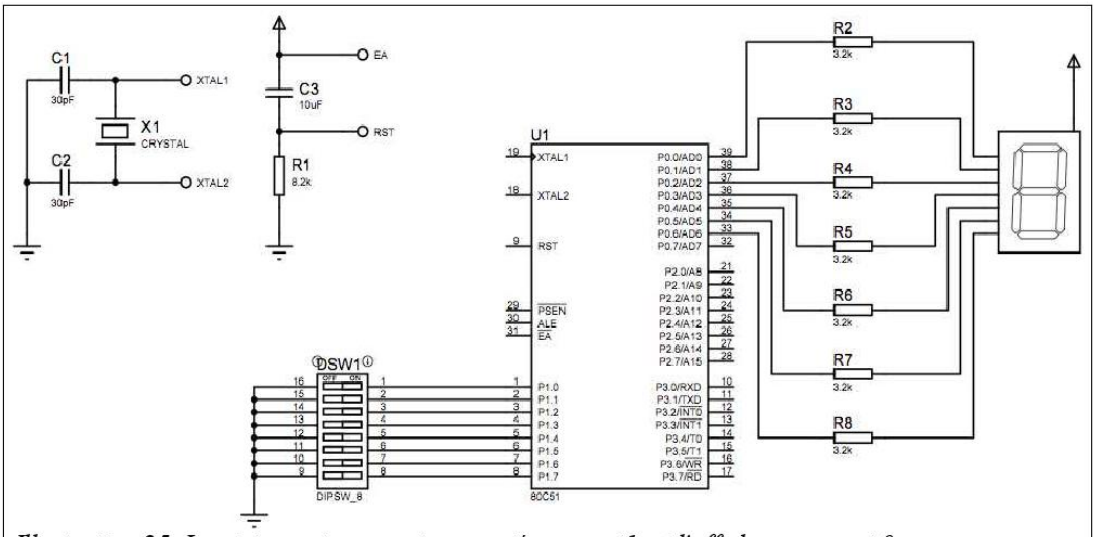



*Illustration 25: Les interrupteurs sont connectés au port1, et l'afficheur au port 0*

Voici la liste des composant à utiliser (voir 16.3)

- Microcontrôleur : 80C51
- Condensateurs : CAP
- Cristal : CRYSTAL
- Résistances : RES
- Interrupteurs : DIPSW_8
- Afficheur LED : 7SEG-COM-ANODE. Ce modèle a, dans sa configuration par défaut, un

courant nominal de 10mA. Les résistances de protection de 3.7kΩ indiquées dans le circuit sont prévues pour un courant nominal de 1mA (voir point 3.3.1). Vous devez donc changer le la propriété « *Segment On Current* » à 1mA.

### 4.2 Le programme

Le programme consiste uniquement à lire du port 1 et à écrire dans le port 0.

```
#include <reg51.h>

/**
 * Point d'entrée du programme.
 * Copie les valeurs du port 1 dans le port 0.
 */
void main(void)
{
      unsigned char s;
      while(1)
      {
            s = P1;
            P0 = s;
      }
}
```

### 4.3 Le résultat

Si tout va bien, le circuit se comporte comme si nous avions connecté les interrupteurs directement à l'afficheur 7 segments :

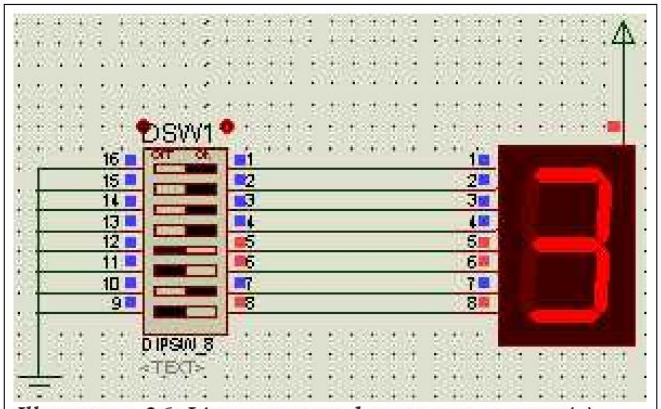



*Illustration 26: L'interrupteur directement connecté à l'afficheur*

### 4.4 Quelques questions

- Pourquoi avons-nous choisi un afficheur en anode commune ? Pourquoi l'avoir connecté au port 0 ?
- Pourquoi les interrupteurs sont directement connectés au port 1, sans résistances de tirage ?

### 4.5 Relation entre software et hardware

Une question qui peut venir à l'esprit en lisant le programme ci-dessus est la suivante : Comment est-ce possible de copier l'état de 7 interrupteurs du port 0 dans 7 LEDs du port 1, en utilisant une

seule variable `s` ?

La réponse se trouve dans la structure interne des variables utilisées. Autant `P0` et `P1` que `s`, sont des variables de type char, d'une taille de un octet. L'instruction `s = P1` copie chacun des bits de la variable `P1` dans la variable `s`.

| s= | P1.7 | P1.6 | P1.5 | P1.4 | P1.3 | P1.2 | P1.1 | P1.0 |
|---|---|---|---|---|---|---|---|---|

L'instruction `P0 = s` copie chacun des bits de la variable `s` dans la variable `P0`. Chaque bit de la variable `P1` est branché sur un des connecteurs du port 1. Chaque bit de la variable `P0` est branché sur un des connecteurs du port 0.

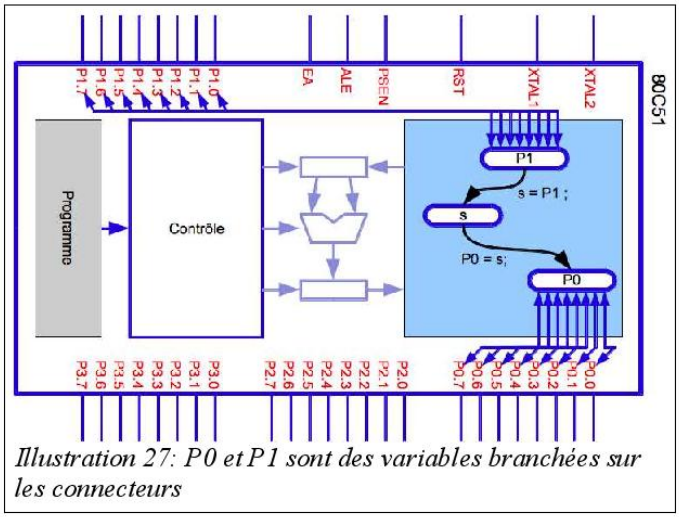



*Illustration 27: P0 et P1 sont des variables branchées sur les connecteurs*

Ce que fait le programme est uniquement de copier la valeur du port 1 dans le port 0, en passant par la variable `s`. Un programme plus évolué pourrait faire quelque chose de plus sophistiqué avec la valeur une fois qu'elle se trouve dans `s`.

# 5. 8051 – Le temps réel

## 5.1 8051 – Les temporisateurs – *Timers*

Le microcontrôleur 80C51 disposent de deux[1] compteurs / temporisateurs de 16 bits (couramment appelés *Timers*, en anglais). Ils sont identifiés comme timer0 et timer1. Ils sont indépendants et peuvent fonctionner en deux modes :

- Mode compteur : Dans ce mode, le dispositif compte le nombre de transitions de « 1 à 0 » sur les connecteurs `T0` ou `T1`. Une interruption est signalée lorsque le compte déborde.
- Mode temporisateur : Dans ce mode, le dispositif compte les cycles d'instruction (Fosc/12, voir point 2.5). Une interruption est signalée lorsque le compte déborde.

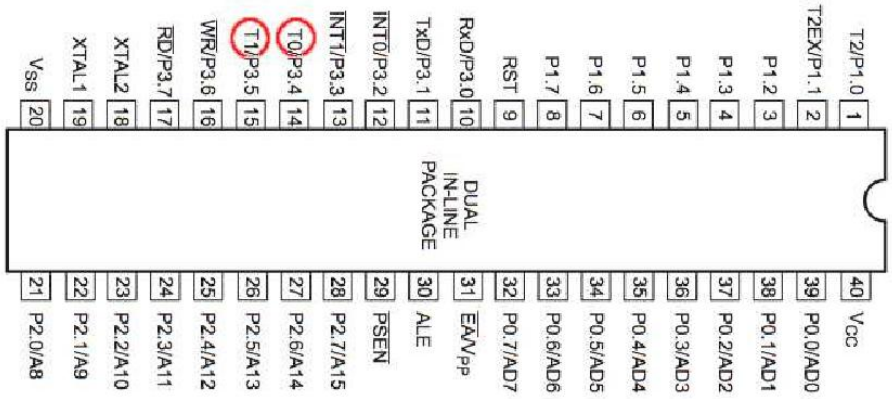



*Illustration 28: Les temporisateurs peuvent compter les transitions sur T0 et T1.*

### 5.1.1 Fonctionnement des temporisateurs

Les temporisateurs fonctionnent de la façon suivante :

- L'application charge une valeur dans les registres TH1-TL1 (timer1) ou TH0-TL0 (timer0)
- Le dispositif incrémente la valeur des registres THx-TLx :
  - En mode temporisateur, l'incrémentation est faite à chaque cycle d'horloge (voir point 2.5)

1 Suivant les modèles de 8051, il peut en exister un troisième, le timer2. Nous ne l'avons pas documenté dans ce texte.

- En mode compteur, l'incrémentation est faite à chaque transition de « 1 à 0 » sur le connecteur `T0` ou `T1`.

- Lorsque le compteur déborde (c'est à dire que THx-TLx passe de 0xFFFF à 0x0000), le bit TF1 (timer1) ou TF0 (timer0) est mis à 1, ce qui provoque une interruption si celle-ci est activée (voir point 5.2.2).
- Dans la routine d'interruption, l'application recharge la valeur appropriée dans THx-TLx.

En mode 2, uniquement TL1 est utilisé pour le décompte. Lorsqu'il déborde, TL1 est automatiquement rechargé avec la valeur de TH1.

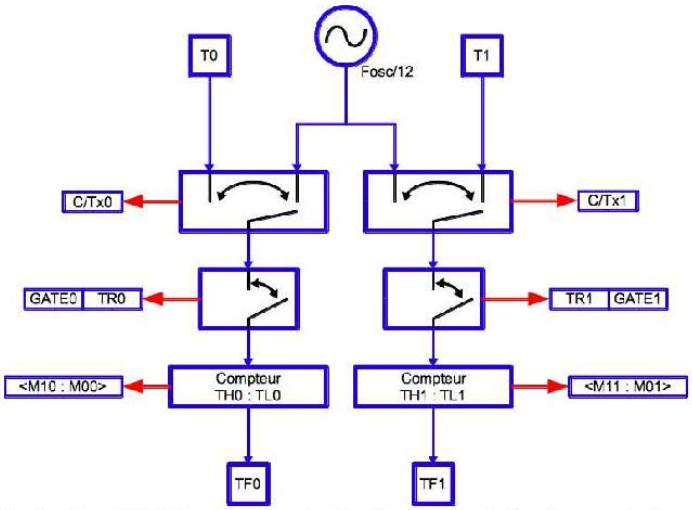



*Illustration 29: Diagramme de fonctionnement des temporisateurs*

### 5.1.2 Configuration des temporisateurs

Les temporisateurs sont configurés au moyen des registres TCON et TMOD :

| TCON ([A80C51], page 86) | | | | | | | |
|---|---|---|---|---|---|---|---|
| **TF1** | **TR1** | **TF0** | **TR0** | **IE1** | **IT1** | **IE0** | **IT0** |
| Bit 7 | | | | | | | Bit 0 |

| | |
|---|---|
| **TF1** | 1 – Indique que le timer1 est arrivé au terme de son compte.<br>Établi à « 1 » par hardware lorsque le compteur du timer1 arrive à 0. Établi à « 0 » par hardware lorsque l'interruption est gérée. |
| **TR1** | 1 – Le timer1 est actif.<br>0 – Le timer1 est inactif. |
| **TF0** | 1 – Indique que le timer0 est arrivé au terme de son compte.<br>Établi à « 1 » par hardware lorsque le compteur du timer1 arrive à 0. Établi à « 0 » par hardware lorsque l'interruption est gérée. |
| **TR0** | 1 – Le timer0 est actif.<br>0 – Le timer0 est inactif. |
| **IE1** | 1 – Une condition d'interruption sur le connecteur $\overline{\texttt{INT1}}$ a été détectée.<br>Remis à zéro automatiquement lors du traitement de l'interruption. |
| **IT1** | 1 – Une transition de « 1 à 0 » sur $\overline{\texttt{INT1}}$ provoque une interruption.<br>0 – Un niveau « 0 » sur $\overline{\texttt{INT1}}$ provoque une interruption. |
| **IE0** | 1 – Une condition d'interruption sur le connecteur $\overline{\texttt{INT1}}$ a été détectée.<br>Remis à zéro automatiquement lors du traitement de l'interruption. |
| **IT0** | 1 – Une transition de « 1 à 0 » sur $\overline{\texttt{INT0}}$ provoque une interruption.<br>0 – Un niveau « 0 » sur $\overline{\texttt{INT0}}$ provoque une interruption. |

| TMOD ([A80C51], page 87) | | | | | | | |
|---|---|---|---|---|---|---|---|
| **GATE1** | **C/T1#** | **M11** | **M01** | **GATE0** | **C/T0#** | **M10** | **M00** |
| Bit 7 | | | | | | | Bit 0 |

| | |
|---|---|
| **GATE1** | 1 – Le timer1 est actif même si TCON.TR1 est à « 0 ». Timer1 peut être utilisé par d'autres périphériques sans affecter le microcontrôleur.<br>0 – Le timer1 n'est actif que si TCON.TR1 est à « 1 ». |
| **C/T1#** | 1 – Mode compteur.<br>0 – Mode temporisateur. |
| **M11**<br>**M01** | <M11 : M01> – Mode de fonctionnement pour le timer1 :<br>00 – Mode 0 – Compteur / Temporisateur de 8 bits (TH1) avec diviseur de 5 bits (TL1)<br>01 – Mode 1 – Compteur / Temporisateur de 16 bits (TH1 et TL1)<br>10 – Mode 2 – Compteur / Temporisateur de 8 bits (TL1), se recharge avec TH1 lorsqu'il arrive à 0.<br>11 – Mode 3 – Compteur / Temporisateur en pause. |
| **GATE0** | 1 – Le timer0 est actif même si TCON.TR0 est à « 0 ».<br>0 – Le timer0 n'est actif que si TCON.TR0 est à « 1 ». |
| **C/T0#** | 1 – Mode compteur.<br>0 – Mode temporisateur. |
| **M10**<br>**M00** | <M10 : M00> – Mode de fonctionnement pour le timer0 :<br>00 – Mode 0 – Compteur / Temporisateur de 8 bits (TH0) avec diviseur de 5 bits (TL0)<br>01 – Mode 1 – Compteur / Temporisateur de 16 bits (TH0 et TL0)<br>10 – Mode 2 – Compteur / Temporisateur de 8 bits (TL0), se recharge avec TH0 lorsqu'il arrive à 0.<br>11 – Mode 3 – Compteur / Temporisateur de 8 bits (TH0) sans diviseur. |

### 5.1.3 Calcul du temps

Les temporisateurs comptent sur une base de Fosc/12, mais chaque mode est légèrement différent dans le comportement.

Comme les compteurs sont incrémentaux, le nombre de cycles à compter se fait depuis le

complément à deux de THx-TLx. Dans les exemples ci-dessous, nous indiquons le complément à 2 comme $XX^{ca2}$. Pour avoir la vraie valeur à déposer dans le registre :

$$XX = 0\text{xFF} - XX^{ca2}$$

**Exemple pour le mode 0**

Dans le mode 0, le temporisateur compte sur THx, mais utilise les 5 premiers bits de TLx pour diviser la fréquence de base Fosc/12. La période est donc :

$$T = \frac{12 \cdot TLx}{Fosc} \cdot (TH^{ca2})$$

Par exemple, pour compter 3ms avec un microcontrôleur cadencé à 12MHz :

$$T = 3 \cdot 10^{-3} = \frac{12 \cdot TLx}{12 \cdot 10^{6}} \cdot THx^{ca2} \rightarrow TLx \cdot THx^{ca2} = 3 \cdot 10^{3}$$

Il y a plusieurs solutions, puisqu'il y a plusieurs valeurs du diviseur de fréquence qui amènent à des valeurs possibles sur le compteur. Admettons que nous voulons avoir une certaine marge sur le compteur : $THx^{ca2} = 0x80$, dans ce cas TLx = 23.

**Exemple pour le mode 1**

Dans le mode 1, le compteur se fait sur les 16 bits de la paire THx et TLx. La fréquence de base est toujours Fosc/12. La période est donc :

$$T = \frac{12 \cdot [THx : TLx]^{cad2}}{Fosc}$$

Par exemple, pour compter 20ms avec un microcontrôleur cadencé à 12MHz :

$$T = 20 \cdot 10^{-3} = \frac{12 \cdot [THx : TLx]^{cad2}}{12 \cdot 10^{6}} \rightarrow [THx : TLx]^{cad2} = 20 \cdot 10^{3}$$

Au final :

- [THx : TLx] = 0x10000 – 20000 = 0xB1E0
- THx = 0xB1
- TLx = 0xE0

**Exemple pour le mode 2**

Le mode 2 se calcule comme le mode 1 mais il n'utilise que TLx comme compteur. Avec seulement 8 bits, il ne peut pas mesurer des laps de temps aussi grands que le mode 1.

Par exemple, sur un microcontrôleur cadencé à 12MHz, le temps maximum mesurable en mode 2 est :

$$T_{max} = \frac{12 \cdot 0\text{xFF}}{Fosc} = \frac{12 \cdot 0\text{xFF}}{12 \cdot 10^{6}} = 256uS$$

## 5.2 Les interruptions

> *En informatique[1], une interruption est un arrêt temporaire de l'exécution normale d'un programme par le microprocesseur afin d'exécuter un autre programme (appelé routine d'interruption).*

1 Définition tirée de Wikipédia, voir: http://fr.wikipedia.org/wiki/Interruption_(informatique)

*Dans son acception la plus stricte, le terme ne désigne que des interruptions dont l'exécution est provoquée par des causes externes au programme: avancement d'une horloge, signalisation de la complétion d'un transfert de données, [un signal externe], etc.*

La routine d'interruption est en charge d'effectuer les traitements pertinents. La nature de ceux-ci dépend de chaque application. Voici quelques exemples :

- Interruptions régulières provoquées par l'avancement d'une horloge :
  - Rafraichir un écran.
  - Incrémenter des compteurs d'heures, minutes, secondes.
  - Surveiller l'état d'un senseur (température, pression, présence...)
  - Etc.
- Interruptions faisant partie d'un processus, comme la complétion d'un transfert de données :
  - Mettre le dispositif de transmission en mode lecture, pour recevoir une réponse.
  - Placer plus de données dans la mémoire tampon.
  - Etc.
- Interruptions ponctuelles provoquées par un signal externe :
  - Afficher ou envoyer un message d'alarme.
  - Allumer / éteindre un dispositif (ventilateur, évacuation d'eau...)
  - Etc.

### 5.2.1 8051 – Les interruptions sur le 8051

Le 8051 a 5 sources[1] possibles d'interruptions :

- Les drapeaux TF1 et TF0, qui s'allument lorsque les temporisateurs 0 et 1 débordent (voir 5.1)
- Les drapeaux IE0 et IE1, qui s'allument selon les transitions détectées sur les connecteurs INT0 et INT1.
- Les drapeaux RI et TI, qui s'allument lorsqu'un caractère a été transmis par la UART (voir chapitre 7)

Pour que l'interruption se déclenche il faut trois conditions :

- Que le drapeau correspondant s'allume (transition de « 0 » à « 1 »).
- Que les interruptions en général soient activées (bit EA doit être à « 1 »).
- Que l'interruption correspondante au drapeau soit activée.

1 Nous avons omis de citer les sources d'interruption du timer2 (voir [A80C51], page 110)

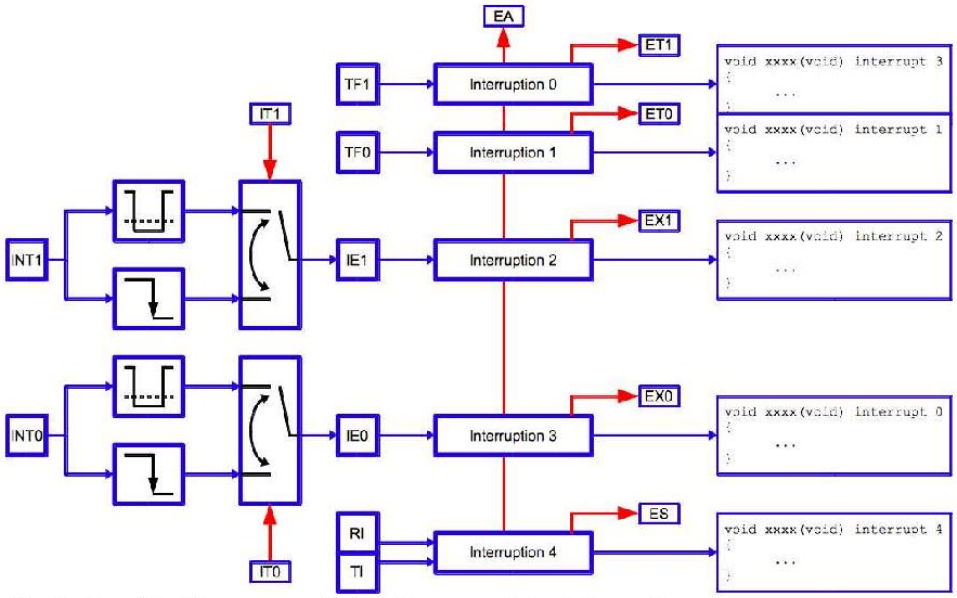



*Illustration 30: Diagramme de fonctionnement des interruptions*

Pour pouvoir prendre en charge le traitement, la routine d'interruption doit avoir un moyen de savoir quelle a été la source de celle-ci. Sur le 8051, chaque source d'interruption est branchée sur une adresse différente. En conséquence il y a une routine spécifique à chaque source d'interruption :

| Source | Adresse | Numéro | Cause |
|---|---|---|---|
| Connecteur $\overline{\texttt{INT0}}$ | 0x0003 | 0 | Une transition sur $\overline{\texttt{INT0}}$ est détectée. |
| timer0 | 0x000B | 1 | Le compteur / temporisateur 0 est arrivé à 0. |
| Connecteur $\overline{\texttt{INT1}}$ | 0x0013 | 2 | Une transition sur $\overline{\texttt{INT1}}$ est détectée. |
| timer1 | 0x001B | 3 | Le compteur / temporisateur 1 est arrivé à 0. |
| Port série | 0x0023 | 4 | Port série – UART |

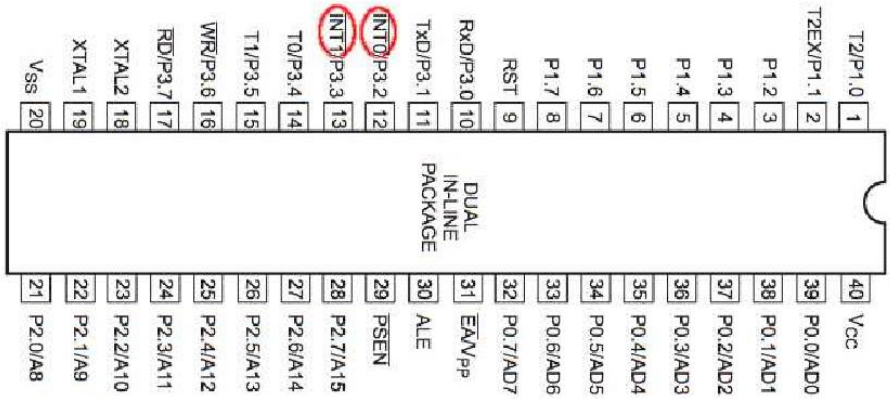



*Illustration 31: Les connecteurs INT1 et INT0 peuvent déclencher des interruptions*

### 5.2.2 Configuration des interruptions

Les interruptions sont activées ou désactivées au moyen du registre IE :

| IE | | | | | | | |
|---|---|---|---|---|---|---|---|
| **EA** | **n/a** | **ET2** | **ES** | **ET1** | **EX1** | **ET0** | **EX0** |
| Bit 7 | | | | | | | Bit 0 |

| | |
|---|---|
| **EA** | 1 – Les interruptions sont activées<br>0 – Toutes les interruptions sont désactivées. |
| **ET2** | 1 – Les interruptions du timer2[1] sont activées.<br>0 – Les interruptions du timer2[1] sont désactivées. |
| **ES** | 1 – Les interruptions de la UART sont activées.<br>0 – Les interruptions de la UART sont désactivées. |
| **ET1** | 1 – Les interruptions du timer1 sont activées.<br>0 – Les interruptions du timer1 sont désactivées. |
| **EX1** | 1 – Les interruptions externes sur le connecteur $\overline{\texttt{INT1}}$ sont activées.<br>0 – Les interruptions externes sur le connecteur $\overline{\texttt{INT1}}$ sont désactivées. |
| **ET0** | 1 – Les interruptions du timer0 sont activées.<br>0 – Les interruptions du timer0 sont désactivées. |
| **EX0** | 1 – Les interruptions externes sur le connecteur $\overline{\texttt{INT0}}$ sont activées.<br>0 – Les interruptions externes sur le connecteur $\overline{\texttt{INT0}}$ sont désactivées. |
| **(1)** | Comme indiqué ailleurs, nous n'avons pas documenté le timer2 dans ce texte. |

Le 8051 supporte deux priorités pour les interruptions : haute et basse. Lorsque deux interruptions sont lancée en même temps, celle de haute priorité est traitée en premier.

| IP | | | | | | | |
|---|---|---|---|---|---|---|---|
| **n/a** | **n/a** | **PT2** | **PS** | **PT1** | **PX1** | **PT0** | **PX0** |
| Bit 7 | | | | | | | Bit 0 |

| | |
|---|---|
| **PT2** | 1 – Les interruptions du timer2 sont de haute priorité.<br>0 – Les interruptions du timer2 sont de haute priorité. |
| **PS** | 1 – Les interruptions de la UART sont de haute priorité.<br>0 – Les interruptions de la UART sont de haute priorité. |
| **PT1** | 1 – Les interruptions du timer1 sont de haute priorité.<br>0 – Les interruptions du timer1 sont de haute priorité. |
| **PX1** | 1 – Les interruptions du connecteur $\overline{\texttt{INT1}}$ sont de haute priorité.<br>0 – Les interruptions du connecteur $\overline{\texttt{INT0}}$ sont de haute priorité. |
| **PT0** | 1 – Les interruptions du timer0 sont de haute priorité.<br>0 – Les interruptions du timer0 sont de haute priorité. |
| **PX0** | 1 – Les interruptions du connecteur $\overline{\texttt{INT0}}$ sont de haute priorité.<br>0 – Les interruptions du connecteur $\overline{\texttt{INT0}}$ sont de haute priorité. |

### 5.2.3 C51 – Mise en place des interruptions depuis *Keil uVision* C

Pour déclarer une routine d'interruption, le compilateur C51 de *Keil uVision* utilise la directive `interrupt` :

```
void maRoutinePourLeTimer0(void) interrupt 1
{
      // Ici le traitement
```

```
        // de l'interruption pour
        // le timer0.
}
```

EA se désactive automatiquement lorsqu'une interruption est gérée. Il se réactive automatiquement lorsque la routine d'interruption est finie.

