



UE A211 : SYSTÈMES EMBARQUÉS I

Professeur : COSTA EMILE

RAPPORT DE LABORATOIRE

Etudiants : KORKUT CANER

TP4 - Gestion des interruptions

Table des matières

1	Objectif du TP	2
2	Introduction	3
2.1	Polling VS Interruptions	3
2.2	Afficheurs 7 segments et <i>multiplexage</i>	4
3	Schéma de principe	5
4	Algorigramme	7
5	Code source	8
6	Analyse du code source	10
6.1	Variables globales	10
6.2	Séquence d'initialisation	11
6.3	Programme principal	11
7	Conclusion	14
A	Annexe : Afficheur 7 segments et <i>multiplexage</i>	15
B	Annexe : Données provenant du datasheet	17
C	Annexe : Logiciel <i>Timer Calculator</i> et calcul des valeurs associées	18

1 Objectif du TP

Le but de cette manipulation est de nous permettre de nous familiariser à la notion d'*interruptions*. Il s'agit d'un mécanisme propre aux microprocesseurs et non au PIC18. Nous verrons son principe de fonctionnement et illustrerons l'utilisation de ce mécanisme à travers l'affichage d'une valeur entière dans un afficheur de 4 blocs de 7 segments.



FIGURE 1 – Exemple d'afficheur de 4 blocs de 7 segments

2 Introduction

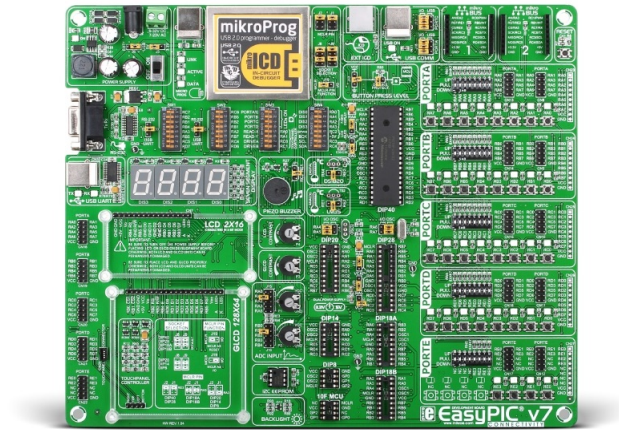


FIGURE 2 – Carte de développement EasyPic 7

2.1 Polling VS Interruptions

Le *Polling* ou encore *scrutement* est un mécanisme qui consiste à bloquer l'exécution du programme en cours et à interroger l'état des pins (ou toute autre événement) afin de lancer une procédure liée à un événement. Dans le cas de l'utilisation du scrutement, il va de soi qu'il ne s'agit pas d'une manière efficace de programmer. En effet, non seulement il consomme un certain temps de traitement pour la surveillance et risque de nous faire rater un court processus qui peut nécessiter une attention immédiate. Pour parer à ce problème, nous utiliserons les **interruptions**.

Comme précité, l'interruption est un mécanisme propre aux processeurs. Il s'agit d'une routine¹ provoquée par un *événement* interne (software) ou externe (hardware). Cette interruption a pour tâche d'arrêter le déroulement normal du programme afin de sauter dans une adresse mémoire contenant un programme lié à l'événement ayant provoqué l'interruption. Lorsque la routine d'interruption est terminée, le programme reprend son exécution à l'adresse où il a été interrompu.

Bien qu'il existe plusieurs niveaux de priorités (ou d'*hiérarchisation*) d'interruptions, nous ne verrons dans le cadre du cours qu'un seul niveau de priorité.

1. Egalement appelé *routine de service d'interruption* ou **ISR**

2.2 Afficheurs 7 segments et *multiplexage*

Le principe de fonctionnement d'un afficheur 7 segments est assez trivial. Il s'agit d'alimenter les pins nécessaires à l'affichage de la valeur *unitaire* souhaitée. Ainsi, voici le schéma permettant d'afficher la valeur **274** :