# 6. Pratique – Temporisateurs et interruptions

Cet exercice pratique démontre de manière très basique le fonctionnement des temporisateur et des interruptions.

## 6.1 Le circuit

Un interrupteur est connecté sur l'entrée $\overline{\texttt{INT1}}$, et un afficheur 7 segment en anode commune sur le port 0.

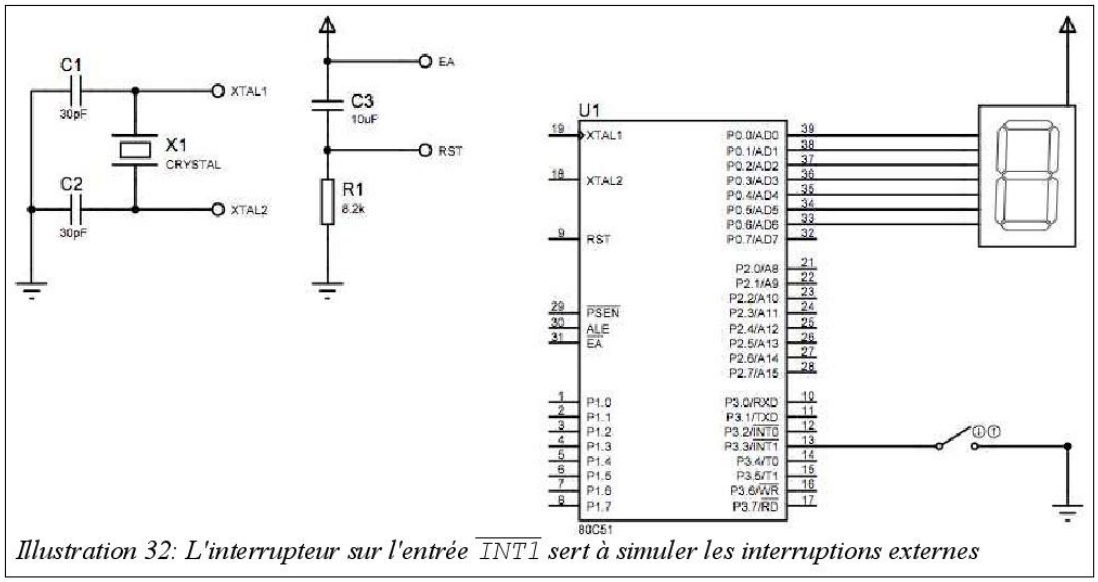



*Illustration 32: L'interrupteur sur l'entrée $\overline{\texttt{INT1}}$ sert à simuler les interruptions externes*

## 6.2 Le programme

Le programme est divisé en trois parties. Le point d'entrée (fonction `main`) configure le timer0 en mode 1 et les interruptions pour le timer0 et $\overline{\texttt{INT1}}$. Ensuite il attend sans rien faire d'autre (boucle `while`). L'action proprement dite se passe au niveau des interruptions.

```
/**
 * Point d'entrée.
 * Configure les temporisateurs et les interruptions,
```

```
 * puis ne fait plus qu'attendre.
 */
void main(void)
{
      // Temporisateur 0 comme temporisateur 16 bits et active
      // la detection sur INT1:
      TCON = 0x14;

      // Temporisateur 0 en mode 1:
      TMOD = 0x01;

      // Active les interruptions pour le temporisateur 0 et
      // la ligne externe INT1
      IE = 0x86;

      TL0 = 0x00;
      TH0 = 0x00;

      // Dodo:
      while(1);
}
```

La routine pour l'interruption du timer0 (interruption n° 1, voir point 5.2.1) incrémente un compteur à chaque fois. Puis, avant de terminer, réinitialise le timer0 (comme il est en mode 1, il ne se recharge pas automatiquement).

```
// Variables globales, pour maintenir le contexte
// d'une interruption à l'autre
unsigned char compteur = 0;

/**
 * Interruptions pour le temporisateur 0.
 */
void temporisateur0(void) interrupt 1
{
      // Actualise le compteur et l'affichage:
      if (compteur != 0)
      {
            compteur+=1;
      }
      P0 = compteur;

      // Réinitialise le temporisateur:
      TL0 = 0x00;
      TH0 = 0x00;
}
```

La routine pour l'interruption externe met le compteur à 0. Ceci a pour effet, compte tenu de la logique pour l'interruption du timer0, d'arrêter le compteur.

```
/**
 * Interruption pour la ligne externe 1.
 */
void ligneExterne1(void) interrupt 2
{
      if (compteur == 0)
      {
```

```
            compteur = 1;
        }
        else
        {
            compteur = 0;
        }
    }
```

Le compteur doit être déclaré comme variable globale (voir point 8.4), puisque sa valeur doit être conservée d'un appel à l'autre des routines des interruptions.

### 6.3 Le résultat

Si tout va bien, le comptage se manifeste sur l'afficheur 7 segments (sans toute fois montrer des chiffres justes). Le comptage peut s'activer et se désactiver en ouvrant et fermant l'interrupteur.

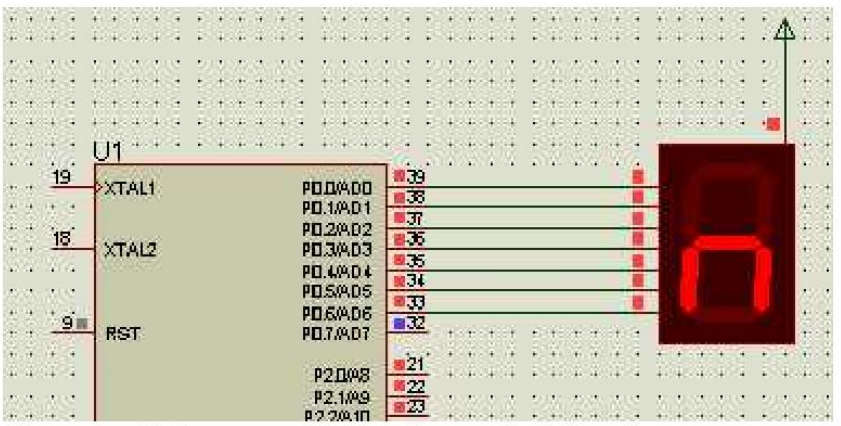



*Illustration 33: Le comptage se manifeste sur l'afficheur 7 segments*

### 6.4 Quelques questions

- De quelle fréquence est l'interruption configurée avec TL0 = 0x00 et TH0 = 0x00, si le microcontrôleur est cadencé à 12MHz ?
- Pourquoi l'interrupteur n'a d'effet que lorsqu'il se ferme ?
- La logique du programme est écartelée sur les deux fonctions de gestion d'interruptions. Si le comportement du programme était plus complexe que cet exemple, ceci pourrait poser un problème de lecture (et donc de maintenance à long terme). Comment le résoudre ?

# 7. 8051 – La UART

## 7.1 La communication série

Un[1] « récepteur / émetteur asynchrone universel » (en anglais « *universal asynchronous receiver / transmitter* », abrévié habituellement comme UART), est un composant électronique qui traduit les données entre les formes parallèle et série. Les UART sont habituellement utilisées avec des protocoles de communication tels que EIA, RS-232, RS-422 et RS-485. La désignation « universelle » vient du fait que le format de données et la vitesse de transmission sont configurables. Les niveaux électriques sont gérés, si nécessaire, avec un composant externe supplémentaire (par exemple, un modem).

Une UART est normalement un circuit intégré, ou fait partie d'un circuit intégré, utilisée pour gérer la communication série avec des ordinateurs ou des périphériques distants. Les UART sont très communes dans les microcontrôleurs.

La UART émet des données via le connecteur TXD, et reçoit des données via le connecteur RXD. Lorsque deux UART sont connectées directement, le connecteur TXD de l'une est branché sur le connecteur RXD de l'autre.

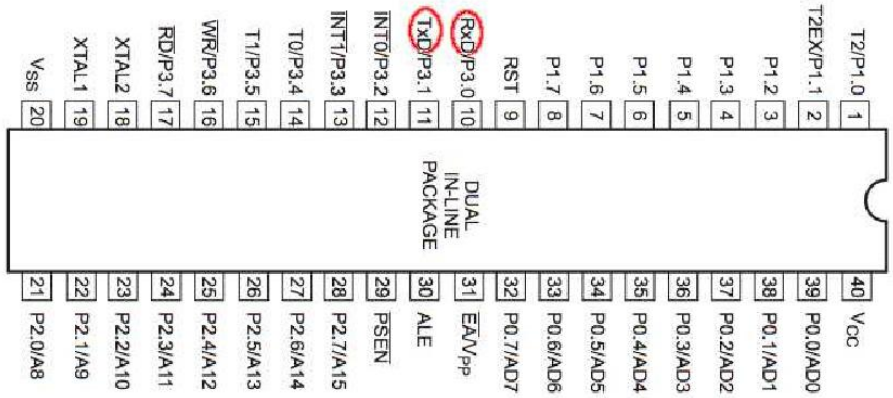



*Illustration 34: La UART est branchée sur les connecteurs TXD et RXD*

1 Extrait de Wikipédia. Voir http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

## 7.2 La UART du 8051

Le 8051 dispose d'une UART relativement flexible. Nous allons décrire ici que quelques éléments de base pour pouvoir l'utiliser comme sortie standard dans un laboratoire (voir point 7.6).

En émission (et modes 1, 2, 3), la UART fonctionne de la façon suivante :

- Lorsqu'une valeur est écrite dans le registre SBUF, la transmission démarre au suivant battement de l'horloge utilisée pour la vitesse de transmission.
- 10 ou 11 bits sont émis à travers TXD: Le bit de Start (toujours « 0 »), les 8 bits de SBUF (bit le moins signifiant en premier), puis, éventuellement, le 9ième bit (TB8), et enfin le bit de Stop (toujours « 1 »).
- Lorsque tous les bits ont été transmis, TI est établi à « 1 ». Si les interruptions sont activées pour le port série, ceci provoque une interruption.
- Lorsque l'application détecte que TI est à « 1 », elle peut remettre une valeur dans SBUF, et continuer ainsi la transmission.

En réception (et modes 1, 2, 3), la UART fonctionne de la façon suivante :

- La réception débute lorsque une transition de « 1 à 0 » est détectée sur RXD.
- Les bits reçus au travers de RXD sont copiés un à un dans SBUF. Le 9ième bit (qui peut être le bit de Stop, ou le 9ième bit) est copié dans RB8
- Lorsque tous les bits ont été reçu, RI est établi à « 1 ». Si les interruptions sont activées pour le port série, ceci provoque une interruption.
- L'application peut alors récupérer la valeur de SBUF, et une nouvelle réception peut démarrer.

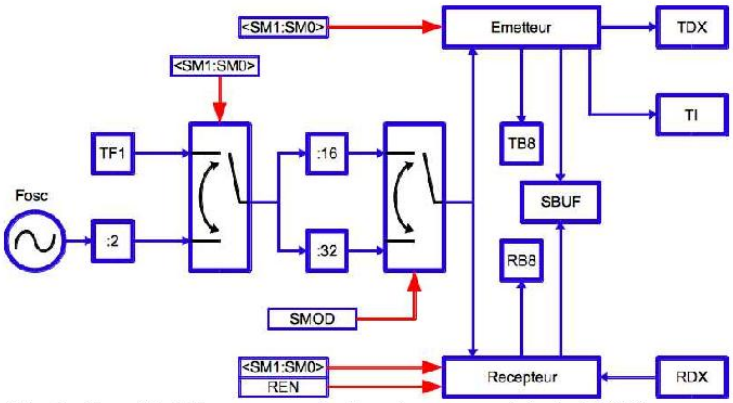



*Illustration 35: Diagramme de fonctionnement de la UART*

La cadence de transmission est déterminée soit par le timer1, soit par l'horloge principale du microcontrôleur. Des diviseurs de fréquence servent à ramener la vitesse de transmission à des valeurs standard (voir point 7.4)

## 7.3 Configuration de la UART

Le registre SCON est le registre principal de configuration du port série :

| SCON ([A80C51], page 96) | | | | | | | |
|---|---|---|---|---|---|---|---|
| **SM0** | **SM1** | **SM2** | **REN** | **TB8** | **RB8** | **TI** | **RI** |
| Bit 7 | | | | | | | Bit 0 |

| | |
|---|---|
| **SM0** | <SM0:SM1> – Mode – Description – Vitesse de transmission<br>00[1] – Mode 0 – 8 bits (SBUF). Les données sortent de RXD, tandis que TXD transmet l'horloge - Fosc/12<br>01[2] – Mode 1 – 10 bits (*Start*[3] + SBUF + *Stop*[4]) – Vitesse dépend de timer1 |
| **SM1** | 10 – Mode 2 – 11 bits (*Start*[3] + SBUF + TB8/RB8 + *Stop*[4]) – Fosc/64 ou Fosc/32<br>11 – Mode 3 – 11 bits (*Start*[3] + SBUF + TB8/RB8 + *Stop*[4]) – Vitesse dépend de timer1 |
| **SM2** | 1 – Communication multi-processeur activée.<br>0 – Communication multi-processeur désactivée. |
| **REN** | 1 – Récepteur activé.<br>0 – Récepteur désactivé. |
| **TB8** | Valeur du 9ème bit à transmettre pour les modes 2 et 3 |
| **RB8** | Valeur du 9ème bit reçu pour les modes 2 et 3 |
| **TI** | Se met à 1 lorsqu'un byte complet a été transmis. Doit être mis à 0 par la routine d'interruption. |
| **RI** | Se met à 1 lorsqu'un byte complet a été reçu. Doit être mis à 0 par la routine d'interruption. |
| **(1)** | Le mode 0 n'est pas utilisé pour transmettre des données ; la UART sert ici comme convertisseur parallèle à série. |
| **(2)** | Mode normal pour la UART |
| **(3)** | Le bit « *Start* » est « 0 » |
| **(4)** | Le bit « *Stop* » est « 1 » |

Le registre PCON contient uniquement le bit SMOD, qui détermine la valeur du diviseur de fréquence qui se trouve en entrée du générateur de vitesse de transmission.

| PCON ([A80C51], page 110) | | | | | | | |
|---|---|---|---|---|---|---|---|
| **SMOD** | **n/a** | **n/a** | **n/a** | **GF1** | **GF0** | **PD** | **IDL** |
| Bit 7 | | | | | | | Bit 0 |

| | | |
|---|---|---|
| **SMOD** | Vitesse de transmission en modes 1 et 3 :<br>1 – Débordement timer1 divisée par 16.<br>0 – Débordement timer1 divisée par 32. | Vitesse de transmission en mode 0 :<br>1 – Fosc / 32<br>0 – Fosc / 64 |
| **GF1** | Pas d'effet sur la UART | |
| **GF0** | Pas d'effet sur la UART | |
| **PD** | Pas d'effet sur la UART | |
| **IDL** | Pas d'effet sur la UART | |

### 7.4 Calcul de la vitesse de transmission

Une communication série tel que le protocole RS-232 se fait sur un seul fil. Uniquement les données sont transmise, pas l'horloge. Pour que le récepteur puisse décoder le signal de donnée, il faut qu'il soit configuré pour la même fréquence que l'émetteur. On considère qu'il ne devrait pas y avoir plus de 5% d'erreur[1].

1 Voir http://www.keil.com/products/c51/baudrate.asp

La fréquence de transmission se mesure en « bauds ». Le baud[2] est l'unité de mesure du nombre de symboles transmissibles par seconde. Dans une communication série tel que implémentée par la UART, le symbole est le bit.

Les fréquences standard pour le protocole RS-232 sont : 115.2k, 57.6k, 38.4k, 19.2k, 9600, 4800, 2400, 1200 et 300.

### 7.4.1 Fréquence de transmission basée sur le timer1

Ce calcul s'applique pour les modes 1 et 3, et il dépend également de la valeur de PCON.SMOD :

$$SMOD = 0 \rightarrow Baud = \frac{F_{osc}}{12 \cdot 32 \cdot T1}$$

$$SMOD = 1 \rightarrow Baud = \frac{F_{osc}}{12 \cdot 16 \cdot T1}$$

Où T1 est le nombre de cycles jusqu'à ce que le timer1 déborde (voir plus bas). La configuration plus naturelle pour utiliser le timer1 comme générateur de fréquence de transmission est la suivante :

- Mode 2 (le registre TL1 est utilisé pour réinitialiser TH1 chaque fois que celui-ci déborde)
- Désactiver les interruptions du timer1, puisque nous en avons pas besoin pour le réinitialiser.

Dans ces conditions, la valeur de TL1 est de 0xFF - T1 + 1.

Voici un exemple de calcul :

- Soit 9600 bauds la fréquence de transmission à obtenir [1].
- En assumant que SMOD = 0 et que la fréquence d'horloge du microcontrôleur est de 10MHz, nous devrions avoir une valeur pour le timer1 de 2.7127. Comme le timer1 ne supporte que les valeurs entières, nous prenons 3 comme résultat [2].
- Pour obtenir tout de même une fréquence de transmission exacte, nous pouvons modifier la fréquence d'horloge du microcontrôleur [3].
- Enfin, la valeur que nous devons mettre dans TH1 est 0xFD [4]

$$Baud = \frac{F_{osc}}{12 \cdot 32 \cdot TH1^{ca2}} = 9600 \qquad [1]$$

$$T1 = \frac{10 \cdot 10^6}{12 \cdot 32 \cdot 9600} = 2.7127 \approx 3 \qquad [2]$$

$$F_{osc} = Baud \cdot 12 \cdot 32 \cdot TH1^{ca2} = 11'059'200 \qquad [3]$$

$$TH1 = 0x100 - TH1^{ca2} = 0xFD \qquad [4]$$

### 7.4.2 Fréquence de transmission basée sur le cycle d'instruction

Dans ce cas la fréquence de transmission est directement Fosc/32 ou Fosc/64, selon le bit SMOD.

## 7.5 Exemple de configuration de la UART

Voici une configuration de base de la UART, pour être utilisée dans la majorité des applications:

- SCON = 0x50 (Mode 1, récepteur activé)
- PCON = 0x00 (SMOD à 0, fait partie de l'exemple de calcul ci-dessus)

2 Le terme « baud » provient du patronyme d'*Émile Baudot*, l'inventeur du code *Baudot* utilisé en télégraphie. Voir http://fr.wikipedia.org/wiki/Baud_(mesure)

- TCON = 0x40 (timer1 actif)
- TMOD = 0x20 (timer1 en mode temporisateur, mode 2)
- TH1 = 0xFD (selon exemple de calcul ci-dessus)

Avec cette configuration, et pour une cadence du microcontrôleur de 11.0592MHz, la UART est programmée pour une vitesse de transmission de 9600 bauds, avec bit de *Start* et bit de *Stop*.

## 7.6 Pratique – Utiliser la UART comme sortie standard – `printf`

L'exercice suivant met en pratique quelques uns des principes de ce chapitre. Il permet également une première prise de contact avec l'environnement de développement. Enfin, il présente une implémentation de `printf` qui est très pratique pour le débogage et les prochains développements.

### 7.6.1 Le circuit

*Proteus* contient un terminal virtuel, qui implémente le protocole RS-232, et donc pourrait être branché aux connecteurs RXD et TXD du 8051 :

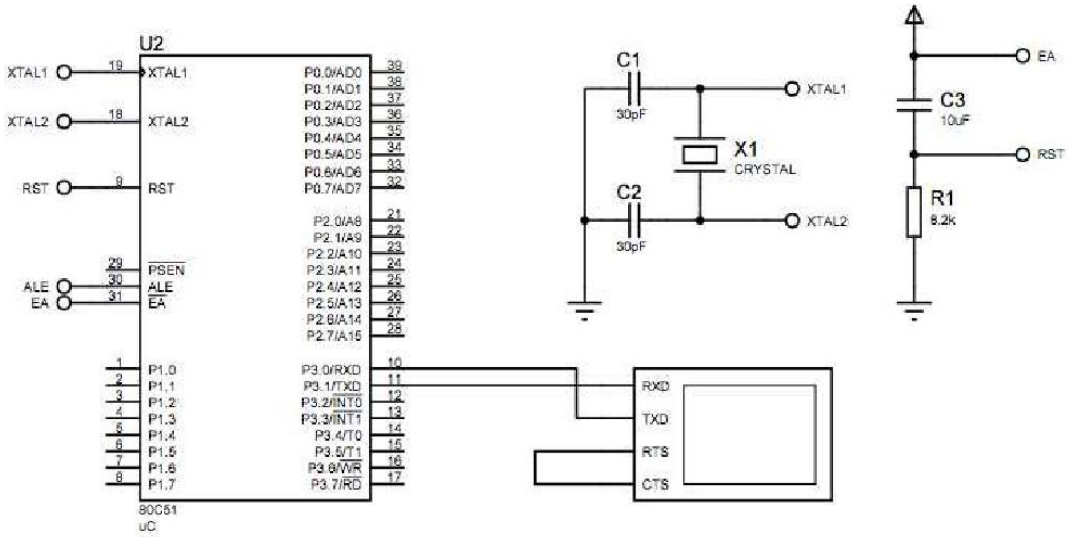



*Illustration 36: Le terminal virtuel de Proteus est compatible avec le 8051*

Vous trouvez le terminal virtuel dans « *Virtual Instruments Mode* ». Vous pouvez le placer comme n'importe quel autre composant. Attention à croiser les connexions RXD et TXD entre le microcontrôleur et le terminal.

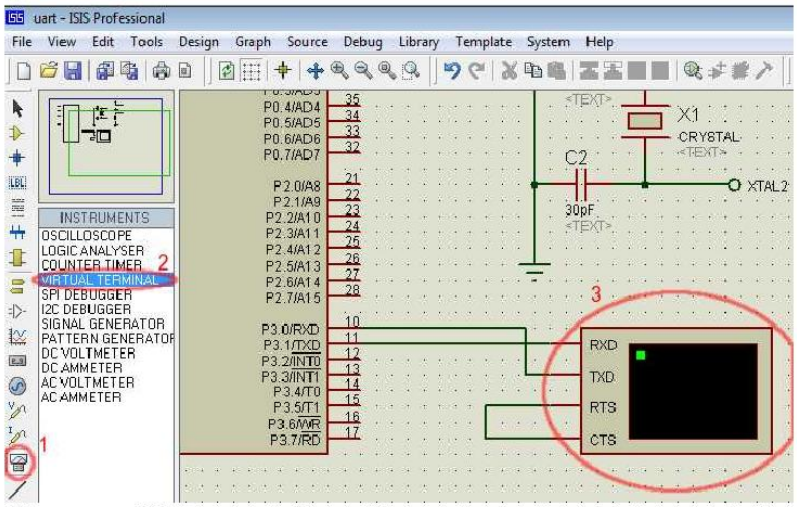


*Illustration 37: Le terminal virtuel se trouve dans une librairie séparée*

### 7.6.2 Le programme – redéfinition de `putchar`

En langage C, `stdio.h` est la librairie qui contient les fonctions de base pour communiquer avec le monde extérieur. Son nom vient de « *Standard Input Output* ». Elle contient, entre autres, la fameuse fonction `printf`, qui permet d'afficher une chaîne de caractères formatée. En interne, `printf` utilise `putch` pour envoyer chaque caractère à la sortie standard.

Un microcontrôleur n'a pas d'affichage, et donc n'a pas de sortie standard. Pour cette raison, le compilateur C51 de *Keil uVision* définit `putchar` de la manière suivante (voir aide de *Keil uVision*, à propos de `putchar`):

```
void putchar(char c)
{
      return;
}
```

Et en conséquence, les appels à `printf` n'ont aucun effet.

Un terminal branché sur les connecteurs P3.0/RXD et P3.1/TXD du 8051 pourrait devenir une sortie standard si `putchar` envoie les caractères à la UART.

Pour ceci il faut :

- Configurer correctement la UART :

```
/**
 * Initialise la UART en 9600 bauds, bit de start,
 * bit de stop, 8 bits de données.
 * Pour que la fréquence de transmission soit correcte,
 * le uC doit être cadencé à 11.0592 MHz.
 */
void UART_Initialize(void)
{
      SCON = 0x50;  // UART en Mode 1, Récepteur activé.
      PCON = 0x00;  // SMOD à 0: divise la fréquence du timer1 par 32.
      TCON = 0x40;  // Timer1 actif.
      TMOD = 0x20;  // Timer1 comme temporisateur, mode 2.
      TH1 = 0xFD;   // Pour obtenir 9600 baud à 10.0592MHz
}
```

- Redéfinir `putchar` :

```
/**
 * Remplace la fonction standard du même nom,
 * pour la rendre compatbile avec la UART.
 * Cette fonction est appelée par printf.
 * Avant d'utiliser cette fonction, il faut initialiser
 * la UART avec UART_Initialize.
 * @param c Le caractère à afficher.
 * @return Le caractère affiché.
 */
char putchar(char c)
{
      SBUF = c;
      TI = 0;
      while(!TI);
      return c;
}
```

Enfin, pour tester la solution, nous implémentons le point d'entrée du programme comme suit :

```
/**
 * Point d'entrée du programme.
 * Affiche "Hello world" un certain nombre de fois.
 */
void main(void)
{
      int n;

      UART_Initialize();

      for(n=0;n<10;n++)
      {
            printf("Hello world %d\r\n",n);
      }

      while(1);
}
```

### 7.6.3 Le résultat

Une fois le programme compilé (sans erreurs ni warnings), vous pouvez le charger dans *Proteus*. Pour que tout fonctionne correctement, vous devez également :

- Configurer le 8051 pour être cadencé à 11.0592MHz
- Configurer le terminal virtuel pour 9600 bauds, 8 bits, 1 bit de stop.

Pour accélérer la simulation, vous pouvez exclure de la simulation tous les condensateurs, résistances et le cristal de la configuration technique.

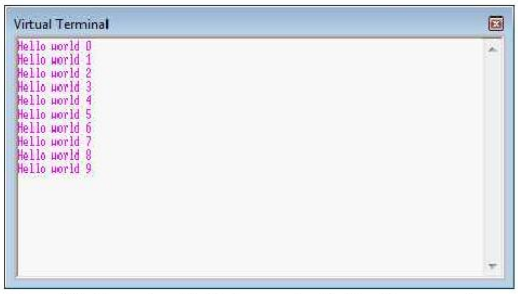



*Illustration 38: Le résultat de `printf` est envoyé au terminal*

### 7.6.4 Quelques questions

- Quel taille fait le programme une fois compilé ? Pourquoi est-il si grand ?
- Que se passe-t'il si la vitesse de transmission n'est pas configurée pareil sur le 8051 et sur le terminal virtuel ? Pourquoi ?
- Comment fonctionne `putchar` ? Pourquoi est-elle appelée, si depuis la fonction main il n'y a pas d'appel ?