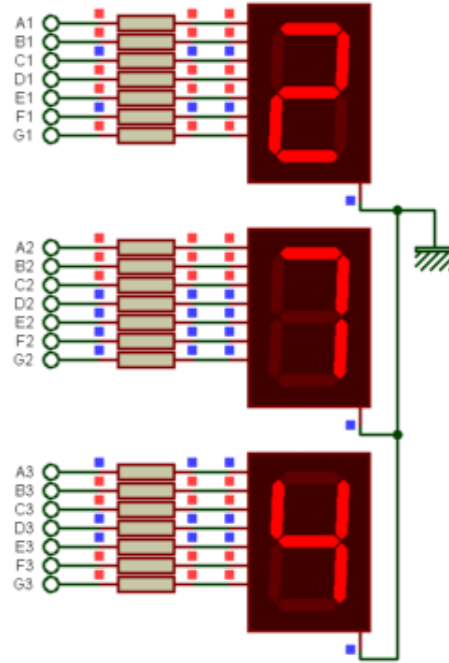


FIGURE 3 – Afficheurs 7 segments **sans** multiplexage

Il va de soi que le nombre de pins mobilisées par l'afficheur sera aussi élevé que la valeur à afficher. Ainsi, pour afficher un nombre à 3 digits, nous mobiliserons un total de $4 \times 7 = 28$ pins. Cela pose un problème évident et le moyen de contourner la situation passe par le **multiplexage**. De manière générale, il s'agit d'une méthode de transmission de données qui consiste à envoyer les bits de données sur un même support.

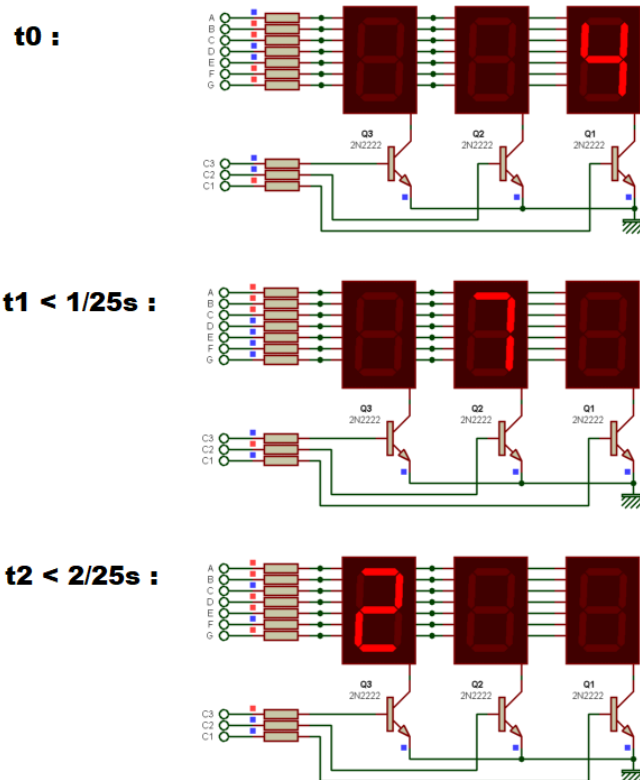


FIGURE 4 – Afficheurs 7 segments **avec** multiplexage

Dans notre cas, nous ferons appel à cette méthode pour sélectionner l'écran sur lequel afficher la valeur. En sélectionnant l'écran adjacent à chaque affichage de la valeur, il est possible d'afficher une valeur différente sur chacun des afficheurs. Ainsi lorsque cette fréquence atteint ou dépasse la fréquence de la persistance rétinienne (qui est de $\frac{1}{25}s \approx 0.05s$), il est aisé d'apercevoir une valeur uniforme à plusieurs digits.

3 Schéma de principe

Le schéma de principe ainsi que la simulation du programme a été réalisé à domicile. Le logiciel **Proteus** nous a été nécessaire afin d'y parvenir. Pour rappel, il s'agit d'un logiciel de conception de CI et de simulation. Le branchement de l'afficheur peut

Les composants que nous avons utilisés pour réaliser cette manipulation sont

les suivants :

- PIC18F45K22 ;
- Bloc de 4 afficheurs 7 segments à anode commune ;
- 4 Inverseur ;
- 7 Interrupteurs avec résistances $1K\Omega$;
- Alimentation 5V.