# 8. Le C en 5 minutes

Ce chapitre n'est qu'un aide mémoire pour les éléments de langage C qui sont utilisés dans le reste du document. Vous y trouverez quelques mots clef qui peuvent vous aider à rechercher de la documentation sur internet, plus quelques mentions aux concepts de base.

Pour une vraie documentation, consultez [KERC] ou [PRIMERC].

## 8.1 Programmation C – `int`, `char` et le modificateur `unsigned`

Les deux types de variables les plus utilisés dans les systèmes simples comme ceux basés sur le 8051 sont `int` et `char` :

- `char`: une variable contenue dans un seul octet. Elle peut prendre des valeurs entre -128 et +127.
- `int` : une variable contenue dans deux octets. Elle peut prendre des valeurs entre -32768 et +32767.

Le modificateur `unsigned` indique que la variable ne contient que des nombres positifs :

- `unsigned char` : comme `char`, mais elle ne peut prendre que des valeurs positives entre 0 et 255.
- `unsigned int` : comme `int`, mais elle ne peut prendre que des valeurs positives entre 0 et 65535.

`char` tient son nom du fait qu'on l'utilise souvent pour représenter des caractères par leur code *ascii*. Le code *ascii* de base va de 0 à 127. Par contre le code *ascii* étendu, tel qu'utilisé en Europe, va de 0 à 255. Il faut utiliser un `unsigned char` pour le code *ascii* étendu.

`int` tient son nom du fait qu'il est souvent utilisé pour représenter des nombres entiers (« *integer* », en anglais). Sur des systèmes limités en mémoire, utiliser deux bytes est souvent un luxe qu'on ne peut pas se permettre. `char` est également capable de représenter des nombres entiers. Chaque fois que vous avez besoin d'une variable qui ne dépasse pas les limites, préférez `char` à la place de `int`.

## 8.2 Programmation C – Opérations booléennes

Les opérateurs booléens permettent de réaliser des opérations binaires sur les variables. Voici quelques exemples :

```
unsigned char c1, c2;
unsigned char r;
c1 = 0xF0;
```

```
c2 = 0x0F;
r = c1 | c2;  // r = 0xFF: OR bit à bit de c1 et de c2.
r = c1 & c2;  // r = 0x00: AND bit à bit de c1 et de c2.
r = c1 ^ c2;  // r = 0x00: OR Exclusif, bit à bit de c1 et de c2.
r = ~c1;      // r = 0x0F: NOT bit à bit de c1.
```

Les opérateurs >> et << réalisent un déplacement des bits de la variable affectée. Voici quelques exemples :

```
unsigned char c1;
c1 = 2;      // En binaire, c1 = '00000010'
c2 = c1 << 2 // En binaire, c2 = '00001000'
c2 = c1 >> 1 // En binaire, c2 = '00000100'
c2 >>=1;    //  En binaire, c2 = '00000010'
c2 <<=4;     // En binaire, c2 = '00100000'
```

### 8.3 Programmation C – Précédence des opérateurs

En mathématiques, les opérateurs de multiplication et division sont plus prioritaires que l'addition et la soustraction :

```
3+4x4+1 = 20
```

En programmation C, les différents opérateurs ont également des priorités différentes :

| Opérateur | Description | Associativité |
|---|---|---|
| `( )`<br>`[ ]`<br>`.`<br>`->`<br>`++`<br>`--` | Appel de fonction<br>Index de tableau<br>Membre de structure<br>Membre de structure (via pointeur)<br>Incrémentation (suffixe)<br>Décrémentation (suffixe) | Gauche à droite |
| `++ --`<br>`+ -`<br>`!`<br>`~`<br>`(type)`<br>`*`<br>`&`<br>`sizeof` | Incrémentation / Décrémentation (préfixe)<br>Plus et Moins unaires, comme dans (-5) ou (+2)<br>Négation logique<br>Négation bit à bit.<br>Changement de type<br>Pointeur<br>Adresse<br>Taille d'une variable | Droite à gauche |
| `* / %` | Multiplication, Division et Modulo | Gauche à droite |
| `+ -` | Addition et Soustraction, comme dans a+b ou a-b | Gauche à droite |
| `<< >>` | Déplacement bit à bit. | Gauche à droite |
| `< <=`<br>`> >=` | Comparaison : plus petit, plus petit ou égal<br>Comparaison : plus grand, plus grand ou égal, | Gauche à droite |
| `== !=` | Comparaison : égal, différent | Gauche à droite |

| Opérateur | Description | Associativité |
|---|---|---|
| & | ET bit à bit | Gauche à droite |
| ^ | OU Exclusif bit à bit | Gauche à droite |
| \| | OU bit à bit | Gauche à droite |
| && | ET logique | Gauche à droite |
| \|\| | OU logique | Gauche à droite |
| ?: | Condition ternaire, comme dans a<b ? a++ : a-- ; | Droite à gauche |
| =<br>+= -=<br>*= /= %=<br>&= ^= \|=<br><<= >>= | Assignation<br>Assignation avec incrémentation ou soustraction<br>Assignation avec multiplication, division, modulo<br>Assignation avec opération bit à bit<br>Assignation avec déplacement bit à bit | Droite à gauche |
| , | Séparateur d'expressions | Gauche à droite |

Les suffixes ++ et - - ont une très haute précédence. Par contre l'exécution de l'incrémentation ou décrémentation peut être laissée à plus tard. Dans une expression comme :

```
y = x * z++;
```

Il n'est pas clair si z va être incrémentée avant ou après l'assignation. La moralité est qu'il vaut mieux ne pas écrire des expression qui dépendent trop de l'ordre de précédence. En cas de doute, simplifiez les expressions et utilisez des parenthèses.

```
z++;
y = x * z;
```

Les parenthèses utilisées pour regrouper des expressions sont évaluées depuis la plus intérieure à la plus extérieure.

### 8.4 Programmation C – Variables locales et variables globales

Les variables locales sont celles déclarées à l'intérieur d'une fonction. Différentes fonctions peuvent utiliser des variables locales du même nom. Même si elles portent le même nom, ce ne sont pas les mêmes variables.

```
void main()
{
	int c,d;
	int a;

	c = 5;
```

```
        d = 8;
        a = addition(c,d);

        // Ici 'c' continue de valoir '5' et 'a' vaut '13'.
}
int addition(int a,int b)
{
        // Ici, 'a' vaut 5, et 'b' vaut 8.
        int c;          // Ici, 'c' ne vaut pas 5.
        c = a + b;
        return c;       // Ici 'c' vaut 13.
}
```

Les variables déclarées comme arguments d'une fonction (ici les arguments 'a' et 'b' de `addition`) sont également des variables locales.

### 8.5 Programmation C – Pointeurs et tableaux

En C, un pointeur est une variable qui contient une adresse de mémoire. Par exemple :

```
char *p = 0x1000;  // p pointe à l'adresse 1000,
                   // où nous allons trouver des caractères.
```

Voici quelques exemples d'utilisation:

```
char c1, c2;
char *p = 0x1000;
c1 = *p;    // c1 contient le caractère stocké à l'adresse 0x1000
p++;        // p pointe à l'adresse 0x1001
c2 = *p;    // c2 contient le caractère stocké à l'adresse 0x1001
*p = 'A';   // Le code ascii 'A' (65) est stocké à l'adresse 0x1001
```

En C, un pointeur peut être considéré également comme un tableau. Une fois déclaré `char *p = 0x1000`, les couples d'expressions suivantes sont équivalentes :

```
char c1 = *p;
char c2 = p[0];

char c3 = *(p+1);
char c4 = p[1];
```

Toutes les variables sont stockées à une adresse. L'opérateur `&` permet d'obtenir l'adresse à la quelle est stockée une variable :

```
int *p;
int n = 0;
p = &n;             // Maintenant 'p' pointe sur l'adresse de 'n'
p[0] += 2;          // Après cette instruction, n=2.
*p += 5;            // Après cette instruction, n=6;
```

La notation suivante permet d'initialiser les valeurs d'un tableau :

```
int t[] = {1, 2, 3, 4, 5};
int s = 0;
int n;
for(n=0; n<5; n++)
{
        s+=t[n];
}
// Ici s = 15.
```

Pour passer un tableau comme paramètre à une fonction :

```
int somme(int t[], int taille)
{
	int n;
	int s;
	for(n=0; n<taille; n++)
	{
		s += t[n];
	}
	return s;
}

void main(void)
{
	int t[] = {1, 2, 3, 4, 5}
	int s;

	s = somme(t,5);
}
```

La fonction `somme` pourrait également être définie comme ceci :

```
int somme(int *t, int taille)
{
	int n;
	int s;
	for(n=0; n<taille; n++)
	{
		s += t[n];
	}
	return s;
}
```

Ou bien comme ceci :

```
int somme(int *t, int taille)
{
	int n;
	int s;
	for(n=0; n<taille; n++)
	{
		s += *t;
		t ++;
	}
	return s;
}
```

### 8.6 Programmation C – Les énumérations

Les énumérations sont des constructions pratiques pour décrire des variables qui ne peuvent prendre qu'un nombre réduit de valeurs différentes. Par exemple, elles sont très bien adaptées pour modéliser l'état interne d'un automate, puisqu'elles limitent naturellement le nombre de valeurs possibles, en plus de donner à chacune un nom explicite.

Les énumérations sont déclarées en deux étapes. D'abord la déclaration de type :

```
enum MonEnumeration
{
	UNEVALEUR,
```

```
        AUTREVALEUR
};
```

On peut déclarer des variables comme appartenant au type préalablement déclaré :

```
enum MonEnumeration me1, me2;
```

Pour assigner la valeur des variables :

```
me1 = UNEVALEUR;
me2 = UNEAUTREVALEUR;
```

Il est possible de tester la valeur des variables comme si il s'agissait de types élémentaires :

```
if (me1 == UNEVALEUR)
{
        ...
}

switch(me2)
{
        case UNEVALEUR:
                ...
                ...
                break;

        case UNEAUTREVALEUR:
                ...
                ...
                break;
}
```

Pour utiliser des variables énumérées dans une fonction :

```
enum MonEnumeration uneFonction(enum MonEnumeration parametre)
{
        if (parametre == UNEAUTREVALEUR)
        {
                return UNEVALEUR;
        }
        else
        {
                return UNEAUTREVALEUR;
        }
}
```

Pour déclarer un tableau d'énumérations :

```
enum MonEnumeration e[] = {UNEVALEUR, UNEAUTREVALEUR};
```

### 8.7 Structures

Les structures permettent de déclarer des groupes de données dans une unique entité. C'est une façon pratique de réduire le nombre de paramètres des fonctions. C'est également la seule manière pour une fonction de rendre plus que une valeur avec `return`.

Pour déclarer une structure :

```
struct MaStructure
{
        int x, y ;
```

```
        enum MonEnumeration status ;
} ;
```

Pour déclarer une variable de type structure :

```
struct MaStructure maStructure ;
```

Pour initialiser une variable de type structure :

```
maStructure.x = 10 ;
maStructure.y = 20 ;
status = UNEAUTREVALEUR ;
```

L'utilisation de structures comme paramètres de fonctions est plus délicat ; les structures ne peuvent pas se passer comme valeur, mais uniquement comme référence. En conséquence :

```
void uneFonction(struct MaStructure *s)
{
      switch(s->status)
      {
            case UNEVALEUR:
                  s->x += s->y;
                  break;
            case UNEAUTREVALEUR:
                  s->x -= s->y;
                  break;
      }
}

void main()
{
      struct MaStructure maStructure;

      maStructure.x = 10 ;
      maStructure.y = 20 ;
      status = UNEAUTREVALEUR ;

      uneFonction(&maStructure);
}
```

### 8.7.1 Collections de structures

Les collections de structures ont les mêmes propriétés que les collections de valeurs simples.

Pour déclarer et initialiser une collection de structure :

```
struct MaStructure mesStructures[] =
{
      {0, 1, UNEVALEUR},
      {2, 3, UNEAUTREVALEUR}
};
int n;
int s = 0;
for(n=0; n<2; n++)
{
      s += mesStructures[n].x;
      s += mesStructures[n].y;
}
// Ici, s=6.
```

Une autre façon d'initialiser une collection de structures :

```
struct MaStructure mesStructures[2];
mesStructures[0].x = 10;
mesStructures[0].y = 20;
mesStructures[1].x = 30;
mesStructures[1].y = 40;
```

Pour envoyer un élément d'une collection de structures à une fonction qui prend une structure comme paramètre :

```
for(n=0; n<2; n++)
{
      // L'opérateur [] est plus prioritaire que &
      uneFonction(&mesStructures[n]);
}
```

Une fonction qui prend comme paramètre une collection de structures est déclarée exactement comme une fonction qui prend une unique structure :

```
void traiteDesStructures(struct MesStructures desStructures[],
                         int taille)
{
      int n;
      for (n=0; n<taille; n++)
      {
            desStructures[n].x = ....;
      }
}
```

Elle pourrait également être déclarée comme ceci :

```
void traiteDesStructures(struct MesStructures *desStructures,
                         int taille)
{
      int n;
      for (n=0; n<taille; n++)
      {
            desStructures[n].x = ....;
      }
}
```

Ou encore comme ceci :

```
void traiteDesStructures(struct MesStructures *desStructures,
                         int taille)
{
      int n;
      for (n=0; n<taille; n++)
      {
            desStructures->x = ....;
            ...
            desStructures++;
      }
}
```

## 8.8 Les unions

Les unions ressemblent à des structures, avec la différence que tous les membres sont stockés sur la même position de mémoire.

```
union MonUnion
{
      unsigned int ui;
      signed int si;
      unsigned char c[2];
};

void main(void)
{
      union MonUnion u;
      u.c[0] = 0xFF;
      u.c[1] = 0xFF;

      // Ceci affiche «65535 - -1»:
      printf("%u - %d", u.ui, u.si);
}
```

### 8.8.1 La virgule fixe

En informatique[1], une représentation d'un nombre en virgule fixe est un type de donnée correspondant à un nombre qui possède (en base deux ou en base dix) un nombre fixe de chiffres après la virgule. Les nombres en virgule fixe sont utiles pour représenter des quantités fractionnaires dans un format utilisant le complément à deux quand le processeur de l'ordinateur n'a aucune unité de calcul en virgule flottante ou quand une virgule fixe permet d'augmenter la vitesse d'exécution ou d'améliorer l'exactitude des calculs. La plupart des processeurs à faible coût (ex: microcontrôleurs) ne disposent pas d'unité de calcul en virgule flottante.

Cette notation était déjà utilisée dans les bouliers de l'Inde antique et, dans un passé plus récent, dans les caisses enregistreuses mécaniques. En réalité la causse enregistreuse affiche un nombre entier (ci-dessous 1225), mais on a peint sur la carcasse l'endroit où se trouve la virgule décimale. À gauche se trouvent les francs. À droite se trouvent les centimes.

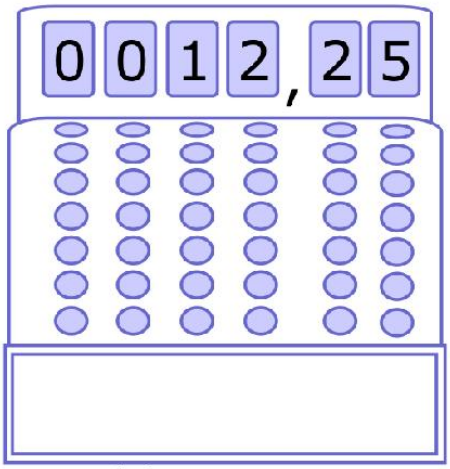



*Illustration 39: Virgule fixe sur une caisse enregistreuse à l'ancienne*

L'immense avantage de la virgule fixe est que les opérations et soustractions décrites plus haut marchent exactement pareil que pour les nombres entiers. La seule différence se trouve au moment

1 Extrait de Wikipédia : http://fr.wikipedia.org/wiki/Virgule_fixe

de l'interprétation du nombre. Mais ceci aussi, découle pratiquement de source:

- En notation entière:

$$1110101 = 1\cdot2^6 +1\cdot2^5 +1\cdot2^4 +0\cdot2^3 +1\cdot2^2 +0\cdot2^1 +1\cdot2^0 \equiv 117$$

- Par contre, en virgule fixe, si nous décidons que les deux bits les mois signifiants sont la partie fractionnaire:

$$11101.01 = 1\cdot2^4 +1\cdot2^3 +1\cdot2^2 +0\cdot2^1 +1\cdot2^0 +0\cdot2^{-1} +1\cdot2^{-2} \equiv 29.25$$

Avec une union il est simple de déclarer un nombre dont la virgule fixe se trouve entre les 8 bits les plus signifiants et les 8 bits les moins signifiants :

```
union VirguleFixe
{
      int w;      // 16 bits en virgule fixe.
      char b;     // Permet d'accéder à la part entière.
}
```

### 8.9 Code distribué sur plusieurs fichiers

Distribuer un programme sur plusieurs fichiers est intéressant dès que le code devient trop long pour le gérer confortablement dans l'éditeur de texte, ou bien que plusieurs personnes doivent travailler en même temps sur le code. Les subtilités de la distribution des fonctionnalités sont infinies, mais l'idée de base est de regrouper les fonctions par catégories, selon un critère clair.

Chaque groupe de fonction est défini en deux fichiers :

- Le fichier « .h » où se trouvent les entêtes des fonctions, plus la déclaration des énumérations et des structures. Dans celui-ci on trouve des directives de compilations qui empêchent le fichier d'être pris en compte deux fois dans la compilation.
- Le fichier « .c » où se trouvent les fonctions, écrites normalement. Dans le fichier « .c » se trouve également une référence au fichier « .h »

**mathematiques.h**

```
#ifndef __MATHEMATIQUES_H
#define __MATHEMATIQUES_H
enum SigneEnum
{
      POSITIF,
      NEGATIF
}
int operation(int a, int b, enum SigneEnum signe);
int somme(int a, int b);
int soustraction(int a, int b);
#endif
```

**mathematiques.c**

```
include "mathematiques.h"
int operation(int a, int b, enum SigneEnum signe)
{
      switch(signe)
      {
            case POSITIF:
                  return somme(a,b);
            case NEGATIF
                  return soustraction(a,b);
      }
}
int somme(int a, int b)
{
      return a + b;
}
int soustraction(int a, int b)
{
      return a - b;
}
```

Pour utiliser les fonctions mathématiques depuis ailleurs, il suffit d'inclure le fichier « .h ». Par exemple :

**main.c**

```
include <stdio.h>
include "mathematiques.h"

void main(void)
{
      int r1 = 10, r2 = 20;
      int s;
      enum SigneEnum signe = POSITIF;

      s = operation(r1, r2, signe);
      printf("Somme: %d", s);
}
```

# 9. Les composants TTL

## 9.1 Concept – « *Chip Select* » et l'état de « Haute Impédance »

On nomme[1] « *chip select* » une entrée de contrôle de nombreux circuits intégrés, tels que les puces mémoires, permettant d'activer ou désactiver le circuit. Quand elle est active, le composant peut être adressé ; quand elle ne l'est pas, le composant est dans un mode dit « *standby* » (au repos). L'économie d'énergie qui en découle est appréciable, surtout quand le nombre de circuits désactivés est important (ce qui est le cas pour les puces formant la mémoire centrale d'un ordinateur par exemple).

Suivant les fabricants et les textes, l'entrée *chip select* (CS) est également nommée *chip enable* (CE), *slave enable* (SE), *slave select* (SS), ou *encore enable* (E).

Lorsque le composant est au repos, il met toutes ses sorties en état de « haute impédance », notée avec Z. Un circuit dont les sorties sont en en état de « haute impédance » est déconnecté du reste du circuit :

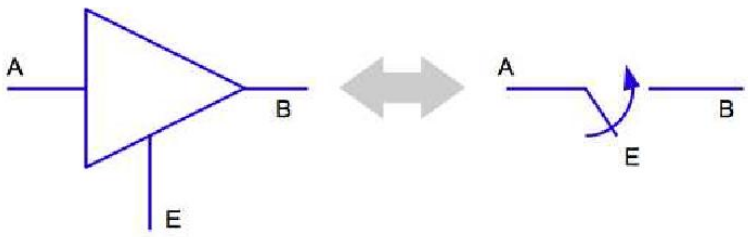



*Illustration 40: L'état de haute impédance agit comme un interrupteur*

Voici un exemple de table de vérité pour un circuit minimaliste qui ne réalise aucune opération :

| A | E | B |
|---|---|---|
| 0 | 0 | Z |
| 1 | 0 | Z |
| 0 | 1 | 0 |

1 Extrait de Wikipédia http://fr.wikipedia.org/wiki/Chip_select

| A | E | B |
|---|---|---|
| 1 | 1 | 1 |

Les composants dotés de sorties à trois états peuvent partager les mêmes connecteurs, à condition un seul d'entre eux soit actif (entrée E) à la fois:

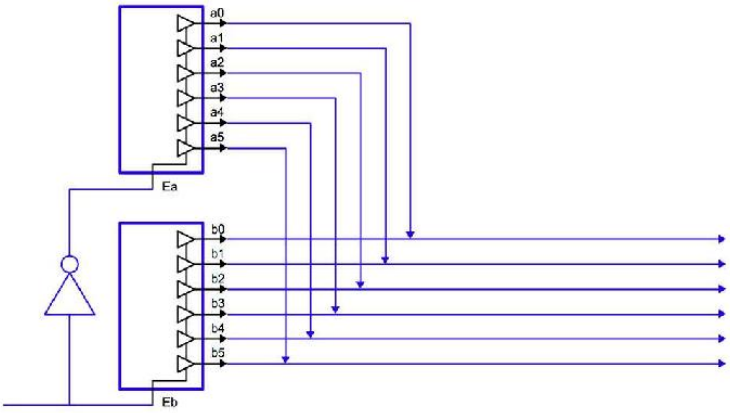



*Illustration 41: Les sorties à trois états peuvent partager les mêmes connecteurs*

## 9.2 Composants – Les séries TTL

Transistor-Transistor Logic ou TTL est une famille de circuits logiques utilisée en électronique inventée dans les années 1960, réalisée avec la technologie du transistor bipolaire.

La série TTL originale tend à disparaître aujourd'hui du fait de sa consommation énergétique élevée[1], mais des sous-familles sont apparue au fur et à mesure pour s'adapter au nouvelles techniques:

- TTL : série standard
- TTL-L (*low power*) : série à faible consommation
- TTL-S (*shottky*) : série rapide (utilisation de diodes *schottky*)
- TTL-AS (*advanced shottky*) : version améliorée de la série S
- TTL-LS (*low power shottky*) : combinaison des technologies L et S, c'est la famille la plus répandue
- TTL-ALS (*advanced low power shottky*) : version améliorée de la série LS
- TTL-F (*FAST : Fairchild Advanced Schottky Technology*)
- TTL-AF (*advanced FAST*) : version améliorée de la série F
- TTL-HC (*high speed C-MOS*) : circuit TTL fabriqué en technologie C-MOS dans un boîtier TTL (tension compatible TTL, mais pas l'emplacement des pattes)
- TTL-HCT (*high speed C-MOS transposed*) : série HC dotée de niveaux logiques compatibles TTL (100% compatible TTL, car le brochage TTL est conservé)

1 Voir plus ici : http://fr.wikipedia.org/wiki/Transistor-Transistor_logic

### 9.2.1 Composants – La famille TTL-HC

La famille à utiliser dans les circuits micro-contrôlés avec le 8051 est la TTL-HC. Voici les caractéristiques générales de cette famille :

- Alimentation de la 74HC: 2 à 6V, des petites fluctuations sont tolérées.
- Les entrées ont une très haute impédance, ce qui a l'avantage d'affecter très peu le circuit au quel elles sont connectées. Par contre cela veut dire qu'il ne faut pas laisser des entrées non connectées, parce qu'elles peuvent très facilement capter du bruit électronique ou de l'électricité statique, et changer rapidement d'état de façon imprévisible.
- Il est fortement déconseillé de connecter des entrées 74HC à des sorties 74LS ; les fourchettes pour le 0 ne sont pas compatibles. Si le circuit contient des 74LS, il vaut mieux utiliser la famille 74HCT.
- Les sorties peuvent admettre des courant jusqu'à 4mA tout en restant dans les fourchettes de tensions standards pour le 1 logique. Dans les cas où ceci n'est pas important, le courant maximum nominal est de 20mA. Pour des courants plus grands, il faut amplifier avec un transistor.
- Une sortie 74HC peut alimenter plus de 50 entrées 74HC. Une sortie 74HC ne peut pas alimenter plus de 10 entrées 74LS.
- Le temps de propagation du signal à travers une porte logique est d'approximativement 10ns.
- La fréquence peut aller jusqu'à 25MHz.
- La consommation de courant est de quelques mW à 1MHz.

### 9.2.2 Composants – Identifiants de circuits TTL à usage courant

Les circuits intégrés de la série TTL sont habituellement préfixés avec le numéro 74, puis le sigle de la famille (L, AS, HC), puis un chiffre qui identifie la fonctionnalité du composant. Par exemple, le 7400 est un circuit avec 4 portes NAND à deux entrées. Le 74L00 est le même, mais en *Low power*. Le 74HC00 est le même, mais en *high speed C-MOS*.