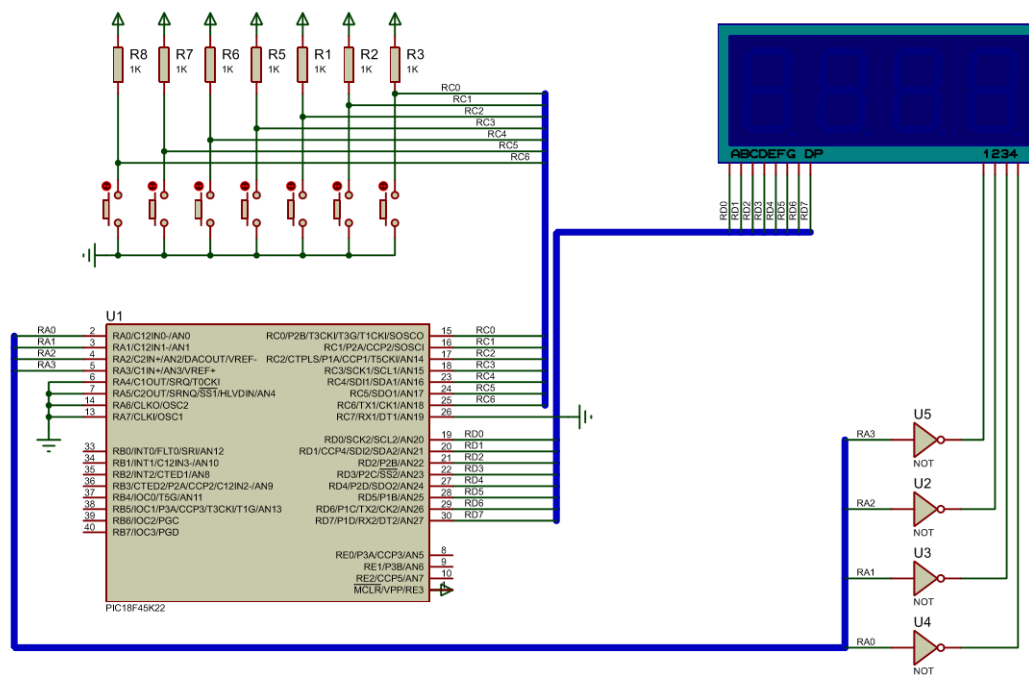


FIGURE 5 – Schéma de principe sous Proteus

4 Algorithme

5 Code source

```
1  /*
2   Fichier: Afficheur7Segments.c
3  */
4
5  // Variables globales
6  unsigned int  compteur, millier, centaine, dizaine, unite;
7  unsigned char index;
8  unsigned char Segment[4]={0,0,0,0};
9
10 // Constantes
11 const unsigned char conversion[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0
    x7D,0x07,0x7F,0x6F};
12
13 bit oldState;
14 int i;
15
16 // Gestion des interruptions
17 void Interrupt(void){
18     if (TMR0IF_bit){ // Timer0 toutes les 5ms
19         TMR0IF_bit = 0;
20         TMR0H = 0xD8;
21         TMR0L = 0xF0;
22         LATA = 0;
23         LATD = Segment[index];
24         LATA = 1 << index;
25         index++;
26         index = index%4;
27     }
28 }
29
30 // Fonction: Update_display
31 void Update_display(void)
32 {
33     millier = (compteur/1000)%10;
34     Segment[3] = conversion[millier];
35     centaine = (compteur/100)%10;
36     Segment[2] = conversion[centaine];
37     dizaine = (compteur/10)%10;
38     Segment[1] = conversion[dizaine];
39     unite = compteur%10;
40     Segment[0] = conversion[unite];
```

```

41 }
42
43 void setValue(int n){
44     if (n == 0 && compteur +1 < 1000){
45         compteur += 1;
46     }
47     else if (n == 1 && compteur -1 > 0){
48         compteur -= 1;
49     }
50     else if (n == 2 && compteur +10 < 1000){
51         compteur += 10;
52     }
53     else if (n == 3 && compteur -1 > 0){
54         compteur -= 10;
55     }
56     else if (n == 4 && compteur +100 < 1000){
57         compteur += 100;
58     }
59     else if (n == 5 && compteur -1 > 0){
60         compteur -= 100;
61     }
62     else{
63         compteur = 0;
64     }
65 }
66
67 void update_value(void){
68     for (i = 0; i < 6; i++){
69         if (Button(&PORTC, i, 1, 1)){
70             oldState = 1;
71         }
72         if (Button(&PORTC, i, 1, 0) && oldState){
73             setValue(i);
74         }
75     }
76 }
77
78 //Fonction principale
79 void main(void) {
80     ANSELA = 0;
81     ANSELD = 0;
82     TRISA = 0;
83     LATA = 0;

```

```

84 TRISD = 0;
85
86 TRISC = 0b01111111;
87 ANSELC = 0;
88 oldState = 0;
89
90 LATD = 0;
91 compteur = 0;
92 index = 0;
93 // Timer 0
94 TOCON = 0x88;
95 TMR0H = 0xD8;
96 TMR0L = 0xF0;
97 INTCON.GIE = 1;
98 INTCON.TMR0IE = 1;
99
100 //Programme :
101 while(1) {
102     Update_display();
103     update_value();
104     delay_ms(100);
105 }
106 }

```

6 Analyse du code source

Dans cette section, nous analyserons le code source associé au point précédent. Nous nous focaliserons plus précisément sur deux points à savoir : **les interruptions** et **le multiplexage**.

6.1 Variables globales

Les variables qui nous ont semblées pertinentes d'analyser sont

```

7 [...]
8 unsigned char Segment[4] = {0,0,0,0};
9
10 // Constantes
11 const unsigned char conversion[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

```

La variable $Segment[4]=\{0,0,0,0\}$ est un tableau auquel on fera appel pour afficher les différents chiffres qui composent notre nombre. La constante $conversion[]$

$= \{0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F\}$ est un tableau représentant les valeurs hexadécimales et nous servira pour la partie de la programmation du multiplexage. Voir annexe A

6.2 Séquence d'initialisation

Avant d'analyser le programme principal, voyons quels sont les nouveaux éléments qui composent la séquence d'initialisation :

```
93 //Timer0
94 TOCON = 0x88;
95 TMR0H = 0xD8;
96 TMR0L = 0xF0;
97 INTCON.GIE = 1;
98 INTCON.TMR0IE = 1;
```

TOCON = 0x88 est le registre associé au Timer0 (voir annexe B). Ensuite nous avons les instructions TMR0H = 0xD8 et TMR0L = 0xF0. Pour comprendre ces lignes, référons-nous au point 11.5 du datasheet situé à l'annexe B :

[...] En mode 8 bits (qui est la valeur par défaut), un débordement dans le registre TMR0 (FFh → 00h) mettra le bit indicateur TMR0IF à 1. En mode 16 bits, un débordement dans la paire de registres TMR0H et TMR0L (FFFFh → 0000h) mettra TMR0IF à 1. L'interruption peut être activée ou désactivée en mettant à 1 ou 0 le bit d'activation TMR0IE du registre INTCON. [...].²

Ainsi, les instructions INTCON.GIE = 1 et INTCON.TMR0IE = 1 permettent d'autoriser ou d'interdire les interruptions. Toute la partie qui précède ainsi que les valeurs de ces registres ont été fixées par le logiciel TIMER CALCULATOR de MIKROC. Le calcul effectué par ce dernier est disponible à l'annexe C.

6.3 Programme principal

Regardons maintenant le contenu du programme principal :

```
101 //Programme :
102 while(1) {
103     Update_display();
```

2. Traduit par <https://www.deepl.com>, le 8 mars 2020

```

104     update_value();
105     delay_ms(100);
106 }
107 }

```

La fonction `void update_value()` ayant été analysée dans un travail ultérieur, nous nous focaliserons ici sur la fonction `void Update_display()` :

```

101 void Update_display(void)
102 {
103     millier = (compteur/1000)%10;
104     Segment[3] = conversion[millier];
105     centaine = (compteur/100)%10;
106     Segment[2] = conversion[centaine];
107     dizaine = (compteur/10)%10;
108     Segment[1] = conversion[dizaine];
109     unite = compteur%10;
110     Segment[0] = conversion[unite];
111 }

```

Cette fonction permet de remplir le tableau situé à la ligne 8 :

```

8 unsigned char Segment[4]={0,0,0,0};

```

Il s'agit du tableau à 4 caractères contenant les 4 digits à envoyer par multiplexage aux différents afficheurs. Afin de voir plus clair, regardons de plus près la fonction `void Interrupt(void)` de notre programme :

```

17 void Interrupt(void){
18     if (TMR0IF_bit){ // Timer0 toutes les 5ms
19         TMR0IF_bit = 0;
20         TMR0H = 0xD8;
21         TMR0L = 0xF0;
22         LATA = 0;
23         LATD = Segment[index];
24         LATA = 1 << index;
25         index++;
26         index = index%4;
27     }
28 }

```

Cette fonction ne fait pas partie du programme principal car il s'agit d'une **routine d'interruption**. Comme expliqué plus haut, elle est générée par le débordement du registre associé à *Timer0* et la levée de l'interruption est réalisée à la ligne 19 :

```
TMROIF_bit = 0;
```