Le tableau suivant[1], non exhaustif, liste quelques circuits TTL usuels :

| Numéro | Fonctionnalité |
|---|---|
| 7400 | *quad 2-input NAND* |
| 7402 | *quad 2-input NOR* |
| 7403 | *quad 2-input NAND with open collector outputs* |
| 7408 | *quad 2-input AND* |
| 7409 | *quad 2-input AND with open collector outputs* |
| 7432 | *quad 2-input OR* |
| 7486 | *quad 2-input EX-OR* |
| 74132 | *quad 2-input NAND with Schmitt trigger inputs, which provide good noise immunity. They are ideal for slowly changing or noisy signals.* |
| 7402 | *quad 2-input NOR* |
| 7410 | *triple 3-input NAND* |
| 7411 | *triple 3-input AND* |

1 Les données du tableau sont extraites de : http://www.kpsec.freeuk.com/components/74series.htm

| Numéro | Fonctionnalité |
|---|---|
| 7412 | *triple 3-input NAND with open collector outputs* |
| 7427 | *triple 3-input NOR* |
| 7420 | *dual 4-input NAND* |
| 7421 | *dual 4-input AND* |
| 7430 | *8-input NAND gate* |
| 7431 | *8-input AND gate* |
| 7404 | *hex NOT* |
| 7405 | *hex NOT with open collector outputs* |
| 7414 | *hex NOT with Schmitt trigger inputs* |
| 7490 | *decade (0-9) ripple counter* |
| 7493 | *4-bit (0-15) ripple counter* |
| 74390 | *dual decade (0-9) ripple counter* |
| 74393 | *dual 4-bit (0-15) ripple counter* |
| 74160 | *synchronous decade counter (standard reset)* |
| 74161 | *synchronous 4-bit counter (standard reset)* |
| 74162 | *synchronous decade counter (synchronous reset)* |
| 74163 | *synchronous 4-bit counter (synchronous reset)* |
| 74192 | *up/down decade (0-9) counter* |
| 74193 | *up/down 4-bit (0-15) counter* |
| 74HC4017 | *decade counter (1-of-10)* |
| 74HC4020 | *14-bit ripple counter* |
| 74HC4040 | *12-bit ripple counter* |
| 74HC4060 | *14-bit ripple counter with internal oscillator* |
| 7442 | *BCD to decimal (1 of 10) decoder* |
| 7447 | *BCD to 7-segment display driver* |
| 74HC4511 | *BCD-to-7 segments latch decoder driver* |
| 74373 | *Octal D-Type transparent latch with 3-state output* |
| 74154 | *4-to-16 line decoder / demultiplexer* |
| 74238 | *3-to-8 line decoder inverting and non-inverting* |

### 9.3 Composants – La bascule

Voici la définition de bascule dans *wikipedia* :

> *Une bascule est un circuit logique capable, dans certaines circonstances, de maintenir les valeurs de ses sorties malgré les changements de valeurs d'entrées*[1]

Il y a plusieurs types de bascules. La bascule D est une unité de mémoire élémentaire. Un bit, en

1 Voir plus ici : http://fr.wikipedia.org/wiki/Bascule_%28circuit_logique%29

quelque sorte :

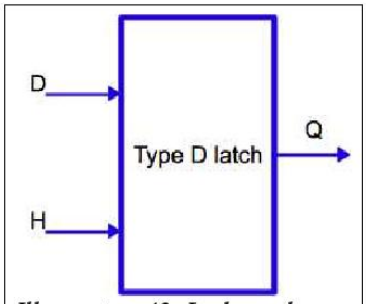



*Illustration 42: La bascule type D se comporte comme une mémoire de 1 bit*

La bascule de type D permet de stocker la valeur de l'entrée à un certain moment, déterminé par l'état actif de H[1].

- Tant que H reste inactif, la sortie Q reste stable, indépendamment des variations de D.
- Lorsque H est actif, la sortie Q acquiert la valeur de l'entrée D.

La bascule D est appelée « *latch* » en anglais.

### 9.3.1 Composants – Le 74HC373

Ce circuit contient 8 bascules de type D, activés simultanément par le signal de contrôle LE. Il représente un octet de mémoire :

- Lorsque LE est à niveau haut, les valeurs des entrées D0..D7 sont stockées.
- Lorsque $\overline{OE}$ est à niveau bas, les sorties Q0..Q7 sont actives, et rendent la valeur stockée.
- Lorsque $\overline{OE}$ est à niveau haut, les sorties Q0..Q7 sont en haute impédance.

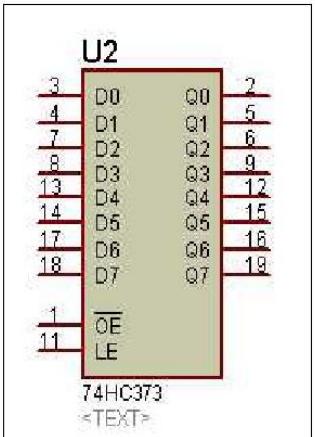



*Illustration 43: Le 74hc373 contient 8 verrous de type D.*

1 H est pour « Horloge », quoique le terme est un peu équivoque. Il s'agit plutôt d'un signal de contrôle qui active l'écriture dans la mémoire.

### 9.4 Composants – Le multiplexeur et le décodeur

Un multiplexeur est un circuit qui a :

- Deux ou plusieurs entrées de données.
- Un sélecteur.
- Une sortie unique, dont la valeur est celle de l'entrée activée par un sélecteur :

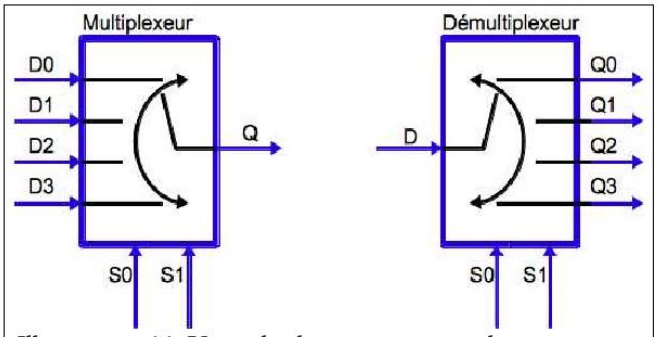



*Illustration 44: Un multiplexeur connecte plusieurs entrées à une seule sortie*

Un démultiplexeur est exactement l'inverse du multiplexeur. Il a :

- Une entrée.
- Un sélecteur.
- Plusieurs sorties. La sortie correspondante au sélecteur est à la valeur de l'entrée. Toute les autres sont à 0.

Il est courant que le démultiplexeur, appelé également décodeur, ait plusieurs entrées et réalise une opération booléenne avec. La valeur de la sortie sélectionnée est dépendante du résultat de l'opération.

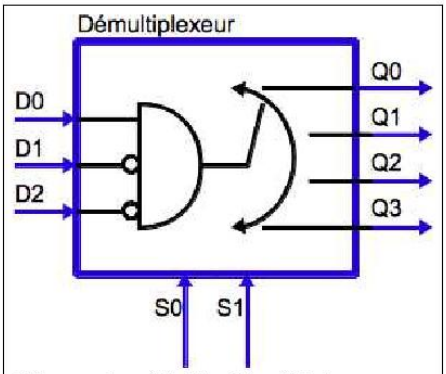



*Illustration 45: Un démultiplexeur peut implémenter une opération booléenne en entrée*

On peut faire un décodeur avec un démultiplexeur, en connectant son entrée à une valeur fixe (par exemple 1).

### 9.4.1 Composants – Décodeurs / démultiplexeurs TTL

Voici deux exemples de décodeurs / démultiplexeurs:

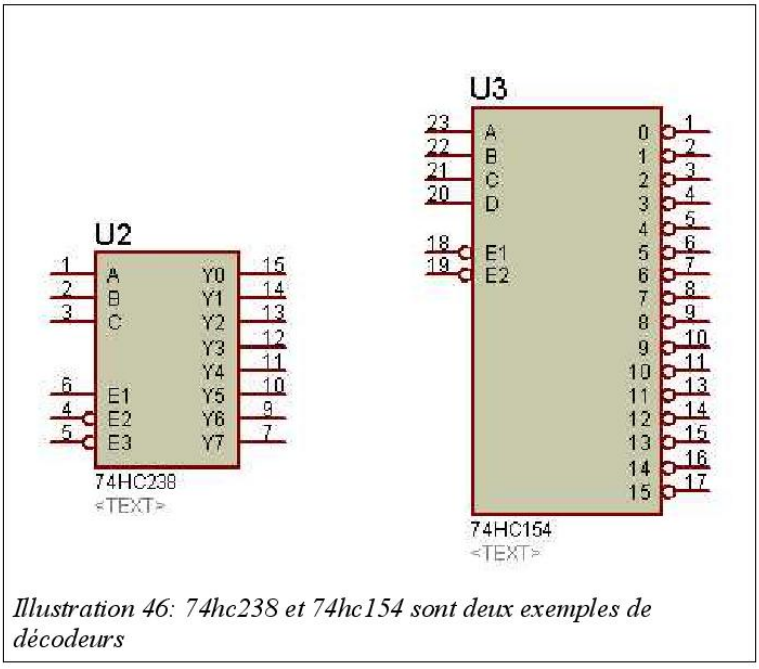



*Illustration 46: 74hc238 et 74hc154 sont deux exemples de décodeurs*

Une seule des sorties est activée, selon l'état du sélecteur :

- 74hc238 :
  - Le sélecteur est composé des entrées A, B et C.
  - La sortie correspondante au sélecteur prend la valeur de : `(E1 and` $\overline{\texttt{E2}}$ `and` $\overline{\texttt{E3}}$`)`
  - Les autres sorties sont à 0.
- 74hc154 :
  - Le sélecteur est composé des entrées A, B, C et D.
    - La sortie correspondante au sélecteur prend la valeur de : `not (`$\overline{\texttt{E1}}$ `and` $\overline{\texttt{E2}}$`)`
    - Toutes les autres sorties sont à 1.

# 10. La mémoire externe

## 10.1 8051 – L'accès à la mémoire de données externe

Au point 2.2 nous avons vu que le 8051 possède une mémoire de données interne de 256 octets. Cette mémoire très limitée est utilisée pour stocker les variables des programmes, et pour accéder aux périphériques (notamment les ports).

Le 8051 peut également être connecté à une mémoire externe au travers des ports 0 et 2 :

- Le port 0 est utilisé pour lire ou écrire les données.
- Le port 0 et le port 2 sont utilisés pour transmettre une adresse de 16 bits à la quelle on veut lire ou écrire.

Grace à ce mécanisme, le 8051 est capable de gérer 65536 octets de mémoire externe. Le port 0 a un double usage : Il sert pour transmettre les 8 bits de la donnée, et également pour transmettre les 8 bits moins significatifs de l'adresse.

La communication entre le 8051 est la mémoire externe est gérée par le connecteurs suivants :

| Connecteur | Acronyme | Direction | Port | Utilisation |
|---|---|---|---|---|
| ALE | *Address Latch Enable* | Sortie | -- | Lorsqu'il est à niveau haut, il indique que le 8051 a mis sur le Port 0 la partie basse de l'adresse. |
| WR | *WRite* | Sortie | Port 3, bit 6 | Lorsqu'il est à niveau bas, il indique qu'une donnée est prête sur le Port 0, et que la mémoire peut la récupérer. |
| RD | *ReaD* | Sortie | Port 3, bit 6 | Lorsqu'il est à niveau bas, il indique que le Port 0 est prêt à recevoir une donnée. |
| PORT0 | -- | Entrée / Sortie | Port 0 | Alternativement la donnée, ou les 8 bits moins significatifs de l'adresse. |
| PORT2 | -- | Sortie | Port 2 | Les 8 bits les plus significatifs de l'adresse. |

### 10.1.1 8051 – Le port 0 en mode mémoire externe

Lorsque le port 0 travaille avec de la mémoire externe, un deuxième transistor s'active. Sa configuration devient totem-pole, et elle est compatible avec les entrées digitales des composants de

mémoire.

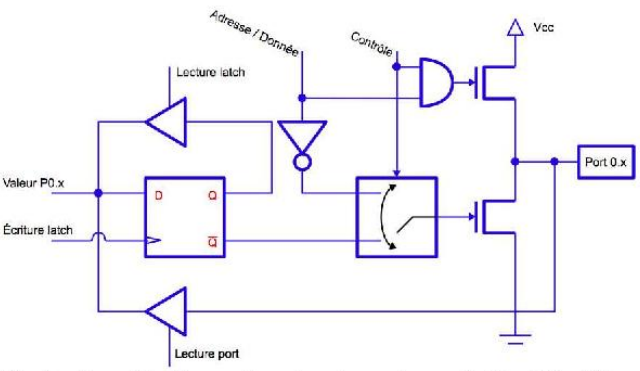



*Illustration 47: En mode mémoire externe, le Port 0 utilise un deuxième transistor*

En mode mémoire externe, les résistances de tirage (voir 2.3.3) ne sont plus nécessaires. Les deux transistors s'ouvrent alternativement pour brancher le connecteur du port 0 directement à Vcc ou à terre.

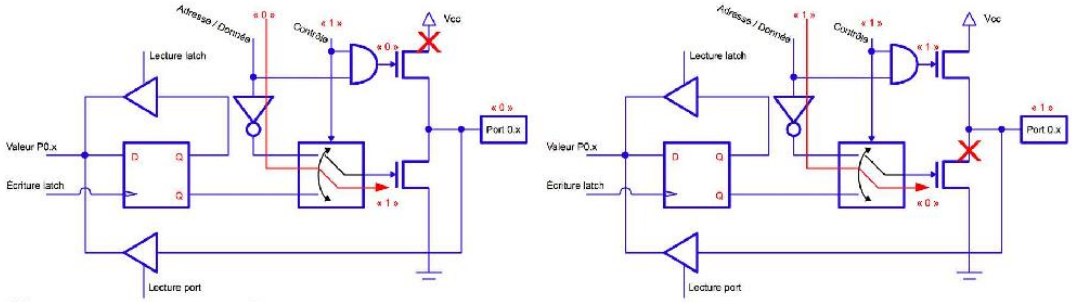



*Illustration 48: En mode mémoire externe, « 0 » est court-circuité à terre, « 1 » est court-circuité à Vcc*

### 10.1.2 8051 – Cycles de lecture et d'écriture de la mémoire externe

Le cycle de lecture est l'opération par la quelle le micro-contrôleur lit une donnée de la mémoire externe.

1. Le micro-contrôleur place l'adresse sur le Port 0 et le Port 2.
2. Le micro-contrôleur met à niveau haut le signal ALE.
3. Le micro-contrôleur remet à niveau bas le signal ALE.
4. Le micro-contrôleur libère le Port 0 (le configure comme une entrée).
5. Le micro-contrôleur met $\overline{\text{RD}}$ à niveau bas.
6. La mémoire place la donnée sur le Port 0.
7. Le micro-contrôleur récupère la donnée du Port 0.
8. Le micro-contrôleur remet $\overline{\text{RD}}$ à niveau haut.
9. La mémoire libère le Port 0 (se met en haute impédance).

Le cycle d'écriture est l'opération par la quelle le micro-contrôleur écrit une donnée sur la mémoire externe.

1 Le micro-contrôleur place l'adresse sur le Port 0 et le Port 2.

2 Le micro-contrôleur met à niveau haut le signal `ALE`.

3 Le micro-contrôleur remet à niveau bas le signal `ALE`.

4 Le micro-contrôleur met la donnée sur le Port 0 (et le configure comme une sortie).

5 Le micro-contrôleur met $\overline{\texttt{WR}}$ à niveau bas.

6 La mémoire récupère la donnée du Port 0, et la stocke.

7 Le micro-contrôleur remet $\overline{\texttt{WR}}$ à niveau haut.

8 La mémoire libère le Port 0 (se met en haute impédance).

9 Le micro-contrôleur dispose du Port 0 pour la prochaine opération.

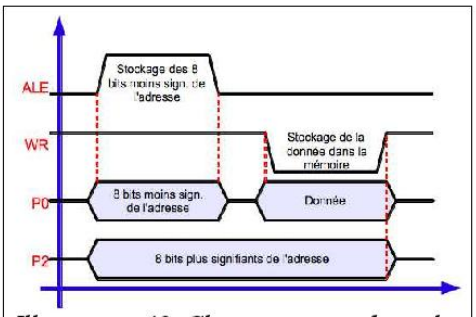



*Illustration 49: Chronogramme du cycle d'écriture*

### 10.1.3 Programmation C pour 8051 – `xdata`

Le modificateur `xdata` sert à indiquer qu'un pointeur se trouve sur la mémoire externe :

```
char xdata *p = 0x0000;
```

Attention, car il n'est pas possible d'initialiser un tableau de la même manière :

```
char xdata q[] = 0x1000;  // Ceci n'est pas correct.
```

La raison est comment le compilateur C interprète les différents éléments. Si au moment de les utiliser, la différence est imperceptible, l'initialisation est très différente :

- `*p` est une variable qui pointe sur une adresse. On peut donc changer sa valeur à tout moment dans le programme.
- `q[]` est une constante qui pointe sur une zone de mémoire. C'est donc le compilateur qui calcule sa valeur, et il n'est plus possible de la modifier en temps d'exécution. En temps d'exécution il est possible de changer le contenu du tableau, mais pas son adresse.

Voici la manière correcte de déclarer un tableau pointant à une adresse précise de la mémoire externe :

```
char xdata *tableau = 0x1000 // Par exemple pointant sur 4096.
char c0, c1, c2;
...
c0 = tableau[0];
c1 = tableau[1];
c2 = tableau[2];
```

## 10.2 Pratique – Lire et écrire dans la mémoire externe

Voici un exemple minimaliste pour connecter une mémoire externe de 2Ko au 8051, et un programme qui démontre comment y lire et écrire.

### 10.2.1 Le circuit

Le latch stocke les 8 bits moins significatifs de l'adresse dès que le signal ALE se met à niveau haut. De cette façon, au moment où $\overline{WR}$ ou $\overline{RD}$ s'activent, le bus `A[0..15]` contient la totalité de l'adresse.

Lorsque le signal $\overline{WR}$ s'active, la mémoire lit la valeur présente sur `D[0..7]` et la stocke à l'adresse présente sur `A[0..15]`. Lorsque le signal $\overline{RD}$ s'active, la mémoire récupère la valeur stockée à l'adresse `A[0..15]`, et la dépose sur `D[0..15]`.

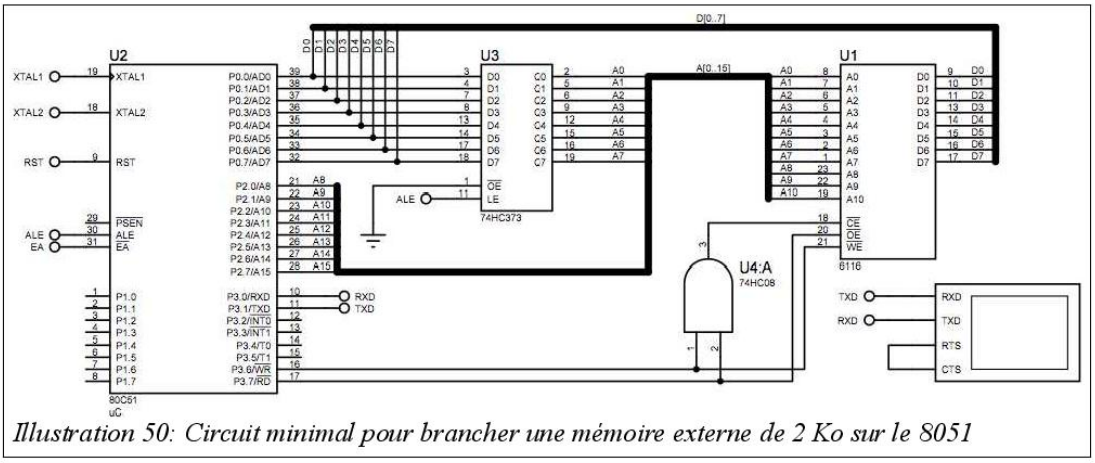



*Illustration 50: Circuit minimal pour brancher une mémoire externe de 2 Ko sur le 8051*

### 10.2.2 Le connecteur $\overline{CE}$

Beaucoup des composants destinés au microprocesseurs ont un connecteur appelé $\overline{CE}$. Il s'agit du connecteur de haute impédance décrit au point 9.1.

La mémoire peut rester en repos tant que aucune opération de lecture ou d'écriture est en cours. En tenant compte que WR et RD sont toutes deux de logique inversée (actives à niveau bas), voici la table de vérité pour CE :

| $\overline{WR}$ | $\overline{RD}$ | $\overline{CE}$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | Indéfini |

Ce qui correspond à un ET logique, d'où la présence d'une porte de ce type sur le circuit.

### 10.2.3 Le programme

Le programme suivant écrit une valeur dans chacune des adresses entre 0 et 2047 (0x3FF), puis ensuite les affiche au travers de la sortie standard :

```
/**
 * Point d'entrée du programme.
 * Ecrit dans la mémoire, puis lit de la mémoire.
 * Une mémoire doit être connectée entre les adresses
 * 0x0000 et 0x03FF
 */
void main(void)
{
	unsigned char xdata *memoire = 0x0000;
	unsigned int adresse;
	unsigned char valeur;

	UART_Initialize();

	valeur=1;
	for(adresse=0 ; adresse<=0x3FF ; adresse++)
	{
		memoire[adresse] = valeur;
		valeur += 2;
	}

	for(adresse=0 ; adresse<=0x3FF ; adresse++)
	{
		valeur = memoire[adresse];
		printf("Adresse %u contient %bu \r\n",adresse,valeur);
	}

	while(1);
}
```

Le fonctions pour gérer la UART vues au point 7.6 sont également nécessaires.

### 10.2.4 Le résultat

Voici un extrait de la sortie du terminal virtuel :

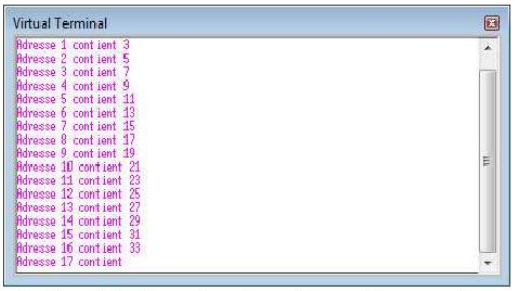



*Illustration 51: La mémoire est peuplée avec les valeurs dans la boucle*

### 10.2.5 Quelques questions.

- Que se passe-t'il si on lit des positions de mémoire au delà de l'adresse 0x3FF ?
- Que se passe-t'il si l'entrée $\overline{CE}$ est directement connectée à terre ? Pourquoi est-ce mieux d'utiliser une porte logique ?

# 11. Pratique - Une mémoire d'affichage

À l'exercice pratique du point 10.2, nous avons vu comment lire et écrire dans une mémoire externe. Dans cet exercice nous démontrons comment construire une petite mémoire (4 octets seulement) avec des bascules et un multiplexeur. Puis nous connectons des afficheurs LED 7 segments aux bascules, pour réaliser un affichage de 4 digits.

## 11.1 Le circuit

Le bus de données est connecté à l'entrée des bascules. Un multiplexeur contrôle l'entrée $\overline{\texttt{LE}}$ de chaque bascule. Pour qu'une bascule s'active il faut que $\overline{\texttt{WR}}$ soit à niveau bas, et que l'adresse correspondante soit sur le bus `A[0..15]` :

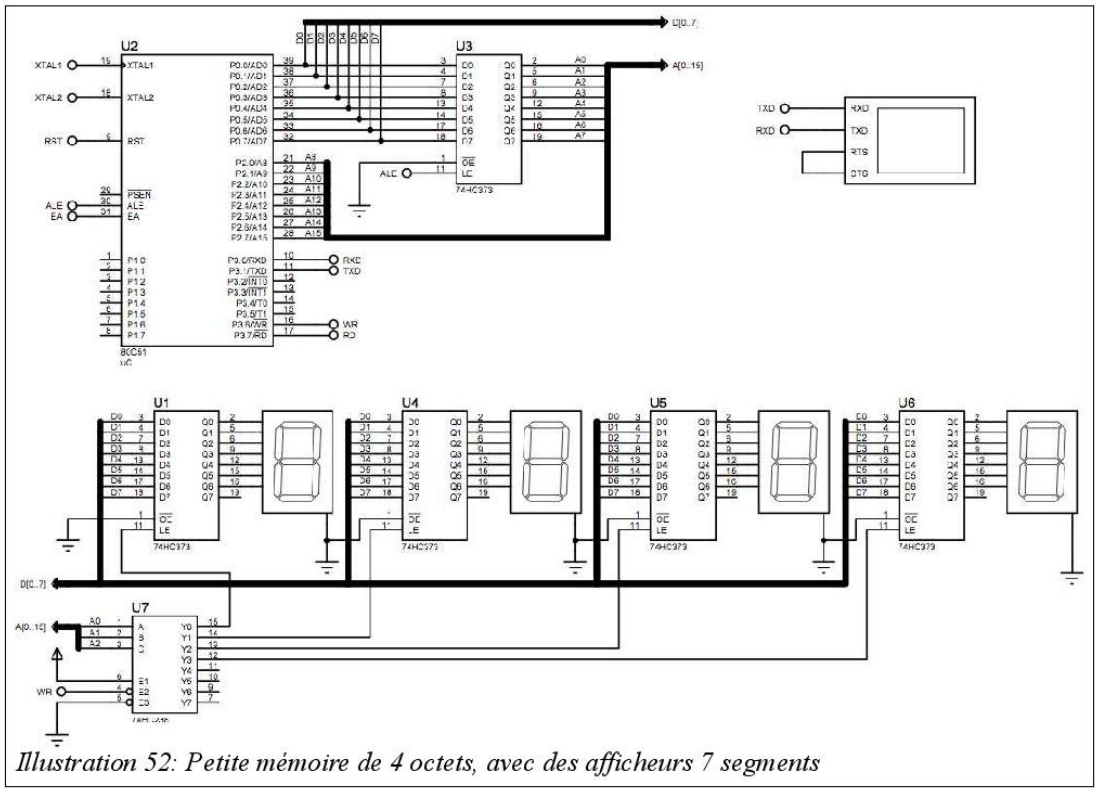



*Illustration 52: Petite mémoire de 4 octets, avec des afficheurs 7 segments*

Ce circuit est très conceptuel. Il fonctionne sur le simulateur, mais il présente quelques problèmes d'ordre pratique, comme par exemple le manque de résistances de protection sur les afficheurs.

### 11.2 Le programme

Le programme consiste uniquement à écrire des valeurs dans les adresses entre 0 et 3 de la mémoire externe, qui est dans cet exemple constituée uniquement des 4 bascules:

```
/**
 * Point d'entrée du programme.
 * Envoie les caractères encodés en BCD 7 segments
 * aux adresses entre 0x0000 et 0x0003 de la mémoire externe.
 * Si des afficheurs 7 segments y sont connectés, ils
 * afficheront.
 */
void main(void)
{
      unsigned char xdata *led7 = 0x0000;

      UART_Initialize();

      printf("Affichage 7 segments...");

      led7[0] = 0x06;
      led7[1] = 0x5B;
      led7[2] = 0x4F;
      led7[3] = 0x66;

      printf("termine");

      while(1);
}
```

### 11.3 Le résultat

Si tout va bien, le résultat est l'affichage de quatre chiffres sur les afficheurs 7 segments :

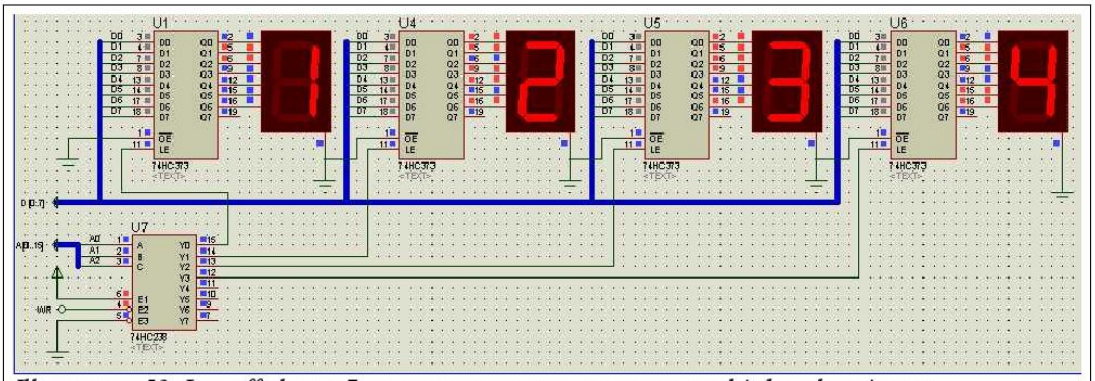



*Illustration 53: Les afficheurs 7 segments montrent ce qui est stocké dans la mémoire*

### 11.4 Quelques questions

- À quoi correspondent les valeurs `0x06`, `0x5B`, `0x4F`, `0x66` du programme ?
- Comment pourrait-on étendre le circuit pour avoir 8 afficheurs ?
- Pourquoi $\overline{WR}$ est connecté sur `E2`, et pas sur `E1` ?