Dès lors, la routine à effectuer pendant l'interruption est placée dès la ligne

22. Analysons chacune de ces instructions :

- $LATA = 0$: On alimente les 4 afficheurs ;
- $LATD = Segment[index]$: On écrit le contenu de la valeur $Segment[index]$ sur **un** des 4 afficheurs ;
- $LATA = 1 \ll index$: On alimente l'afficheur suivant ;
- $index++$: On incrémente $index$ afin de traiter la valeur et l'afficheur suivants ;
- $index = index \% 4$: On borne $index$ entre 0 et 3.

Notons qu'une interruption système doit laisser la main sur le déroulement normal du programme le plus vite possible, sous peine de *louper* des événements faisant partie du programme principal.

7 Conclusion

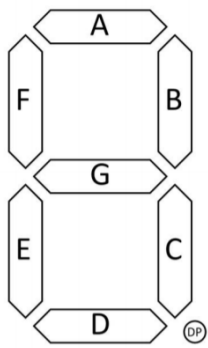
Tout d'abord nous retenons de cette séance que le *multiplexage* est un moyen de transmettre des données sur un même bus de communication. Cela permet d'inhiber un nombre considérable de fils. Nous l'avons utilisé afin de transmettre les données aux différents afficheurs tout en sélectionnant celui qui sera destiné à les afficher. Il s'agit d'une pratique répandue dans les systèmes de communication et est utilisée entre autres dans la téléphonie.

Ensuite cette séance nous a permis de nous familiariser à la notion d'interruptions. C'est une notion fondamentale dans la programmation de μC qui ne possèdent pas de multitâches. Il existe une hiérarchie au sein des interruptions. Ces derniers peuvent être soit logicielles ou matérielles, selon leur nature. Il est primordial de s'assurer de la *levée* de l'interruption et que la routine d'interruption se termine *le plus rapidement* possible.

Enfin, nous constatons qu'il existe des *framework* telles que TIMER CALCULATOR afin de faciliter la mise en place de certaines structures et de nous épargner du temps. Cependant il ne faut jamais perdre de vue le mécanisme et les calculs qui sont derrière sous peine d'être limité par l'utilisation de ce genre de plateforme.

A Annexe : Afficheur 7 segments et *multiplexage*

La programmation du multiplexage



Affichage	Bit 7 DP	Bit 6 G	Bit 5 F	Bit 4 E	Bit 3 D	Bit 2 C	Bit 1 B	Bit 0 A	Valeur Hexa
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F

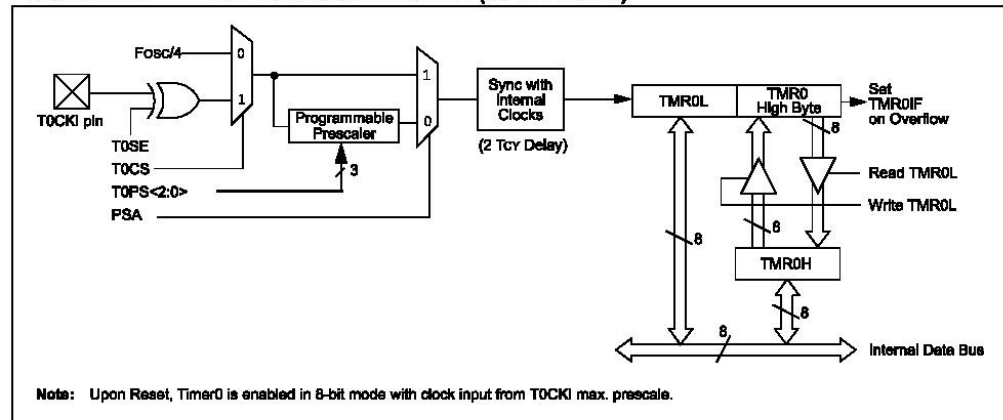
En langage C : `const unsigned char conversion[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};`

FIGURE 6 – Extrait du cours sur l'afficheur 7 segments et le multiplexage

B Annexe : Données provenant du datasheet

PIC18(L)F2X/4XK22

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



11.4 Prescaler

An 8-bit counter is available as a prescaler for the Timer0 module. The prescaler is not directly readable or writable; its value is set by the PSA and T0PS<2:0> bits of the T0CON register which determine the prescaler assignment and prescale ratio.