# 12. Le clavier

## 12.1 Composant – Le clavier

Un clavier est une matrice d'interrupteurs. Les interrupteurs sont organisés en lignes et en colonnes. Tous les interrupteurs de la même ligne ont leurs entrées connectées au même conducteur. Tous les interrupteurs de la même colonne sont connectés au même conducteur.

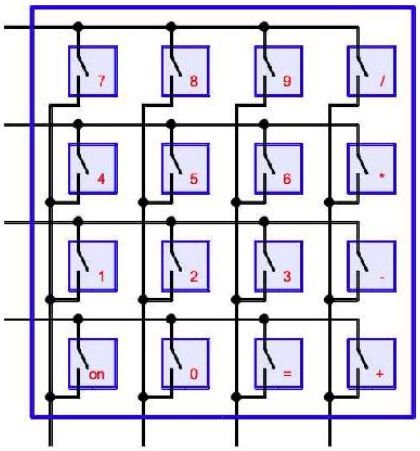



*Illustration 54: Un clavier est une matrice d'interrupteurs organisés en lignes et en colonnes*

Pour détecter si un interrupteur est enfoncé, on met le conducteur correspondant à la ligne à niveau haut, et on vérifie l'état du conducteur correspondant à la colonne[1]. Voici un exemple :

1 On peut également faire le contraire : mettre le conducteur correspondant à la colonne à niveau haut, et vérifier l'état du conducteur correspondant à la ligne.

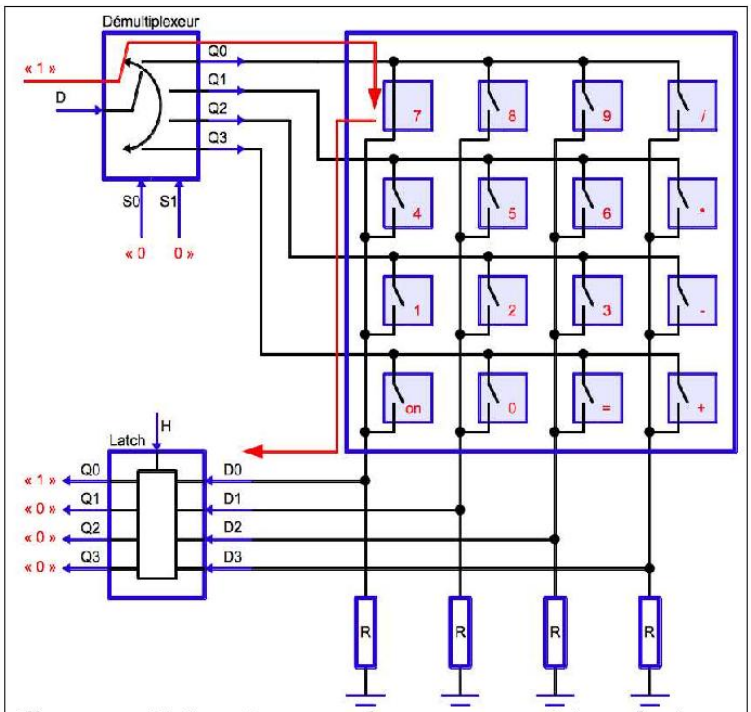



*Illustration 55: Les résistances de tirage servent à éviter les états de haute impédance (voir point 3.2)*

Il faut également éviter de provoquer un court-circuit si deux touches sont enfoncées en même temps. Pour éviter des problèmes avec les niveaux logiques, il faut que `r << R`.

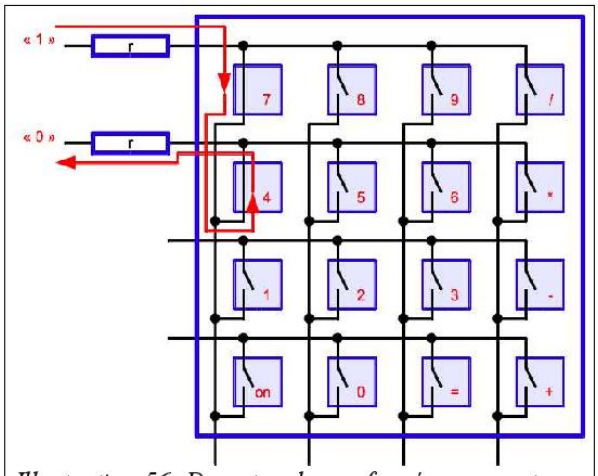



*Illustration 56: Deux touches enfoncées peuvent provoquer un court-circuit*

### 12.1.1 *Proteus* – Les claviers virtuels

Dans *Proteus* on trouve plusieurs claviers prédéfinis, tous sous le principe de la matrice d'interrupteurs.

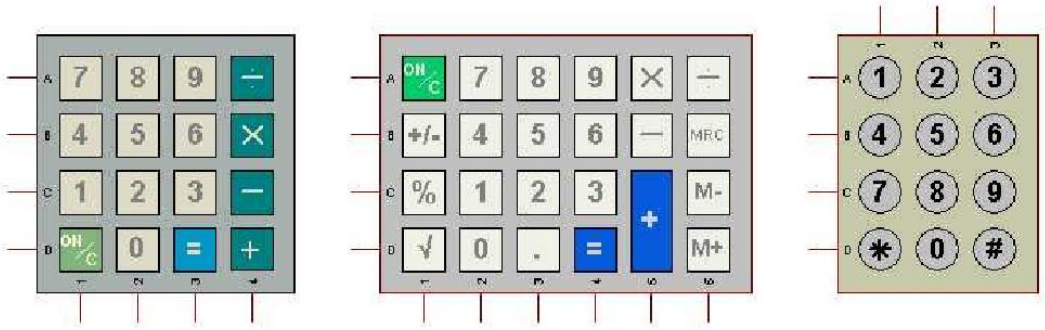



*Illustration 57: KEYPAD-SMALLCALC, KEYPAD-CALCULATOR et KEYPAD-PHONE*

Pour les exercices de ce chapitre, utilisez le `KEYPAD-SMALLCALC`.

## 12.2 Pratique - Le clavier comme une mémoire externe

Dans cet exercice, nous allons considérer l'ensemble de composants clavier, plus démultiplexeur d'entrée, plus le latch de sortie, comme une mémoire de lecture seule, de 4 octets :

- L'adresse (numéro de ligne) est contrôlée par les entrées du démultiplexeur.
- La valeur de l'élément de mémoire (état de la ligne) est obtenu à la sortie du latch.

### 12.2.1 Le circuit

Chaque élément de mémoire représente une ligne du clavier. La sortie du latch est en haute impédance tant que le microcontrôleur ne fait pas d'opération de lecture ($\overline{RD}$ est connecté à $\overline{OE}$).

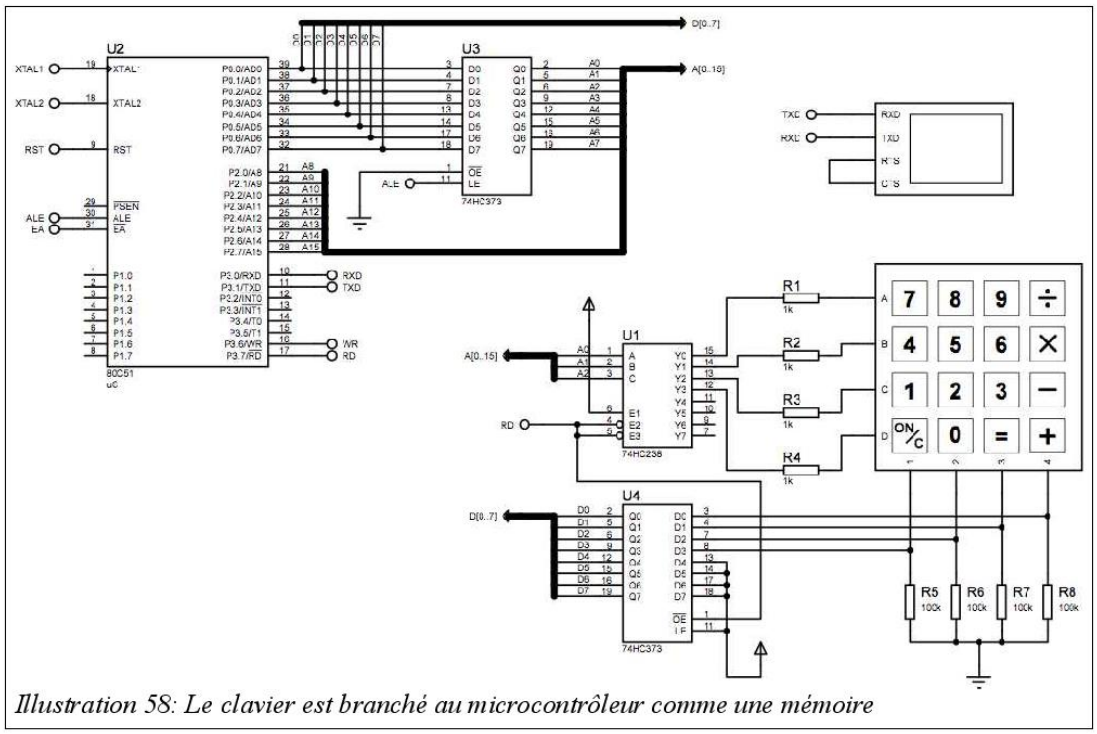



*Illustration 58: Le clavier est branché au microcontrôleur comme une mémoire*

Attention : pour les résistances de protection et de tirage autour du clavier, vous ne pouvez pas utiliser le modèle digital. Celui-ci ne gère pas les niveaux intermédiaires entre le « 1 » et le « 0 », du coup il ne supporte pas de connecter deux résistances en série.

### 12.2.2 Le programme

Le programme consiste à parcourir les adresses de 0 à 3 de la pseudo-mémoire, et d'afficher les valeurs obtenues en binaire :

```
/**
 * Point d'entrée du programme.
 * Parcoure la mémoire externe en lisant entre les
 * adresses 0x0000 et 0x0003, et affiche en binaire
 * les valeurs qui s'y trouvent.
 */
void main(void)
{
	unsigned char xdata *clavier = 0x0000;
	unsigned char ligne;
	unsigned char n,z;

	UART_Initialize();

	while(1)
	{
		printf("----------\r\n");
		for(n=0;n<4;n++)
		{
			// Obtient l'état de la n-ième ligne du clavier:
			ligne = clavier[n];

			// Affiche la valeur en binaire:
```

```
                        for (z = 128; z > 0; z >>= 1)
                        {
                              if ((ligne & z) == z)
                              {
                                    putchar('1');
                              }
                              else
                              {
                                    putchar('0');
                              }
                        }

                        // Saut de ligne:
                        printf("\r\n");
                  }

                  // Boucle d'attente
                  for(n=0;n<255;n++)
                  {
                        for(z=0;z<255;z++);
                  }
            }
      }
```

La boucle autour de la variable `z` sert à réaliser l'affichage binaire. La boucle d'attente sert à ne pas saturer le terminal, et le rendre plus lisible.

### 12.2.3 Le résultat

Si tout va bien, vous devriez voir quelque chose qui ressemble à ceci sur la sortie du terminal :

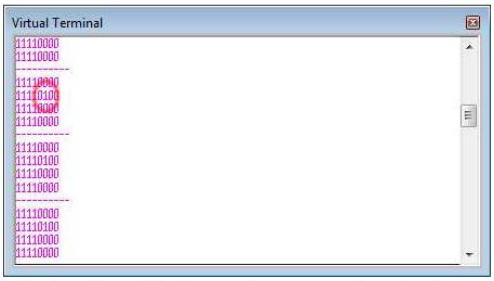



*Illustration 59: Les valeurs représentent l'état des touches*

Lorsque vous enfoncez une touche, un 1 apparaît à la place correspondante.

### 12.2.4 Quelques questions

- Comment fonctionne l'affichage binaire ?
- Comment implémenter une fonction `scanf` ? Est-ce possible d'implémenter quelque chose de similaire à ce qui a été fait pour `printf` ?
- Pourquoi les 4 bits les plus signifiants sont toujours à 1 ?

# 13. Le Bus

## 13.1 Concept – Le bus

Un bus informatique est un système de communication entre les composants d'un ordinateur. Dans sa plus simple expression il relie le micro-contrôleur à la mémoire (comme décrit au point 10.1 et, plus concrètement, à l'illustration du point 10.2). Dans cette configuration, le bus consiste uniquement en une série de conducteurs qui peuvent être classifiés en :

- Bus de données (entrée / sortie) : `D[0..N-1]` Sur le 8051, N = 8, donc le bus de données est `D[0..7]`.
- Bus d'adresses (seulement sortie) : `A[0..M-1]` Sur le 8051, M=16, donc le bus d'adresses est `A[0..15]`.
- Bus de contrôle (seulement sortie) : Suivant la famille de microprocesseurs / micro-contrôleurs il peut changer, mais consiste toujours en une variation autour de $\overline{WR}$ et $\overline{RD}$.

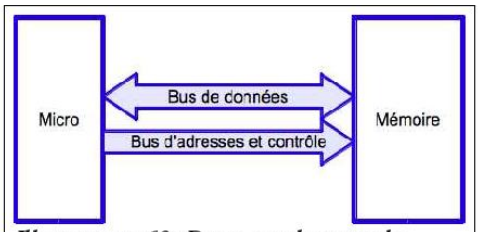



*Illustration 60: Dans sa plus simple expression, un bus est une série de conducteurs*

À un niveau de complexité supérieur, un bus est un ensemble de connecteurs plus des circuit annexes qui permettent de relier plusieurs périphériques à un microprocesseur ou micro-contrôleur.

Tous les périphériques sont reliés sur le même bus d'adresses et le même bus de données. Pour pas qu'ils n'entrent en conflit :

- Seul le micro-contrôleur écrit sur le bus de contrôle et sur le bus d'adresses.
- Un dispositif spécial, le contrôleur de bus, s'assure que tous les périphériques sauf un, soient en haute impédance par rapport au bus de données (voir point 9.1).

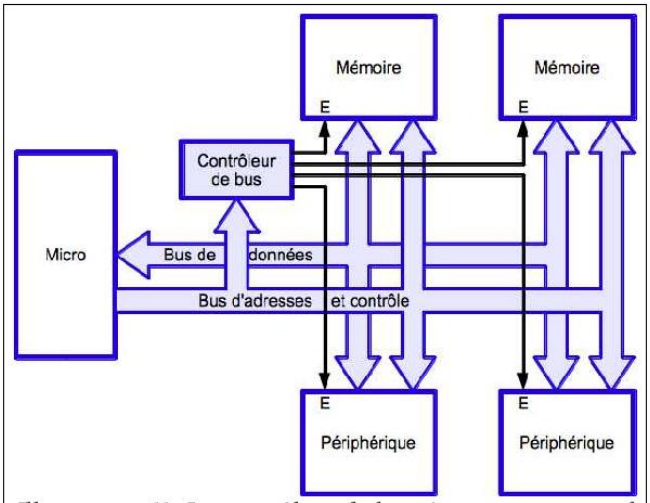



*Illustration 61: Le contrôleur de bus s'assure que un seul des périphériques soit actif.*

Dans cette configuration le microprocesseur est toujours le maître du bus[1].

Les avantages de la structure microprocesseur – bus – périphériques sont les suivants :

- Du point de vue du système informatique, chaque périphérique peut être considéré comme une boîte noire, dans la quelle seuls compte sa connexion au bus, ce qui simplifie la description
- Du point de vue de leur fonctionnement interne, les périphériques sont complètement isolés les un des autres, ce qui réduit les possibilités d'effets de bord.
- Du point de vue de la modularité, ajouter de nouveaux périphériques ne rend pas le système plus complexe.
- Du point de vue du microprocesseur, la totalité des modules de mémoire et périphériques sont une unique grosse mémoire, uniforme et homogène. Ceci standardise (et donc simplifie) la manière de coder l'accès aux périphériques.

## 13.2 Pratique – Contrôleur de bus

Dans les exercices précédents, nous avons connecté au 8051 successivement une mémoire (voir point 10.2), des afficheurs LED 7 segments (voir chapitre 11) et un clavier (voir point 12.2). Avec le bus il est possible de les connecter tous à la fois :

1 À un niveau de complexité encore supérieur, plusieurs maîtres peuvent se connecter au même bus. Dans ce cas, le bus doit implémenter un mécanisme d'arbitrage pour donner le contrôle du bus à un seule maître chaque fois.

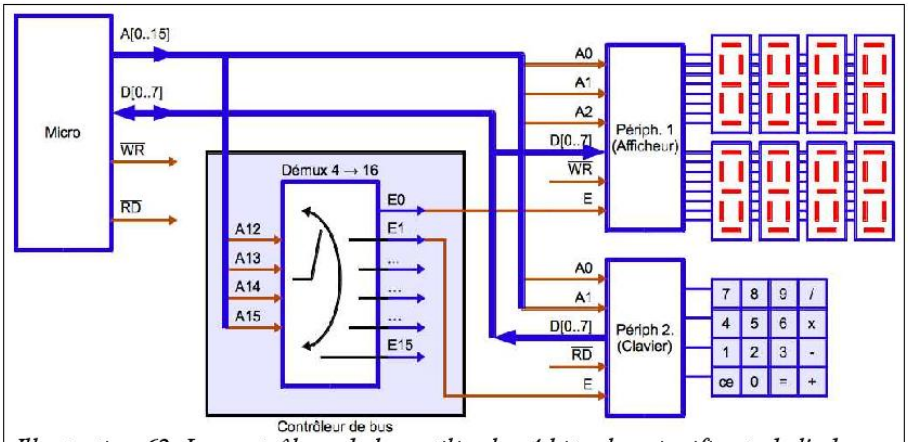



*Illustration 62: Le contrôleur de bus utilise les 4 bits plus signifiants de l'adresse pour activer un des 16 périphériques*

### 13.2.1 Le contrôleur de bus

Le contrôleur de bus, capable de gérer 16 périphériques avec 4096 bytes de mémoire chacun, est implémenté avec un démultiplexeur à 4 entrées et 16 sorties :

- Entrées : `A[11..15]` : en provenance du 8051.
- Sorties : `E0`, `E1`, `E2`... jusqu'à `E15`.

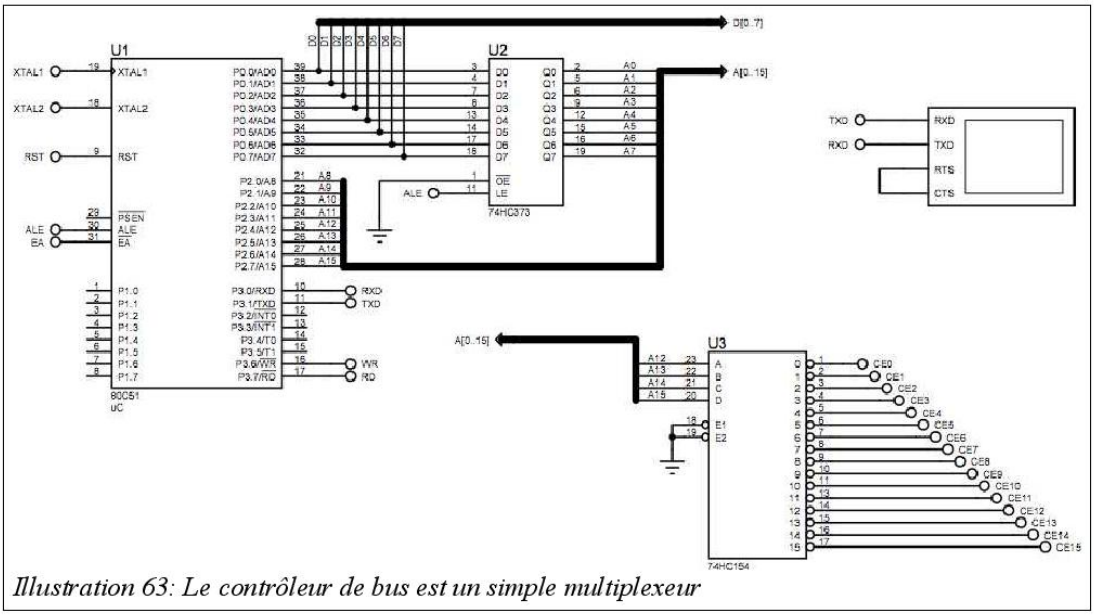



*Illustration 63: Le contrôleur de bus est un simple multiplexeur*

### 13.2.2 Les trois périphériques

Les sorties CE0 à CE15 sont à connecter aux « *chip select* » de chaque périphérique :

- Pour la mémoire, puisqu'elle dispose d'une entrée $\overline{CE}$, on en profite pour y connecter directement $\overline{CE0}$.

- Pour les afficheurs, le bus D[0..7] est connecté sur l'entrée des bascules ; il n'y a donc pas de risques de conflits. Il faut juste éviter qu'une des bascules admette une valeur au mauvais moment. On profite ET logique du multiplexeur pour connecter CE2 sur une des entrées inversées et WR sur l'autre. De cette façon, pour qu'un des bascules admette une valeur il faut que CE2 et WR soient actives en même temps.
- Pour le clavier, la sortie de la bascule est connectée sur le bus D[0..7] ; le risque de conflit existe. Il faut éviter que la bascule interfère avec le bus pendant qu'il établit la partie basse de l'adresse (voir 10.1.2). La porte OU gouverne $\overline{LE}$ pour que la sortie de la bascule reste en haute impédance tant que $\overline{RD}$ et $\overline{CE1}$ ne soient toutes les deux actives.

| $\overline{RD}$ | $\overline{CE1}$ | $\overline{OE}$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

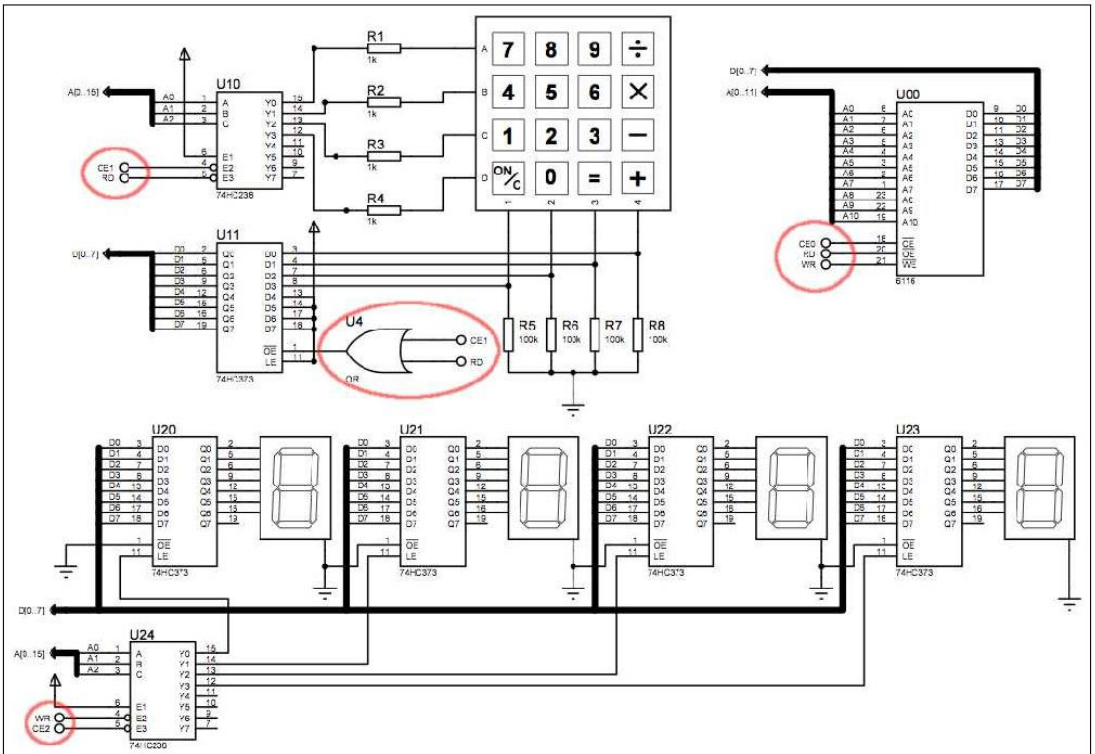



*Illustration 64: Chaque périphérique est connecté à une des sorties du contrôleur de bus*

### 13.2.3 Le programme

Le programme consiste en une concaténation des précédents exemples pratiques, en plus simplifié. À noter que l'initialisation des variables `memoire`, `clavier` et `led7` doit être fait en correspondance avec la connexion des périphériques sur le contrôleur de bus.

```
/**
```

```
 * Point d'entrée du programme.
 * Teste l'ecriture sur les afficheurs led, l'accès à la mémoire
 * et l'accès au clavier.
 */
void main(void)
{
      unsigned char xdata *memoire = 0x0000;
      unsigned char xdata *clavier = 0x1000;
      unsigned char xdata *led7 = 0x2000;

      unsigned char n,v;

      UART_Initialize();

      printf("Affichage 7 segments...");

      // Premier afficheur, pour indiquer que le programme s'exécute:
      led7[0] = 0x06;

      // Écriture dans la mémoire:
      led7[1] = 0x5B;
      for(n=0 ; n<100 ; n++)
      {
            memoire[n] = n*2;
      }

      // Lecture de la mémoire:
      led7[2] = 0x4F;
      for(n=0 ; n<100 ; n++)
      {
            printf("A l'adresse %bu il y a un %bu\r\n", n,
memoire[n]);
      }

      // Test de clavier:
      led7[3] = 0x66;
      while(1)
      {
            for(n=0 ; n<4 ; n++)
            {
                  printf("| %bu = %bu ", n, clavier[n]);
            }
            printf("\r\n");
      }
}
```

# 14. Les test unitaires

## 14.1 Qu'est-ce qu'un test unitaire

En programmation informatique, un test unitaire est une unité de code qui teste une autre unité de code, vérifiant qu'elle fonctionne en accord avec ses spécifications. Une unité de code peut être un module, un service. En programmation procédurale, une unité de code peut être une simple fonction.

Chaque unité du système devrait avoir au moins un test unitaire qui lui soit propre, de façon à faciliter le diagnostique des tests en échec. Si une unité est incorrecte, il est probable que plusieurs tests unitaires échouent : le test unitaire propre à l'unité, plus les tests des unités qui dépendent de l'unité défectueuse, mais il est plus facile de déboguer le test propre à l'unité car il n'est pas nécessaire de descendre profond dans le code.

Lorsqu'un un bug est détecté hors d'un test unitaire, un nouveau test unitaire devrait être immédiatement écrit pour le reproduire. Seulement après le bug est corrigé. De cette façon les tests unitaires finissent par contenir la totalité des spécifications du code principal, dans un langage qui est aussi précis que le code principal.

Ré-écrire du code est une activité courante lorsque le projet grandit : les solutions qui ont été prises initialement, souvent dans un souci d'économie et de vitesse d'implémentation, deviennent simplistes lorsque le projet grandit. Il faut en trouver d'autres, et ré-écrire une partie du code. Les tests unitaires permettent de garantir que le nouveau code respecte les spécifications.

Pour ces deux raisons, et pour autant qu'ils soient correctement utilisés, c'est à dire qu'ils soient mis à jour en même temps que le code, et qu'ils soient complets, les tests unitaires accumulent une valeur inestimable tout au long de la vie d'un projet informatique. Il est généralement conseillé d'écrire les test unitaire en même temps que le code, voir avant le code.

## 14.2 Programmation C – Structure pour tests unitaires

Certains environnements permettent d'inclure les tests unitaires dans le processus de développement[1]. Ce n'est en général pas le cas de C, et en particulier pas le cas de *Keil uVision*. Mais il est tout de même possible de créer, maintenir et exécuter un jeu de tests unitaires de manière satisfaisante, en utilisant les « *directives du préprocesseur* ».

### 14.2.1 Programmation C – Le préprocesseur

Pour la majorité des compilateurs (non seulement les compilateurs C), la première étape de la

1 Typiquement, les tests unitaires sont exécutes à chaque compilation du projet, et il n'est pas possible de déployer de nouvelles versions dans que tous les tests ne passent pas correctement.

compilation est le préprocesseur. Le but d'un préprocesseur est d'éliminer ou d'ajouter du texte dans les fichiers source avant la compilation. Les actions courantes du préprocesseur sont les suivantes :

- Remplacer tous les commentaires par un unique espace.
- Remplacer les sauts de ligne et les espaces supplémentaires de la mise en page, pour laisser une instruction par ligne.
- Remplacer les directives `#include` par les fichiers correspondants.
- Exécuter d'autres directives de précompilation, qui se trouvent mélangée au code source, et qui commencent toutes par le caractère « # ».

Une autre utilisation courante du préprocesseur est la compilation conditionnelle. Les directives de la compilation conditionnelle[1] sont évaluées par le préprocesseur. Selon le résultat de l'évaluation, le code est conservé ou supprimé du fichier source.

```
// Définition de la variable d'environnement directement dans le
// préprocesseur:
#define XXX

#ifdef XXX
      // Le code ici n'est pris en compte que si la
      // variable d'environnement XXX a été définie.
      ...
      ...
#endif

#ifndef XXX
      // Le code ici n'est pris en compte que si la
      // variable d'environnement XXX n'a pas été définie.
      ...
      ...
#endif
```

### 14.2.2 *Keil uVision* – Créer un environnement de test

Dans *Keil uVision* il est possible de définir des variables d'environnement pour le préprocesseur depuis les options pour chaque *target*. Voici la procédure :

1 Ouvrez l'option de menu *Project → Manage → Projects, Environments, Books...*

2 Dans le dialogue correspondant, la liste des *target* se trouve sur la colonne de gauche. Au départ il n'y a que `Target1`.

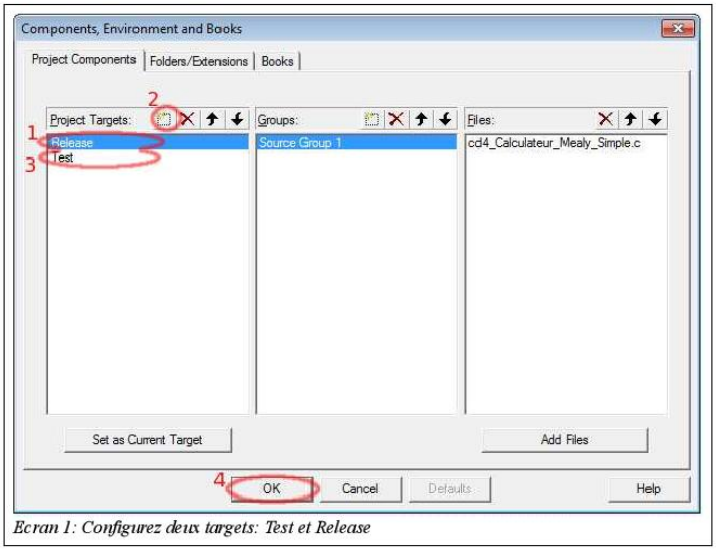
  2.1 Double-cliquez sur `Target1`, et renommez-la `Release`.

  2.2 Puis ajoutez une nouvelle *target* en cliquant sur le bouton « *New (Insert)* ».

  2.3 Appelez-la `Test`

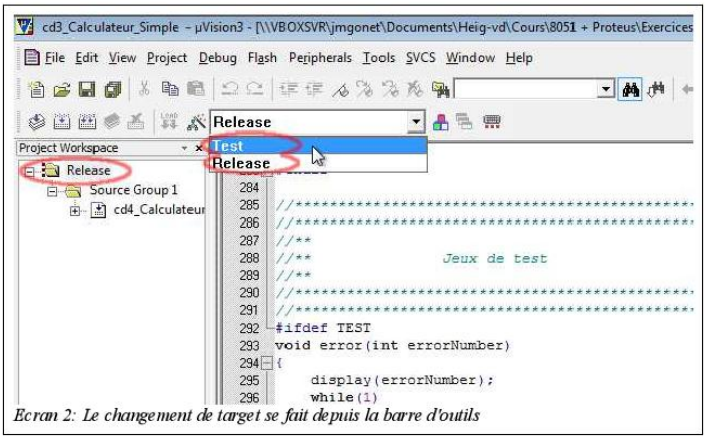
  2.4 Acceptez les changements

1 Les directives de la compilation conditionnelle sont : `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`.



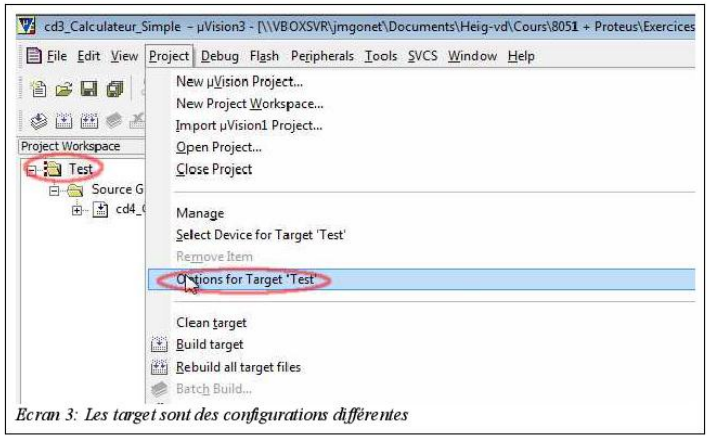
*Ecran 1: Configurez deux targets: Test et Release*

3 Une fois créées, il est possible de changer de *target* directement depuis la barre d'outils.



*Ecran 2: Le changement de target se fait depuis la barre d'outils*

4 Ouvrez les options pour le *target* Test



*Ecran 3: Les target sont des configurations différentes*

5 Changez le fichier de sortie du *target* `TEST`.

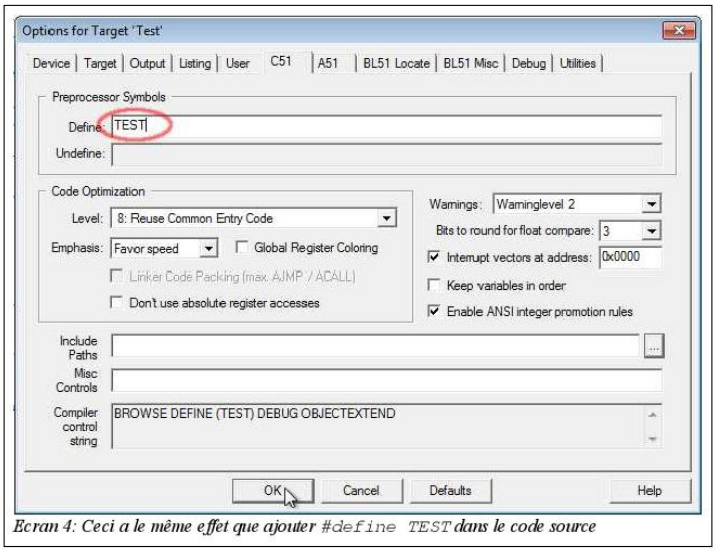
5.1 Allez sur l'onglet « *Output* »

5.2 Dans le champ « *Name of Executable* », donnez un nom reconnaissable au fichier. Par exemple `calculateur_test.omf`.

6 Ajoutez la variable d'environnement `TEST` au *target* `Test`.

6.1 Allez sur l'onglet « *C51* »

6.2 Dans la section « *Preprocessor Symbols* », ajoutez `TEST` dans le champ « *Define* ».



*Ecran 4: Ceci a le même effet que ajouter `#define TEST` dans le code source*

### 14.2.3 Programmation C – Structure de base pour les tests unitaires

Le service minimum pour les tests unitaires est le suivant:

1 Être maintenus en même temps que le code principal.

2 Être faciles à lancer.

3 Voir facilement si tous les tests passent.

4 Voir facilement quel test échoue.

La structure typique d'un programme C (et tout autre langage procédural) est une collection de fonctions, plus un point d'entrée unique (la fonction `main`). Avec les directives de la compilation conditionnelle, il est possible d'avoir deux points d'entrée. Le premier contient l'application principale. Le deuxième contient les tests unitaires :

```
fonction1(argument1, argument2, ...)
{
      ...
}

fonction2(argument1)
{
      ...
}

fonction3(argument1, argument2)
{
      ...
}

#ifndef TEST
void main()
{
      // Ici le point d'entrée de
      // l'application principale.
}
#endif

#ifdef TEST
void main()
{
      // Ici les tests unitaires pour
      // fonction1, fonction2, fonction3, etc.
}
#endif
```

Cette structure a l'avantage de mettre ensemble les fonctions avec les tests unitaires correspondants, ce qui rend la maintenance plus simple, et nous permet d'atteindre le premier objectif.

Pour lancer les tests unitaires depuis *Keil*, il suffit de sélectionner le *target* de test (voir la copie d'écran de la page 91), compiler, puis lancer la simulation sur *Proteus* comme si il s'agissait du programme principal. Mieux encore ; *Keil uVision* possède un débogueur intégré. Il est possible de lancer l'exécution des tests directement (voir 14.2.4)

Finalement nous devons afficher le résultat des tests de façon à retrouver facilement le test en échec. Pour l'instant, la manière la plus simple de communiquer avec l'utilisateur est la méthode `printf` (voir chapitre 7). On peut l'utiliser de la manière suivante :

```
#ifdef TEST
/**
 * Affiche un code d'erreur, puis bloque l'application.
 */
void displayError(int errorNumber)
{
      printf("Test en echec %d", errorNumber);
}

/**
 * Point d'entrée pour les tests unitaires.
 */
```

```
void main()
{
	int value;
	int expectedValue;

	// Un test unitaire:
	expectedValue = 10;
	value = fonctionXXX(x, y, z);
	if (value != expectedValue)
	{
		displayError(1001); // Un code d'erreur unique.
	}

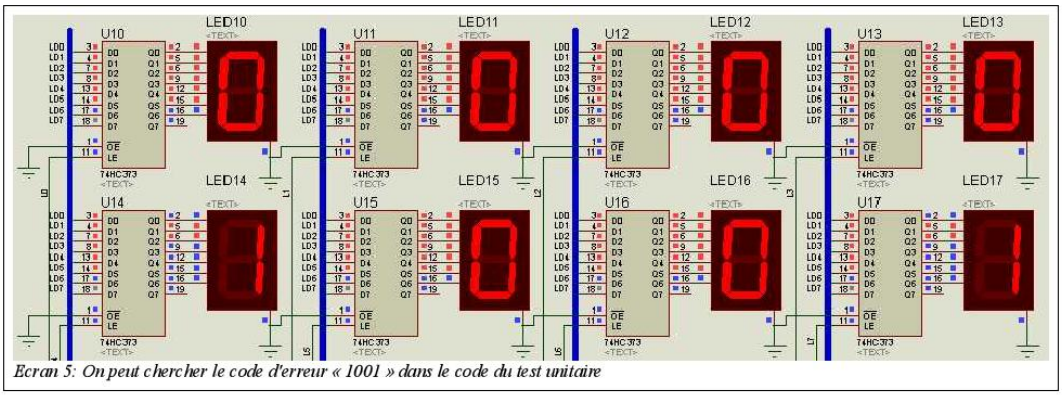
	// Un autre test unitaire:
	...
	...
	if (...)
	{
		displayError(1002);
	}

	// D'autres tests:
	...
	...

	// Si on arrive ici, c'est qu'il n'y a pas eu d'erreur
	displayError(0);
}
#endif
```

Si un test échoue, `displayError` affiche le code d'erreur. Pour retrouver le test en échec, il suffit de chercher le code d'erreur dans le programme.

La raison pour séparer cette fonction est de pouvoir facilement changer son exécution. Par exemple, nous pourrions avoir un environnement de test embarqué sur une circuit intégré spécial qui aurait un afficheur LED comme ci-dessous. Dans ce cas, `displayError` devrait afficher le code d'erreur puis bloquer l'exécution du programme.




*Ecran 5: On peut chercher le code d'erreur « 1001 » dans le code du test unitaire*

Dans le but de rendre plus compact le code des tests unitaires, on ajoute encore une méthode :

```
/**
 * Contrôle l'identité de deux valeurs, et signale un code
 * d'erreur le cas échéant.
 * @param value Première valeur.
 * @param expectedValue Deuxième valeur.
 * @param errorNumber Code d'erreur.
```

```
 **/
void assertEquals(int value, int expectedValue, int errorNumber)
{
      if (value != expectedValue)
      {
            displayError(errorNumber);
      }
}
```

Maintenant, le code du test unitaire est comme ceci :

```
/**
 * Point d'entrée pour les tests unitaires.
 */
void main()
{
      // Un test:
      assertEquals(fonctionXXX(x, y, z), 10, 1001);

      // Un autre test:
      assertEquals(fonctionYYY(x, y, z), 200, 1002);

      // D'autres tests:
      ...
      ...

      // Si on arrive ici, c'est qu'il n'y a pas eu d'erreur
      displayError(0);
}
```

Finalement, en prévision d'un code plus grand, avec de nombreux modules (et sur plusieurs fichiers), on peut grouper les tests d'une unité ensemble. La fonction main appèle les jeux de tests :

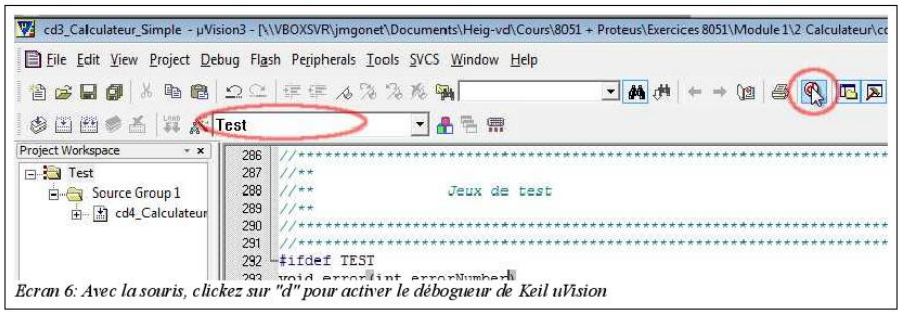
```
/**
 * Tests pour l'unité XXX
 */
void unitXXXTest()
{
      assertEquals(functionXXX(x, y, z), 10, 1001);
      assertEquals(functionXXX(a, b, c), 22, 1002);
      assertEquals(functionXXX(m, n, o), 35, 1003);
}

/**
 * Tests pour l'unité YYY
 */
void unitYYYTest()
{
      assertEquals(functionYYY(0, 0), 0, 2001);
      assertEquals(functionYYY(0, 1), 1, 2001);
      assertEquals(functionYYY(1, 0), 2, 2002);
      assertEquals(functionYYY(1, 1), 3, 2003);
}

/**
 * Point d'entrée pour les tests unitaires.
 */
void main()
{
      unitTestXXX();
```

```
        unitTestYYY();
        displayError(0);
}
```

### 14.2.4 *Keil uVision* – Lancer les tests unitaires directement



*Ecran 6: Avec la souris, clickez sur "d" pour activer le débogueur de Keil uVision*

## 14.3 Concept – Le développement piloté par les tests

Voici la définition qu'en donne Wikipedia[1] :

> *Le « Test Driven Development (TDD) » ou en Français développement piloté par les tests est une méthode de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.*

Le cycle préconisé par TDD comporte cinq étapes :

1. Écrire un premier test ;
2. Vérifier qu'il échoue (car le code qu'il teste n'existe pas), afin de vérifier que le test est valide ;
3. Écrire juste le code suffisant pour passer le test ;
4. Vérifier que le test passe ;
5. Si nécessaire, « *refactoriser* » le code, c'est-à-dire l'améliorer tout en gardant les mêmes fonctionnalités.
6. Ou alors, recommencer le processus pour la fonctionnalité suivante.

## 14.4 Exemple pratique de test unitaire

L'exemple ci-dessous montre un test unitaire pour vérifier le fonctionnement d'un chronomètre.

Pour l'affichage, celui-ci déclare deux deux variables globales, `minutes` et `secondes`, pour compter le temps :

```
/**
 * Le compteur de minutes.
 */
int minutes = 0;

/**
 * Le compteur de secondes.
 */
int secondes = 0;
```

1 La définition et le cycle TDD sont extraits de Wikipédia (http://fr.wikipedia.org/wiki/Test_Driven_Development). Voir également l'article en anglais (http://en.wikipedia.org/wiki/Test-driven_development), plus complet.

Puis une fonction `COMPTEUR_compte` qui, chaque fois qu'elle est appelée, incrémente de 1 le compteur des secondes. Si le compteur des secondes dépasse 59, il revient à 0, et le compteur des minutes est incrémenté.

Voici comment pourrait être le test unitaire pour cette fonction :

```
#ifdef TEST
/**
 * Tests pour la fonction COMPTEUR_compte
 */
void COMPTEUR_compte_unitTest()
{
	unsigned char n;
	for(n=0;n<30;n++)
	{
		COMPTEUR_compte();
	}
	assertEquals(minutes, 0, 1001);
	assertEquals(secondes, 30, 1002);

	for(n=0;n<30;n++)
	{
		COMPTEUR_compte();
	}
	assertEquals(minutes, 1, 1003);
	assertEquals(secondes, 0, 1004);

	for(n=0;n<300;n++)
	{
		COMPTEUR_compte();
	}
	assertEquals(minutes, 5, 1005);
	assertEquals(secondes, 30, 1006);
}
#endif
```

Le chronomètre est implémenté au moyen d'une machine à états. Il a deux boutons :

- Le bouton MARCHEARRET lance le chronomètre ou le met en pause. Lorsque le chronomètre est lancé il utilise COMPTEUR_compte pour compter les secondes. Lorsqu'il est en arrêt, il ne compte pas les secondes.
- Lorsque le chronomètre est en pause, le bouton ZEROLAPS met les minutes / secondes à zéro.

La machine à états pourrait être implémentée comme ceci :

```
#ifdef TEST
/**
 * Tests pour la machine à états du chronomètre.
 */
void chronometre_unitTest()
{
	unsigned char n;

	// Lance le chronomètre, et compte 30 secondes:
	chronometre(MARCHEARRET);
	for(n=0;n<30;n++)
	{
		chronometre(SECONDE);
	}
```

```
	chronometre(ZEROLAPS); // Cette action devrait être ignorée.
	assertEquals(minutes, 0, 2001);
	assertEquals(secondes, 30, 2002);

	// Arrête le chronomètre, et compte 30 secondes:
	chronometre(MARCHEARRET);
	for(n=0;n<30;n++)
	{
		chronometre(SECONDE);
	}
	assertEquals(minutes, 0, 2003);
	assertEquals(secondes, 30, 2004);
	chronometre(ZEROLAPS);
	assertEquals(minutes, 0, 2005);
	assertEquals(secondes, 0, 2006);
}
#endif
```

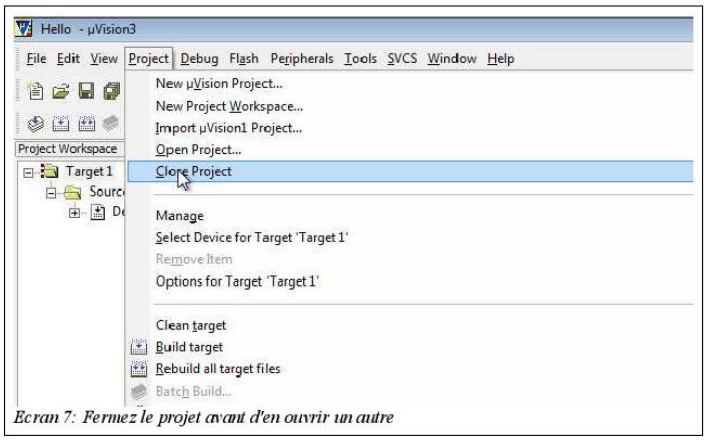
# 15. *Keil uVision*

## 15.1 Introduction

*Keil uVision* est un éditeur et compilateur pour un grand nombre de micro-contrôleurs, dont le 8051. Il existe une version d'évaluation gratuite dont les limitations sont suffisantes pour débuter et écrire les premiers programmes.

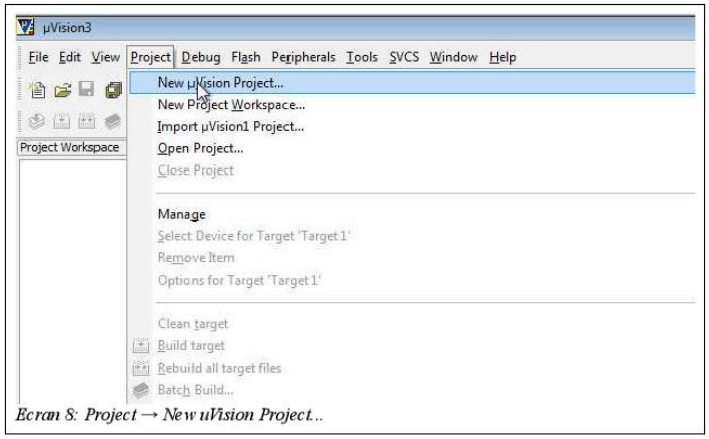
## 15.2 Création et configuration du projet

Lorsque vous lancez l'application, elle s'ouvre automatiquement le dernier projet sur le quel vous avez travaillé. Si c'est le cas, fermez-le :




*Ecran 7: Fermez le projet avant d'en ouvrir un autre*

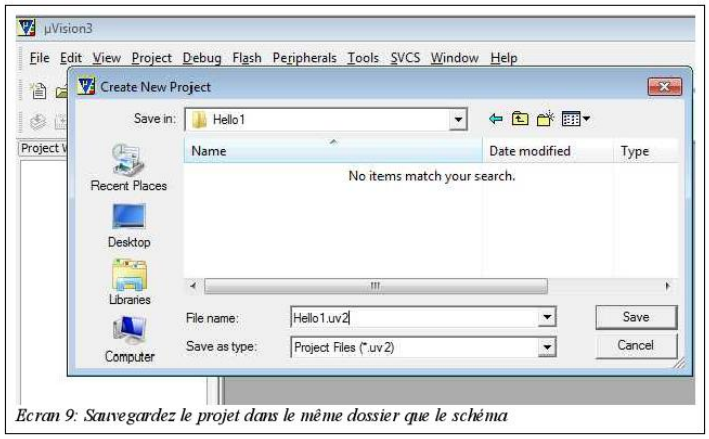
Vous pouvez créer un nouveau projet avec l'option de menu « *File → New uVision Project...* »



*Ecran 8: Project → New uVision Project...*

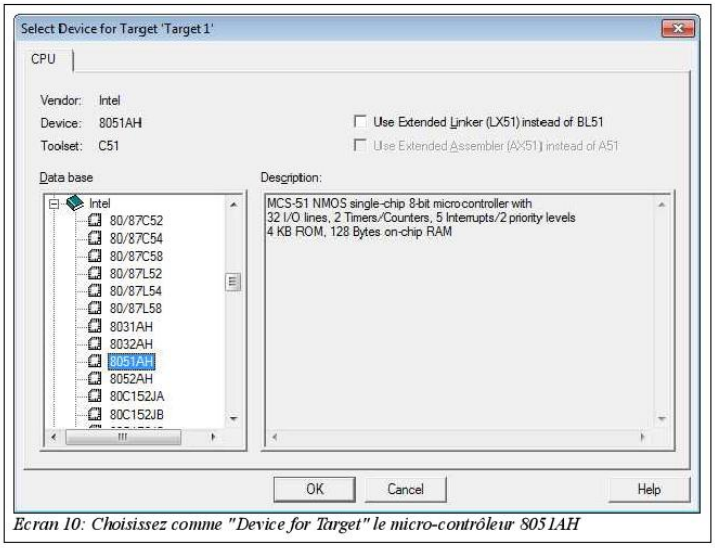
**Note importante : *Proteus* gère mal le fait que le fichier exécutable soit dans un dossier différent que le schéma. Pour cette raison il convient de sauvegarder tous les fichiers dans le même dossier.**

Pour cette raison, vous devez sauvegarder le nouveau projet dans le même dossier où se trouve le schéma *Proteus* (fichier `*.DSN`). À moins d'avoir une bonne raison, donnez au projet le même nom que le schéma :



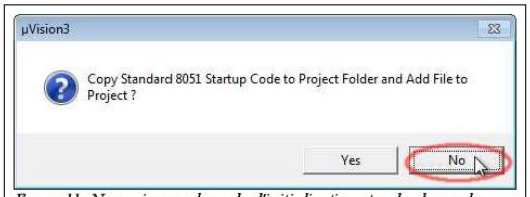
*Ecran 9: Sauvegardez le projet dans le même dossier que le schéma*

À l'étape suivante, choisissez le 8051AH :