Clearing the PSA bit assigns the prescaler to the Timer0 module. When the prescaler is assigned, prescale values from 1:2 through 1:256 in integer power-of-2 increments are selectable.

When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g., CLRF TMR0, MOVWF TMR0, BSF TMR0, etc.) clear the prescaler count.

Note: Writing to TMR0 when the prescaler is assigned to Timer0 will clear the prescaler count but will not change the prescaler assignment.

11.4.1 SWITCHING PRESCALER ASSIGNMENT

The prescaler assignment is fully under software control and can be changed "on-the-fly" during program execution.

11.5 Timer0 Interrupt

The TMR0 interrupt is generated when the TMR0 register overflows from FFh to 00h in 8-bit mode, or from FFFFh to 0000h in 16-bit mode. This overflow sets the TMR0IF flag bit. The interrupt can be masked by clearing the TMR0IE bit of the INTCON register. Before re-enabling the interrupt, the TMR0IF bit must be cleared by software in the Interrupt Service Routine.

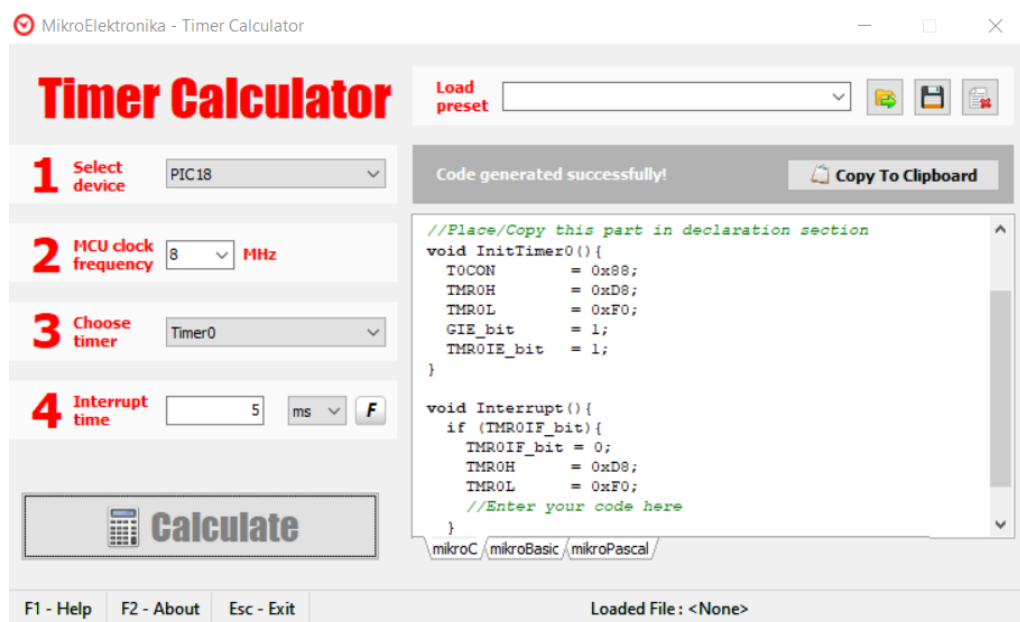
Since Timer0 is shut down in Sleep mode, the TMR0 interrupt cannot awaken the processor from Sleep.

TABLE 11-1: REGISTERS ASSOCIATED WITH TIMER0

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	109
INTCON2	RBPV	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	110
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS<2:0>			154
TMR0H	Timer0 Register, High Byte								—
TMR0L	Timer0 Register, Low Byte								—
TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	151

Legend: — = unimplemented locations, read as '0'. Shaded bits are not used by Timer0.

C Annexe : Logiciel *Timer Calculator* et calcul des valeurs associées



TOCON = 0x88; // 0b10001000
 TMR0H = 0xD8;
 TMR0L = 0xF0;

Chaque interruption du timer 0 aura lieu toutes les **5ms**!

Démonstration:

Fréquence quartz = 8MHz

Prescaler = prédiviseur = 1

Valeur de chargement du Timer 0 = 0xD8F0 = 55536

$$T_{ins} = \frac{4}{F_{quartz}} \cdot \text{prescaler} = 500\text{ns}$$

$$\text{Timer 0 Overflow} = (65536 - 55536) \cdot T_{ins} = 5\text{ms}$$

FIGURE 8 – Logiciel *Timer Calculator* et calcul provenant du cours