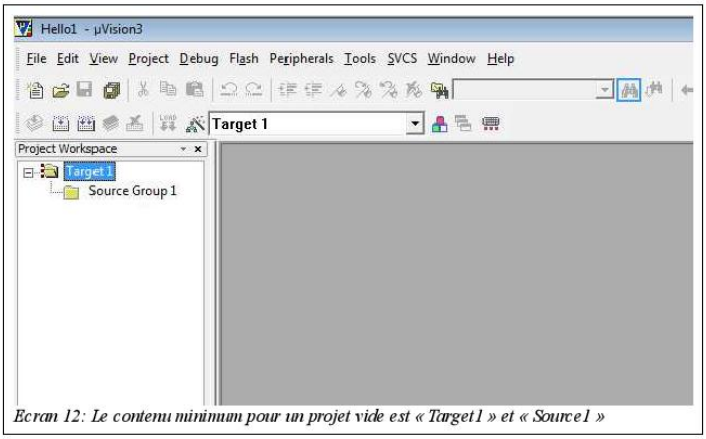
*Ecran 10: Choisissez comme "Device for Target" le micro-contrôleur 8051AH*

Enfin, désactivez le « *Standard 8051 Startup Code* »



*Ecran 11: Ne copiez pas le code d'initialisation standard pour le 8051*

Si tout s'est bien passé, vous devriez voir une arborescence dans la fenêtre « *Project Workspace* » qui contient les éléments « *Target1* » et « *Source1* » :

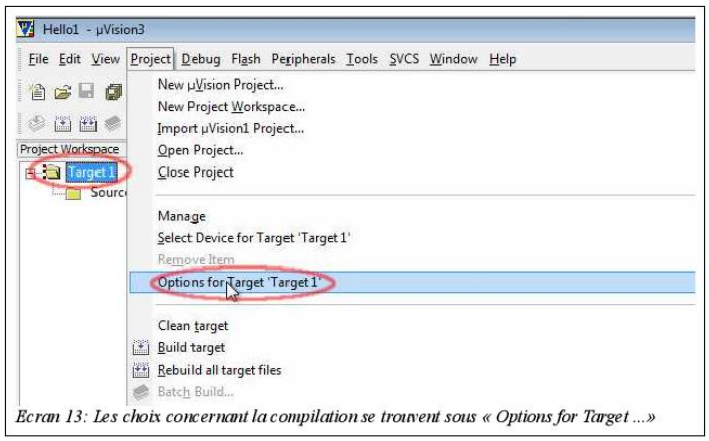


*Ecran 12: Le contenu minimum pour un projet vide est « Target1 » et « Source1 »*

### 15.3 Configurer un projet pour *Proteus VSM*

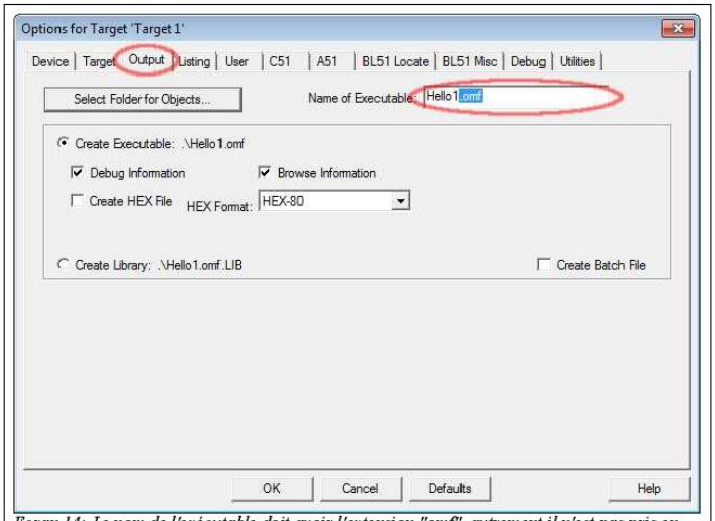
Lors de la compilation, *Keil uVision* permet de choisir plusieurs types de fichiers exécutables. Celui qui convient le mieux pour la simulation dans *Proteus VSM* est le format OFM. Pour configurer le

projet pour produire un fichier exécutable au format « OMF », sélectionnez « `Target1` » puis allez sur « *Project → Options for target1* »




*Ecran 13: Les choix concernant la compilation se trouvent sous « Options for Target ...»*

Allez sur l'onglet « *Output* », puis ajoutez l'extension « `.omf` » sur le champ




*Ecran 14: Le nom de l'exécutable doit avoir l'extension "omf", autrement il n'est pas pris en compte par Proteus*

### 15.4 Ajouter des fichiers dans le projet

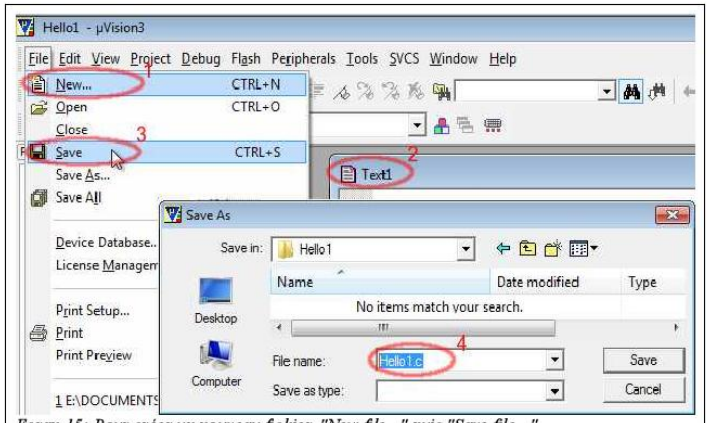
Avant d'ajouter un nouveau fichier dans le projet, il faut le créer. Pour créer un nouveau fichier :

7 Faites *File → New...*

8 Ceci fait apparaître une nouvelle fenêtre avec « `Text1` » comme titre.

9 Puis faites *File → Save*

10 Sauvegardez le fichier avec le même nom que le schéma, mais avec extension « `.c` » (ou « .h » si vous allez créer un fichier d'entête)




*Ecran 15: Pour créer un nouveau fichier, "New file..." puis "Save file..."*

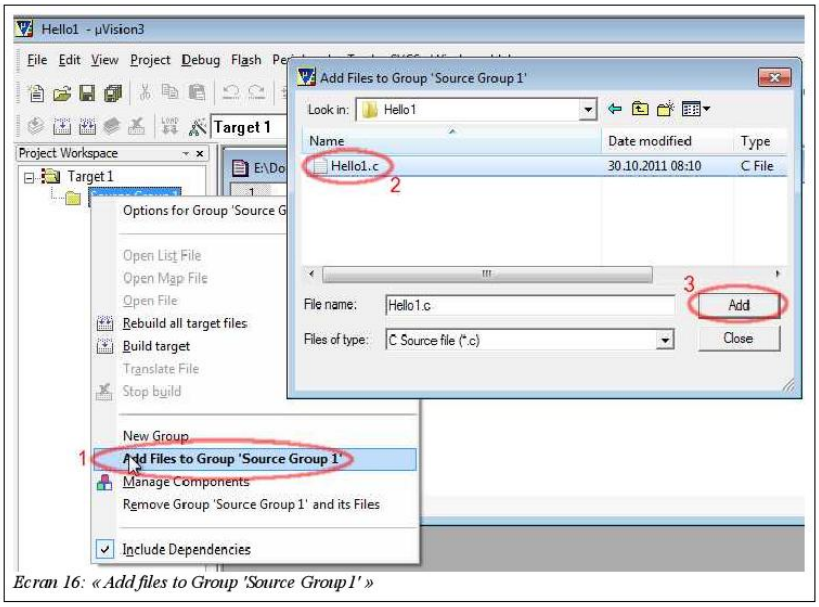
Pour ajouter un fichier existant dans le projet (voir ci-dessus pour créer un nouveau fichier) :

1 Avec le bouton droit, cliquez sur « *Source Group1* », puis choisissez « *Add files to Group 'Source Group1'* »

2 Sélectionnez le fichier que vous venez de créer.

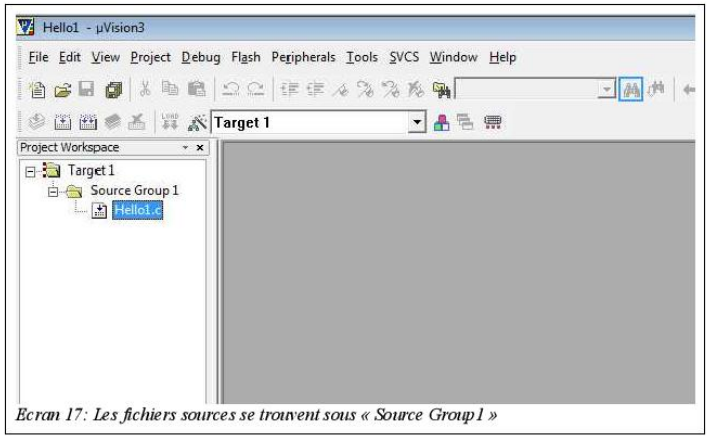
3 Puis cliquez sur le bouton « *Add* ».

4 Puis fermez le sélecteur de fichiers avec « *Close* ».




*Ecran 16: « Add files to Group 'Source Group1' »*

Si tout va bien, le fichier apparaît comme un nouvel élément dans la hiérarchie, en dessous de « Source Group1 »



*Ecran 17: Les fichiers sources se trouvent sous « Source Group1 »*

## 15.5 Quelques particularités du C pour 8051

*Keil uVision* inclut des fichiers d'entête qui définissent la plupart des constantes à utiliser dans les programmes pour le 8051. Le fichier d'entête pour le 8051 est « `reg51.h` ».

Un microcontrôleur n'a pas de système opératif. En conséquence les programmes ne doivent pas se terminer. Si votre programme n'est pas naturellement infini, ajoutez une boucle à la fin de la fonction principale `main`.

Voici un exemple minimaliste :

```
#include <reg51.h>

void main(void)
{
      // Ici le programme:
      ...
      ...
      ...

      // Le programme ne se termine jamais
      while(1)
      {
      }
}
```

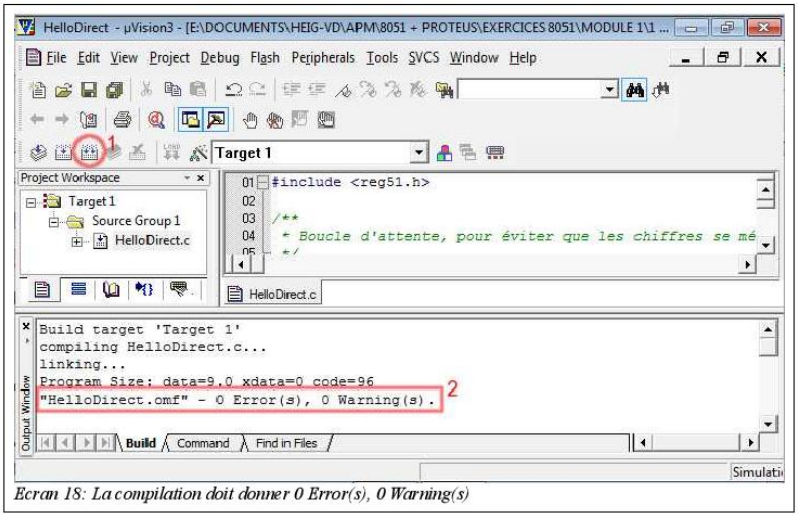
## 15.6 Compilation

1 Cliquez sur le bouton pour lancer la compilation.

2 Vérifiez le dernier message de la simulation :

  2.1 Le nom du fichier doit finir par « `*.omf` »

  2.2 La simulation ne doit donner aucune erreur, ni aucun avertissement : « `Error(s), 0 Warning(s)` »




*Ecran 18: La compilation doit donner 0 Error(s), 0 Warning(s)*

# 16. *Proteus*

## 16.1 Introduction

*Proteus* est un programme qui permet de simuler le fonctionnement d'un circuit électronique. Sur d'autres programmes sur le marché, il a les avantages d'être plutôt simple à utiliser (du moins comparé à d'autres outils équivalents) et de simuler des microcontrôleurs tel que le 8051 et les PIC au niveau du programme.

Il est donc possible de dessiner un circuit, y inclure le microcontrôleur, charger le code exécutable dans le microcontrôleur, et de valider le fonctionnement de l'ensemble.

En général il faut considérer que un circuit qui ne fonctionne pas dans la simulation a peu de chances de fonctionner dans la réalité. Comme tout programme de simulation électronique, *Proteus* est d'un réalisme limité. Il n'est donc pas garanti qu'un circuit qui fonctionne correctement dans la simulation fonctionnera une fois monté en vrai.

## 16.2 ISIS, ARES et VSM

*Proteus* contient 3 programmes différents

- *Proteus ISIS* : Sert à créer des schémas électroniques.
- *Proteus VSM* : Sert à simuler les schémas créés avec ISIS.
- *Proteus ARES* : Sert à produire les transparences pour les circuits imprimés, sur la base des schémas créés avec ISIS.

C'est dans ISIS que nous allons créer et simuler les circuits microcontrôlés.

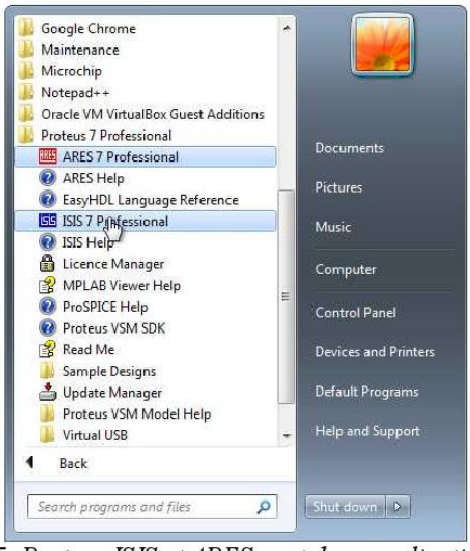



*Illustration 65: Proteus ISIS et ARES sont deux applications distinctes*

Si ISIS et ARES sont des applications distinctes que l'on trouve dans la barre de démarrage de Windows, *Proteus* VSM est une librairie qui se trouve intégrée dans ISIS. La frontière entre les deux n'est pas directement visible, sauf pour le menu d'aide et les boutons de contrôle de simulation, dans la fenêtre principale de ISIS.

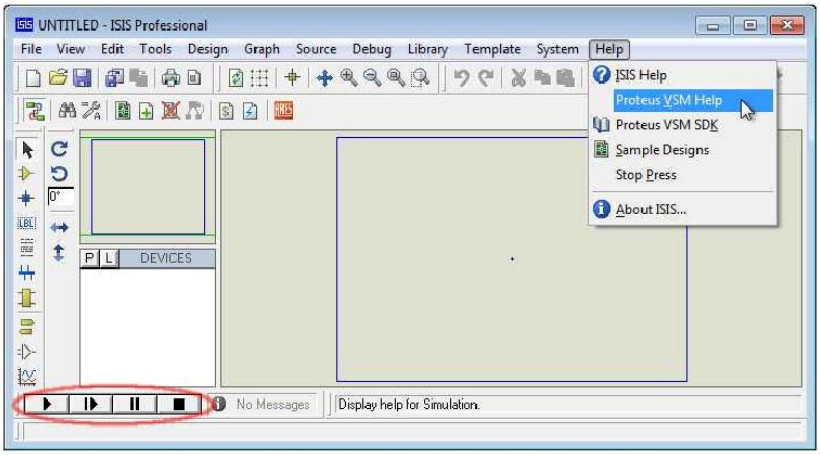



*Illustration 66: Proteus VSM est intégrée dans Proteus ISIS*

## 16.3 Placer les composants

Avant de pouvoir les placer, il faut ajouter les composants dans la liste de composants du circuit. La liste des composants du circuit est composée à partir de la librairie de composants.

### 16.3.1 Choisir les composants du circuit

Pour ouvrir la librairie de composants, cliquez sur l'icône 'P' avec la souris.

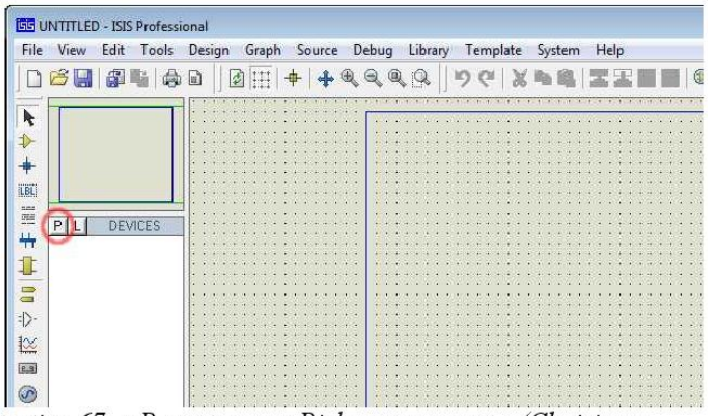



*Illustration 67: « P » est pour « Pick a component » (Choisir un composant)*

Depuis la libraire de composants :

3 Tapez le nom ou une partie du nom du composant dans le champ « *Keywords* ».

4 Puis double-cliquez sur le composant de votre choix dans la liste « *Results* ».

5 Les composants choisis apparaissent dans la liste de composants du circuit, « *DEVICES* ».

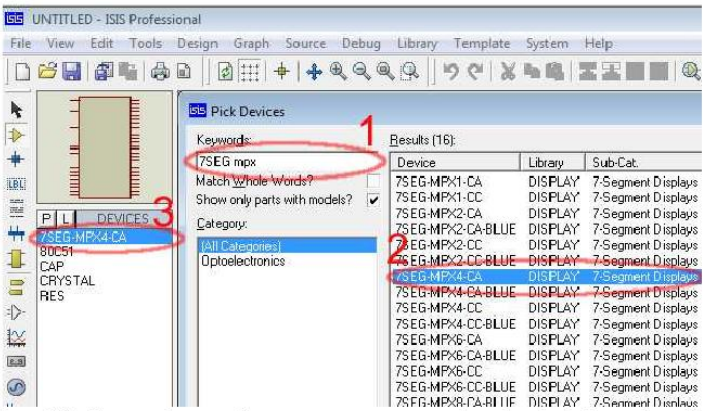



*Illustration 68: Pour ajouter des composants au circuit, choisissez-les de la librairie*

Lorsque vous avez sélectionné tous les composants dont vous avez besoin, vous pouvez fermer la librairie de composants.

### 16.3.2 Placer les composants dans le circuit

Pour placer un composant sur le circuit, sélectionnez-le de la liste « DEVICES », puis posez-le à un endroit approprié.

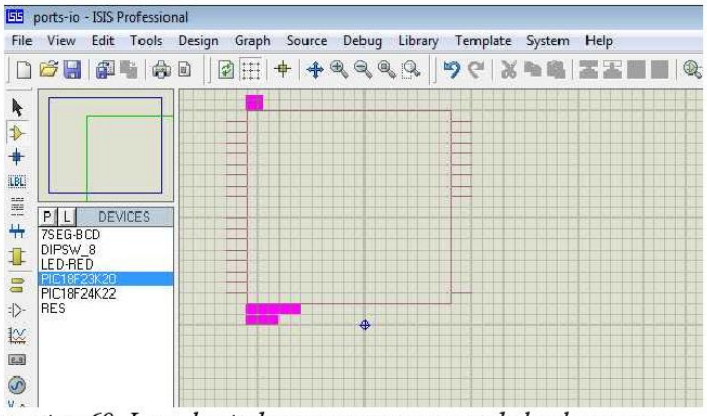



*Illustration 69: Le gabarit du composant permet de le placer avec précision*

### 16.3.3 Remplacer un composant par un autre

Il est possible de remplacer un composant par un autre sur le circuit, pour autant qu'ils aient les mêmes connecteurs. Par exemple, le PIC18F23K20 et le PIC18F23K22 diffèrent uniquement par la mémoire disponible.

Pour remplacer un composant :

1 Sélectionner le nouveau composant.

2 Le placer exactement sur le composant déjà existant.

3 Confirmer « *Replace Component* »

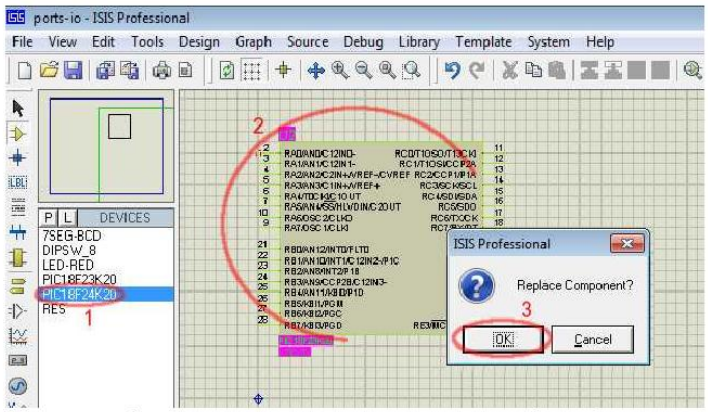



*Illustration 70: Remplacer un composant par un autre, si ils ont les mêmes connecteurs*

## 16.4 Connecter les composants

Les composants sont connectés entre eux soit par des fils conducteurs simples, soit par des bus.

### 16.4.1 Les simples conducteurs

Les simples conducteurs peuvent se tendre entre :

- Les connecteurs des composants.
- Entre un autre simple conducteur et le connecteur d'un composant.

- Entre deux simples conducteurs.

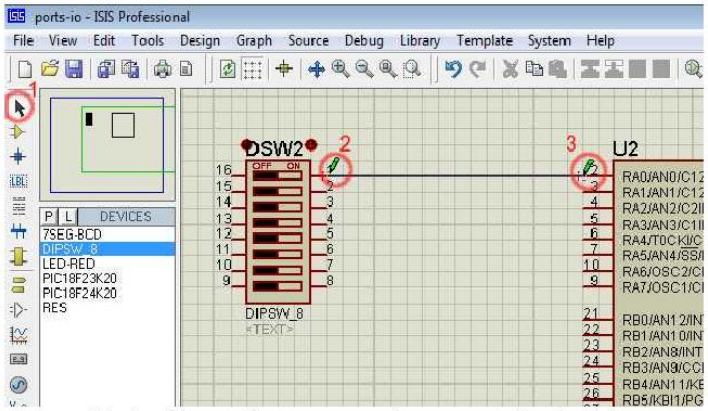



*Illustration 71: Le fils conducteurs simples servent à relier des connecteurs*

Pour placer un conducteur simple :

1 Choisir l'outil pointeur.

2 Pointer avec la souris sur un connecteur. Le curseur devient alors un crayon vert.

3 Tendre le fil jusqu'à un autre connecteur, et cliquer dessus pour effectuer la connexion.

### 16.4.2 Les bus

En microélectronique il est courant de d'avoir des nappes de connecteurs parallèles. Pour éviter de devoir gérer chaque connecteur séparément, il est possible d'utiliser des bus.

Les bus se tendent comme les conducteurs simples, mais ils ne peuvent pas relier directement deux composants. Vous devez vous contenter de les placer à proximité des points de départ et de destination.

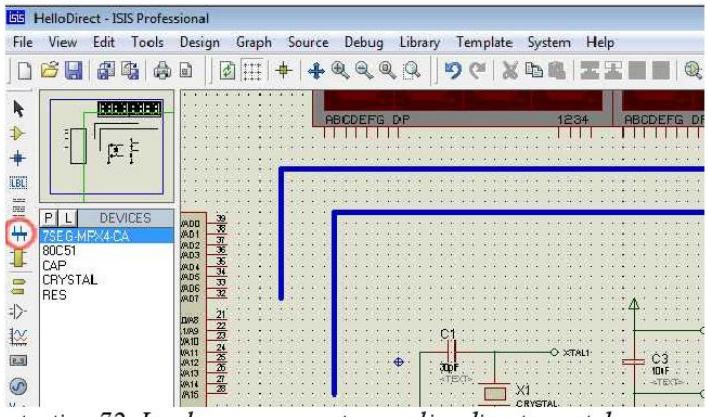



*Illustration 72: Les bus ne peuvent pas relier directement des composants*

Une fois placés, les bus doivent être configurés :

1 Sélectionnez l'outil « LBL » (pour label, qui veut dire étiquette)

2 Cliquez sur le bus.

3 Donnez un nom au bus. Le nom doit garder un format particulier, de la forme XX[N..M] :

3.1 XX est le nom du bus. Le nom ne doit pas se finir par un chiffre. En fait il vaut mieux d'éviter les chiffres dans le nom.

3.2 N est le numéro du premier conducteur. En règle générale, N=0.

3.3 M est le numéro du dernier conducteur.

3.4 Par exemple : D[0..7], A[0..15], DIP[0..7], etc.

4 Le nom du bus est affiché à l'endroit sélectionné.

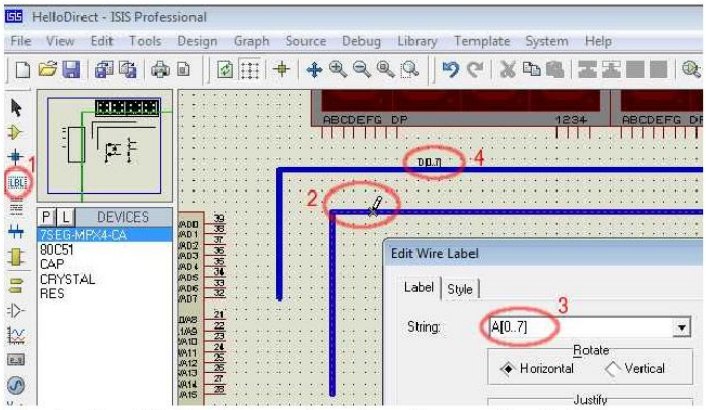



*Illustration 73: Les bus sont configurés avec l'outil LBL*

Finalement il faut relier chaque connecteur du composant au bus, au moyen d'un conducteur simple :

1 Tendez des conducteurs simples entre chaque connecteur et le bus.

2 Avec l'outil « *LBL* », nommez chaque conducteur avec le nom du bus, suivi du numéro du conducteur dans le bus.

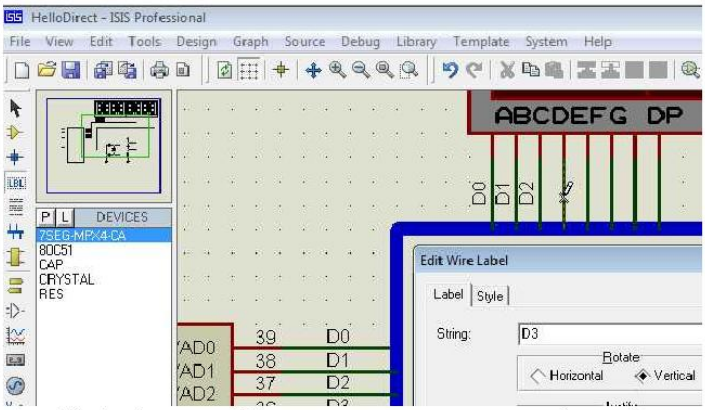



*Illustration 74: Le bus est relié au connecteurs au moyen de conducteurs simples*

### 16.4.3 Les terminaux

Pour placer un terminal :

1 Sélectionnez l'outil « *Terminals* »

2 Placez le terminal sur le circuit, non loin d'un connecteur.

3 Donnez-lui un nom.

4 Vous devez expressément utiliser un conducteur simple pour relier le terminal à un connecteur.

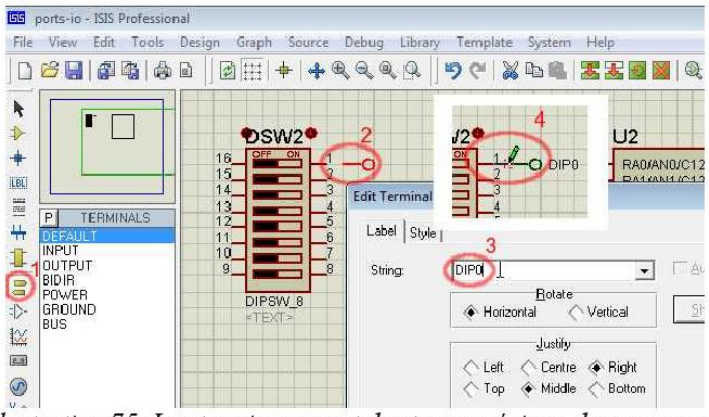



*Illustration 75: Les terminaux sont des types spéciaux de composants*

Du moment que deux terminaux ont le même nom, ils sont considérés comme connectés ensemble. Ceci est pratique pour relier deux points distants sur un circuit.

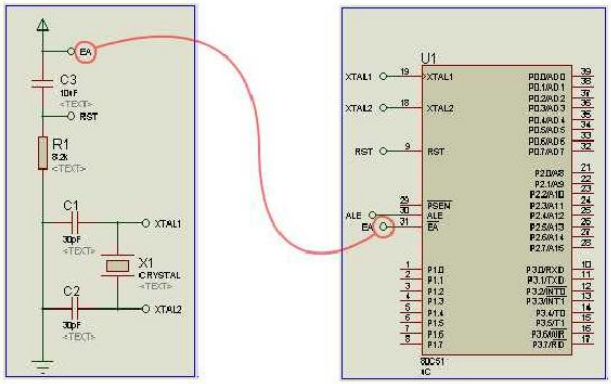



*Illustration 76: Les terminaux du même nom sont considérés comme reliés ensemble*

Les bus ont également des terminaux.

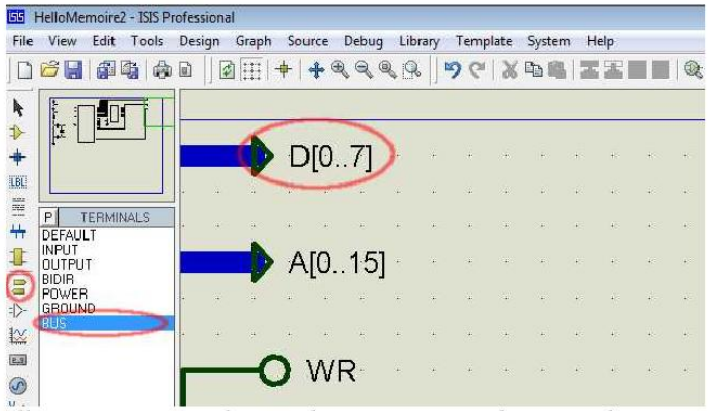



*Illustration 77: Les bus ont leur propre type de terminal*

### 16.4.4 L'alimentation et la terre

L'alimentation et la terre sont des types de terminaux qui représentent le pole + et le pole – de l'alimentation du circuit.

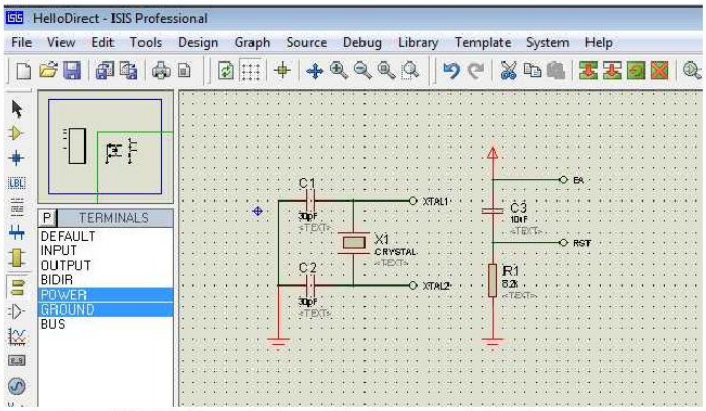



*Illustration 78: L'alimentation et la terre sont des types de terminaux*

## 16.5 Naviguer sur le circuit

Les touches suivantes permettent de cadrer et d'afficher les zones désirées du circuit :

- [F5] : Centre le circuit sur le pointeur de la souris.
- [F6] : Approche le zoom.
- [F7] : Eloigne le zoom.
- [F8] : Ajuste le zoom pour que tout le circuit soit visible.

En cliquant sur la zone supérieure gauche, on peut contrôler la position de la zone visible de l'écran.

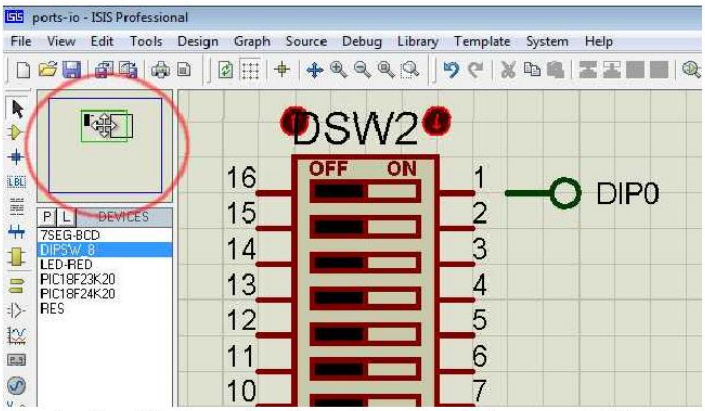



*Illustration 79: Le rectangle vert représente la zone visible du circuit*

### 16.5.1 Gérer la taille de la feuille

Il est possible de spécifier la taille de la feuille pour faire tenir plus de composants dedans

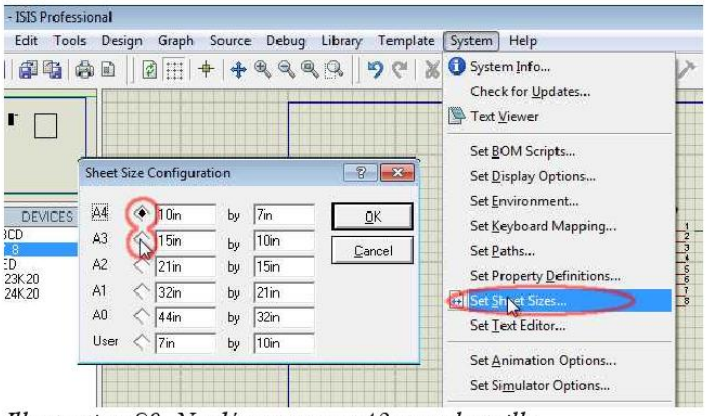



*Illustration 80: Ne dépassez pas A3 pour la taille*

### 16.5.2 Distribuer un circuit sur plusieurs feuilles

Lorsque le circuit devient trop grand, il est possible de le distribuer sur plusieurs feuilles.

- Chaque feuille (« *Sheet* ») devrait contenir un module. Par exemple le microcontrôleur, ou le module de mémoire.
- Il est pratique de mettre les modules clavier et écran sur la même feuille, de manière à pouvoir interagir avec la simulation.

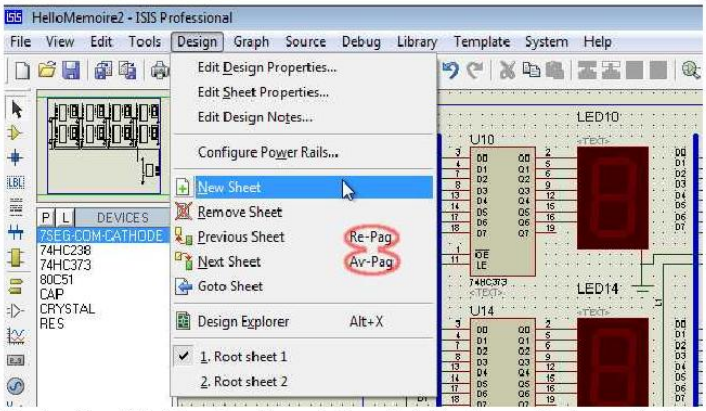



*Illustration 81: Un circuit peut être distribué sur plusieurs feuilles*

Les modules qui se trouvent sur des feuilles différentes sont connectés entre eux au moyen de terminaux.

## 16.6 Simuler le circuit

Pour les composants de type microprocesseur ou microcontrôleur, il est nécessaire de charger le programme exécutable avant de lancer la simulation. *Proteus VSM* supporte les formats des compilateurs les plus populaires :

- Format COFF, produit par le compilateur *XC8* de *MPLABX*, pour les microcontrôleurs *PIC* de *Microchip*.
- Format OMF, produit par le compilateur *Keil uVision* pour les microcontrôleurs 8051.
- Etc[1].

1 *Proteus VSM* supporte également le format HEX, qui est un des standards les plus répandus. Cependant ce format ne

Pendant la simulation, *Proteus VSM* est non seulement capable de reproduire le comportement du programme, il permet également de mettre la simulation en pause et inspecter les variables, exécuter ligne à ligne, etc. Pour que ceci soit possible, *Proteus VSM* a besoin du code source du programme, en plus de l'exécutable.

### 16.6.1 Chargement du programme avec MPLABX

Pour que *Proteus VSM* soit capable de trouver les sources associées au programme exécutable, il est nécessaire que le fichier *.DSN avec le circuit électrique se trouve dans le dossier racine du projet.

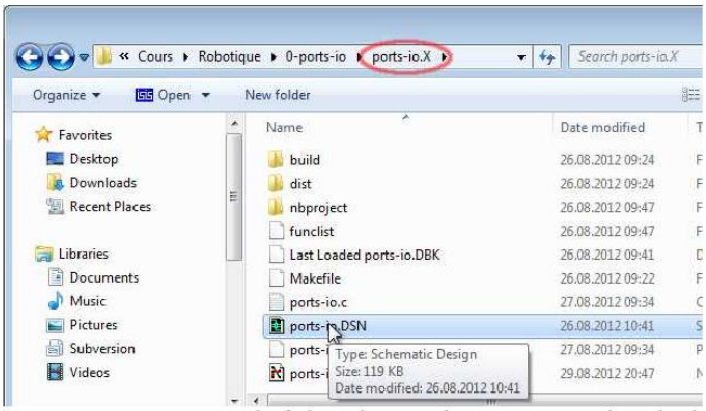



*Illustration 82: Avec MPLABX, le fichier *.DSN doit se trouver dans le dossier projet*

Dans le cas des microcontrôleurs de la famille PIC18, utilisez soit l'assembleur *mpasm*, soit le compilateur *XC8*; les deux outils produisent un fichier *.COF que *Proteus* peut utiliser comme code exécutable. En plus du programme, le fichier *.COF contient les liens nécessaires vers le code source en assembleur ou en C, ce qui permet d'exécuter le programme ligne à ligne.

Pour sélectionner le programme exécutable depuis *Proteus* :

1. Double-cliquez sur le composant.
2. Ouvrez le sélecteur de fichier du champ « *Program File* ».
3. Si elle n'est pas déjà sélectionnée par défaut, choisissez « *Files of type : Program Files* »
4. Naviguez jusqu'au sous-dossier `dist / default / production`, c'est là que MPLAB dépose les fichiers exécutables.
5. Sélectionnez celui de type « *COF File* »
6. Puis faites click sur « *Open* »
7. Et acceptez avec « *OK* ».

contient pas d'informations sur le code source, ce qui le rend peu pratique pour les séances de débogage. À moins que vous ne soyez obligés, n'utilisez donc pas le format HEX.

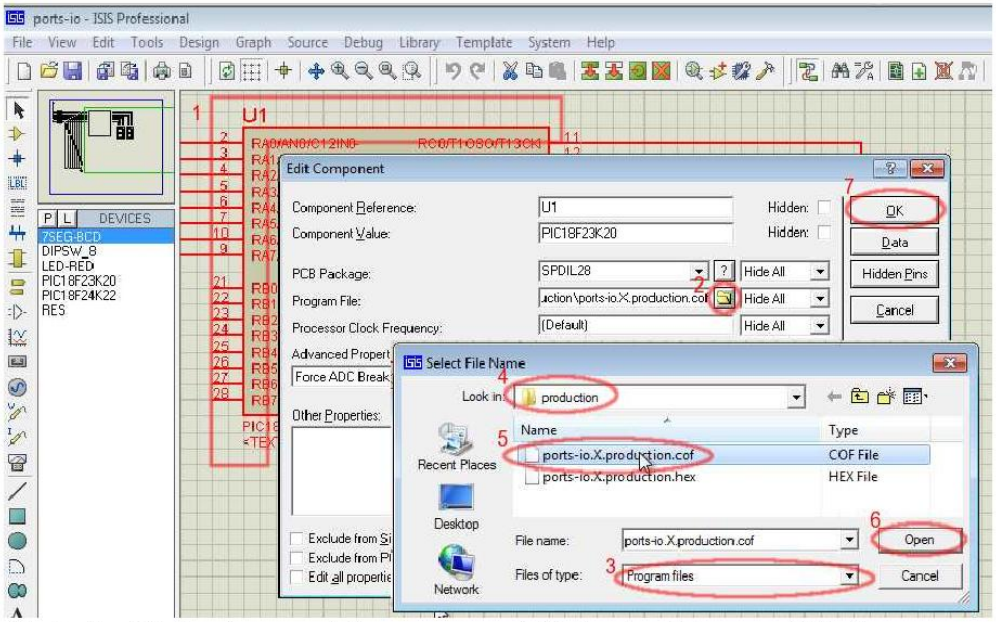



*Illustration 83: Le chargement du programme s'effectue depuis les propriétés du composant*

### 16.6.2 Chargement de programmes avec *Keil uVision*.

Dans le cas de programmes

### 16.6.3 Lancement de la simulation

Une fois le programme exécutable chargé dans le composant, lancez la simulation au moyen des boutons du bas de la fenêtre principale. Vous pouvez constater que la simulation est en cours parce que « ANIMATING » apparaît dans la case de status, au bas de la fenêtre.

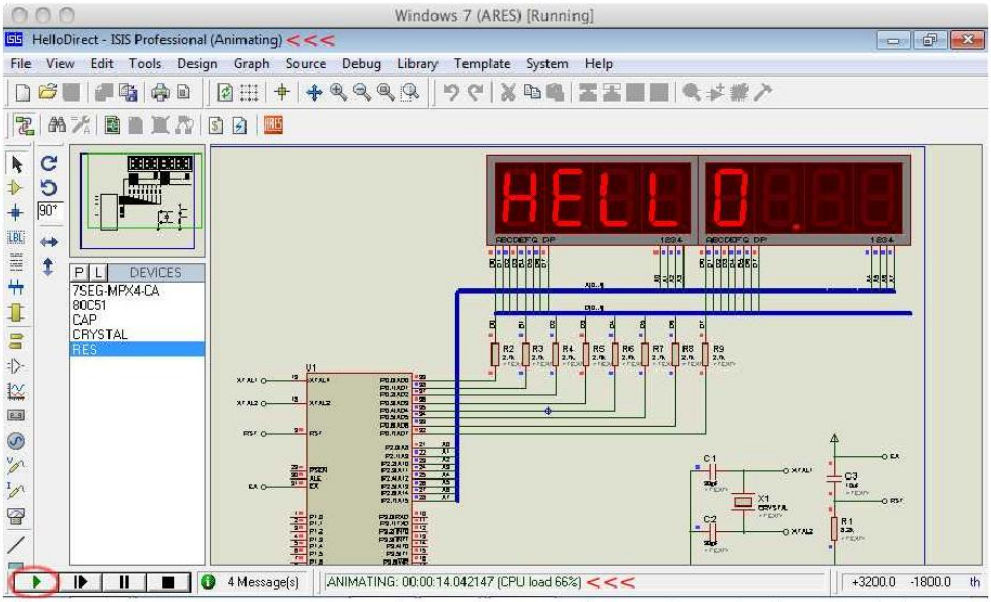



*Illustration 84: Dans Proteus VSM, la simulation est lancée depuis les boutons du bas*

### 16.6.4 Exécution pas à pas

Pour mettre en pause la simulation, utilisez le bouton « Pause », qui se trouve dans la partie inférieur de la fenêtre.

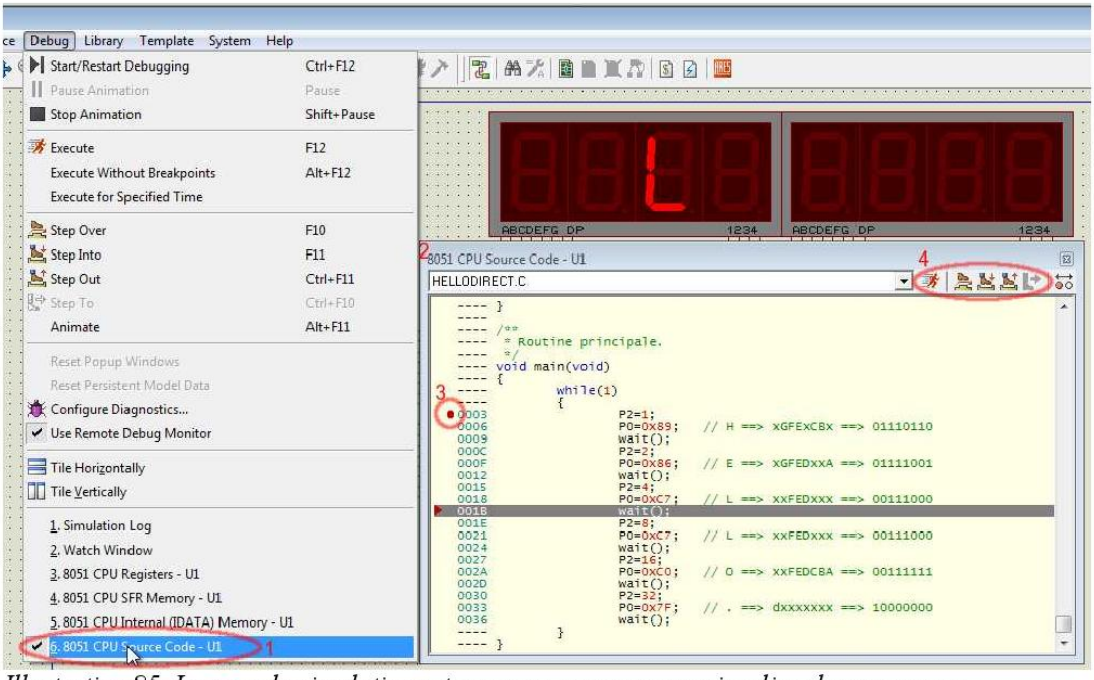



*Illustration 85: Lorsque la simulation est en pause, vous pouvez visualiser le programme*

1 Si elle ne vient pas automatiquement, vous pouvez afficher la fenêtre du code source depuis le menu « *Debug → CPU Source Code* ».

2 Dans la fenêtre qui s'ouvre, vous devriez reconnaître le programme que vous avez écrit depuis l'éditeur.

3 Vous pouvez établir un point d'arrêt avec un double-clic sur le côté droit de la fenêtre de code.

4 Vous pouvez gérer l'exécution pas à pas depuis les boutons de contrôle.

# Appendice A. Table des illustrations

Illustration 1: Le conducteur est le microprocesseur de l'autobus....5
Illustration 2: Sélection du programme, sur une machine à laver....6
Illustration 3: La pomme de Apple est-elle un hommage à Alan Turing?....8
Illustration 4: La Zuse Z1, construite dans l'appartement des parents de Konrad Zuse....9
Illustration 5: Architecture Harvard: le programme est physiquement séparé des données....10
Illustration 6: Architecture von Neumann: le programme et les données sont sur le même support. 10
Illustration 7: Les ports sont des éléments de mémoire connectés au monde extérieur....11
Illustration 8: Vue d'ensemble du 8081....14
Illustration 9: Mémoire de données interne du 8051....14
Illustration 10: Les ports du 8051 sont reliés à la mémoire de données....15
Illustration 11: Schéma pour un connecteur d'un port. Chaque port a huit connecteurs....15
Illustration 12: Pour avoir un "1" en sortie du port, le uC écrit un "1" sur la bascule....16
Illustration 13: Le port 0 (contrairement aux ports 1, 2 et 3) n'a pas de résistance de tirage....16
Illustration 14: Le port 0 peut soit « consommer » du courant, soit « bloquer » le courant. ....17
Illustration 15: Avec les résistances de tirage, le port 0 devient identique aux ports 1, 2 et 3....17
Illustration 16: Configuration minimum pour le 8051....18
Illustration 17: Un « DIP switch » de 8 éléments....19
Illustration 18: Interrupteur sans résistance de tirage....20
Illustration 19: Interrupteur avec résistance de tirage....20
Illustration 20: Afficheur LED 7 segments....21
Illustration 21: Afficheur LED 7 segments en cathode commune et anode commune....21
Illustration 22: Afficheurs 7 segments multiples....22
Illustration 23: Schéma interne d'un afficheur 7 segments multiple en anode commune....22
Illustration 24: Le décodeur BCD 7 segments....24
Illustration 25: Les interrupteurs sont connectés au port1, et l'afficheur au port 0....25
Illustration 26: L'interrupteur directement connecté à l'afficheur....26
Illustration 27: P0 et P1 sont des variables branchées sur les connecteurs....27
Illustration 28: Les temporisateurs peuvent compter les transitions sur T0 et T1....29
Illustration 29: Diagramme de fonctionnement des temporisateurs....30
Illustration 30: Diagramme de fonctionnement des interruptions....34
Illustration 31: Les connecteurs INT1 et INT0 peuvent déclencher des interruptions....34
Illustration 32: L'interrupteur sur l'entrée INT1 sert à simuler les interruptions externes....37
Illustration 33: Le comptage se manifeste sur l'afficheur 7 segments....39
Illustration 34: La UART est branchée sur les connecteurs TXD et RXD....41
Illustration 35: Diagramme de fonctionnement de la UART....42
Illustration 36: Le terminal virtuel de Proteus est compatible avec le 8051....45

Illustration 37: Le terminal virtuel se trouve dans une librairie séparée ..... 46
Illustration 38: Le résultat de printf est envoyé au terminal ..... 48
Illustration 39: L'état de haute impédance agit comme un interrupteur ..... 55
Illustration 40: Les sorties à trois états peuvent partager les mêmes connecteurs ..... 56
Illustration 41: La bascule type D se comporte comme une mémoire de 1 bit ..... 59
Illustration 42: Le 74hc373 contient 8 verrous de type D. ..... 59
Illustration 43: Un multiplexeur connecte plusieurs entrées à une seule sortie ..... 60
Illustration 44: Un démultiplexeur peut implémenter une opération booléenne en entrée ..... 60
Illustration 45: 74hc238 et 74hc154 sont deux exemples de décodeurs ..... 61
Illustration 46: En mode mémoire externe, le Port 0 utilise un deuxième transistor ..... 64
Illustration 47: En mode mémoire externe, « 0 » est court-circuité à terre, « 1 » est court-circuité à Vcc ..... 64
Illustration 48: Chronogramme du cycle d'écriture ..... 65
Illustration 49: Circuit minimal pour brancher une mémoire externe de 2 Ko sur le 8051 ..... 66
Illustration 50: La mémoire est peuplée avec les valeurs dans la boucle ..... 67
Illustration 51: Petite mémoire de 4 octets, avec des afficheurs 7 segments ..... 69
Illustration 52: Les afficheurs 7 segments montrent ce qui est stocké dans la mémoire ..... 70
Illustration 53: Un clavier est une matrice d'interrupteurs organisés en lignes et en colonnes ..... 71
Illustration 54: Les résistances de tirage servent à éviter les états de haute impédance (voir point 3.2) ..... 72
Illustration 55: Deux touches enfoncées peuvent provoquer un court-circuit ..... 72
Illustration 56: KEYPAD-SMALLCALC, KEYPAD-CALCULATOR et KEYPAD-PHONE ..... 73
Illustration 57: Le clavier est branché au microcontrôleur comme une mémoire ..... 74
Illustration 58: Les valeurs représentent l'état des touches ..... 75
Illustration 59: Dans sa plus simple expression, un bus est une série de conducteurs ..... 77
Illustration 60: Le contrôleur de bus s'assure que un seul des périphériques soit actif. ..... 78
Illustration 61: Le contrôleur de bus utilise les 4 bits plus signifiants de l'adresse pour activer un des 16 périphériques ..... 79
Illustration 62: Le contrôleur de bus est un simple multiplexeur ..... 79
Illustration 63: Chaque périphérique est connecté à une des sorties du contrôleur de bus ..... 80
Illustration 64: Proteus ISIS et ARES sont deux applications distinctes ..... 92
Illustration 65: Proteus VSM est intégrée dans Proteus ISIS ..... 92
Illustration 66: « P » est pour « Pick a component » (Choisir un composant) ..... 93
Illustration 67: Pour ajouter des composants au circuit, choisissez-les de la librairie ..... 93
Illustration 68: Le gabarit du composant permet de le placer avec précision ..... 94
Illustration 69: Remplacer un composant par un autre, si ils ont les mêmes connecteurs ..... 94
Illustration 70: Le fils conducteurs simples servent à relier des connecteurs ..... 95
Illustration 71: Les bus ne peuvent pas relier directement des composants ..... 95
Illustration 72: Les bus sont configurés avec l'outil LBL ..... 96
Illustration 73: Le bus est relié au connecteurs au moyen de conducteurs simples ..... 96
Illustration 74: Les terminaux sont des types spéciaux de composants ..... 97
Illustration 75: Les terminaux du même nom sont considérés comme reliés ensemble ..... 97
Illustration 76: Les bus ont leur propre type de terminal ..... 97
Illustration 77: L'alimentation et la terre sont des types de terminaux ..... 98
Illustration 78: Le rectangle vert représente la zone visible du circuit ..... 98
Illustration 79: Ne dépassez pas A3 pour la taille ..... 99
Illustration 80: Un circuit peut être distribué sur plusieurs feuilles ..... 99
Illustration 81: Avec MPLABX, le fichier *.DSN doit se trouver dans le dossier projet ..... 100
Illustration 82: Le chargement du programme s'effectue depuis les propriétés du composant ..... 101
Illustration 83: Dans Proteus VSM, la simulation est lancée depuis les boutons du bas ..... 102
Illustration 84: Lorsque la simulation est en pause, vous pouvez visualiser le programme ..... 102